Bkd-Tree: Un árbol kd dinámico y escalable

Octavian Procopiuc[1] (^), Pankaj K. Agarwal[(1)^^], Lars Arge[1]), y Jeffrey Scott Vitter[2] (†).

[1] Departamento de Informática, Duke University Durham, NC 27708, EE.UU.
{tavi,pankaj,large}@cs.duke.edu
[2] Departamento de Informática, Purdue University West Lafayette, IN 47907 EE.UU.
jsv@purdue.edu

Resumen. En este trabajo proponemos una nueva estructura de índices, denominada árbol Bkd, para indexar grandes conjuntos de datos puntuales multidimensionales. El árbol Bkd es una estructura de datos dinámica eficiente en E/S basada en el árbol kd. Presentamos los resultados de un amplio estudio experimental que demuestra que, a diferencia de los intentos anteriores de hacer dinámicas las versiones externas del árbol kd, el árbol Bkd mantiene su alta utilización del espacio y su excelente rendimiento de consulta y actualización, independientemente del número de actualizaciones por formadas en él.

1 Introducción

El problema de la indexación de conjuntos de datos puntuales multidimensionales se plantea en muchas aplicaciones y se ha estudiado ampliamente. Se han desarrollado numerosas estructuras, lo que pone de manifiesto la dificultad de optimizar los múltiples requisitos interrelacionados que deben satisfacer estos índices multidimensionales. Más concretamente, un índice eficiente debe tener una alta utilización del espacio y ser capaz de procesar consultas rápidamente, y estas dos propiedades deben mantenerse bajo una carga significativa de actualizaciones. Al mismo tiempo, las actualizaciones también deben procesarse rápidamente, lo que significa que

[s] Con el apoyo de la National Science Foundation a través de la beca de investigación EIA-9870734 y de la Army Research Office a través de la beca MURI DAAH04-96- 1-0013. Parte de este trabajo se realizó durante una visita a BRICS, Universidad de Aarhus, Dinamarca.

[ss] Con el apoyo de la subvención MURI DAAH04-96-1-0013 de la Oficina de Investigación del Ejército, de una beca Sloan, de las subvenciones ITR-333-1050, EIA-9870724 y CCR-9732787 de la NSF y de una subvención de la Fundación Científica Binacional EE.UU.-Israel.

[(sss)] Financiado en parte por la National Science Foundation a través de la beca ESS EIA-9870734, la beca RI EIA-9972879 y la beca CAREER CCR-9984099. Parte de este trabajo se realizó durante una visita a BRICS, Universidad de Aarhus, Dinamarca.

(†) Financiado en parte por la National Science Foundation a través de las becas de investigación CCR-9877133 y EIA-9870734 y por la Army Research Office a través de la beca MURI DAAH04-96-1-0013. Parte de este trabajo se realizó durante una visita a BRICS, Universidad de Aarhus, Dinamarca. Parte de este trabajo se realizó durante una estancia en la Universidad de Duke.

T. Hadzilacos et al. (Eds.): SSTD 2003, LNCS 2750, pp. 46-65, 2003.

la estructura debe cambiar lo menos posible durante las inserciones y supresiones. Esto dificulta el mantenimiento de una buena utilización del espacio y el rendimiento de las consultas a lo largo del tiempo. En consecuencia, la calidad de la mayoría de las estructuras de indexación se deteriora a medida que se realizan en ellas un gran número de actualizaciones, y el problema de manejar cargas masivas de actualización manteniendo al mismo tiempo una alta utilización del espacio y un bajo tiempo de respuesta de las consultas ha sido reconocido como un importante problema de investigación [9].

En este trabajo proponemos una nueva estructura de datos, denominada árbol Bkd, que mantiene su alta utilización del espacio y su excelente rendimiento de consulta y actualización independientemente del número de actualizaciones que se realicen en él. El Bkd-tree se basa en una conocida extensión del kd-tree, llamada K-D-B-tree [22], y en el llamado método logarítmico para dinamizar una estructura estática. Como demostramos mediante amplios estudios experimentales, el árbol Bkd es capaz de alcanzar casi el 100% de utilización del espacio y el rápido procesamiento de consultas de un árbol K-D-B estático. Sin embargo, a diferencia del árbol K-D-B, estas propiedades se mantienen con actualizaciones masivas.

Resultados anteriores. Una de las consultas más fundamentales en las bases de datos espaciales *es la consulta de rango* ortogonal o *consulta de ventana*. En dos dimensiones, una consulta de ventana es un rectángulo alineado con un eje y el objetivo es encontrar todos los puntos de la base de datos dentro del rectángulo. Se han propuesto numerosas estructuras de indexación de puntos multidimensionales prácticamente eficientes para las consultas de ventana, la mayoría de las cuales también pueden responder a otros tipos de consultas. Entre ellas se encuentran los árboles K-D-B [22], los árboles hB [18, 10] y los árboles R [13, 6]. Si N es el número total de puntos y B es el número de puntos que caben en un bloque de disco, $\Omega(+ K/B)$ es el límite inferior teórico del número de E/S que necesita un índice de espacio lineal para responder a consulta de ventana 15]. Aquí K es el número de puntos del rectángulo de consulta. En la práctica, las estructuras de indexación anteriores suelen responder a las consultas con un número de E/S mucho menor. Sin embargo, su rendimiento puede deteriorarse seriamente tras un gran número de actualizaciones. Recientemente, se han desarrollado una serie de estructuras espaciales lineales con un rendimiento de consulta y actualización eficiente garantizado en el peor de los casos (véase, por ejemplo, [5, 15, 12]). Los llamados cross-trees [12] y O-trees [15] responden a consultas de ventana en el número óptimo de E/S y pueden actualizarse, teóricamente, en $O(\log_B N)$ E/S, pero tienen un interés práctico limitado porque un análisis teórico muestra que su rendimiento medio de consulta está cerca del rendimiento en el peor de los casos. Véase
p. ej. [11, 3] para estudios más completos de estructuras de indexación multidimensionales. Aunque algunas de las estructuras de indexación anteriores están diseñadas específicamente para memoria externa, muchas de ellas son adaptaciones de estructuras diseñadas para memoria principal. En este artículo sólo nos centramos en las adaptaciones a memoria externa del árbol kd original propuesto por Bentley [7] (véase también [23]).

Árboles kd dinámicos de memoria externa. Mientras que las versiones estáticas del árbol kd han demostrado tener un excelente rendimiento de consulta en muchas situaciones prácticas, ha resultado difícil desarrollar una versión dinámica eficiente. A continuación, presentamos una breve descripción de la estructura del árbol kd de memoria interna y analizamos los dos enfoques anteriores más importantes para obtener árboles kd dinámicos de memoria externa. En dos dimensiones, el árbol kd consiste en un árbol binario de altura $\lceil \log_2 N \times$.

que representa una descomposición recursiva del plano mediante líneas ortogonales al eje que dividen el conjunto de puntos en dos subconjuntos iguales[1] En los niveles pares, la línea es ortogonal *al eje x*, mientras que en los niveles impares es ortogonal *al eje y*. Los propios puntos de datos se almacenan en las hojas, que forman una partición de el plano en regiones rectangulares disjuntas que contienen un punto cada una. En el En el peor de los casos, una consulta de ventana en un árbol kd requiere un tiempo de $O(N+K)$ [16], pero El análisis de casos medios [24] y los experimentos han demostrado que, en la práctica, suele funcionar mucho mejor. Una forma de realizar una inserción en un árbol kd consiste en buscar primero en el árbol la hoja al rectángulo que contiene el punto y, a continuación, dividir esta hoja en dos para acomodar el nuevo punto. Aunque este procedimiento de inserción se ejecuta eficientemente en tiempo $O(\log_2 N)$, el árbol kd puede desequilibrarse cada vez más cuando se muchas inserciones, lo que deteriora el rendimiento de la consulta. De hecho, el árbol resultante ya no es un árbol kd, puesto que las líneas de los nodos internos ya no dividen los puntos en conjuntos de igual tamaño. Desafortunadamente, mientras que muchas otras estructuras de árbol pueden reequilibrarse eficientemente en un tiempo proporcional a la ruta raíz-hoja, se puede demostrar que para reequilibrar un árbol kd después de una inserción, puede que necesitemos reorganizar grandes partes del árbol [23]. Por lo tanto, parece difícil soportar inserciones de forma eficiente y, al mismo tiempo, mantener un buen rendimiento de las consultas. Estas consideraciones demuestran que el árbol kd es principalmente una estructura de datos estática con un rendimiento de consulta de ventana muy bueno.

Uno de los principales problemas a la hora de adaptar el árbol kd a la memoria externa es cómo asignar nodos a bloques de disco para obtener una buena utilización del espacio (usar cerca de N/B bloques de disco) y un buen rendimiento de las consultas de E/S. En la primera adaptación del árbol kd a la memoria externa, denominada árbol K-D-B [22], el árbol kd se organiza como un árbol B^+ . Más concretamente, un árbol K-D-B es un árbol multidireccional con todas las hojas en el mismo nivel. Cada nodo interno v corresponde a una región rectangular y los hijos de v definen una partición disjunta de esa región obtenida mediante un esquema de partición kd-árbol. Los puntos se almacenan en las hojas del árbol, y cada hoja y nodo interno se almacenan en un bloque de disco. Al igual que un árbol kd, un árbol K-D-B puede cargarse de forma masiva, de modo que presenta una excelente utilización del espacio (utiliza cerca de N/B bloques) y responde a las consultas de forma eficiente en E/S (en el peor de los casos, de forma óptima en $O(N/B+K/B)$, pero a menudo mucho mejor en la práctica). Desgraciadamente, también presenta las características de inserción del árbol kd. Para insertar un punto en un árbol K-D-B-, se sigue una ruta de raíz a hoja en $\log_B(N/B)$ E/S y, tras insertar el punto en una hoja, la hoja y posiblemente otros nodos de la ruta se dividen igual que en un árbol B^+ -. Sin embargo, a diferencia del árbol B^+ pero de forma similar al árbol kd, la división de un nodo interno v puede dar lugar a la necesidad de dividir varios de los subárboles enraizados en los hijos de v (véase la Figura 1). Como resultado, las actualizaciones pueden ser muy ineficientes y, lo que es quizá más importante, la utilización del espacio puede disminuir drásticamente, ya que el proceso de división puede generar muchas hojas casi vacías [22].

Tras el árbol K-D-B, se han propuesto otras adaptaciones del árbol kd a la memoria externa. Un importante avance se produjo con el resultado

[1] Para simplificar, en este sólo consideramos árboles kd bidimensionales. Sin embargo, todos nuestros resultados funcionan en d dimensiones.

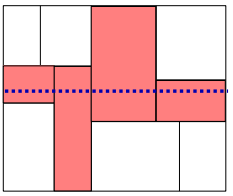


Fig. 1. División de un nodo del árbol K-D-B. El rectángulo exterior corresponde a un nodo *v* que se divide. Las regiones más oscuras corresponden a los hijos que deben dividirse recursivamente cuando *v* se divide.

de Lomet y Salzberg [18]. Su estructura, denominada hB-tree (holey brick tree), mejoró significativamente el rendimiento de actualización con respecto al K-D-B-tree. El mejor rendimiento se obtuvo dividiendo sólo los nodos en una ruta de raíz a hoja después de una inserción. Sin embargo, para poder hacerlo, hubo que cambiar la definición de los nodos internos de modo que ya no correspondieran a rectángulos simples, sino a rectángulos de los que se han eliminado rectángulos más pequeños (holey bricks). El algoritmo de actualización del árbol hB es teóricamente eficaz, aunque bastante complicado. Como mostramos en nuestros resultados experimentales, el árbol hB todavía puede sufrir una utilización degenerada del espacio, aunque en menor medida que el árbol K-D-B (véase también [10]). Todos los demás intentos de externalizar el árbol kd sufren ineficiencias similares.

Nuestros resultados. En este trabajo presentamos la primera adaptación *dinámica* teórica y prácticamente eficiente del árbol kd a la memoria externa. Nuestra estructura, a la que llamamos árbol Bkd, mantiene la alta utilización del almacenamiento y la eficiencia de consulta de un árbol K-D-B estático, a la vez que soporta actualizaciones de E/S de forma eficiente. Hemos llevado a cabo experimentos exhaustivos que demuestran que el árbol Bkd supera a los enfoques anteriores en términos de utilización del almacenamiento y tiempo de actualización, al tiempo que mantiene un rendimiento de consulta similar.

Los principales ingredientes utilizados en el diseño del árbol Bkd son un algoritmo de carga masiva de árboles K-D-B eficiente en E/S y el llamado método logarítmico para dinamizar una estructura de datos estática [8, 21]. En lugar de mantener un árbol y reequilibrarlo dinámicamente tras una inserción, mantenemos un conjunto de $\log_2(N/M)$ árboles K-D-B estáticos y realizamos actualizaciones reconstruyendo un conjunto cuidadosamente elegido de las estructuras a intervalos regulares (M es la capacidad del buffer de memoria, en número de puntos). De este modo, mantenemos una utilización del espacio cercana al 100% del árbol K-D-B estático. La idea de mantener múltiples árboles para acelerar el tiempo de inserción también ha sido utilizada por O'Neill et al. [20] y Jagadish et al. [14]. Sus estructuras se utilizan para indexar puntos en un único atributo y sus técnicas no pueden extenderse para manejar eficientemente puntos multidimensionales.

Para responder a una consulta de ventana utilizando el árbol Bkd, tenemos que consultar todas las estructuras $\log_2(N/M)$ en lugar de sólo una, pero teóricamente mantenemos el límite de consulta óptimo en el peor de los casos $O(\sqrt{N/B} + K/B)$. Utilizando un algoritmo de carga masiva de E/S, se realiza una inserción en mal $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$

$O(\frac{1)}{B}(\log_{M/B}\frac{N}{B})(\log_2\frac{N)}{M})$ E/S amortizadas. Este límite es mucho menor que el $O(\log_B N)$ B^+ -tree update bound para todos los propósitos prácticos. Una desventaja de la reconstrucción periódica es, por supuesto, que el límite de actualización varía de una actualización a otra (de ahí el resultado amortizado). Sin embargo, las consultas pueden responder mientras se realiza una actualización (reconstrucción) y (al menos teóricamente) el límite de actualización se puede hacer en el peor de los casos utilizando almacenamiento adicional [21]. Aunque nuestro árbol Bkd tiene buenas propiedades teóricas, la principal contribución de este artículo es una prueba de su viabilidad práctica. Presentamos el resultado de un amplio estudio experimental del rendimiento del árbol Bkd en comparación con el árbol K-D-B- utilizando datos reales (TIGER) y generados artificialmente (uniformes). Además, utilizamos una familia de conjuntos de datos cuidadosamente elegidos para demostrar que tanto el árbol K-D-B como el árbol $hB^{(\Pi)}$ (una versión mejorada del árbol hB, véase [10]) pueden tener una mala utilización del espacio (tan baja como el 28% para el árbol K-D-B y el 36% para el árbol hB^{Π}), mientras que la utilización del espacio del árbol Bkd es siempre superior al 99%. Al mismo tiempo, una inserción en un árbol Bkd puede ser hasta 100 veces más rápida que una inserción en el árbol K-D-B, en el sentido amortizado. La principal cuestión práctica es, por supuesto, cómo afecta el uso de estructuras $\log_2(N/M)$ al rendimiento de la consulta. Aunque se mantiene la eficiencia teórica en el peor de los casos, la consulta de varias estructuras en lugar de una sola da lugar a un mayor número de E/S aleatorias en comparación con las E/S más localizadas en una sola estructura. Nuestros experimentos demuestran que la diferencia es nula o relativamente pequeña y, por tanto, que la dinámica Bkd mantiene el excelente rendimiento de consulta del árbol K-D-B estático. Por último, consideramos que la demostración de la eficacia práctica del loga-

El método logarítmico es una importante contribución general de este artículo; aunque el objetivo principal del artículo es hacer que el árbol kd sea dinámico, el método logarítmico es aplicable a cualquier estructura de índice para la que se conozca un algoritmo de carga masiva eficiente. Así pues, nuestros resultados sugieren que, en general, podríamos hacer que las estructuras de índices estáticas eficientes en la práctica fueran dinámicamente eficientes utilizando el método.

El resto de este artículo se divide en tres secciones. En la sección 2 se presentan los detalles del árbol Bkd. A continuación, en la sección 3, se describen el hardware, el software y los conjuntos de datos utilizados en nuestro estudio experimental. Los resultados del experi- mento se presentan y analizan en la sección 4.

2 Descripción del árbol Bkd

Como ya se ha mencionado, el árbol Bkd está formado por un conjunto de árboles kd equilibrados. Cada árbol kd se dispone (o *bloquea*) en el disco de forma similar a como se el árbol K-D-B. Para almacenar un árbol kd determinado en el disco, primero modificamos las hojas para que contengan B puntos, en lugar de sólo uno. De este , los puntos se empaquetan en bloques N/B. Para empaquetar los nodos internos del árbol kd, ejecutamos el algoritmo siguiente. Sea B_i el número de nodos que caben en un . Supongamos primero que N/B es una potencia exacta de B_i, es decir, $N/B = {}^{B(p)}{}_{i}$, para algún p, y que B_i es una potencia exacta de 2. En este caso, los nodos internos pueden almacenarse fácilmente en bloques $O(N/(BB_i)$ de forma natural. Partiendo de la raíz v del árbol kd, almacenamos juntos los nodos obtenidos

realizando una búsqueda exhaustiva empezando por v, hasta que se hayan recorrido B_i nodos. A continuación, se bloquea recursivamente el resto del árbol. Con este procedimiento, el número de bloques necesarios para todos los nodos internos es $O(N/(BB_i)$, y el número de bloques tocados por una ruta raíz-hoja -la ruta recorrida durante la búsqueda de la raíz- *es* $O(N/(BB_i)$.
es $log_{(B)_i}(N/B)+1 = \Theta(\log_B(N/B))$. Si N/B no es una potencia de B_i, llenamos el bloque que contiene la raíz del árbol kd con menos de B_i nodos para poder bloquear el resto del árbol como se ha indicado anteriormente. Si N/B no es una potencia de 2, el árbol kd está desequilibrado y el algoritmo de bloqueo anterior puede acabar infrautilizando los bloques de disco. Para paliar este problema modificamos el método de división del árbol kd
y se dividen en elementos de rango potencia de 2, en lugar de en los elementos medianos. Más concretamente, al construir los dos hijos de un nodo v a partir de un conjunto de p puntos, asignamos $2^{\lfloor \log_2 p \rfloor}$ puntos al hijo izquierdo, y el resto al hijo derecho. Este
sólo los bloques que contienen el camino más a la derecha -como máximo $\log_{B_i}(N/B)$- pueden estar infrautilizados.

A partir de ahora, cuando nos refiramos a un árbol kd, nos referiremos a un árbol almacenado en disco como se ha descrito anteriormente.

2.1 Carga masiva de árboles kd

Clásicamente, un árbol kd se construye de arriba abajo, como se indica en la Figura 2 (columna izquierda). El primer paso consiste en ordenar la entrada en ambas coordenadas. A continuación (en el paso 2) se construyen los nodos de recursiva, empezando por la raíz. Para un nodo $v$determinamos la posición de división leyendo la mediana de uno de los dos conjuntos ordenados asociados a v (cuando la división es ortogonal *al eje* x, utilizamos el archivo ordenado en x, y cuando la división es ortogonal *al eje* y, utilizamos el archivo ordenado en y). Finalmente escaneamos estos conjuntos ordenados y distribuimos cada uno de ellos en dos conjuntos y recursivamente construimos los hijos de v. Dado que el árbol kd en N puntos tiene altura $\log_2(N/B)$ y cada punto de entrada se lee dos veces y se escribe dos veces en cada nivel, el algoritmo realiza $O((N/B)\log_2(N/B))$ E/S, más el coste de la ordenación, que es $O((N/B)\log_{M/B}(N/B))$ E/S [2], para un total de $O((N/B)\log_2(N/B))$ E/S.

En [1] se propuso un método mejorado de carga masiva. En lugar de construir un nivel cada vez, este algoritmo construye un subárbol completo de $\Theta(\log_2(M/B))$-altura del árbol kd cada vez. Los principales pasos del algoritmo se describen en la figura 2 (columna derecha). Como antes, el primer paso es ordenar la entrada en ambos
coordenadas. A continuación (en el paso 2) construimos los niveles $\log_2 t$ superiores del árbol kd utilizando
sólo tres pasadas sobre el archivo de entrada, donde $t = \Theta(\min\{M/B, M\})$.
Conseguimos
Para ello, primero se determina una cuadrícula $t \times t$ en los puntos de entrada: se eligen t líneas de cuadrícula horizontales (verticales) (en el paso 2a) de modo que cada franja horizontal (vertical) contenga N/t puntos (véase la figura 3(a)). A continuación (en el paso 2b) se calcula el número de puntos de cada celda de la cuadrícula simplemente escaneando el archivo de entrada. Estos recuentos se almacenan en una matriz de cuadrícula $t \times t$ A, guardada en la memoria interna (el tamaño de la matriz, t^2, es como máximo M). El subárbol superior de altura $\log_2 t$ se calcula ahora (en el paso 2c) utilizando un enfoque descendente. Supongamos que el nodo raíz divide los puntos mediante una línea vertical. Esta línea de división puede determinarse calculando primero (utilizando los recuentos de celdas de la matriz A) la franja *vertical* X_k que contiene la línea. Después podemos

Alg oritmo Bulk Load (binario)	**Algoritmo Bulk Load (grid)**
(1) Crea dos listas ordenadas;	(1) Crear dos listas ordenadas;
(2) Construye el árbol kd de arriba abajo: Empezando por el nodo raíz, haga los siguientes pasos para cada nodo, y *t* líneas de rejilla ortogonales al eje y:	(2) Construir $\log_2 t$ niveles del árbol kd: (a) Calcule *t* líneas de rejilla ortogonales a *t* líneas de rejilla ortogonales al eje *x* t ortogonales al eje *y*;
(a) Halla la línea de partición;	(b) Crear la matriz de rejilla *A* contener- celdas de la cuadrícula;
(b) Distribuya las entradas en dos conjuntos, basado en la línea de partición;	(c) Crear un subárbol de altura $\log_2 t$, utilizando los recuentos de la rejilla ma- trix; (d) Distribuir la entrada en *t* conjuntos, cor- respondiendo a las *t* hojas;
	(3) Construir los niveles inferiores en la memoria principal o repitiendo el paso (2).

Fig. 2. Dos algoritmos para la carga masiva de un árbol kd

calcula fácilmente qué bloque leer de la lista ordenada por *la* coordenada *x* para determinar el punto que define la división. *A* continuación, la matriz de la cuadrícula *A* se divide en dos nuevas matrices, $A^<$ y $A^>$, que almacenan los recuentos de celdas de la cuadrícula desde la izquierda y desde derecha de la línea de división, respectivamente. Esto puede hacerse escaneando el contenido de la franja *vertical* X_k. La figura 3(b) muestra cómo una celda $C_{j,k}$ de la matriz original se divide en dos celdas, $C^<_{j,k}$ y $C^>_{j,k}$. El número de puntos en $C^<_{j,k}$ se almacena en $A^<_{j,k}$, y el número de puntos en $C^>_{j,k}$ se almacena en $A^>_{j,k}$, para cada j, $1 \le j \le t$. Utilizando las matrices $A^<$ y $A^>$, la división correspondiente a dos hijos de *v* puede ser calculada recursivamente. Para cada nodo producido, el tamaño de la matriz *A* en la memoria interna crece en *t* celdas. Como $t \le O(M)$, aún cabe en memoria después de produciendo $\log_2 t$ niveles, es decir $2^{\log_2 t} = t$ nodos, del árbol. Después de producir este número de niveles, el subárbol resultante determina una partición del espacio en *t*

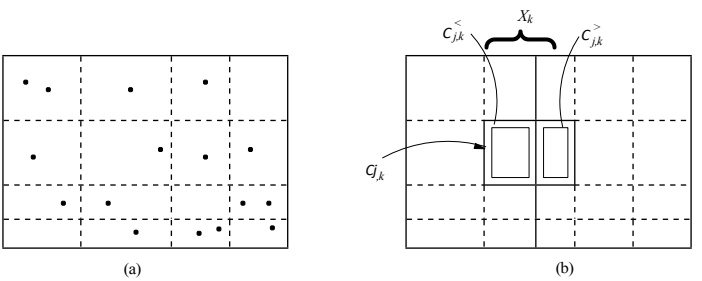


Fig. 3. Hallazgo de la mediana utilizando celdas de cuadrícula. (a) Cada franja contiene *N/t* puntos.
(b) Las celdas $C^<_{j,k}$ y $C^>_{j,k}$ se calculan dividiendo la celda $C_{j,k}$

rectángulos. En este punto distribuimos los puntos de entrada en estos rectángulos escaneando la entrada y, para cada punto p, utilizando el subárbol construido para encontrar el rectángulo que contiene a p (Paso 2d). Si la memoria principal puede contener $t+1$ bloques-
uno para cada rectángulo de la partición, más uno para la entrada: la distribución bution
puede realizarse en $2N/B$ E/S. Esto explica la elección de $t = \Theta(\min(M/B, \quad M))$.
Por último, los niveles inferiores del árbol se construyen (en la etapa 3) repitiendo la etapa 2 o, si el conjunto de puntos cabe en la memoria internaen memoria y aplicándole el algoritmo de carga masiva binaria.

Dado que el paso 2 escanea los puntos de entrada dos veces, se deduce que se pueden construir $\Theta(\log_2(M/B))$ niveles del árbol kd usando $O(N/B)$ E/Ss. Por lo tanto, todo el árbol kd se construye en $O((N/B)\ \log_{M/B}(N/B))$ E/S. Esto es un factor de $\Theta(\log_2(M/B))$ mejor que el algoritmo binario de carga masiva. A efectos prácticos, el factor logarítmico es como máximo 3, por lo que la complejidad de la carga masiva es lineal.

El algoritmo presentado en esta sección utiliza sólo las características de la memoria interna kd-tree, y no la disposición específica del disco. En consecuencia, otras estructuras de datos eficientes en E/S basadas en el árbol kd pueden cargarse masivamente utilizando este algoritmo. En particular, el algoritmo puede usarse fácilmente para cargar en bloque un árbol hB$^{\Pi}$- , que fue mencionado como un problema abierto en [10].

2.2 Actualizaciones dinámicas

Un árbol Bkd sobre N puntos del plano está formado por $\log_2(N/M)$ árboles kd. *i-ésimo* árbol kd, T_i, está vacío o contiene exactamente $2^i M$ puntos. Así, T_0 almacena en como máximo M puntos. Además, una estructura T^M_0 que contenga a lo sumo M puntos es
en la memoria interna. La figura 4 muestra la organización del árbol Bkd. Esta organización es similar a la utilizada por el método logarítmico [8, 21].

Los algoritmos para insertar y borrar un punto se describen en la Figura 5. El más sencillo de los dos es el algoritmo de borrado. El más sencillo de los dos es el algoritmo de borrado. Basta con consultar cada uno de los árboles para encontrar el árbol T_i que contiene el punto y borrarlo de T_i. Como hay un máximo de $\log_2(N/M)$ árboles, el número de E/S realizadas por un borrado es $O(\log_B(N/B)\ \log_2(N/M))$.

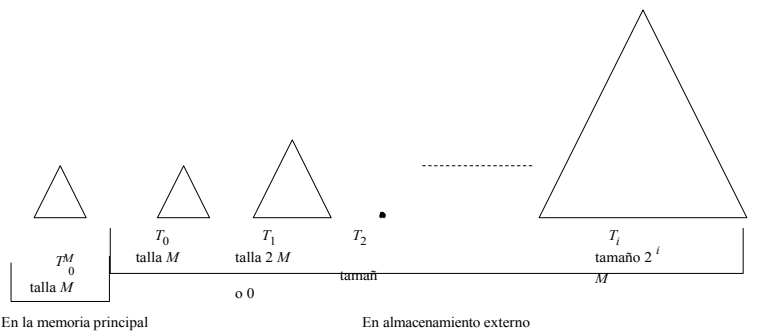


Fig. 4. El bosque de árboles que compone la estructura de datos. En este , T_2 está vacío

Algoritmo Insert(p)
(1) Insertar p en la memoria intermedia T_0^M;
(2) Si T_0^M no está lleno, devolver; en caso contrario, encontrar el primer árbol vacío T_k y extrazar todos los puntos de T_0^M y T_i, $0 \le i < k$ en un fichero F;
(3) Carga masiva T_k de los artículos de F;
(4) Vacío T_0^M y T_i, $0 \le i < k$.

Algoritmo Delete(p)
(1) Consultar T_0^M con p; si se encuentra, borrarlo y volver;
(2) Consultar cada árbol no vacío del bosque (empezando por T_0) con p; si se encuentra, borrarlo y devolver;

Fig. 5. Los algoritmos **Insert** y **Delete** para el árbol Bkd

El algoritmo de inserción es fundamentalmente diferente. La mayoría de las inserciones ($M - 1$ de cada M) se realizan directamente en la estructura en memoria T_0^M. Siempre que T_0^M se llena, encontramos el k más pequeño tal que T_k es un vacío kd-árbol. A continuación extraemos todos los puntos de T_0^M y T_i, $0 \le i < k$, y cargamos en bloque el árbol T_k a partir de estos puntos. Nótese que el número de puntos almacenados ahora en T_k es efectivamente $2^k M$ ya que T_0^M almacena M puntos y cada T_i, $1 \le i < k$, almacena exactamente $2^i M$ puntos (T_k fue el primer árbol kd vacío). Por último, vaciamos T_0^M y T_i, $0 \le i < k$. En otras palabras, los puntos se insertan en la estructura en memoria y se "empujan" gradualmente hacia árboles kd más grandes mediante reorganizaciones periódicas de pequeños árboles kd en un gran árbol kd. Cuanto mayor sea árbol kd, menos veces habrá que reorganizarlo.

Para calcular el número amortizado de E/S realizadas por una inserción, considere N inserciones consecutivas en un árbol Bkd inicialmente vacío. Cada vez que se construye un nuevo árbol kd T_k, éste sustituye a todos los árboles kd T_j, $1 \le j < k$, y a la estructura en memoria T_0^M. Esta operación requiere $O((2^k M/B) \log_{M/B}(2^k M/B))$ E/S (carga masiva T_k) y mueve exactamente $2^k M$ puntos en el árbol kd más grande T_k. Si dividimos la construcción de T_k entre estos puntos, cada uno de ellos tiene que pagar $O((1/B) \log_{M/B}(2^k M/B)) = O((1/B) \log_{M/B}(N/B))$ E/S. Como los puntos sólo se mueven a kd-árboles más grandes, y hay como mucho $\log_2(N/M)$ kd-árboles, un punto puede ser cargado como mucho $\log_2(N/M)$ veces. Por tanto, el coste final amortizado de una inserción es $O\left(\frac{\log_{(M/B)}(N/B)\log_2(N/M)}{B}\right)$ E/S.

2.3 Consultas

Para responder a una consulta de ventana en el árbol Bkd basta con consultar todos los $\log_2(N/M)$ árboles kd. En el peor de los casos, el rendimiento de una consulta de ventana en un árbol kd que almacena N puntos es de $O(\sqrt{N/B} + K/B)$ E/S, donde K es el número de puntos de la ventana de consulta. Dado que los árboles kd que forman el árbol Bkd son geométricamente crecientes en tamaño, el rendimiento en el peor de los casos del árbol Bkd es también $O(\sqrt{N/B} + K/B)$ E/Ss. Sin embargo, dado que el rendimiento medio de consulta de ventana de un árbol kd suele ser mucho mejor que este rendimiento en el peor de los casos [24], es importante investigar cómo influye el uso de varios árboles kd en el rendimiento práctico del árbol Bkd en comparación con el árbol kd.

3 Plataforma experimental

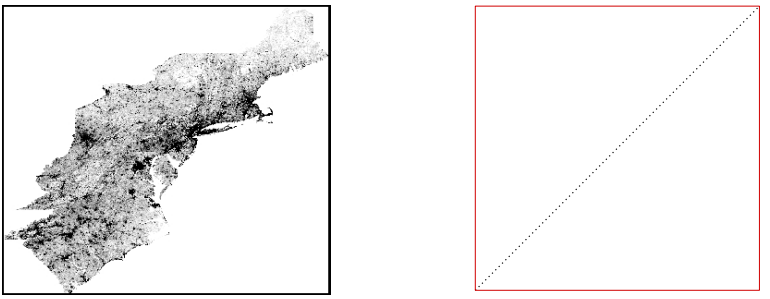
En esta sección describimos la configuración de nuestros estudios experimentales, proporcionando información detallada sobre el software, el hardware y los conjuntos de datos que se utilizaron.

Plataforma de software. Hemos implementado el árbol Bkd en C++ utilizando TPIE. TPIE [4] es una biblioteca de plantillas que proporciona soporte para implementar algoritmos y estructuras de datos eficientes en E/S. En nuestra implementación utilizamos un tamaño de bloque de 16 KB para los nodos internos (siguiendo las sugerencias de Lomet [17] para el árbol B). En nuestra implementación usamos un tamaño de bloque de 16KB para los nodos internos (siguiendo las sugerencias de Lomet [17] para el árbol B), lo que resulta en un fanout máximo de 512. Las hojas de un árbol kd, almacenadas también en bloques de 16 KB, contienen un máximo de 1364 elementos (clave, puntero). Hemos implementado el árbol Bkd utilizando el algoritmo grid bulk loading durante las inserciones y
una matriz lineal como estructura de memoria interna T_0^M (datos más sofisticados
para mejorar el rendimiento de la CPU). Para comparar, también hemos implementado el árbol K-D-B, siguiendo de cerca los detalles proporcionados en el artículo original [22] en relación con el algoritmo de inserción. Como se menciona en la Introducción, el árbol K-D-B es el punto de partida del árbol hB [18] y del árbol hB^{Π}- [10]. Este último es el más avanzado en estructuras de indexación de datos para puntos multidimensionales. Para los experimentos de utilización del espacio utilizamos la implementación de los autores del árbol hB^{Π}. La implementación proporcionada está en memoria, pero simula las E/S contando los accesos a los bloques de datos. Para el resto de los experimentos, decidimos no utilizar esta implementación del árbol hB^{Π}, ya que queríamos enfatizar los tiempos de ejecución del árbol Bkd para conjuntos de datos mucho mayores que la memoria principal.

Conjuntos de datos. W e eligieron tres tipos diferentes de conjuntos de puntos para nuestros experimentos: puntos reales de los datos de TIGER/Line [25], puntos distribuidos uniformemente y puntos a lo largo de la diagonal de un cuadrado. Los datos reales consisten en seis conjuntos de puntos generados a partir de las características de las carreteras en los archivos TIGER/Line. El conjunto TIGER *i*, 1 i 6, consta de todos los puntos de los CD-ROMs 1 a *i*. Obsérvese que el conjunto más grande, el conjunto TIGER 6, contiene todos los puntos de las características de las carreteras de todo Estados Unidos y su tamaño es de 885MB. La tabla 1 contiene los tamaños de los 6 conjuntos de datos. La figura 6(a) muestra el conjunto TIGER 1, que representa 15 estados del este de Estados Unidos. Puede observarse que los puntos están algo agrupados, correspondiendo los conglomerados a zonas urbanas. Los datos uniformes constan de seis conjuntos, cada uno de los cuales contiene puntos distribuidos uniformemente en una región cuadrada. El conjunto más pequeño contiene 20 millones de puntos, mientras que el más grande contiene 120 millones de puntos. La tabla 2 contiene los tamaños de los 6 conjuntos. El último grupo de conjuntos contiene puntos dispuestos en la diagonal de un cuadrado, como se muestra en la figura 6(b). Utilizamos estos conjuntos sólo para experimentos de utilización del espacio. En todos los , un punto consta de

Tabla 1. Tamaños de los conjuntos TIGER.

Establecer	1	2	3	4	5	6
Número de puntos	15483533	29703113	39523372	54337289	66562237	77383213
Tamaño (MB)	177.25	340.00	452.38	621.94	761.82	885.69



(a) (b)

Fig. 6. (a) Una imagen del conjunto TIGER 1 (todos los puntos de las características de las carreteras de 15 estados del este de EE.UU.). La zona blanca no contiene puntos. Las regiones más oscuras tienen la mayor densidad de puntos. (b) Un conjunto de datos diagonal

de tres valores enteros: la *coordenada x*, la *coordenada y* y un ID, para un total de 12 bytes por punto. Así, el mayor conjunto de datos con el que hemos realizado pruebas, que contiene 120 millones de puntos, utiliza 1,34 GB de almacenamiento.

Plataforma de hardware. Utilizamos una estación de trabajo dedicada Dell PowerEdge 2400 con un procesador Pentium III/500MHz, que ejecutaba FreeBSD 4.3. Se utilizó un disco SCSI de 36 GB (IBM Ultrastar 36LZX) para almacenar todos los archivos necesarios: los puntos de entrada, la estructura de datos y los archivos temporales. La máquina tenía 128 MB de memoria, pero restringimos la cantidad de memoria que TPIE podía utilizar a 64 MB. El resto se utilizaba en y los demonios del sistema operativo. Utilizamos deliberadamente una pequeña, , para obtener una gran relación entre el tamaño de los datos y el tamaño de la memoria.

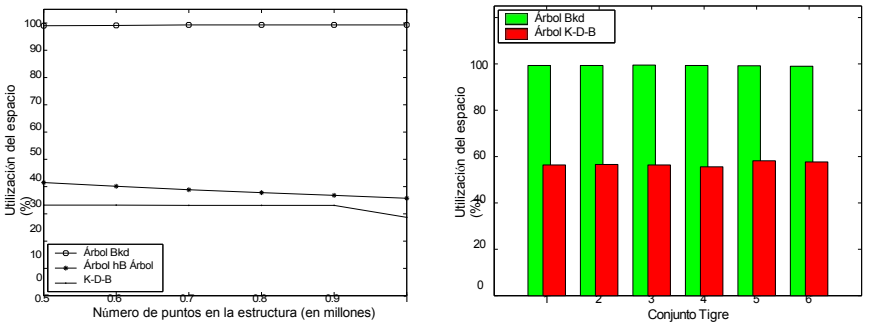
4 Resultados experimentales

4.1 Utilización del espacio

Como ya se ha mencionado, el árbol Bkd utiliza casi el 100% del espacio. Para contrastarlo con la utilización del espacio del árbol K-D-B y el árbol hB$^{\Pi}$, insertamos los puntos de cada uno de conjuntos de datos diagonales, ordenados por *coordenada x*, en las tres estructuras de datos, y medimos la utilización final del espacio. Los resultados se muestran en la figura 7(a). Como era de esperar, la utilización del espacio del árbol Bkd es casi del 100% (entre el 99,3% y el 99,4%). En el caso del árbol K-D-B, la utilización del espacio es tan baja como el 28%, mientras que en el caso del árbol hB$^{\Pi}$ es tan baja como el 38%. En el caso del árbol

Tabla 2. Tamaños de los conjuntos de datos uniformes

Establecer	1	2	3	4	5	6
Número de puntos (millones)	20	40	60	80	100	120
Tamaño (MB)	228.88	457.76	686.65	915.53	1144.41	1373.29



(a) (b)

Fig. 7. (a) Utilización del espacio para los conjuntos diagonales. (b) Utilización del espacio para los conjuntos TIGER.

K-D-B-tree, el patrón diagonal hace que la mayoría de las hojas del árbol estén dentro de rectángulos largos y delgados, con puntos concentrados en un extremo del rectángulo. Cuando se divide un nodo interno, algunas de estas hojas se cortan, dando lugar a hojas vacías. A medida que los conjuntos de datos se hacen más grandes, el efecto se agrava (las hojas vacías también se dividen), lo que resulta en una utilización del espacio cada vez menor. En el caso del árbol hB(, las divisiones de nodos no se propagan a las hojas. De hecho, la utilización del espacio de las hojas se mantiene en el 50% o más, como se indica en [10]. Sin embargo, las divisiones de nodos causan redundancia: Algunos nodos del árbol kd se almacenan en varios nodos del árbol hB. En consecuencia, el tamaño del índice crece drásticamente, lo que resulta en un bajo fanout, una gran altura del árbol y una mala utilización general del espacio. En nuestros experimentos, el árbol K-D-B tenía menor altura que el correspondiente árbol hB^{Π}.

Estos resultados subrayan la sensibilidad del árbol K-D-B y del árbol $hB^{(\Pi)}$. a la distribución de los datos y al orden de inserción. De hecho, se obtiene una utilización del espacio mucho mejor cuando los puntos de un conjunto de datos diagonal se insertan en orden aleatorio, en lugar de ordenados según la coordenada *x*.

Para investigar la utilización del espacio para conjuntos de datos más realistas en la práctica, repetimos el experimento utilizando los datos de TIGER. Las estructuras se construyeron mediante inserciones repetidas, y el orden de inserción viene dado por el orden en que se almacenaron los puntos en los datos originales de TIGER/Line. Por desgracia, no pudimos realizar los experimentos con el árbol hB(Π) en un tiempo razonable. Los experimentos con conjuntos de datos TIGER más pequeños muestran que la utilización del espacio del árbol hB^{Π} es de alrededor del 62% (consistente con los resultados reportados en [10] para datos geográficos similares). Aunque no son tan extremos como los conjuntos diagonales, los conjuntos de datos de la vida real dan como resultado una utilización del espacio relativamente pobre (véase la Figura 7(b)). Para estos conjuntos, la utilización del espacio del árbol K-D-B se sitúa en torno al 56%, aún lejos de la utilización del 99,4% del árbol Bkd.

4.2 Rendimiento de carga a granel

Para comparar los dos algoritmos de carga masiva de árboles kd presentados en la sección 2.1, los probamos con conjuntos de datos uniformes y reales. La Figura 8 muestra el rendimiento de los conjuntos de datos uniformes y la Figura 9, el de los conjuntos de datos TIGER. Las cifras reflejan únicamente el tiempo de construcción, sin tener en cuenta el tiempo necesario para ordenar el conjunto de datos en cada coordenada, que es común a los dos métodos.

Los experimentos con datos distribuidos uniformemente (Figura 8(a)) muestran que, en términos de tiempo de ejecución, el método de malla es al menos dos veces más rápido que el método binario y, como predice el análisis teórico, el aumento de velocidad aumenta con el incremento del tamaño del conjunto. Si se compara el número de E/S (Figura 8(b)), la diferencia es aún mayor. Para entender mejor la diferencia en el número de E/S realizadas por los dos métodos, podemos hacer un cálculo "retrospectivo": para el mayor tamaño probado, el método binario lee el archivo de entrada 14 veces y lo escribe 13 veces (dos lecturas y dos escrituras para cada uno de los niveles superiores, y dos lecturas y una escritura para los niveles inferiores, que se calculan en memoria), mientras que el método de rejilla lee el archivo de entrada 5 veces y lo escribe 3 veces (una lectura para calcular la matriz de rejilla, dos lecturas y dos escrituras para todos los niveles superiores, y dos lecturas y una escritura para los niveles inferiores, que se calculan en memoria). Esto significa que el método de la rejilla ahorra 9 lecturas de todo el archivo y, lo que es más importante, 10 escrituras de todo el archivo de entrada. Dicho de otro modo, el método de rejilla realiza menos de un tercio menos de E/S que el método binario. Esto se corresponde perfectamente con los resultados de la figura 8(b). La diferencia entre el aumento de velocidad del tiempo de ejecución (aproximadamente 2) y el aumento de velocidad de E/S (aproximadamente 3) refleja el hecho de que el método de rejilla es más intensivo en CPU.

Los experimentos con los datos de TIGER (Figura 9) muestran un patrón similar. Obsérvese que el rendimiento de la carga masiva del árbol kd es independiente de la distribución de los datos.

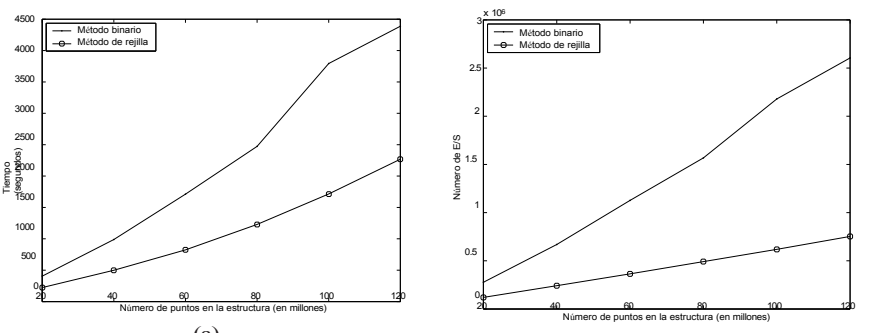
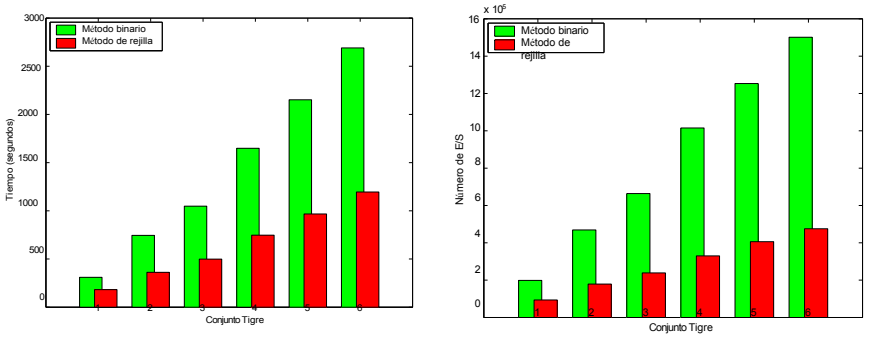


Fig. 8. Rendimiento de la carga masiva con datos uniformes: (a) Tiempo (en segundos), (b) Número de E/S



(a) (b)

Fig. 9. Rendimiento de la carga masiva en datos TIGER: (a) Tiempo (en segundos), (b) Número de E/S

Esto significa que el rendimiento de la carga masiva puede predecirse con gran exactitud sólo a partir del número de puntos que deben indexarse. Para ilustrarlo, consideremos el conjunto distribuido uniformemente, que contiene 40 millones de puntos, y el conjunto TIGER 3, que contiene 39,5 millones de puntos. Comparando los tiempos de carga masiva de los dos conjuntos , encontramos valores prácticamente idénticos.

4.3 Rendimiento de inserción

Para comparar el rendimiento medio de inserción del árbol Bkd con el del árbol K-D-B, insertamos todos los puntos de cada conjunto TIGER en una estructura inicialmente vacía y dividimos los resultados globales por el número de puntos insertados. La figura 10 muestra el tiempo medio y el número medio de E/S para una in- serción. En términos de transcurrido, la inserción de un árbol Bkd es sólo dos veces más rápida que la de un árbol K-D-B. Sin embargo, si se tienen en cuenta las E/S, la inserción de un árbol K-D-B es mucho más rápida que la de un árbol Bkd. Sin embargo, cuando se cuentan las E/S, los valores del árbol Bkd ni siquiera son visibles en el gráfico, ya que están muy por debajo de 1. Esta diferencia en dos métricas de rendimiento puede explicarse fácilmente por la disposición de los datos de TIGER y los efectos del almacenamiento en caché. Dado que los puntos se insertan en el árbol K-D-B en el orden en el que aparecen en los conjuntos de datos originales (los puntos del mismo condado se almacenan juntos), el árbol K-D-B aprovecha la localidad existente en este orden concreto y el hecho de que almacenamos en caché las rutas raíz-hoja durante las inserciones. Si el siguiente punto que se inserta está junto al , podría utilizarse la misma ruta, y la inserción podría no realizar ninguna E/S.

También comparamos el rendimiento medio de inserción del árbol Bkd y el árbol K-D-B utilizando los datos generados artificialmente. Las inserciones en estos experimentos muestran menos (o ninguna) localidad, ya que los puntos se insertaron en orden aleatorio. La Figura 11 muestra el tiempo medio y el número de E/S para una inserción, utilizando los conjuntos de datos uniformes. Para el árbol Bkd, los valores se obtuvieron insertando todos los puntos uno a uno en una estructura inicialmente vacía y calculando la media. Para el árbol K-D-B, sin embargo, no hemos podido realizar el mismo experimento.

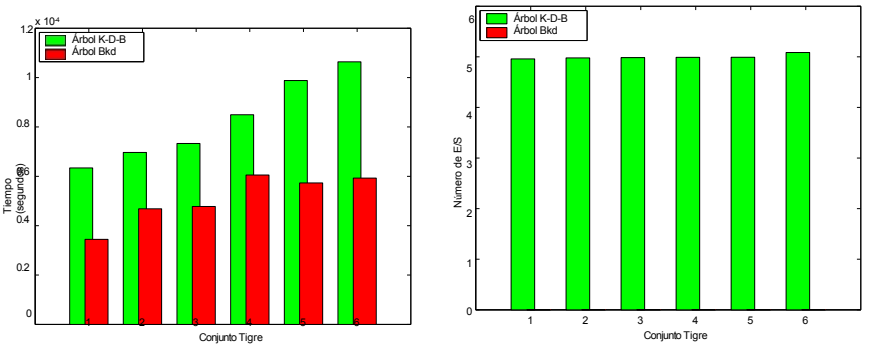


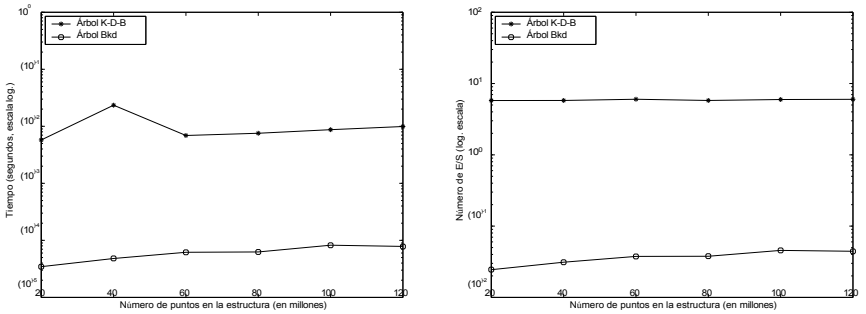
Fig. 10. Rendimiento de inserción en árboles K-D-B y Bkd (datos TIGER): (a) Tiempo (en segundos), (b) Número de E/S

Incluso para el conjunto más pequeño, que contiene 20 millones de puntos, ¡insertarlos uno a uno lleva más de 2 días! Esto se debe a la falta de localidad en el patrón de inserción; incluso si todos los nodos internos se almacenan en caché, cada inserción sigue realizando al menos dos E/S (para leer y escribir la hoja correspondiente) porque lo más probable es que la hoja relevante no esté en la caché. El resultado son 40 millones de E/S aleatorias para el conjunto de 20 millones de puntos.

Como no pudimos construir el árbol K-D-B mediante inserciones repetidas, diseñamos un experimento diferente para medir el rendimiento de inserción del árbol K-D-B. Cargamos en bloque un árbol K-D-B utilizando los puntos de entrada (llenando cada hoja y nodo hasta el 70% de su capacidad) y, a continuación, insertamos 1.000 puntos aleatorios en esa estructura. Como predecía el análisis teórico, la inserción de un árbol Bkd es varios órdenes de magnitud más rápida que la de un árbol K-D-B, tanto en términos de tiempo transcurrido como de número de E/S; en términos de tiempo transcurrido, la inserción de un árbol Bkd es más de 100 veces más rápida que la de un árbol K-D-B, para todos los tamaños de datos. En cuanto al número de E/S, el árbol Bkd es hasta 230 veces más rápido. La discrepancia entre ambas cifras se debe, una vez más, al hecho de que almacenamos en caché los nodos y las hojas. Dado que el árbol Bkd utiliza implícitamente toda la memoria principal como caché, permitimos que el árbol K-D-B hiciera lo mismo. Sin embargo, debido a la aleatoriedad de los datos, muy pocas hojas se encontraron en la caché.

4.4 Rendimiento de las consultas

Aunque los límites asintóticos en el peor de los casos para una consulta de ventana en un árbol Bkd y un árbol K-D-B son idénticos, esperamos que el árbol Bkd realice más E/S, debido a los múltiples árboles en los que hay que buscar. Para , consultamos un árbol Bkd y un árbol K-D-B con la misma ventana. La Figura 12 muestra los tiempos de ejecución y el número de E/S de una consulta con una ventana cuadrada que cubre el 1% de los puntos de cada uno de los conjuntos de datos uniformes. Estos valores se obtienen promediando 10 consultas del mismo tamaño, cuya posición se elige aleatoriamente en el área



(a) (b)

Fig. 11. Rendimiento de inserción en árboles K-D-B y Bkd (datos distribuidos uniformemente): (a) Tiempo (en segundos), (b) Número de E/S

cubierto por los puntos. Se puede observar que el árbol Bkd realiza aproximadamente el mismo número de E/S que el árbol K-D-B. Este resultado algo inesperado es consecuencia de varios factores. En primer lugar, el número medio de árboles kd que forman el árbol Bkd es inferior a $\log_2(N/M)$ (el máximo posible). La tabla 3 muestra el número de árboles kd no vacíos y el número de árboles kd máximos para cada uno de los 6 conjuntos de datos uniformes. Se puede demostrar fácilmente que en el transcurso de $2^p M$ inserciones en un árbol Bkd inicialmente vacío, el número medio de árboles kd no vacíos es $p/2$. Como resultado, el número de árboles kd que hay que buscar durante una consulta de ventana es menor que el máximo. En segundo lugar, los árboles kd individuales del árbol Bkd tienen alturas menores que el árbol K-D-B construido con el mismo conjunto de datos. Esto se debe al tamaño geométricamente decreciente de los árboles kd y al hecho de que, como se indica en la sección 3, el abanico del árbol Bkd es mayor que el abanico del árbol K-D-B. Como resultado, el número de árboles kd internos es menor que el del árbol K-D-B. Como resultado, el número de nodos internos leídos durante una consulta de ventana es pequeño. En tercer lugar, el rendimiento de la consulta del árbol kd es muy eficiente para estos conjuntos de datos. La tabla 4 muestra, para los conjuntos de datos uniformes, el número de puntos devueltos por la consulta como porcentaje del número total de puntos recuperados. Como resultado, ambas estructuras de datos leen aproximadamente el mismo número de bloques a nivel de hoja, lo que se aproxima al nivel óptimo.

En términos de tiempo de ejecución, el árbol K-D-B es más rápido que el árbol Bkd. Esto puede explicarse por el hecho de que las consultas se realizan en un árbol K-D-B cargado masivamente. Los árboles construidos mediante los algoritmos de carga masiva descritos en

Tabla 3. El número de árboles kd no vacíos y el número máximo de árboles kd, para cada árbol Bkd construido sobre los conjuntos de datos uniformes.

Número de puntos (en millones)	20	40	60	80	100	120
Árboles kd no vacíos	3	3	3	4	4	4
Max kd-trees $(\log_2(N/M))$	4	5	6	6	7	7

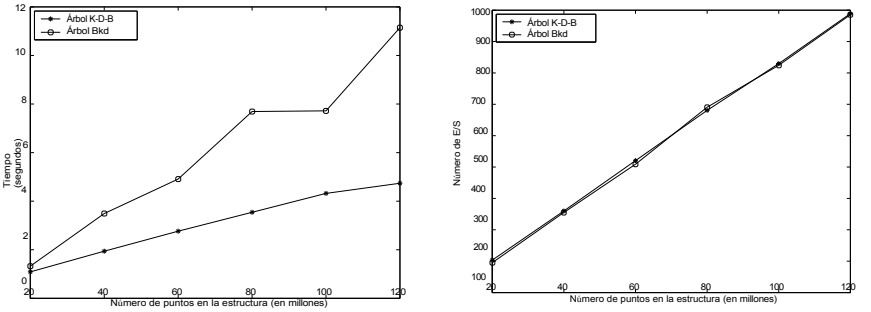


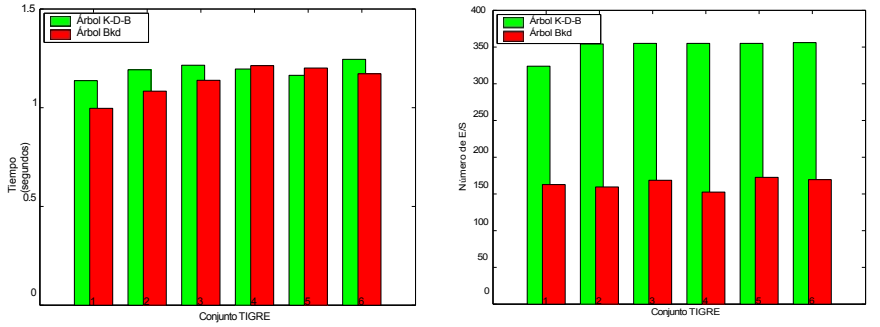
Fig. 12. Rendimiento de la consulta de rango en los datos uniformes (el área de rango es el 1% del área total): (a) Tiempo (en segundos), (b) Número de E/S

La sección 2.1 muestra un alto nivel de localidad, en el sentido de que es probable que los puntos cercanos en el disco estén espacialmente próximos. Las consultas realizadas en el árbol K-D-B pueden aprovechar esta localidad, lo que da lugar a un patrón de acceso más secuencial. Por otro lado, el árbol Bkd tiene menos localidad, ya que hay que consultar varios árboles para obtener el resultado final. En una base de datos espacial del mundo real , el árbol K-D-B suele obtenerse mediante inserciones repetidas. Esto suele lugar a una estructura con una baja utilización del espacio y una escasa localidad. Este comportamiento puede observarse en los experimentos realizados con los conjuntos TIGER. Como se explica en la sección 4.3, el árbol K-D-B de los conjuntos TIGER se obtuvo mediante inserciones repetidas. Como resultado, presenta mucha menos localidad. La Figura 13 muestra que las dos estructuras tienen un rendimiento similar en términos de tiempo, lo que demuestra que ambas tienen que realizar algunas E/S aleatorias (el árbol Bkd porque consulta varios árboles kd y el árbol K-D-B porque presenta menos localidad). En términos de E/S, el árbol Bkd realiza la mitad de E/S que el árbol K-D-B. Esto se debe a la escasa utilización del espacio. Esto se debe a la escasa utilización del espacio del árbol K-D-B, que en el caso de los conjuntos de datos TIGER (véase la sección 4.1) se situó en torno al 56%.

Para medir el efecto del tamaño de la ventana en el rendimiento de la consulta, realizamos una serie de experimentos con distintos tamaños de ventana. La figura 14 muestra los resultados de estos experimentos. Tanto el árbol K-D-B como el árbol Bkd se construyen con el conjunto de datos más grande, que contiene 120 millones de puntos distribuidos uniformemente.

Tabla 4. El número de puntos devueltos por una consulta de ventana como del número total de puntos recuperados. Para cada conjunto, la ventana cubre el 1% del número total de puntos.

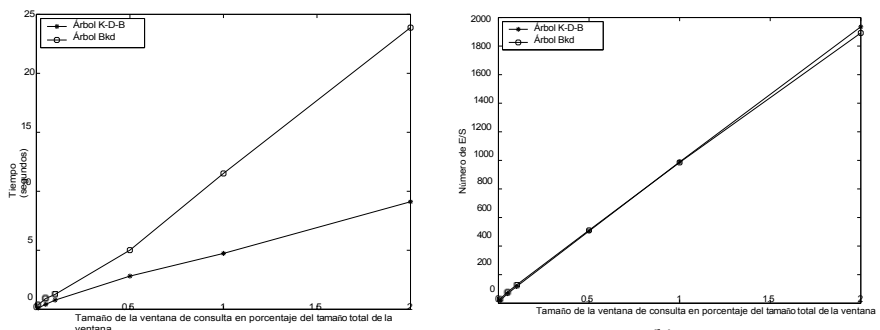
Número de puntos (en millones)	20	40	60	80	100	120
Árbol Bkd	78.4	84.7	88.1	86.5	90.4	90.6
Árbol K-D-B	74.8	83.6	86.2	87.9	90.2	90.6



(a) (b)

Fig. 13. Rendimiento de la consulta de rango en los datos de TIGER: (a) Tiempo (en segundos),
(b) Número de E/S

En el gráfico que muestra el tiempo transcurrido, volvemos a ver los efectos de un árbol K-D-B recién cargado en masa, que da lugar a un patrón de E/S más secuencial que el del árbol Bkd. Pero el rendimiento de E/S de las dos estructuras es prácticamente idéntico para toda la gama de tamaños de consulta, lo que confirma los resultados obtenidos en la consulta del 1%, es decir, que el rendimiento de consulta de ventana del árbol Bkd está a la par con el de las estructuras de datos existentes. Así pues, sin sacrificar el rendimiento de la consulta de ventana, el árbol Bkd mejora significativamente el rendimiento de inserción y la utilización del espacio : las inserciones son hasta 100 veces más rápidas que las inserciones del árbol K-D-B, y la utilización del espacio se acerca a un perfecto 100%, incluso con inserciones masivas.



(a) (b)

Fig. 14. Rendimiento de las consultas de rango de tamaño creciente (el conjunto de datos consta de 120 millones de puntos distribuidos uniformemente en un cuadrado): (a) Tiempo (en segundos),
(b) Número de E/S

Agradecimientos

Agradecemos a Georgios Evangelidis que nos haya proporcionado el código del árbol hB$^{\Pi}$.

Referencias

[1] P. K. Agarwal, L. Arge, O. Procopiuc y J. S. Vitter. A framework for index bulk loading and dynamization. En *Proc. Intl. Colloq. Automata, Languages and* Programming, páginas 115-127, 2001. 51

[2] A. Aggarwal y J. S. Vitter. The Input/Output complexity of sorting and related problems. *Commun. ACM*, 31:1116-1127, 1988. 51

[3] L. Arge. Estructuras de datos de memoria externa. En J. Abello, P. M. Pardalos, y M. G. C. Resende, editores, *Handbook of Massive Data Sets*, páginas 313-358. Kluwer, 2002. 47

[4] L. Arge, O. Procopiuc y J. S. Vitter. Implementing I/O-efficient data structures using TPIE. En *Proc. European Symp. on Algorithms*, páginas 88-100, 2002. 55

[5] L. Arge, V. Samoladas y J. S. Vitter. On two-dimensional indexability and op- timal range search indexing. En *Proc. ACM Symp. Principles of Database Systems*, páginas 346-357, 1999. 47

[6] N. Beckmann, H.-P. Kriegel, R. Schneider y B. Seeger. El árbol R*: Un método de acceso eficiente y robusto para puntos y rectángulos. En *Proc. SIGMOD Intl. Conf. on Management of Data*, páginas 322-331, 1990. 47

[7] J. L. Bentley. Árboles de búsqueda binarios multidimensionales utilizados para la búsqueda asociativa.
Commun. ACM, 18(9):509-517, sept. 1975. 47

[8] J. L. Bentley. Problemas de búsqueda descomponibles. *Inform. Process. Lett.*, 8:244- 251, 1979. 49, 53

[9] S. Berchtold, C. B¨ohm, y H.-P. Kriegel. Improving the query performance of high-dimensional index structures by bulk load operations. En *Proc. Intl. Conf. on Extending Database Technology*, volumen 1377 de *Lecture Notes Comput. Sci.*, páginas 216-230, 1998. 47

[10] G. Evangelidis, D. Lomet y B. Salzberg. El árbol hB$^{\Pi}$: Un índice multiatributo soportando concurrencia, recuperación y consolidación de nodos. *The VLDB Journal*, 6:1-25, 1997. 47, 49, 50, 53, 55, 57

[11] V. Gaede y O. Gu¨nther. Métodos de acceso multidimensional. *ACM Computing Encuestas*, 30(2):170-231, 1998. 47

[12] R. Grossi y G. F. Italiano. Efficient cross-tree for external . En J. Abello y J. S. Vitter, editores, *External Memory Algorithms and Visualization*, páginas 87-106. American Mathematical Society, 1999. Versión revisada disponible en ftp://ftp.di.unipi.it/pub/techreports/TR-00-16.ps.Z. 47

[13] A. Guttman. Árboles R: Una estructura de índice dinámica para la búsqueda espacial. En *Proc. SIGMOD Intl. Conf. on Management of Data*, páginas 47-57, 1984. 47

[14] H. V. Jagadish, P. P. S. Narayan, S. Seshadri, S. Sudarshan y R. Kanneganti. Incremental organization for data recording and warehousing. En *Proc. Intl. Conf. on Very Large Data Bases*, páginas 16-25, 1997. 49

[15] K. V. R. Kanth y A. K. Singh. Optimal dynamic range searching in non-replicating index structures. En *Proc. Intl. Conf. on Database Theory*, volumen 1540 de *Lecture Notes Comput. Sci.*, páginas 257-276, 1999. 47

[16] D. T. Lee y C. K. Wong. Worst-case analysis for region and partial region searches in multidimensional binary search trees and balanced quad trees. *Acta Informatica*, 9:23-29, 1977. 48

[17] D. Lomet. B-tree page size when caching is considered. *SIGMOD Record*, 27(3):28-32, 1998. 55

[18] D. B. Lomet y B. Salzberg. El árbol hB: Un método de indexación multiatributo con un buen rendimiento garantizado. *ACM Trans. on Database Systems*, 15(4):625- 658, dic. 1990. 47, 49, 55

[19] J. Nievergelt, H. Hinterberger y K. C. Sevcik. El archivo grid: An adaptable, symmetric multikey file structure. *ACM Trans. on Database Systems*, 9(1):38-71, Mar. 1984.

[20] P. E. O'Neil, E. Cheng, D. Gawlick y E. J. O'Neil. The log-structured merge- tree (LSM-tree). *Acta Informatica*, 33(4):351-385, 1996. 49

[21] M. H. Overmars. *The Design of Dynamic Data Structures*, volumen 156 de *Lecture Notes Comput.* Springer-Verlag, 1983. 49, 50, 53

[22] J. T. Robinson. El árbol K-D-B: Una estructura de búsqueda para grandes índices dinámicos multidimensionales. En *Proc. SIGMOD Intl. Conf. on Management of Data*, páginas 10-18, 1981. 47, 48, 55

[23] H. Samet. *Diseño y análisis de estructuras de datos espaciales*. Addison-Wesley, 1990. 47, 48

[24] Y. V. Silva Filho. Análisis del caso medio de búsqueda de regiones en árboles k-d equilibrados.
Informar. Lett. Lett., 8:219-223, 1979. 48, 54

[25] *TIGER/Line Files, Documentación técnica de 1997*. Oficina del Censo de EE.UU., 1998.
http://www.census.gov/geo/tiger/TIGER97D.pdf. 55