Índices

José Ramón Paramá Gabía

Capítulo 1

Índices

En este capítulo suponemos que existe ya un fichero con alguna organización básica de las expuestas en el capítulo anterior. Un índice sobre un fichero, es una estructura auxiliar diseñada para acelerar las operaciones que no son soportadas eficientemente por las organizaciones básicas de ficheros. Las estructuras de índices proporcionan caminos alternativos para acceder a los registros sin afectar a la posición física de los registros en el fichero.

Permite un acceso eficiente a registros basándose en *campos de indexación* que se utilizan para construir el índice. Básicamente, cualquier campo del fichero puede usarse para crear un índice; también se puede construir sobre el mismo fichero, múltiples índices sobre varios campos, algo que no era posible con las técnicas de hash. Otra de las ventajas con respecto a la técnicas de hash son las búsquedas de rango (por ejemplo, los empleados que ganen más de 20000 €). Estas consultas son ineficientes en el caso de los ficheros hash, ya que sólo pueden resolver eficazmente consultas por igualdad en la clave del fichero. Como veremos, los índices ayudan en todo tipo de consultas, aunque en las consultas por igualdad en la clave del fichero, los ficheros hash son más eficientes en general. Pero dado que los índices son estructuras auxiliares, podemos construir índices sobre ficheros hash, con lo que podemos obtener los beneficios de ambas técnicas.

Hay gran variedad de índices, cada uno de ellos usa una estructura de datos particular para agilizar la búsqueda. Para encontrar un registro o registros del fichero basándonos en cierto criterio de selección de un campo índice, primero hay que acceder al índice, que apunta a uno o más bloques del fichero donde se encuentra el registro buscado. Los tipos de índice más utilizados se basan en ficheros ordenados (índices de un sólo nivel) y estructuras de datos en árbol (árboles B y árboles B^+).

1.1. Índices ordenados de un solo nivel

La idea en la que se basa una estructura de acceso de índice ordenada es similar a la que subyace en la usada en un libro de texto, que enumera los términos importantes al final del libro en orden alfabético junto con una lista de los números de página en los que aparecen los términos en el libro. Si utilizamos este índice, podemos localizar directamente la palabra buscada por su página. La alternativa es leer todo el libro.

El índice suele definirse sobre un solo campo del fichero, lo que denominaremos el *campo de indexación*. Por lo general, el índice almacena todos los valores del campo de indexación

junto con una lista de punteros a todos los bloques del disco que contienen registros con cada valor del campo de indexación. Los valores en el índice están ordenados para que podamos efectuar búsquedas binarias en el índice. Como el fichero del índice es mucho más pequeño que el de datos, una búsqueda binaria en un índice es bastante eficiente, sobre todo si cabe en memoria principal.

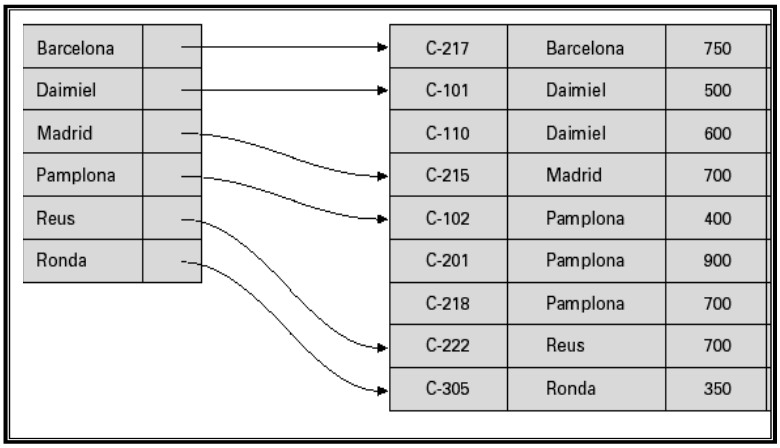

Figura 1.1: Índice de un solo nivel.

1.2. Índices basados en árboles

Los índices basados en árboles son jerarquías de índices. La raíz o primer nivel del índice apunta al segundo nivel del árbol. Cada nivel del índice apunta a niveles más bajos hasta llegar al nivel más bajo o nodos hoja del árbol. Los nodos hoja pueden ser los propios registros del fichero de datos, o bien, pueden contener punteros únicamente a los registros del fichero de datos.

Los índices basados en árboles proporcionan un mejor rendimiento que los índices ordenados, sobre todo a la hora de insertar y borrar registros.

Los ficheros ordenados proporcionan la mejora de la búsqueda binaria sobre el índice, que debido a que el índice es más pequeño que el fichero de datos, es normalmente una búsqueda más corta que si realizáramos la búsqueda binaria directamente sobre el fichero de datos. Aún así, la búsqueda binaria sobre el índice puede seguir siendo bastante cara.

La búsqueda binaria requiere del orden de $log_2(N)$ accesos a disco (siendo N el número de registros). El factor de bloqueo puede favorecer las lecturas de registros próximos, así que podemos simplificar N al número de bloques físicos del fichero de datos. Mediante un índice ordenado podemos rebajar los accesos a disco a $log_2(N/F)$, siendo F el número de entradas de índice en cada bloque físico del índice.

La búsqueda binaria en un caso particular de búsqueda en un árbol que sólo tiene dos

nodos hijo por nodo, generalizando a árboles con más hijos por nodo (*fan-out*) se puede acortar los tiempos de búsqueda a $log_b(N)$, siendo b el fan-out. Así, por ejemplo, en un fichero con 1,000,000 registros con un factor de bloqueo 10, una búsqueda binaria sobre el propio fichero requeriría 17 accesos. Si utilizásemos un índice ordenado con 100 entradas por bloque físico, el número de accesos sería 10. Finalmente, utilizando un árbol con fan-out de 50, el número de accesos sería alrededor de 3.

1.2.1. Árboles heterogéneos y homogéneos

Los índices basados en árboles se pueden clasificar en heterogéneos y homogéneos. Los *árboles heterogéneos* son aquellos donde cada nodo del árbol contiene sólo un tipo de punteros, pero los punteros de los nodos hoja son de distinto tipo que los de los nodos no hoja. Los punteros de los nodos hoja apuntan a los registros del fichero de datos, mientras que los punteros de los nodos no hoja apuntan a otros nodos (en niveles inferiores) del árbol (ver figura 1.2).

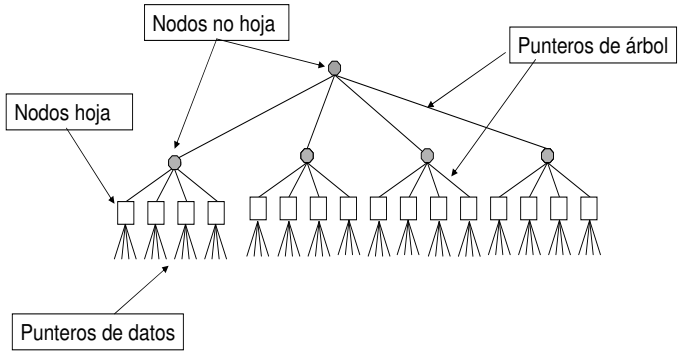

Figura 1.2: Árbol heterogéneo.

Los *árboles homogéneos* son aquellos en los que cada nodo contiene dos tipos de punteros, punteros a registros (punteros de datos) y punteros a otros nodos (punteros de árbol). Todos los nodos son idénticos respecto a su estructura, los nodos hoja tienen punteros de árbol vacíos y punteros de datos activos. Los nodos no hoja, tienen ambos tipos de punteros activos.

Todo nodo debe tener una serie de entradas, cada una con un valor del campo de indexación, más un puntero de datos. Además, cada nodo, debe tener un puntero de árbol más que punteros de datos (ver Figura 1.3).

Cada puntero de árbol apunta a un nodo del árbol cuyos valores del campo de indexación están acotados por por los valores del campo de indexación a los dos lados del puntero. El primer puntero de árbol en un nodo apunta a otro nodo cuyos valores del campo de indexación son menores o iguales al primer valor de campo de indexación de dicho nodo.

Para localizar un registro, la búsqueda comienza en la raíz del árbol. Se compara la el valor buscado con los valores de campo de indexación en el nodo raíz. Puede ocurrir que el valor buscado esté en ese mismo nodo, que entre dos de los valores del nodo, que esté antes del primero o después del último. En el primer caso, se utiliza el puntero de datos para acceder al registro y en los últimos tres casos, se sigue el puntero apropiado para acceder a otro nodo del árbol y continuar así el proceso.

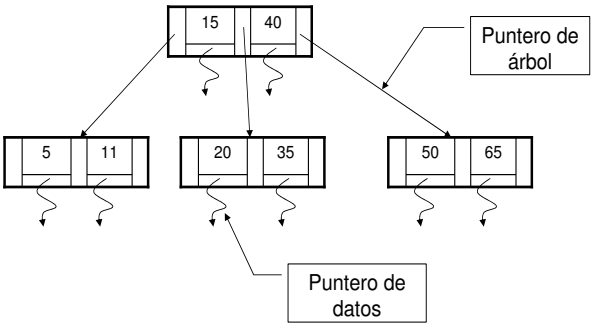

Figura 1.3: Árbol homogéneo.

Así, en el ejemplo de la Figura 1.3 si buscamos el registro con valor 11 en el campo de indexación, primero se comienza la búsqueda por el nodo raíz. Al no estar en la raíz, vemos como el valor 11 es anterior a primer valor de campo de indexación en la raíz (15). Se sigue por lo tanto el puntero a la izquierda del 15, y llegamos ya a un nodo que contiene el puntero a registro con valor 11.

Hay dos grandes diferencias entre los árboles homogéneos y heterogéneos, la altura del árbol y la longitud de la búsqueda media. La longitud media de la búsqueda será mayor en los árboles heterogéneos que en los homogéneos. Esto es porque en los heterogéneos la búsqueda siempre tiene que llegar a los nodos hoja, mientras que en los homogéneos puede acabar en cualquier nivel. El precio que se debe pagar, es el espacio necesario para el doble juego de punteros en cada nodo y unos algoritmos un poco más complejos.

Altura de árboles y fan-out

Como ya hemos apuntado anteriormente, el fan-out (b) define el número de punteros de árbol que tiene como máximo los nodos de un árbol. En la figura 1.3 se observa un árbol con dos niveles y un fan-out de 3.

Comparemos ahora los árboles homogéneos y heterogéneos con el mismo número de niveles y fan-out. Supongamos un árbol heterogéneo de 3 niveles y fan-out 4, este árbol puede indexar 4^3 o 64 registros de datos. Sin embargo, un árbol homogéneo de los mismos niveles y fan-out puede indexar como máximo 63 registros de datos.

En un árbol heterogéneo de fan-out b y n niveles, el número máximo de registros de datos indexados es b^n, dado que en cada nivel intermedio hay 0 punteros de datos y b punteros de árbol, mientras que en los nodos hoja hay b punteros de datos. En el caso de los árboles homogéneos (de fan-out b y n niveles) el máximo número de registros indexados es $b^0(b-1) + b^1(b-1) + \ldots + b^{n-1}(b-1)$, dado que en cada nivel hay $(b-1)$ punteros a datos y b punteros de de árbol.

1.2.2. Árboles B

El origen del término árboles B no está claro, algunos opinan que B proviene de *balanceado*, otros de Boeing, dado que buena parte del trabajo en las etapas tempranas del su desarrollo se realizó en esa empresa, finalmente también se le atribuye a Bayer que también realizó gran

parte del trabajo inicial y las primeras publicaciones sobre árboles B. En cualquier caso no viene de árbol binario como veremos.

Un *árbol B de orden d* se define como un árbol homogéneo con las siguientes características:

1. Cada nodo puede alojar como mucho $2d$ valores del campo de indexación con sus punteros a datos y $2d + 1$ punteros de árbol.
2. Ningún nodo, excepto el nodo raíz, puede tener menos de d valores del campo de indexación.
3. Todos los nodos hoja están en el mismo nivel.

Por estas restricciones, el fan-out de un árbol B siempre estarán entre $d + 1$ y $2d + 1$. El árbol de la Figura 1.3 es un árbol B de orden d=1. d se escoge de tal modo que cada nodo quepa en un bloque físico de disco.

Añadir

A la hora de añadir o borrar registros del fichero de datos hay dos acciones:

1. Añadir o borrar el registro del fichero de datos. Esto se hará según la organización utilizada (secuencial ordenado, acceso directo, etc.).
2. Añadir o borrar la entrada del índice.

La primera acción se comentó en su momento para cada una de las posibles organizaciones de ficheros, ahora nos vamos a ocupar de añadir y borrar entradas del índice.

Para añadir o borrar entradas del árbol es necesario disponer de un buffer de nodos que sea un poco más grande que el tamaño normal de un nodo. Concretamente, es necesario que contenga espacio para $2d + 1$ valores del campo de indexación (con sus punteros de datos) y para $2d + 2$ punteros de árbol. También para esta operación y otras que se verán más adelante es bueno disponer de una pila en memoria para almacenar nodos del árbol en memoria durante las distintas operaciones, y de este modo, ahorrar lecturas a disco. Normalmente la pila contendrá los nodos que van del nodo raíz, al nodo procesado en un momento dado.

Las nuevas entradas siempre se añaden en los nodos hoja. El proceso consiste en localizar el nodo hoja apropiado e insertar la entrada (valor del campo más el puntero de datos) dentro de él. Siempre existe la posibilidad de que el nodo que le corresponde a la entrada que queremos añadir ya esté lleno, en tal caso se produce un desbordamiento, en tal caso, el primer remedio es *redistribuir* las entradas entre el nodo objeto de la inserción, su padre y un nodo hermano adyacente. Si esto no es posible, el nodo se divide en dos nodos, con la entrada que sería el valor medio del nodo promocionando al nodo padre.

Supongamos que en el árbol B de la Figura 1.4 (de orden 1) se desea insertar la entrada con el valor 4. En primer lugar se localiza el nodo hoja que le corresponde. Partiendo de la raíz, es un valor menor que 8, por lo que se baja por la rama izquierda del nodo raíz, a continuación, en el segundo nivel, se compara con el 2, como 4 es mayor se baja por la rama derecha, de modo que se ha localizado un nodo hoja (el ocupado ahora únicamente por el valor 5). En este caso, hay espacio en el nodo para la nueva entrada, si inserta por lo tanto en dicho nodo. Esto es el caso normal, más que la excepción, a la hora de realizar inserciones. El resultado se puede observar en la Figura 1.5.

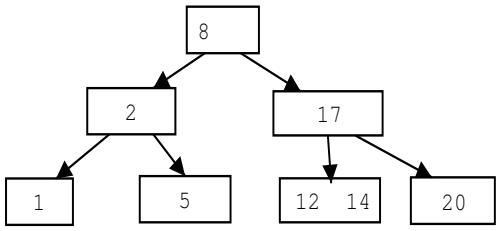

Figura 1.4: Árbol B de orden 1.

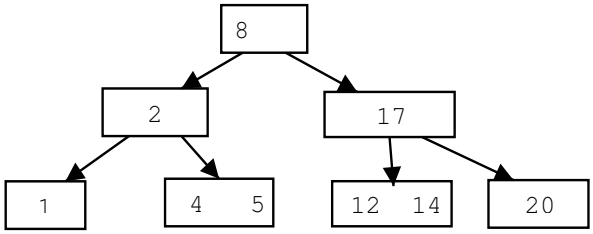

Figura 1.5: Después de la inserción de la entrada 4.

Ahora se desea insertar el registro con clave 6, el nodo que le corresponde es el ahora ocupado por las entradas 4 y 5, que (al ser el orden 1) ya está lleno. La solución en este caso es la *redistribución*. Existe un nodo hermano (y solo uno en este caso) adyacente (está justo a su izquierda) y que no está lleno, las tres condiciones para poder realizar la redistribución.

Si hubiera dos nodos hermanos adyacentes con espacio, habría que escoger una política *primero izquierda* o bien, *primero derecha*. Aunque como hemos indicado, en este ejemplo, no hay duda.

En este caso se distribuyen las entradas de los dos nodos implicados igualmente ($(1+3)/2 = 2$)), es decir, dos entradas en cada uno de los nodos. La entrada que está en el nodo padre separando los dos nodos (y que guía por tanto la búsqueda) se debe ajustar para que dicha búsqueda funcione correctamente. La redistribución se realiza tomando las entradas del nodo izquierdo, más las del derecho (que incluye en la sección de desbordamiento la nueva entrada), más la entrada del nodo padre que separa a los dos nodos. Se ordena ese conjunto, la entrada que se encuentra en la mitad es la que promociona al nodo padre, las que son menores van al nodo izquierdo, y las que son mayores, al nodo derecho. Así, en este caso, las entradas 1 y 2 quedarían en el nodo izquierdo, 5 y 6 en el nodo derecho y la entrada 4 pasaría al nodo padre a "separar"los dos nodos. El proceso se esboza en la Figura 1.6, y el resultado se puede observar en la Figura 1.7.

A continuación, en nuestro ejemplo se debe insertar la entrada de índice con valor 7. El nodo que le corresponde ya está lleno, y su único nodo hermano adyacente (el ocupado por el 1 y 2) está completo también, por lo que no es posible la redistribución.

Se debe crear por lo tanto un nuevo nodo del árbol. Las claves se deben distribuir entre el nodo desbordado y el nuevo, de un modo similar al caso de la redistribución. Se toman las entradas del nodo antiguo más la nueva entrada. Se ordena ese conjunto, la entrada que se encuentra en la mitad es la que promociona al nodo padre, las que son menores van al nodo izquierdo, y las que son mayores, al nodo derecho. Así en este caso, la entrada con el 5, se

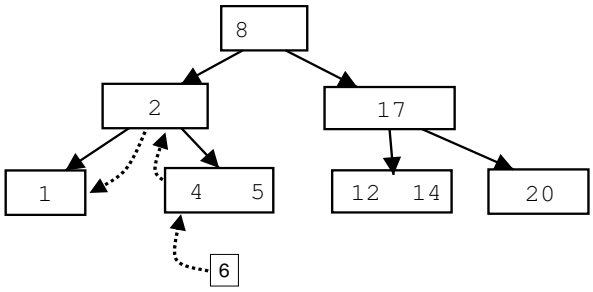

Figura 1.6: Redistribución.

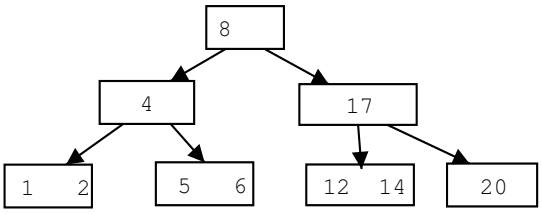

Figura 1.7: Resultado después de la inserción de la entrada 6.

queda en el nodo antiguo, la entrada con el valor 7 se va al nodo nuevo, y la entrada con valor 6 promociona al nodo padre. El resultado se puede observar en la Figura 1.8.

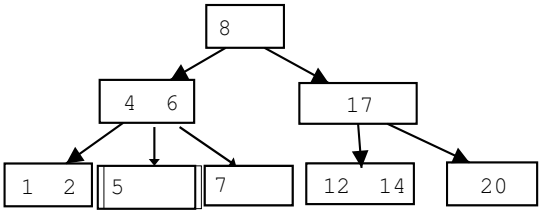

Figura 1.8: Resultado después de la inserción de la entrada 7.

La promoción de una entrada al nodo padre, cabe en dicho nodo en la mayoría de los casos, pero puede ocurrir que no quepa en dicho nodo. Entonces habría que recurrir una vez más, o bien a la redistribución con un nodo hermano adyacente, o bien a la creación de un nuevo nodo como el caso que estamos considerando.

Borrar

El proceso de borrado de entradas es casi el inverso del de añadir. La única diferencia está en el borrado de entradas que no están en nodos hoja.

Para borrados en nodos hoja, una vez se elimina la entrada del nodo correspondiente, se debe comprobar que el número de entradas en el nodo no baje de d. En tal caso, el primer recurso es intentar la redistribución de claves con un nodo hermano adyacente con más de d

claves. En tal caso, las entradas son redistribuidas entre los dos nodos del mismo modo que se hizo cuando se estaba añadiendo. Supongamos que en el árbol de la Figura 1.8 se borra la entrada con valor 20. El nodo que ocupaba esa estrada queda ahora vacío, por lo tanto contiene menos de d claves. Pero hay un nodo hermano adyacente con más de d claves (el ocupado por las entradas 12 y 14). Se realiza la redistribución, se toman las entradas 12, 14 y 17 (la del nodo padre que separa a los 2 nodos), y después de ordenarlas, la entrada del medio (14) promociona al nodo padre, las menores de 14 (12) se van al nodo de la izquierda y las mayores que 14 (17) se insertan en el nodo de la derecha. El resultado se muestra en la Figura 1.9.

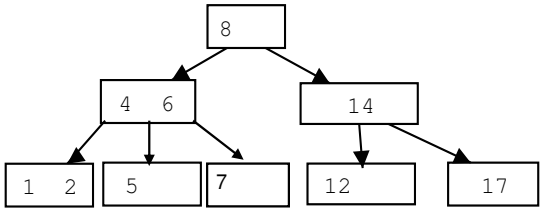

Figura 1.9: Resultado después del borrado de 20.

A continuación se borra la clave 12, una vez más el nodo se queda vacío y hay que buscar alguna alternativa para que albergue d claves. La redistribución no es posible, el único nodo hermano adyacente (el ocupado por la entrada 17) tiene ya tan sólo d claves.

Hay que refundir por lo tanto dos nodos, el que tiene ahora menos de d claves, más un hermano adyacente (en este caso sólo hay un hermano adyacente, pero si hubiera 2, se utilizaría una política de primero izquierdo o primero derecha). En este caso, como hemos dicho, se funden el nodo con menos de d claves (ahora vacío completamente) y su hermano adyacente (el ocupado ahora por la entrada 17). Se toman las claves de los dos nodos más la entrada del nodo padre que separa a los dos nodos (en este caso el 14) y se colocan en el nuevo nodo. Es decir, el nuevo nodo fruto de la fusión tendrá, en este caso, las entradas 14 y 17. El problema es que al tomar la entrada 14 del nodo padre, resulta que el nodo padre pasa a tener menos de d claves también. Por lo tanto se debe seguir con el proceso, tal y como se realizó para los nodos hoja.

El nodo padre, tiene un hermano adyacente con más de d claves (el nodo con entradas 4 y 6), podemos redistribuir las entradas entre los dos nodos. Se toman las entradas 4, 6 y 8 (la entrada del nodo raíz que separa a los dos nodos que se están redistribuyendo), la del medio promociona, la entrada 4 queda a la izquierda y la entrada 8 a la derecha. Ahora hay que tener cuidado y reajustar los punteros de modo que las búsquedas sigan funcionando correctamente. Así, observe en la Figura 1.10, el puntero del nodo con la entrada 7 se recoloca apropiadamente.

En el caso de borrar una entrada que no está en un nodo hoja, como comentamos, hay alguna diferencia. El problema se soluciona, reemplazando la entrada en el nodo no hoja, con una entrada de un nodo hoja. La entrada seleccionada, puede ser la entrada más a la derecha del subárbol izquierdo, o la entrada más a la izquierda del subárbol derecho, se utilizará una política (primero izquierda o primero derecha) para realizar la elección.

Supongamos ahora que en nuestro ejemplo se borra la entrada 6. En principio, la entrada borrada puede ser sustituida o bien por la entrada 5 (entrada más a la derecha del subárbol

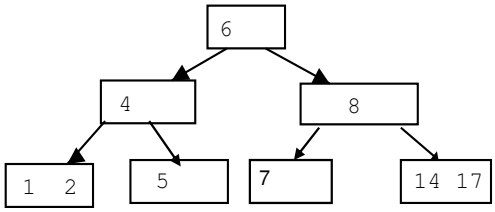

Figura 1.10: Resultado después del borrado de 12.

izquierdo), o por la entrada 7 (entrada más a la izquierda del subárbol derecho). Supongamos que la política es, primero derecha. Al eliminar la entrada 7 del nodo hoja, dicho nodo pasa a tener menos de d claves, este problema se soluciona como cualquier borrado en un nodo hoja. Así en este caso, se realiza una redistribución con el nodo hermano adyacente, quedando el resultado mostrado en la Figura 1.11.

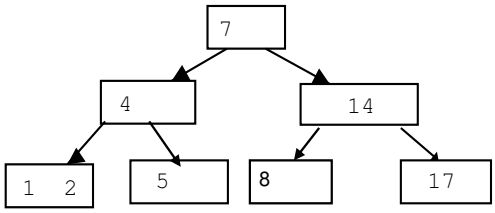

Figura 1.11: Resultado después del borrado de 6.

Leer un registro con un valor específico

Este algoritmo es muy sencillo y mucho menos complejo que añadir o borrar entradas. El nodo raíz es el primero en ser examinado para encontrar:

- Una entrada que contiene el valor buscado, y por lo tanto obtenemos el puntero al registro deseado en el fichero de datos.
- El lugar en la secuencia de valores del campo de indexación donde el valor buscado debería estar.

Si se encuentra la entrada el proceso termina. Si no es este el caso se sigue el puntero de árbol correspondiente para acceder al siguiente nodo, en el siguiente nivel del árbol. Si no hay siguiente nivel, la búsqueda termina concluyendo que el registro no está en el fichero de datos.

Leer todos los registros en el orden del campo de indexación

Para leer todos los registros en orden del campo de indexación, se debe realizar un recorrido *en orden* del árbol, cada vez que se lee una entrada, se accede al puntero a datos y se pasa a la siguiente entrada del índice.

El recorrido comienza con la entrada más a la izquierda del nodo hoja más a la izquierda del árbol, y continúa hasta la entrada más a la derecha del nodo hoja más a la derecha del árbol. El recorrido va arriba o abajo según es necesario para visitar todas las entradas *en orden*.

Eficiencia de los árboles B

Los árboles B permiten búsquedas, borrados e inserciones de registros utilizando muy pocas lecturas/escrituras físicas. Supongamos que el tamaño de cada bloque físico sea suficiente para albergar en cada bloque un número razonable de entradas, digamos 10 o más (es decir árboles B de orden 5 o más), en tal caso será realmente raro (como comentamos anteriormente) la fusiones o divisiones de nodos por inserciones o borrados. Además, cuando se producen esas operaciones, seguramente se limitarán a los nodos hoja (y sus padres) y no afectarán a niveles superiores. Por lo tanto, podemos casi despreciar el costo de las lecturas/escrituras debidas a reorganizaciones de los árboles B.

A la hora de realizar búsquedas, el número medio de lecturas físicas será el número de niveles del árbol (n) dividido por dos. ¿Cuántos niveles tiene un árbol B? Para los tamaños normales de campos de indexación, punteros y bloques físicos, entre cuatro y tres niveles son suficientes para todos los ficheros, salvo los casos extremos.

Ejemplo 1.2.1 Supongamos que disponemos de bloques físicos de 4096 bytes, que el tamaño del campo de indexación sea 4 bytes y que el de los punteros 8 bytes. Entonces deseamos encontrar el mayor entero (k) tal que $4k + 8(2k + 1) \leq 4096$. Ese valor es $k = 204$, por lo tanto nuestro árbol será de orden 102.

Supongamos que el nodo medio tiene una ocupación media entre d y $2d$, es decir, un bloque típico tiene 153 punteros de datos. Con tres niveles tendríamos $154^0(153)+154^1(153)+154^2(153) = 3652263$, es decir, sobre 3.5 millones de punteros a datos. Con cuatro niveles, ya se podría indexar los fichero extremadamente grandes, $154^0(153) + 154^1(153) + 154^2(153) + 154^3(153) = 562448655$, es decir, 562.5 millones de registros.□

Para la mayoría de las aplicaciones, un árbol B de tres niveles sería suficiente. El número medio de lecturas físicas para alcanzar el puntero a datos será, de media, algo superior a 2 y menor claramente a 3. Sin embargo, el nodo raíz (por donde empiezan todas las búsquedas) se puede mantener en memoria principal, incluso bajo ciertas circunstancias puede tener sentido mantener también el segundo nivel en memoria principal, de modo que muchas búsquedas requerirán cero lecturas físicas y como mucho suponga una lectura física.

1.2.3. Árboles B^+

La mayoría de las implementaciones de índices basados en árboles emplean una variación del árbol B llamada *árbol* B^+. Los árboles B^+ se diferencian de los árboles B en tres aspectos:

1. Es un árbol heterogéneo, en lugar de homogéneo. Todos los valores indexados aparecen en los nodos hoja. Los punteros de datos sólo aparecen en los nodos hoja, y los punteros de árbol sólo en los nodos no hoja.

2. Los valores del campo de indexación están duplicados en el árbol. Dado que todos los valores aparecen en los nodos hoja, algunos de ellos deben aparecer duplicados en los nodos no hoja para guiar la búsqueda.

3. Además de los punteros a datos, cada nodo hoja tiene un puntero al nodo hermano siguiente en la secuencia de nodos hoja. Esto permite recorrer todos los nodos hoja (que contienen todos los punteros a datos) de izquierda a derecha. Con esto se puede recorrer todos los registros del fichero en orden del campo de indexación sin necesidad de recorrer el árbol *en orden*, como sí era necesario en el caso del árbol B.

Con estas excepciones, los árboles B^+ se gestionan de modo similar a los árboles B. Los algoritmos correspondientes a los árboles B son fácilmente modificables para adecuarse a los árboles B^+.

Para acceder a un puntero a datos son necesarias unas pocas más lecturas físicas, puesto que siempre hay que llegar a un nodo hoja para obtener un puntero a datos.

Ejemplo 1.2.2 En las mismas condiciones del Ejemplo 1.2.1, desearíamos entonces encontrar el mayor entero (k) tal que $4k + 8(k + 1) \leq 4096$. k en este caso es 340, suponiendo una vez más una ocupación media de cada bloque, un nodo tendrá una media de 255 punteros. Con un árbol de tres niveles, tendríamos 255^3 punteros a datos, unos 16.6 millones de registros indexados. Para la mayoría de las aplicaciones sería más que suficiente.

Si suponemos que los dos primeros niveles del árbol están en memoria, una búsqueda requeriría una lectura física, es decir, poco más o menos lo mismo que en el caso del árbol B, con la desventaja para el árbol B de que con 3 niveles indexa muchos menos registros (considerando el mismo tamaño de bloque físico) y que para leer todos los registros en el orden del campo de indexación en el árbol B hay que hacer una búsqueda *en orden* por el árbol, mientras que en el árbol B^+ disponemos de punteros que enlazan los nodos hoja.□

Las diferencias con los algoritmos de inserción y borrado de los árboles B es que tenemos que mantener los punteros a datos siempre en los nodos hoja, y en los nodos intermedios copias de los valores en los nodos hoja.

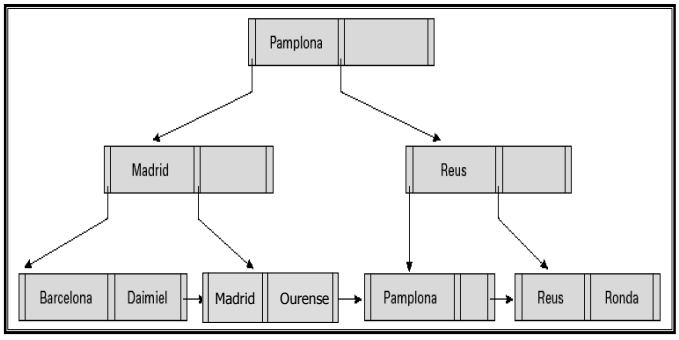

Figura 1.12: Ejemplo de árbol B^+.

Si en el árbol B^+ (de orden 1) de la Figura 1.12 se desea insertar la entrada con valor *Cádiz*, como en el caso del árbol B, se debe dividir el nodo. Se toman las tres entradas ordenadas (Barcelona, Damiel y Cádiz), la mitad inferior más uno se queda en el nodo antiguo (Barcelona y Cádiz) y el resto va al nodo nuevo. Aquí, hay la primera diferencia, todas las entradas deben quedar en nodos hoja. La primera entrada del nodo derecho (Damiel) promociona al nodo padre, pero a diferencia del árbol B, sólo como puntero de árbol (sin puntero de datos) para guiar la búsqueda. El resultado se muestra en la Figura 1.13.

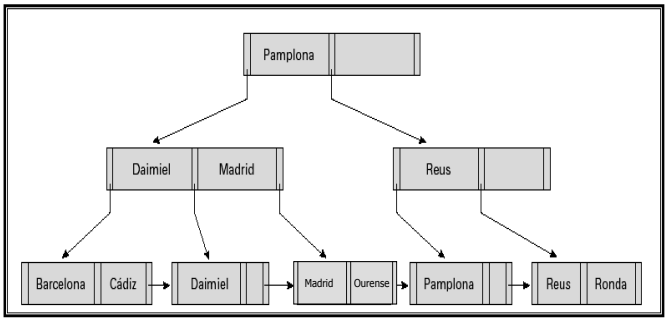

Figura 1.13: Resultado después de la inserción de Cádiz.

1.2.4. Variaciones de los árboles B y B^+

Para concluir esta sección, vamos a realizar una breve mención a algunas variaciones de los árboles B y B^+. En algunos casos la restricción de que en cada nodo debe haber al menos d claves se puede modificar de modo que exija que todos los nodos (salvo la raíz) estén ocupados por lo menos hasta las dos terceras partes de su capacidad. A este tipo de árboles se les ha llamado *árboles* B^*.

En general, algunos sistemas permiten que el usuario elija un *factor de llenado* de entre 0.5 y 1; este último valor indica que los nodos del árbol deben estar completamente llenos. También es posible especificar dos factores de llenado para los árboles B^+: uno para el nivel de hoja y otro para los nodos internos del árbol.

Al construirse inicialmente el índice, todos los nodos se ocupan hasta alcanzar aproximadamente los factores de llenado especificados. En fechas recientes algunos investigadores han sugerido que el requerimiento de que un nodo esté lleno hasta la mitad sea menos riguroso, y se permita que llegue a estar completamente vacío antes de efectuarse una fusión, a fin de simplificar el algoritmo de eliminación. Hay estudios de simulación que indican que esto no desperdicia demasiado espacio adicional si las inserciones y eliminaciones se distribuyen en forma aleatoria.

1.2.5. Duplicados

Los algoritmos vistos hasta el momento ignoran el problema de la presencia de duplicados, es decir, asumen que sólo hay una entrada en el índice por valor del campo de indexación. Una manera de satisfacer esta suposición en la presencia de duplicados es la utilización de bloques de overflow (cadena de bloques físicos enlazados con el bloque físico del fichero de datos apuntado por la entrada del índice para el valor considerado).

Sin embargo, generalmente se utiliza una alternativa distinta. Una posibilidad es considerar las entradas duplicadas igual que las entradas normales. Para recuperar todas las entradas de un valor dado, se accede a la entrada más a la izquierda en el árbol, en el caso de los árboles B^+ se sigue recorriendo los nodos hoja siguiendo los punteros que los enlazan hasta que se recuperen todas las entradas. En los árboles B, se debe realizar el recorrido *en orden*.

Otra alternativa es tener una entrada por valor, que apunta (en lugar de al registro) a una lista de punteros, que finalmente son los que apuntan al fichero de datos. Una variación

sería que la propia entrada contuviese la lista de punteros, pero esto conllevaría una serie de problemas al no ser todas las entradas del índice del mismo tamaño.

1.3. Índices multiclave

Hasta ahora se ha asumido implícitamente que se utiliza solamente un índice por un campo de indexación para procesar una consulta. Sin embargo, para cierto tipo de consultas es ventajoso el uso de múltiples índices si éstos existen.

Supongamos que disponemos un archivo con datos de la cuentas que tiene los campos *número de cuenta, nombre sucursal y saldo*, además de dos índices por los campos *nombre sucursal* y *saldo*. Consideremos la consulta: "encontrar todos los números de cuenta de la sucursal Pamplona con saldo igual a 1.000 €".

Hay tres estrategias para procesar esta consulta:

1. Usar el índice sobre *nombre de sucursal* para encontrar todos los registros pertenecientes a la sucursal de Pamplona. Luego se examinan estos registros para ver si $saldo = 1000$.

2. Usar el índice sobre *saldo* para encontrar todos los registros pertenecientes a cuentas con saldos de 1000 €. Luego se examinan estos registros para ver si *nombre sucursal=Pamplona*.

3. Usar el índice sobre *nombre sucursal* para encontrar punteros a registros pertenecientes a la sucursal Pamplona. Y también usar el índice sobre el campo *saldo* para encontrar los punteros a todos los registros pertenecientes a cuentas con un saldo de 1000 €. Se realiza la intersección de estos dos conjuntos de punteros. Aquellos punteros que están en la intersección apuntan a los registros pertenecientes a la vez a Pamplona y a las cuentas con un saldo de 1000 €.

La tercera estrategia es la única de las tres que aprovecha la ventaja de tener dos índices. Sin embargo, incluso esta estrategia podría ser una pobre elección si sucediera lo siguiente:

1. Hay muchos registros pertenecientes a la sucursal de Pamplona.

2. Hay muchos registros pertenecientes a cuentas con un saldo de 1000 €.

3. Hay solamente unos cuantos registros pertenecientes a *ambos*, a la sucursal de Pamplona y a las cuentas con un saldo de 1000 €.

Si estas condiciones ocurrieran, se tendrían que examinar un gran número de punteros para producir un resultado pequeño.

Una estrategia más eficiente para este caso es crear y utilizar un índice con un campo de indexación compuesto (*nombre sucursal, saldo*), esto es, el campo de indexación consiste en la concatenación de *nombre sucursal* y *saldo*. La estructura del índice es la misma que para cualquier otro índice, con la única diferencia de que el campo de indexación no es un simple atributo, sino una concatenación de atributos. El campo de indexación se puede representar como una tupla de valores, de la forma $(a_1, a_2, \ldots, a_n)$, donde los atributos indexados son $A_1, A_2, \ldots, A_N$. El orden de los valores de la clave de búsqueda es el orden *lexicográfico*. Por ejemplo, para el caso de dos atributos en el campo de indexación, $(a_1, b_1) < (a_2, b_2)$ si $a_1 < a_2$, o bien $a_1 = a_2$ y $b_1 < b_2$. El orden lexicográfico es básicamente el mismo que el alfabético.

El empleo de una estructura de índice con múltiples atributos concatenados tiene algunas deficiencias. Por ejemplo, consideremos la siguiente consulta: "obtener el número de cuenta de las cuentas cuyo nombre de sucursal sea menor que Pamplona y su saldo sea de 1000€". Se puede responder a esta consulta usando un índice con campo de indexación (*nombre sucursal, saldo*) de la manera siguiente: para cada valor de *nombre sucursal* que es menor que "Pamplona" alfabéticamente, localizar los registros con un saldo de 1000€. Sin embargo, debido a la ordenación de los registros en el archivo, es probable que cada registro esté en un bloque diferente de disco, causando muchas operaciones de E/S.

Existen diversas alternativas para solucionar esta problemática como veremos en las siguientes secciones.

1.3.1. Ficheros multilista

Un fichero multilista es una colección de listas enlazadas. Cada lista enlazada conecta registros que contienen el mismo valor para un atributo dado. Para permitir un acceso rápido a los registros, las listas están indexadas. El índice más común de este tipo es un índice de dos niveles. El nivel 1 indexa la lista de atributos y cada valor tiene un puntero a una partición del nivel 2. Este último indexa la lista de todos los valores del atributo en cuestión.

A	B	C	Nº Registro
30	a	x	1
30	a	y	2
30	b	x	3
30	a	y	4
30	c	x	5
40	a	x	6
40	b	x	7
40	c	z	8
40	d	y	9
50	b	y	10
50	c	y	11
50	d	z	12

Tabla 1.1: Un archivo con tres atributos

Utilizando el fichero de la Tabla 1.1, en la Figura 1.14 se muestra el índice multilista correspondiente.

Para una consulta sobre un único atributo, se utilizan los dos niveles del modo común (en el ejemplo se presentan índices ordenados, pero podrían ser índices basados en árboles). El puntero del nivel 2 apunta al registro cabeza de la lista, una vez en ese registro, se pueden obtener todos los registros con ese mismo valor siguiendo los punteros.

Para una búsqueda de rango la búsqueda es más lenta puesto que es necesario recorrer en el nivel 2 para obtener los punteros apropiados. Sin embargo, si las consultas de rango se conocen a priori, se pueden añadir entradas al índice de nivel 2 para estos casos.

Para una consulta que involucre varios campos el mecanismo funciona como sigue. Para cada valor en el índice de nivel 2 se mantiene un contador de los registros enlazados que tienen

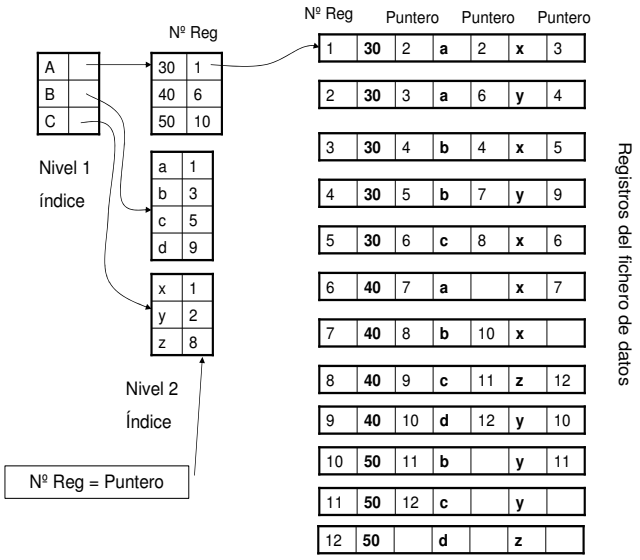

Figura 1.14: Ejemplo de un índice multilista.

ese valor. Se selecciona la lista más corta que cumple los criterios de búsqueda, y se recorre comprobando los valores de los otros atributos.

Los ficheros multilista requieren espacio de almacenamiento, primero para los índices y segundo, para los punteros y los registros de datos. Los punteros incrementan el tamaño del fichero de datos y esto puede afectar al rendimiento global, puesto que las ventajas del factor de bloqueo se ven mermadas.

1.3.2. Ficheros invertidos

Un fichero invertido es aquel que dispone una estructura de índice sobre uno o más atributos que no son la clave primaria del archivo, de modo que sirven de ayuda para realizar búsquedas que obtienen como resultado la clave primaria de los registros. Un fichero con un simple índice ordenado denso por un atributo que no sea parte de la clave primaria ya es un fichero invertido.

Supongamos que tenemos un fichero con información de los distintos tornillos disponibles en una ferretería. Los campos del fichero son *Referencia, longitud, diámetro cabeza, diámetro cuerpo, tipo*. La referencia es la clave primaria. La mayoría de las consultas serán a partir de los atributos que no son clave primaria, por ejemplo, "tornillos de más de 2cm de longitud y 4mm de diámetro", y lo que se desea es obtener la referencia para poder localizar el tornillo deseado en el almacén. Con un índice denso por cualquiera de los atributos que no es *referencia* tenemos una ayuda para responder a consultas de este tipo y conformaría un fichero invertido. Un fichero que tiene un índice por cada uno de los atributos que no forma la clave primaria es un *fichero totalmente invertido*, si tiene índices densos sobre atributos que no forman parte de la clave primaria pero no sobre todos, se dice que es *parcialmente invertido*.

Una posible mejora sobre lo apuntado anteriormente es disponer de un índice multinivel junto con una lista para acceder a los registros que tienen un valor determinado en el campo de indexación como se muestra en la Figura 1.15 para el fichero de la Tabla 1.1. Este es el

tipo de fichero invertido más común. Un primer nivel para indexar los atributos del fichero, y un segundo nivel para indexar los valores de los atributos.

Para búsquedas por rango, las listas de acceso correspondientes a los valores que están dentro del rango, se mezclan para obtener una lista de acceso a todos los registros que cumplen el criterio. Para realizar búsquedas multiatributo, las lista de acceso de cada uno de los atributos involucrados en la búsqueda se interseccionan para obtener la lista de acceso deseada. Esta intersección puede ser costosa, pero los ficheros invertidos se pueden combinar con la concatenación de los atributos, tal y como comentamos al comienzo de la Sección 1.3.

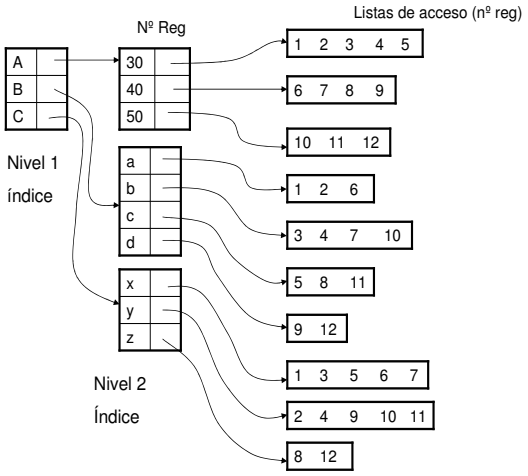

Figura 1.15: Ejemplo de un fichero invertido.

En ocasiones, si el fichero está totalmente invertido, no es necesario mantener el fichero de datos, el propio índice contiene todas la información que se desea almacenar.

1.3.3. Archivos rejilla o en retícula

En la Figura 1.16 se muestra una parte de un *archivo rejilla o en retícula* para los campos de indexación *nombre sucursal* y *saldo* en un archivo de cuentas. El array bidimensional de la figura se llama *array en retícula* y los arrays unidimensionales se llaman *escalas lineales*. El archivo en retícula tiene un único array en retícula y una escala lineal por cada atributo indexado.

Los campos de indexación se asignan a las celdas como se describe a continuación. Cada celda en el array en retícula contiene un puntero a un slot que contiene los registros. Sólo se muestran en la figura algunos de los slots y punteros desde las celdas para simplificar la figura. Para conservar espacio se permite que varios elementos del array puedan apuntar al mismo slot. Los recuadros punteados de la figura señalan las celdas que apuntan al mismo slot.

Supongamos que se quiere insertar en el índice de archivo en retícula un registro cuyo valor de los campos de indexación es ("Barcelona", 500.000). Para encontrar la celda asignada a este valor se localizan por separado la fila y la columna de la celda correspondiente.

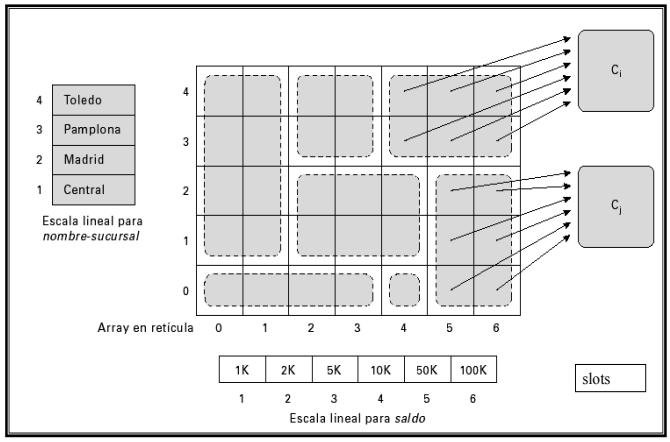

Figura 1.16: Ejemplo de un fichero rejilla o retícula.

Primero se utilizan la escala lineal en *nombre sucursal* para localizar la fila de la celda asignada al valor ("Barcelona", 500.000). Para ello se busca en el array el menor elemento que es mayor que "Barcelona". En este caso es el primer elemento, así que la fila asignada al valor buscado es la 0. Si fuera el $i-$ésimo elemento, se asignaría a la fila $i-1$. Si el valor del campo de indexación es mayor o igual que todos los elementos de la escala lineal, se le asignaría la última fila. A continuación se utiliza la escala lineal en *saldo* para encontrar de la misma manera qué columna le corresponde al valor 500.000, en este caso, la columna 6. Por lo tanto el valor ("Barcelona", 500.000) tiene asignado la celda de la fila 0 y la columna 6. Del mismo modo ("Damiel", 60.000) tendría asignado la celda de la fila 1 y la columna 5. Ambas celdas apuntan al mismo slot (como se indica en el recuadro punteado), así que en los dos casos, los registros de datos están almacenados en el slot C_j de la figura.

Para realizar una búsqueda que responda a la consulta:

$$\textit{nombre sucursal} < \text{"Pamplona"} \text{ and } \textit{saldo} = 1000$$

buscamos todas las filas con nombre de sucursal menores que "Pamplona", utilizando la escala lineal de *nombre sucursal*. En este caso, son las filas 0, 1 y 2. La fila 3 y posteriores contienen nombres de sucursal mayores o iguales que "Pamplona". De igual modo se obtiene que sólo la columna 1 puede tener un saldo de 1.000 €. Así, solamente las celdas en la columna 1, filas 0, 1 y 2 pueden contener entradas que satisfagan la condición de búsqueda.

A continuación, hay que examinar todos los registros en los slots apuntados por estas tres celdas. En este caso, sólo hay dos slots, ya que dos de las celdas apuntan al mismo slot, como se indica con los recuadros punteados de la figura. Los slots podrían contener algunos registros que no satisfacen la condición de búsqueda, de manera que se debe buscar dentro del slot de nuevo los registros que satisfacen la condición, aunque esta búsqueda, al realizarse en memoria, tiene un coste despreciable. De cualquier modo, sólo hay que examinar un pequeño número de slots para responder a la consulta.

Las escalas lineales se deben escoger de tal manera que los registros estén uniformemente distribuidos a través de las celdas. Si el slot –llamémosle A– queda lleno y se tiene que insertar una entrada en él, se crea un nuevo slot B. Si más de una celda apunta a A, se cambian los punteros a la celda de tal manera que algunos apunten a A y otros a B. Los registros en el slot A y el nuevo registro se redistribuyen entre A y B basándose en las celdas que tengan asignado. Si se diera el caso de que sólo una celda apuntase al slot A, se tendría que reorganizar el archivo en retícula extendiendo el array en retícula y escalas lineales de modo similar a la expansión del directorio del hash extensible. Del mismo modo cuando el factor de ocupación de los cubos, debido a borrados, cae por debajo de cierto valor, se pueden fundir cubos, una vez más, de modo similar al hash extensible.

Aunque aquí se ha mostrado el archivo en retícula para un archivo indexado por dos atributos, es sencillo expandir esta estructura a n atributos (n dimensiones).

Esta estructura no sólo es adecuada para consultas por varios atributos, también es adecuada para consultas por un único atributo. Supongamos la consulta: "números de cuenta de las cuentas de la sucursal de Pamplona". La escala lineal de *nombre sucursal* indica que las celdas de la fila 3 satisfacen esta condición. Como no hay condición sobre el saldo, se inspeccionan todos los slots apuntados por las celdas de la fila 3.

De este modo un índice en retícula puede hacer el papel de tres (considerando dos dimensiones, como en nuestro ejemplo) índices distintos. Si cada índice se mantuviera por separado, los tres juntos ocuparían más espacio y el coste se su actualización sería mayor.

Los archivos en retícula proporcionan un descenso en el tiempo de procesamiento de consultas multiatributo o multiclave. Sin embargo, implican un gasto adicional de espacio (el array en retícula puede llegar a ser grande), así como una degradación en el rendimiento a la hora de insertar o borrar registros. Además, es difícil elegir una división en los rangos de las claves para que la distribución de los registros sea uniforme. Si las inserciones en el archivo son frecuentes, la reorganización se tendrá que realizar periódicamente y eso puede tener un coste mayor.