CAPÍTULO

12 INDEXACIÓN Y ASOCIACIÓN

Muchas consultas hacen referencia sólo a una pequeña parte de los registros de un archivo. Por ejemplo, la pregunta «Buscar todas las cuentas de la sucursal Pamplona» o «Buscar el saldo del número de cuenta C-101» hace referencia solamente a una fracción de los registros de la relación cuenta. No es eficiente para el sistema tener que leer cada registro y comprobar que el campo *nombre-sucursal* contiene el nombre «Pamplona» o el valor C-101 del campo *número-cuenta*. Lo más adecuado sería que el sistema fuese capaz de localizar directamente estos registros. Para facilitar estas formas de acceso se diseñan estructuras adicionales que se asocian con archivos.

12.1. CONCEPTOS BÁSICOS

Un índice para un archivo del sistema funciona como el índice de este libro. Si se va a buscar un tema (especificado por una palabra o una frase) en este libro, se puede buscar en el índice al final del libro, encontrar las páginas en las que aparece y después leer esas páginas para encontrar la información que estamos buscando. Las palabras de índice están ordenadas, lo que hace fácil la búsqueda del término que se esté buscando. Además, el índice es mucho más pequeño que el libro, con lo que se reduce aún más el esfuerzo necesario para encontrar las palabras en cuestión.

Los catálogos de fichas en las bibliotecas funcionan de manera similar (aunque se usan poco). Para encontrar un libro de un autor en particular, se buscaría en el catálogo de autores y una ficha de este catálogo indicaría dónde encontrar el libro. Para ayudarnos en la búsqueda en el catálogo, la biblioteca guardaría en orden alfabético las fichas de los autores con una ficha por cada autor de cada libro.

Los índices de los sistemas de bases de datos juegan el mismo papel que los índices de los libros o los catálogos de fichas de las bibliotecas. Por ejemplo, para recuperar un registro *cuenta* dado su número de cuenta, el sistema de bases de datos buscaría en un índice para encontrar el bloque de disco en que se encuentra el registro correspondiente, y entonces extraería ese bloque de disco para obtener el registro *cuenta*.

Almacenar una lista ordenada de números de cuenta no funcionaría bien en bases de datos muy grandes con millones de cuentas, ya que el propio índice sería muy grande; más aún, incluso al mantener ordenado el índice se reduce el tiempo de búsqueda, encontrar una cuenta puede consumir mucho tiempo. En su lugar se usan técnicas más sofisticadas de indexación. Algunas de estas técnicas se discutirán más adelante.

Hay dos tipos básicos de índices:

- **Índices ordenados.** Estos índices están basados en una disposición ordenada de los valores.
- **Índices asociativos** (*hash indices*). Estos índices están basados en una distribución uniforme de los valores a través de una serie de cajones (*buckets*). El valor asignado a cada cajón está determinado por una función, llamada *función de asociación* (*hash function*).

Se considerarán varias técnicas de indexación y asociación. Ninguna de ellas es la mejor. Sin embargo, cada técnica es la más apropiada para una aplicación específica de bases de datos. Cada técnica debe ser valorada según los siguientes criterios:

- **Tipos de acceso.** Los tipos de acceso que se soportan eficazmente. Estos tipos podrían incluir la búsqueda de registros con un valor concreto en un atributo, o buscar los registros cuyos atributos contengan valores en un rango especificado.
- **Tiempo de acceso.** El tiempo que se tarda en buscar un determinado elemento de datos, o conjunto de elementos, usando la técnica en cuestión.
- **Tiempo de inserción.** El tiempo empleado en insertar un nuevo elemento de datos. Este valor incluye el tiempo utilizado en buscar el lugar apropiado donde insertar el nuevo elemento de datos, así como el tiempo empleado en actualizar la estructura del índice.
- **Tiempo de borrado.** El tiempo empleado en borrar un elemento de datos. Este valor incluye el tiempo utilizado en buscar el elemento a borrar, así como el tiempo empleado en actualizar la estructura del índice.

- **Espacio adicional requerido.** El espacio adicional ocupado por la estructura del índice. Como normalmente la cantidad necesaria de espacio adicional suele ser moderada, es razonable sacrificar el espacio para alcanzar un rendimiento mejor.

A menudo se desea tener más de un índice por archivo. Volviendo al ejemplo de la biblioteca, nos damos cuenta de que la mayoría de las bibliotecas mantienen varios catálogos de fichas: por autor, por materia y por título.

Los atributos o conjunto de atributos usados para buscar en un archivo se llaman **claves de búsqueda**. Hay que observar que esta definición de *clave* difiere de la usada en *clave primaria*, *clave candidata* y *superclave*. Este doble significado de *clave* está (por desgracia) muy extendido en la práctica. Usando nuestro concepto de clave de búsqueda vemos que, si hay varios índices en un archivo, existirán varias claves de búsqueda.

12.2. ÍNDICES ORDENADOS

Para permitir un acceso directo rápido a los registros de un archivo se puede usar una estructura de índice. Cada estructura de índice está asociada con una clave de búsqueda concreta. Al igual que en el catálogo de una biblioteca, un índice almacena de manera ordenada los valores de las claves de búsqueda, y asocia a cada clave los registros que contienen esa clave de búsqueda.

Los registros en el archivo indexado pueden estar a su vez almacenados siguiendo un orden, semejante a como los libros están ordenados en una biblioteca por algún atributo como el número decimal Dewey. Un archivo puede tener varios índices según diferentes claves de búsqueda. Si el archivo que contiene los registros está ordenado secuencialmente, el índice cuya clave de búsqueda especifica el orden secuencial del archivo es el **índice primario**. (El término *índice primario* se emplea algunas veces para hacer alusión a un índice según una clave primaria. Sin embargo, tal uso no es normal y debería evitarse.) Los índices primarios también se llaman **índices con agrupación** (*clustering indices*.) La clave de búsqueda de un índice primario es normalmente la clave primaria, aunque no es así necesariamente. Los índices cuyas claves de búsqueda especifican un orden diferente del orden secuencial del archivo se llaman **índices secundarios** o **índices sin agrupación** (*non clustering indices*).

12.2.1. Índice primario

En este apartado se asume que todos los archivos están ordenados secuencialmente según alguna clave de búsqueda. Estos archivos con índice primario según una clave de búsqueda se llaman **archivos secuenciales indexados**. Representan uno de los esquemas de índices más antiguos usados por los sistemas de bases de datos. Se emplean en aquellas aplicaciones que demandan un procesamiento secuencial del archivo completo así como un acceso directo a sus registros.

En la Figura 12.1 se muestra un archivo secuencial de los registros *cuenta* tomados del ejemplo bancario. En esta figura, los registros están almacenados según el orden de la clave de búsqueda, siendo esta clave *nombre-sucursal*.

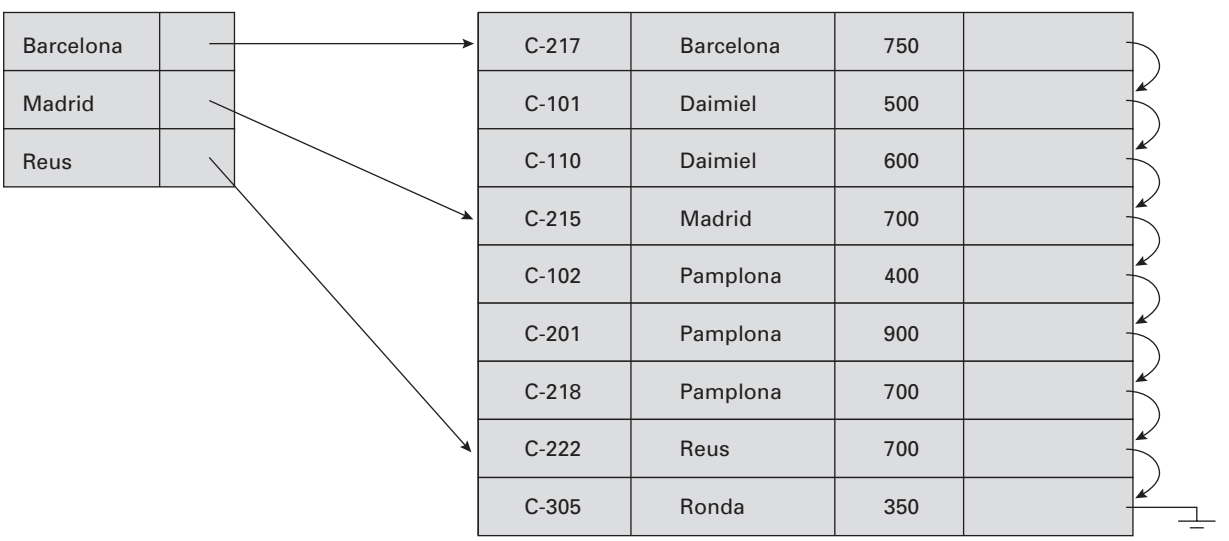

FIGURA 12.1. Archivo secuencial para los registros *cuenta*.

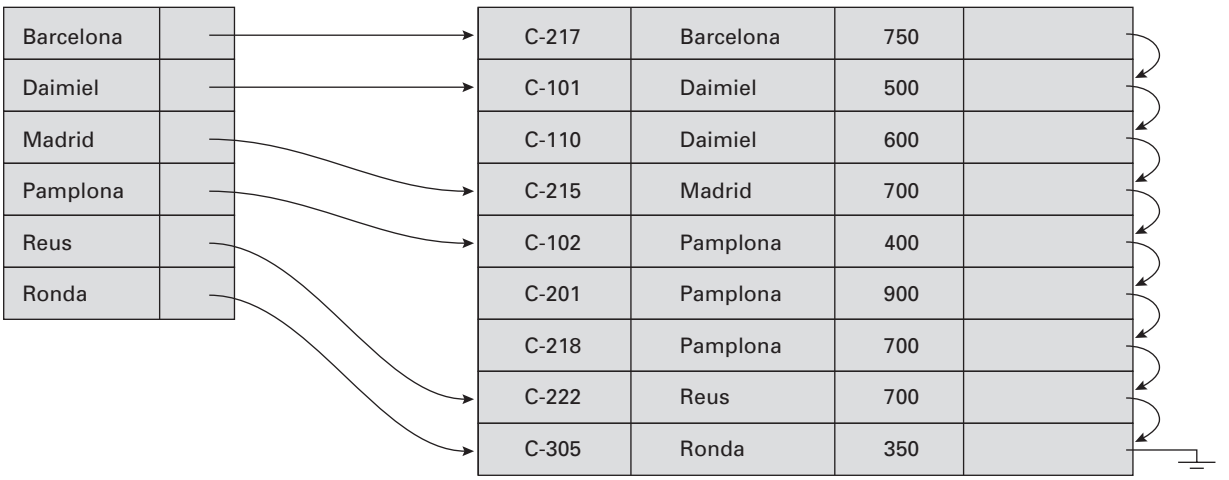

FIGURA 12.2. Índice denso.

12.2.1.1. Índices densos y dispersos

Un **registro índice** o **entrada del índice** consiste en un valor de la clave de búsqueda y punteros a uno o más registros con ese valor de la clave de búsqueda. El puntero a un registro consiste en el identificador de un bloque de disco y un desplazamiento en el bloque de disco para identificar el registro dentro del bloque.

Hay dos clases de índices ordenados que se pueden emplear:

- **Índice denso.** Aparece un registro índice por cada valor de la clave de búsqueda en el archivo. El registro índice contiene el valor de la clave y un puntero al primer registro con ese valor de la clave de búsqueda. El resto de registros con el mismo valor de la clave de búsqueda se almacenan consecutivamente después del primer registro, dado que, ya que el índice es primario, los registros se ordenan sobre la misma clave de búsqueda.

  Las implementaciones de índices densos pueden almacenar una lista de punteros a todos los registros con el mismo valor de la clave de búsqueda; esto no es esencial para los índices primarios.
- **Índice disperso.** Sólo se crea un registro índice para algunos de los valores. Al igual que en los índices densos, cada registro índice contiene un valor de la clave de búsqueda y un puntero al primer registro con ese valor de la clave. Para localizar un registro se busca la entrada del índice con el valor más grande que sea menor o igual que el valor que se está buscando. Se empieza por el registro apuntado por esa entrada del índice y se continúa con los punteros del archivo hasta encontrar el registro deseado.

Las Figuras 12.2 y 12.3 son ejemplos de índices densos y dispersos, respectivamente, para el archivo *cuenta*. Supongamos que se desea buscar los registros de la

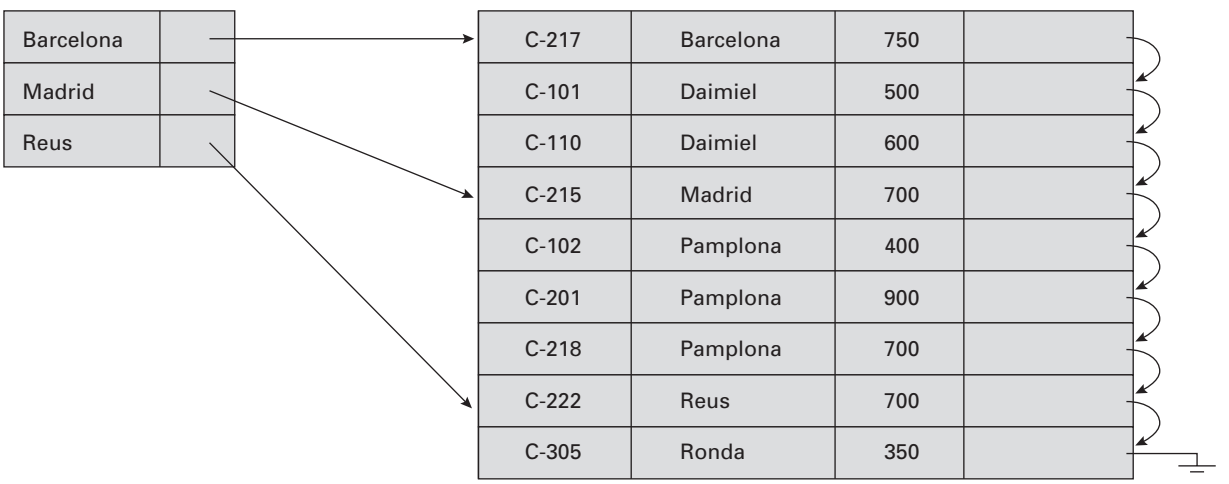

FIGURA 12.3. Índice disperso.

sucursal Pamplona. Mediante el índice denso de la Figura 12.2, se sigue el puntero que va directo al primer registro de Pamplona. Se procesa el registro y se sigue el puntero en ese registro hasta localizar el siguiente registro según el orden de la clave de búsqueda (*nombre-sucursal*). Se continuaría procesando registros hasta encontrar uno cuyo nombre de sucursal fuese distinto de Pamplona. Si se usa un índice disperso (Figura 12.3), no se encontraría entrada del índice para «Pamplona». Como la última entrada (en orden alfabético) antes de «Pamplona» es «Madrid», se sigue ese puntero. Entonces se lee el archivo *cuenta* en orden secuencial hasta encontrar el primer registro Pamplona, y se continúa procesando desde este punto.

Como se ha visto, generalmente es más rápido localizar un registro si se usa un índice denso en vez de un índice disperso. Sin embargo, los índices dispersos tienen algunas ventajas sobre los índices densos, como el utilizar un espacio más reducido y un mantenimiento adicional menor para las inserciones y borrados.

Existe un compromiso que el diseñador del sistema debe mantener entre el tiempo de acceso y el espacio adicional requerido. Aunque la decisión sobre este compromiso depende de la aplicación en particular, un buen compromiso es tener un índice disperso con una entrada del índice por cada bloque. La razón por la cual este diseño alcanza un buen compromiso reside en que el mayor coste de procesar una petición en la base de datos pertenece al tiempo empleado en traer un bloque de disco a la memoria. Una vez traído el bloque, el tiempo en examinar el bloque completo es despreciable. Usando este índice disperso se localiza el bloque que contiene el registro solicitado. De este manera, a menos que el registro esté en un bloque de desbordamiento (véase el Apartado 11.7.1) se minimizan los accesos a bloques mientras mantenemos el tamaño del índice (y así, nuestro espacio adicional requerido) tan pequeño como sea posible.

Para generalizar la técnica anterior hay que tener en cuenta cuando los registros de una clave de búsqueda ocupan varios bloques. Es fácil modificar el esquema para acomodarse a esta situación.

12.2.1.2. Índices multinivel

Incluso si se usan índices dispersos, el propio índice podría ser demasiado grande para un procesamiento eficiente. En la práctica no es excesivo tener un archivo con 100.000 registros, con 10 registros almacenados en cada bloque. Si tenemos un registro índice por cada bloque, el índice tendrá 10.000 registros. Como los registros índices son más pequeños que los registros de datos, podemos suponer que caben 100 registros índices en un bloque. Por tanto, el índice ocuparía 100 bloques. Estos índices de mayor tamaño se almacenan como archivos secuenciales en disco.

Si un índice es lo bastante pequeño como para guardarlo en la memoria principal, el tiempo de búsqueda para encontrar una entrada será breve. Sin embargo, si el índice es tan grande que se debe guardar en disco, buscar una entrada implicará leer varios bloques de disco. Para localizar una entrada en el archivo índice se puede realizar una búsqueda binaria, pero aun así ésta conlleva un gran coste. Si el índice ocupa b bloques, la búsqueda binaria tendrá que leer a lo sumo $\lceil \log_2(b) \rceil$ bloques. ($\lceil x \rceil$ denota al menor entero que es mayor o igual a x; es decir, se redondea hacia abajo.) Para el índice de 100 bloques, la búsqueda binaria necesitará leer siete bloques. En un disco en el que la lectura de un bloque tarda 30 milisegundos, la búsqueda empleará 210 milisegundos, lo que es mucho. Obsérvese que si se están usando bloques de desbordamiento, la búsqueda binaria no sería posible. En ese caso, lo normal es una búsqueda secuencial, y eso requiere leer b bloques, lo que podría consumir incluso más tiempo. Así, el proceso de buscar en un índice grande puede ser muy costoso.

Para resolver este problema se trata el índice como si fuese un archivo secuencial y se construye un índice disperso sobre el índice primario, como se muestra en la Figura 12.4. Para localizar un registro se usa en primer lugar una búsqueda binaria sobre el índice más externo para buscar el registro con el mayor valor de la clave de búsqueda que sea menor o igual al valor deseado. El puntero apunta a un bloque en el índice más interno. Hay que examinar este bloque hasta encontrar el registro con el mayor valor de la clave que sea menor o igual que el valor deseado. El puntero de este registro apunta al bloque del archivo que contiene el registro buscado.

Usando los dos niveles de indexación y con el índice más externo en memoria principal, tenemos que leer un único bloque índice en vez de los siete que se leían con la búsqueda binaria. Si al archivo es extremadamente grande, incluso el índice exterior podría crecer demasiado para caber en la memoria principal. En este caso se podría crear todavía otro nivel más de indexación. De hecho, se podría repetir este proceso tantas veces como fuese necesario. Los índices con dos o más niveles se llaman índices **multinivel**. La búsqueda de registros usando un índice multinivel necesita claramente menos operaciones de E/S que las que se emplean en la búsqueda de registros con la búsqueda binaria. Cada nivel de índice se podría corresponder con una unidad del almacenamiento físico. Así, podríamos tener índices a nivel de pista, cilindro o disco.

Un diccionario normal es un ejemplo de un índice multinivel en el mundo ajeno a las bases de datos. La cabecera de cada página lista la primera palabra en el orden alfabético en esa página. Este índice es multinivel: las palabras en la parte superior de la página del índice del libro forman un índice disperso sobre los contenidos de las páginas del diccionario.

Los índices multinivel están estrechamente relacionados con la estructura de árbol, tales como los árboles binarios usados para la indexación en memoria. Examinaremos esta relación posteriormente en el Apartado 12.3.

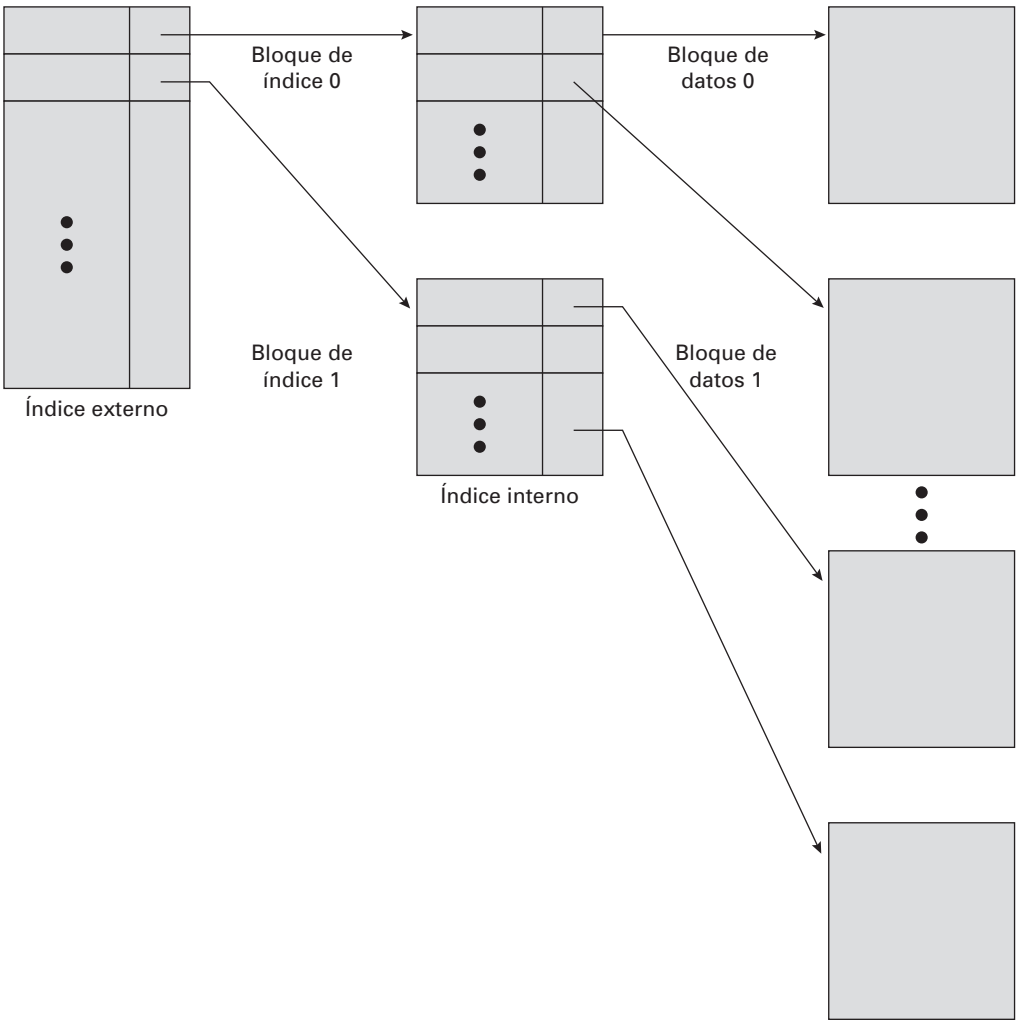

FIGURA 12.4. Índice disperso de dos niveles.

12.2.1.3. Actualización del índice

Sin importar el tipo de índice que se esté usando, los índices se deben actualizar siempre que se inserte o borre un registro del archivo. A continuación se describirán los algoritmos para actualizar índices de un solo nivel.

- **Inserción.** Primero se realiza una búsqueda usando el valor de la clave de búsqueda del registro a insertar. Las acciones que emprende el sistema a continuación dependen de si el índice es denso o disperso.
  - Índices densos:
    1. Si el valor de la clave de búsqueda no aparece en el índice, el sistema inserta en éste un registro índice con el valor de la clave de búsqueda en la posición adecuada.
    2. En caso contrario se emprenden las siguientes acciones:
       a. Si el registro índice almacena punteros a todos los registros con el mismo valor de la clave de búsqueda, el sistema añade un puntero al nuevo registro en el registro índice.
       b. En caso contrario, el registro índice almacena un puntero sólo hacia el primer registro con el valor de la clave de búsqueda. El sistema sitúa el registro insertado después de los otros con los mismos valores de la clave de búsqueda.
  - Índices dispersos: se asume que el índice almacena una entrada por cada bloque. Si el sistema crea un bloque nuevo, inserta el primer valor de la clave de búsqueda (en el orden de la clave de búsqueda) que aparezca en el nuevo bloque del índice. Por otra parte, si el nuevo registro tiene el menor valor de la clave de búsqueda en su bloque, el sistema actualiza la entrada del índice que apunta al bloque; si no,

el sistema no realiza ningún cambio sobre el índice.

- **Borrado.** Para borrar un registro, primero se busca el índice a borrar. De nuevo, las acciones que emprende el sistema a continuación dependen de si el índice es denso o disperso.
  - Índices densos:
    1. Si el registro borrado era el único registro con ese valor de la clave de búsqueda, el sistema borra el registro índice correspondiente del índice.
    2. En caso contrario se emprenden las siguientes acciones:
       a. Si el registro índice almacena punteros a todos los registros con el mismo valor de la clave de búsqueda, el sistema borra del registro índice el puntero al registro borrado.
       b. En caso contrario, el registro índice almacena un puntero sólo al primer registro con el valor de la clave de búsqueda. En este caso, si el registro borrado era el primer registro con el valor de la clave de búsqueda, el sistema actualiza el registro índice para apuntar al siguiente registro.
  - Índices dispersos:
    1. Si el índice no contiene un registro índice con el valor de la clave de búsqueda del registro borrado, no hay que hacer nada.
    2. En caso contrario se emprenden las siguientes acciones:
       a. Si el registro borrado era el único registro con la clave de búsqueda, el sistema reemplaza el registro índice correspondiente con un registro índice para el siguiente valor de la clave de búsqueda (en el orden de la clave de búsqueda). Si el siguiente valor de la clave de búsqueda ya tiene una entrada en el índice, se borra en lugar de reemplazarla.
       b. En caso contrario, si el registro índice para el valor de la clave de búsqueda apunta al registro a borrar, el sistema actualiza el registro índice para que apunte al siguiente registro con el mismo valor de la clave de búsqueda.

Los algoritmos de inserción y borrado para los índices multinivel se extienden de manera sencilla a partir del esquema descrito anteriormente. Al borrar o al insertar se actualiza el índice de nivel más bajo como se describió anteriormente. Por lo que respecta al índice del segundo nivel, el índice de nivel más bajo es simplemente un archivo de registros; así, si hay algún cambio en el índice de nivel más bajo, se tendrá que actualizar el índice del segundo nivel como ya se describió. La misma técnica se aplica al resto de niveles del índice, si los hubiera.

12.2.2. Índices secundarios

Los índices secundarios deben ser densos, con una entrada en el índice por cada valor de la clave de búsqueda, y un puntero a cada registro del archivo. Un índice primario puede ser disperso, almacenando sólo algunos de los valores de la clave de búsqueda, ya que siempre es posible encontrar registros con valores de la clave de búsqueda intermedios mediante un acceso secuencial a parte del archivo, como se describió antes. Si un índice secundario almacena sólo algunos de los valores de la clave de búsqueda, los registros con los valores de la clave de búsqueda intermedios pueden estar en cualquier lugar del archivo y, en general, no se pueden encontrar sin explorar el archivo completo.

Un índice secundario sobre una clave candidata es como un índice denso primario, excepto en que los registros apuntados por los sucesivos valores del índice no están almacenados secuencialmente. Por lo general, los índices secundarios están estructurados de manera diferente a como lo están los índices primarios. Si la clave de búsqueda de un índice primario no es una clave candidata, es suficiente si el valor de cada entrada en el índice apunta al primer registro con ese valor en la clave de búsqueda, ya que los otros registros podrían ser alcanzados por una búsqueda secuencial del archivo.

En cambio, si la clave de búsqueda de un índice secundario no es una clave candidata, no sería suficiente apuntar sólo al primer registro de cada valor de la clave. El resto de registros con el mismo valor de la clave de búsqueda podrían estar en cualquier otro sitio del archivo, ya que los registros están ordenados según la clave de búsqueda del índice primario, en vez de la clave de búsqueda del índice secundario. Por tanto, un índice secundario debe contener punteros a todos los registros.

Se puede usar un nivel adicional de indirección para implementar los índices secundarios sobre claves de búsqueda que no sean claves candidatas. Los punteros en estos índices secundarios no apuntan directamente al archivo. En vez de eso, cada puntero apunta a un cajón que contiene punteros al archivo. En la Figura 12.5 se muestra la estructura del archivo *cuenta*, con un índice secundario que emplea un nivel de indirección adicional, y teniendo como clave de búsqueda el *saldo*.

Siguiendo el orden de un índice primario, una *búsqueda secuencial* es eficiente porque los registros del archivo están guardados físicamente de la misma manera a como está ordenado el índice. Sin embargo, no se puede (salvo en raros casos excepcionales) almacenar el archivo ordenado físicamente por el orden de la clave de búsqueda del índice primario y la clave de búsqueda del índice secundario. Ya que el orden de la cla-

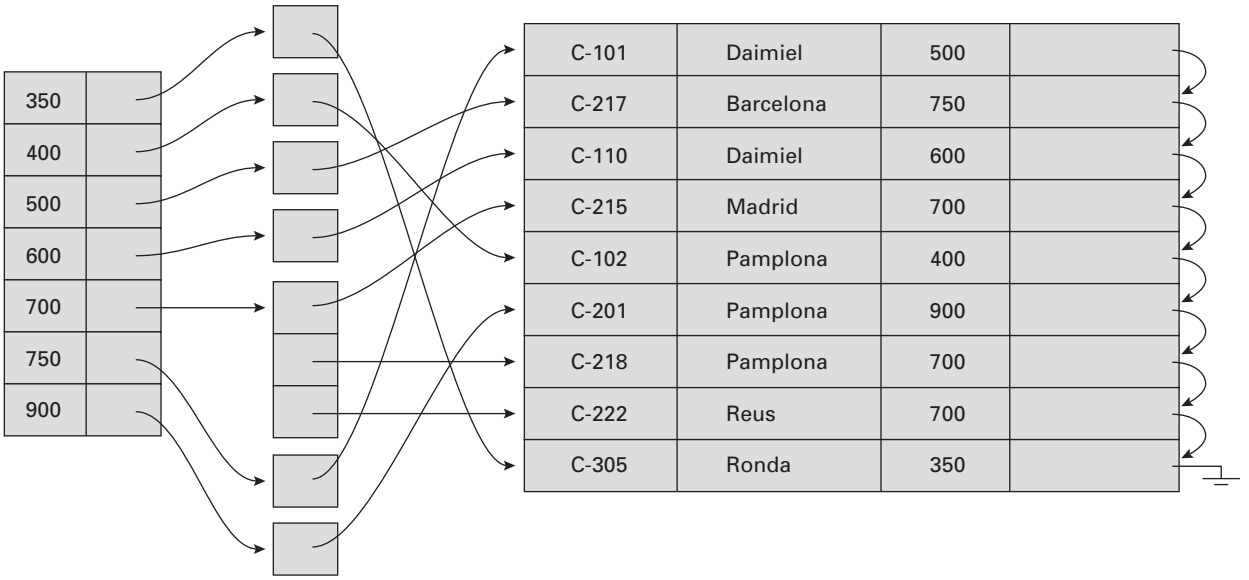

FIGURA 12.5. Índice secundario del archivo *cuenta*, con la clave no candidata *saldo*.

ve secundaria y el orden físico difieren, si se intenta examinar el archivo secuencialmente por el orden de la clave secundaria, es muy probable que la lectura de cada bloque suponga la lectura de un nuevo bloque del disco, lo cual es muy lento.

El procedimiento ya descrito para borrar e insertar se puede aplicar también a los índices secundarios; las acciones a emprender son las descritas para los índices densos que almacenan un puntero a cada registro del archivo. Si un archivo tiene varios índices, siempre que se modifique el archivo, se debe actualizar *cada* uno de ellos.

Los índices secundarios mejoran el rendimiento de las consultas que emplean claves que no son la de búsqueda del índice primario. Sin embargo, implican un tiempo adicional importante al modificar la base de datos. El diseñador de la base de datos debe decidir qué índices secundarios son deseables, según una estimación sobre las frecuencias de las consultas y las modificaciones.

12.3. ARCHIVOS DE ÍNDICES DE ÁRBOL B$^+$

El inconveniente principal de la organización de un archivo secuencial indexado reside en que el rendimiento, tanto para buscar en el índice como para buscar secuencialmente a través de los datos, se degrada según crece el archivo. Aunque esta degradación se puede remediar reorganizando el archivo, el rendimiento de tales reorganizaciones no es deseable.

La estructura de índice de **árbol B$^+$** es la más extendida de las estructuras de índices que mantienen su eficiencia a pesar de la inserción y borrado de datos. Un índice de árbol B$^+$ toma la forma de un **árbol equilibrado** donde los caminos de la raíz a cada hoja del árbol son de la misma longitud. Cada nodo que no es una hoja tiene entre $\lceil n/2 \rceil$ y n hijos, donde n es fijo para cada árbol en particular.

Se verá que la estructura de árbol B$^+$ implica una degradación del rendimiento al insertar y al borrar, además de un espacio extra. Este tiempo adicional es aceptable incluso en archivos con altas frecuencias de modificación, ya que se evita el coste de reorganizar el archivo. Además, puesto que los nodos podrían estar a lo sumo medio llenos (si tienen el mínimo número de hijos) hay algo de espacio desperdiciado. Este gasto de espacio adicional también es aceptable dados los beneficios en el rendimiento aportados por las estructura de árbol B$^+$.

12.3.1. Estructura de árbol B$^+$

Un índice de árbol B$^+$ es un índice multinivel pero con una estructura que difiere del índice multinivel de un archivo secuencial. En la Figura 12.6 se muestra un nodo

P_1	K_1	P_2	...	P_{n-1}	K_{n-1}	P_n

FIGURA 12.6. Nodo típico de un árbol B$^+$.

típico de un árbol B^+. Puede contener hasta $n - 1$ claves de búsqueda K_1, K_2, ..., K_{n-1} y n punteros P_1, P_2, ..., P_n. Los valores de la clave de búsqueda de un nodo se mantienen ordenados; así, si $i < j$, entonces $K_i < K_j$.

Consideraremos primero la estructura de los nodos hoja. Para $i = 1, 2, ..., n - 1$, el puntero P_i apunta, o bien a un registro del archivo con valor de la clave de búsqueda K_i, o bien a un cajón de punteros, cada uno de los cuales apunta a un registro del archivo con valor de la clave de búsqueda K_i. La estructura cajón se usa solamente si la clave de búsqueda no forma una clave primaria y si el archivo no está ordenado según la clave de búsqueda. El puntero P_n tiene un propósito especial que discutiremos más adelante.

En la Figura 12.7 se muestra un nodo hoja en el árbol B^+ del archivo *cuenta*, donde n vale tres y la clave de búsqueda es *nombre-sucursal*. Obsérvese que, como el archivo cuenta está ordenado por *nombre-sucursal*, los punteros en el nodo hoja apuntan directamente al archivo.

Ahora que se ha visto la estructura de un nodo hoja, se mostrará cómo los valores de la clave de búsqueda se asignan a nodos concretos. Cada hoja puede guardar hasta $n - 1$ valores. Está permitido que los nodos hojas contengan al menos $\lceil (n - 1)/2 \rceil$ valores. Los rangos de los valores en cada hoja no se solapan. Así, si L_i y L_j son nodos hoja e $i < j$, entonces cada valor de la clave de búsqueda en L_i es menor que cada valor de la clave en L_j . Si el índice de árbol B^+ es un índice denso, cada valor de la clave de búsqueda debe aparecer en algún nodo hoja.

Ahora se puede explicar el uso del puntero P_n. Dado que existe un orden lineal en las hojas basado en los valores de la clave de búsqueda que contienen, se usa P_n para encadenar juntos los nodos hojas en el orden de la clave de búsqueda. Esta ordenación permite un procesamiento secuencial eficaz del archivo.

Los nodos internos del árbol B^+ forman un índice multinivel (disperso) sobre los nodos hoja. La estructura de los nodos internos es la misma que la de los nodos hoja excepto que todos los punteros son punteros a nodos del árbol. Un nodo interno podría guardar hasta n punteros y *debe* guardar al menos $\lceil n/2 \rceil$ punteros. El número de punteros de un nodo se llama *grado de salida* del nodo.

Consideremos un nodo que contiene m punteros. Para $i = 2, 3, ..., m - 1$, el puntero P_i apunta al subárbol que contiene los valores de la clave de búsqueda menores que K_i y mayor o igual que K_{i-1}. El puntero P_m apunta a la parte del subárbol que contiene los valores de la clave mayores o iguales que K_{m-1}, y el puntero P_1 apunta a la parte del subárbol que contiene los valores de la clave menores que K_1.

A diferencia de otros nodos internos, el nodo raíz puede tener menos de $\lceil n/2 \rceil$; sin embargo, debe tener al menos dos punteros, a menos que el árbol consista en un solo nodo. Siempre es posible construir un árbol B^+, para cualquier n, que satisfaga los requisitos anteriores. En la Figura 12.8 se muestra un árbol B^+ completo para el archivo *cuenta* ($n = 3$). Por simplicidad se omiten los punteros al propio archivo y los punteros nulos. Como un ejemplo de un árbol B^+ en el cual la raíz debe tener menos de $\lceil n/2 \rceil$ valores, en la Figura 12.9 se muestra un árbol B^+ para el archivo *cuenta* con $n = 5$.

En todos los ejemplos mostrados de árboles B^+, éstos están equilibrados. Es decir, la longitud de cada camino desde la raíz a cada nodo hoja es la misma. Esta propiedad es un requisito de los árboles B^+. De hecho, la «B» en árbol B^+ proviene del inglés *balanced* (equilibrado). Es esta propiedad de equilibrio de los árboles B^+ la que asegura un buen rendimiento para las búsquedas, inserciones y borrados.

12.3.2. Consultas con árboles B^+

Considérese ahora cómo procesar consultas usando árboles B^+. Supóngase que se desean encontrar todos los registros cuyo valor de la clave de búsqueda sea V. La Figura 12.10 muestra el pseudocódigo para hacerlo. Primero se examina el nodo raíz para buscar el menor valor de la clave de búsqueda mayor que V. Supóngase que este valor de la clave de búsqueda es K_i. Siguiendo el puntero P_i hasta otro nodo. Si no se encuentra ese valor, entonces $k >= K_{m-1}$, donde m es el número de punteros del nodo. Es este caso se sigue P_m hasta otro nodo. En el nodo alcanzado anteriormente se busca de nuevo el

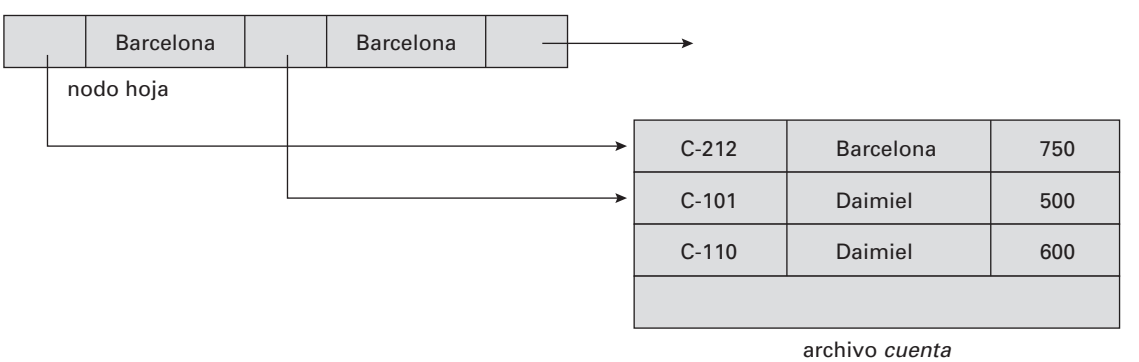

FIGURA 12.7. Nodo hoja para el índice del árbol B^+ de *cuenta* ($n = 3$).

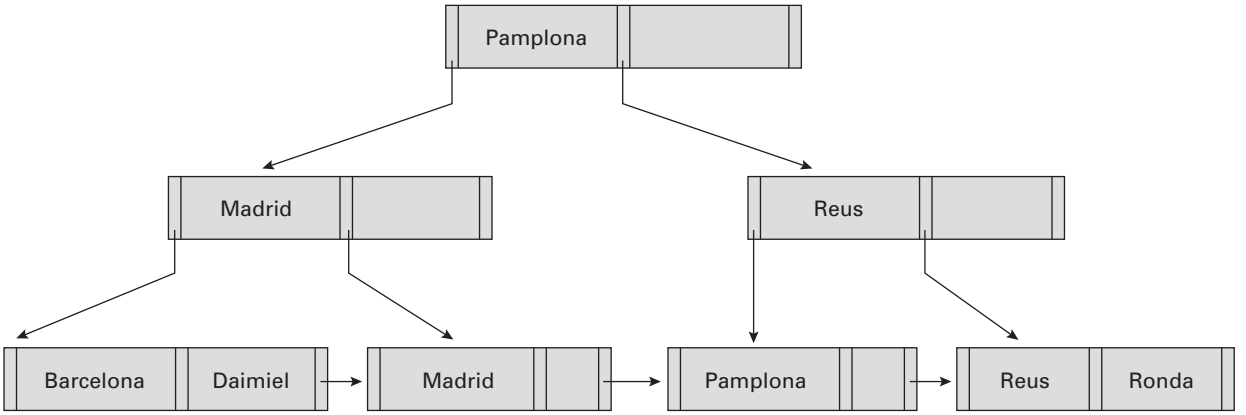

FIGURA 12.8. Árbol B$^+$ para el archivo *cuenta* (n = 3).

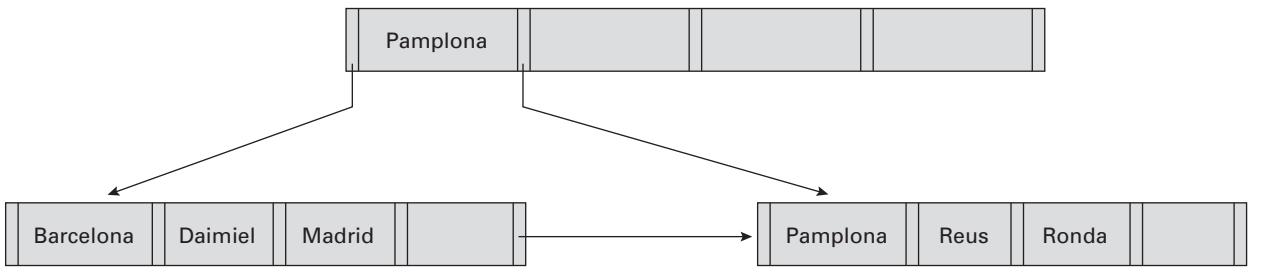

FIGURA 12.9. Árbol B$^+$ para el archivo *cuenta* (n = 5).

menor valor de la clave de búsqueda que es mayor que V para seguir el puntero correspondiente. Finalmente se alcanza un nodo hoja. En este nodo hoja, si se encuentra que el valor K_i es igual a V, entonces el puntero P_i nos ha conducido al registro o cajón deseado. Si no se encuentra el valor V en el nodo hoja, no existe ningún registro con el valor clave V.

De esta manera, para procesar una consulta, se tiene que recorrer un camino en el árbol desde la raíz hasta algún nodo hoja. Si hay K valores de la clave de búsqueda en el archivo, este camino no será más largo que $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$.

En la práctica sólo se accede a unos cuantos nodos. Generalmente un nodo se construye para tener el mismo tamaño que un bloque de disco, el cual ocupa normalmente 4 KB. Con una clave de búsqueda del tamaño de 12 bytes y un tamaño del puntero a disco de 8 bytes, n está alrededor de 200. Incluso con una estimación más conservadora de 32 bytes para el tamaño de la clave de búsqueda, n está en torno a 100. Con n = 100, si se tienen un millón de valores de la clave de búsqueda en el archivo, una búsqueda necesita solamente $\lceil \log_{50}(1.000.000) \rceil = 4$ accesos a nodos. Por tanto, se necesitan leer a lo sumo cuatro bloques del dis-

```
procedure buscar(valor V)
    set C = nodo raíz
    while C no sea un nodo raíz begin
        Sea Ki = menor valor de la clave de búsqueda, si lo hay, mayor que V
        if no hay tal valor then begin
            Sea m = el número de punteros en el nodo
            Sea C = nodo apuntado por Pm
        end
            else Sea C = el nodo apuntado por Pi
    end
    if hay un valor de clave Ki en C tal que Ki = C
        then el puntero Pi conduce al registro o cajón deseado
        else no existe ningún registro con el valor clave k
end procedure
```

FIGURA 12.10. Consulta con un árbol B$^+$.

co para realizar la búsqueda. Normalmente, el nodo raíz del árbol es el más accedido y por ello se guarda en una memoria intermedia; así solamente se necesitan tres o menos lecturas del disco.

Una diferencia importante entre las estructuras de árbol B^+ y los árboles en memoria, tales como los árboles binarios, está en el tamaño de un nodo y, por tanto, la altura del árbol. En un árbol binario, cada nodo es pequeño y tiene a lo sumo dos punteros. En un árbol B^+, cada nodo es grande —normalmente un bloque del disco— y un nodo puede tener un gran número de punteros. Así, los árboles B^+ tienden a ser bajos y anchos, en lugar de los altos y estrechos árboles binarios. En un árbol equilibrado, el camino de una búsqueda puede tener una longitud de $\lceil \log_2 (K) \rceil$, donde K es el número de valores de la clave de búsqueda. Con $K = 1.000.000$, como en el ejemplo anterior, un árbol binario equilibrado necesita alrededor de 20 accesos a nodos. Si cada nodo estuviera en un bloque del disco distinto, serían necesarias 20 lecturas a bloques para procesar la búsqueda, en contraste con las cuatro lecturas del árbol B^+.

12.3.3. Actualizaciones en árboles B^+

El borrado y la inserción son más complicados que las búsquedas, ya que podría ser necesario **dividir** un nodo que resultara demasiado grande como resultado de una inserción, o **fusionar** nodos si un nodo se volviera demasiado pequeño (menor que $\lceil n/2 \rceil$ punteros). Además, cuando se divide un nodo o se fusionan un par de ellos, se debe asegurar que el equilibrio del árbol se mantiene. Para presentar la idea que hay detrás del borrado y la inserción en un árbol B^+, asumiremos que los nodos nunca serán demasiado grandes ni demasiado pequeños. Bajo esta suposición, el borrado y la inserción se realizan como se indica a continuación.

- **Inserción.** Usando la misma técnica que para buscar, se busca un nodo hoja donde tendría que aparecer el valor de la clave de búsqueda. Si el valor de la clave de búsqueda ya aparece en el nodo hoja, se inserta un nuevo registro en el archivo y, si es necesario, un puntero al cajón. Si el valor de la clave de búsqueda no aparece, se inserta el valor en el nodo hoja de tal manera que las claves de búsqueda permanezcan ordenadas. Luego insertamos el nuevo registro en el archivo y, si es necesario, creamos un nuevo cajón con el puntero apropiado.
- **Borrado.** Usando la misma técnica que para buscar, se busca el registro a borrar y se elimina del archivo. Si no hay un cajón asociado con el valor de la clave de búsqueda o si el cajón se queda vacío como resultado del borrado, se borra el valor de la clave de búsqueda del nodo hoja.

Pasamos ahora a considerar un ejemplo en el que se tiene que dividir un nodo. Por ejemplo, queremos insertar un registro en el árbol B^+ de la Figura 12.8, cuyo valor de *nombre-sucursal* es «Cádiz». Usando el algoritmo de búsqueda, «Cádiz» debería aparecer en el nodo que incluye «Barcelona» y «Daimiel». No hay sitio para insertar el valor de la clave de búsqueda «Cádiz». Por tanto, se *divide* el nodo en otros dos nodos. En la Figura 12.11 se muestran los nodos hoja que resultan de insertar «Cádiz» y de dividir el nodo que incluía «Barcelona» y «Daimiel». En general, si tenemos n valores de la clave de búsqueda (los $n - 1$ valores del nodo hoja más el valor a insertar), pondremos $\lceil n/2 \rceil$ en el nodo existente y el resto de valores en el nuevo nodo.

Para dividir un nodo hoja hay que insertar un nuevo nodo hoja en el árbol B^+. En el ejemplo, el nuevo nodo tiene a «Daimiel» como el valor más pequeño de la clave de búsqueda. Luego hay que insertar este valor de la clave de búsqueda en el padre del nodo hoja dividido. En el árbol B^+ de la Figura 12.12 se muestra el resultado de la inserción. El valor «Daimiel» de la clave de búsqueda se ha insertado en el padre. Ha sido posible llevar a cabo esta inserción porque había sitio para añadir un valor de la clave de búsqueda. Si no hubiera sitio, se tendría que dividir el padre. En el peor de los casos, todos los nodos en el camino hacia la raíz se tendrían que dividir. Si la propia raíz se tuviera que dividir, el árbol sería más profundo.

La técnica general para la inserción en un árbol B^+ es determinar el nodo hoja h en el cual realizar la inserción. Si es necesario dividir, se inserta el nuevo nodo dentro del padre del nodo h. Si esta inserción produce otra división, procederíamos recursivamente o bien hasta que una inserción no produzca otra división o bien hasta crear una nueva raíz.

En la Figura 12.13 se bosqueja el algoritmo de inserción en pseudocódigo. En el pseudocódigo $L.K_i$ y $L.P_i$ denotan al i-ésimo valor y el i-ésimo puntero en el nodo L, respectivamente. El pseudocódigo también hace uso de la función *padre*(L) para encontrar el padre del nodo L. Se puede obtener una lista de los nodos en el camino de la raíz a la hoja mientras buscamos el nodo hoja

FIGURA 12.11. División del nodo hoja tras la inserción de «Cádiz».

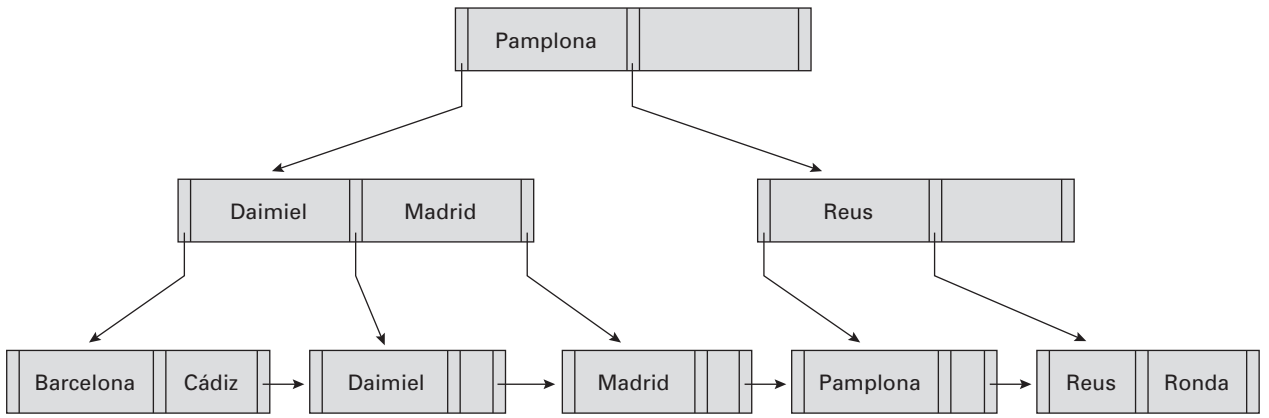

FIGURA 12.12. Inserción de «Cádiz» en el árbol B+ de la Figura 12.8.

para usarlos después a la hora de encontrar eficazmente el padre de cualquier nodo del camino. El pseudocódigo hace referencia a insertar una entrada (V, P) en un nodo. En el caso de nodos hoja, realmente el puntero a una entrada precede al valor de la clave, de tal manera que efectivamente se almacena P en el nodo hoja precediendo a V. Para los nodos internos, se almacena P justo después de V.

```
procedure insertar (valor V, puntero P)
    encontrar el nodo hoja L que debe contener el valor V
    insertar_entrada(L, V, P)
end procedure

procedure insertar_entrada (nodo L, valor V, puntero P)
    if (L tiene espacio para (V, P))
        then   insertar (V, P) en L
        else   begin /* Dividir L */
            Crear el nodo L'
            Sea V' el valor en K1, ...,Kn–1 tal que exactamente ⌈n/2⌉ de los valores L.K1, ...,L.Kn–1, V son menores que V'
            Sea m el menor valor tal que L.Km ≥ V'
            /* Nota: V' tiene que ser L.Km o V */
            if (L es una hoja) then begin
                mover L.Pm, L.Km, ..., L.Pn–1, L.Kn–1 a L'
                if (V < V') then insertar (V, P) en L
                else insertar (V, P) en L'
            end
            else   begin
                if (V = V') /* V es el menor valor que irá en L' */
                    then añadir P, L.Km, ..., L.Pn–1, L.Kn–1, L.Pn a L'
                    else añadir L.Pm, ..., L.Pn–1, L.Kn–1, L.Pn a L'
                borrar L.Km, ..., L.Pn–1, L.Kn–1, L.Pn de L
                if (V < V') then insertar (V, P) en L
                else if (V < V') then insertar (V, P) en L'
                /* El caso V = V' ya se trató anteriormente */
            end
            if (L no es la raíz del árbol)
                then   insertar_entrada(padre(L), V', L');
                else   begin
                    crear un nuevo nodo R con hijos los nodos L y L' y con el único valor V'
                    hacer de R la raíz del árbol
                end
            if (L) es un nodo hoja then begin /* Fijar los siguientes punteros a los hijos */
                hacer L'.Pn = L.Pn;
                hacer L.Pn = L'
            end
        end
end procedure
```

FIGURA 12.13. Inserción de una entrada en un árbol B+.

A continuación se consideran los borrados que provocan que el árbol se quede con muy pocos punteros. Primero se borra «Daimiel» del árbol B^+ de la Figura 12.12. Para ello se localiza la entrada «Daimiel» usando el algoritmo de búsqueda. Cuando se borra la entrada «Daimiel» de su nodo hoja, la hoja se queda vacía. Ya que en el ejemplo $n = 3$ y $0 < \lceil (n-1)/2 \rceil$, este nodo se debe borrar del árbol B^+. Para borrar un nodo hoja se tiene que borrar el puntero que le llega desde su padre. En el ejemplo, este borrado deja al nodo padre, el cual contenía tres punteros, con sólo dos punteros. Ya que $2 >= \lceil n/2 \rceil$, el nodo es todavía lo suficientemente grande y la operación de borrado se completa. El árbol B^+ resultante se muestra en la Figura 12.14.

Cuando un borrado se hace sobre el padre de un nodo hoja, el propio nodo padre podría quedar demasiado pequeño. Esto es exactamente los que ocurre si se borra «Pamplona» del árbol B^+ de la Figura 12.14. El borrado de la entrada Pamplona provoca que un nodo hoja se quede vacío. Cuando se borra el puntero a este nodo en su padre, el padre sólo se queda con un puntero. Así, $n = 3$, $\lceil n/2 \rceil = 2$ y queda tan sólo un puntero, que es demasiado poco. Sin embargo, ya que el nodo padre contiene información útil, no podemos simplemente borrarlo. En vez de eso, se busca el nodo hermano (el nodo interno que contiene al menos una clave de búsqueda, Madrid). Este nodo hermano tiene sitio para colocar la información contenida en el nodo que quedó demasiado pequeño, así que se fusionan estos nodos, de tal manera que el nodo hermano ahora contiene las claves «Madrid» y «Reus». El otro nodo (el nodo que contenía solamente la clave de búsqueda «Reus») ahora contiene información redundante y se puede borrar desde su padre (el cual resulta ser la raíz del ejemplo). En la Figura 12.15 se muestra el resultado. Hay que observar que la raíz tiene solamente un puntero hijo como resultado del borrado, así que éste se borra y el hijo solitario se convierte en la nueva raíz. De esta manera la profundidad del árbol ha disminuido en una unidad.

No siempre es posible fusionar nodos. Como ejemplo se borrará «Pamplona» del árbol B^+ de la Figura 11.11. En este ejemplo, la entrada «Daimiel» es todavía parte del árbol. Una vez más, el nodo hoja que contiene «Pamplona» se queda vacío. El padre del nodo hoja se queda también demasiado pequeño (únicamente con un puntero). De cualquier modo, en este ejemplo, el nodo hermano contiene ya el máximo número de punteros: tres. Así, no puede acomodar a un puntero adicional. La solución en este caso es **redistribuir** los punteros de tal manera que cada hermano tenga dos punteros. El resultado se muestra en la Figura 12.16. Obsérvese que la redistribución de los valores necesita de un cambio en el valor de la clave de búsqueda en el padre de los dos hermanos.

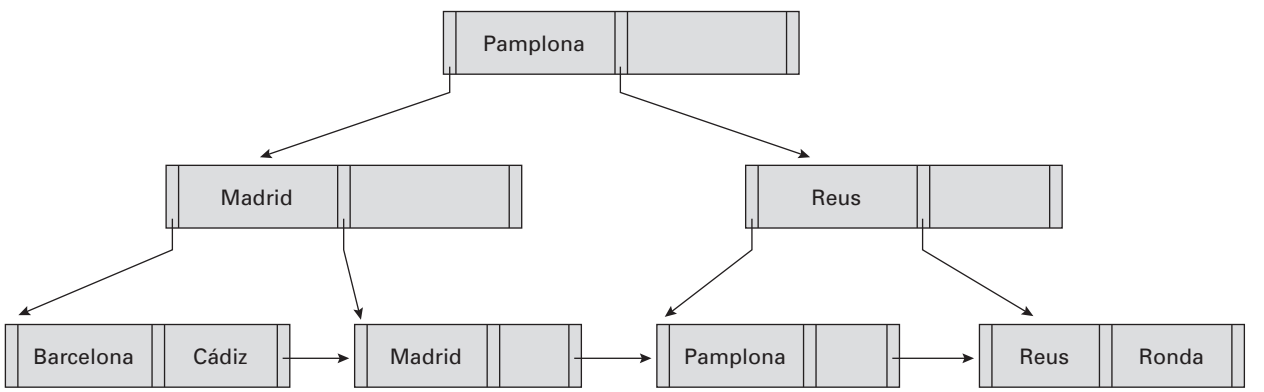

FIGURA 12.14. Borrado de «Daimiel» del árbol B^+ de la Figura 12.12.

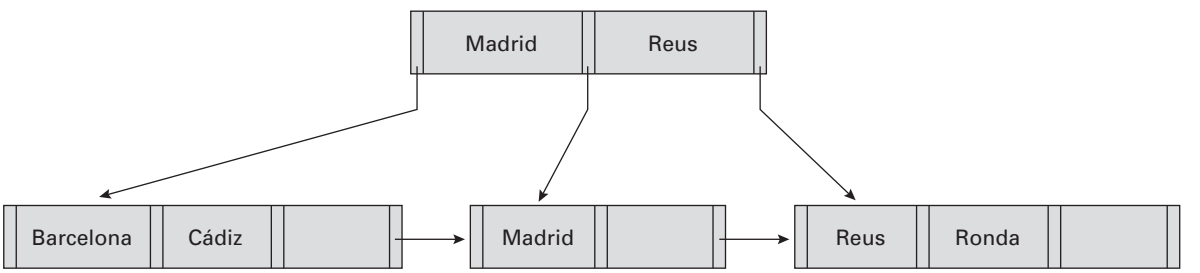

FIGURA 12.15. Borrado de «Pamplona» del árbol B^+ de la Figura 12.14.

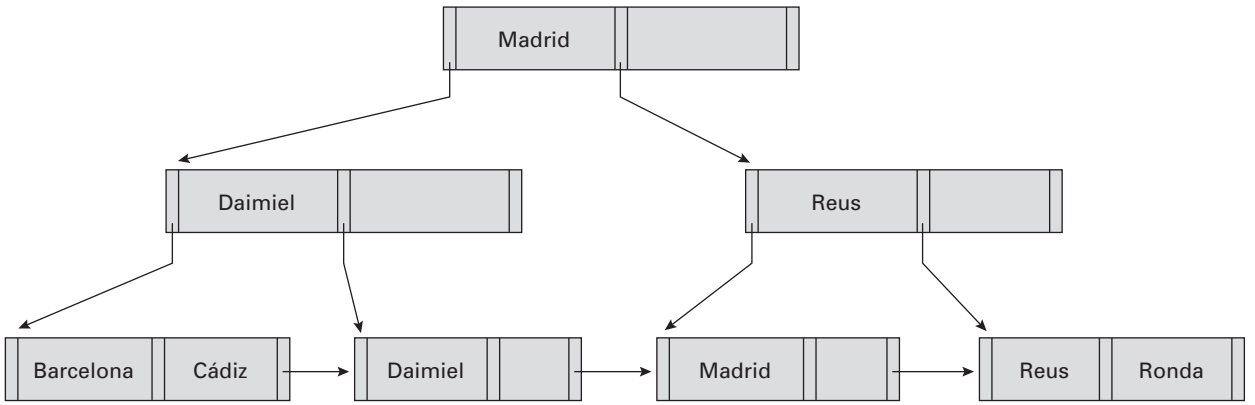

FIGURA 12.16. Borrado de «Pamplona» del árbol B^+ de la Figura 12.12.

En general, para borrar un valor en un árbol B^+ se realiza una búsqueda según el valor y se borra. Si el nodo es demasiado pequeño, se borra desde su padre. Este borrado se realiza como una aplicación recursiva del algoritmo de borrado hasta que se alcanza la raíz, un nodo padre queda lleno de manera adecuada después de borrar, o hasta aplicar una redistribución.

En la Figura 12.17 se describe el pseudocódigo para el borrado en un árbol B^+. El procedimiento intercambiar_variables(L, L') simplemente cambia de lugar los valores (punteros) de las variables L y L'; este cambio no afecta al árbol en sí mismo. El pseudocódigo utiliza la condición «muy pocos valores/punteros». Para nodos internos, este criterio quiere decir menos que $\lceil n/2 \rceil$ punteros; para nodos hoja, quiere decir menos que $\lceil (n-1)/2 \rceil$ valores. El pseudocódigo realiza la redistribución tomando prestada una sola entrada desde un nodo adyacente. También se puede redistribuir mediante la distribución equitativa de entradas entre dos nodos. El pseudocódigo hace referencia al borrado de una entrada (V, P) desde un nodo. En el caso de los nodos hoja, el puntero a una entrada realmente precede al valor de la clave; así, el puntero P precede al valor de la clave V. Para nodos internos, P sigue al valor de la clave V.

Es interesante señalar que como resultado de un borrado, un valor de la clave de un nodo interno del árbol B^+ puede que no esté en ninguna hoja del árbol.

Aunque las operaciones inserción y borrado en árboles B^+ son complicadas, requieren relativamente pocas operaciones, lo que es un beneficio importante dado el coste de las operaciones E/S. Se puede demostrar que el número de operaciones E/S necesarias para una inserción o borrado es, en el peor de los casos, proporcional a $\log_{\lceil n/2 \rceil}(K)$, donde n es el número máximo de punteros en un nodo y K es el número de valores de la clave de búsqueda. En otras palabras, el coste de las operaciones inserción y borrado es proporcional a la altura del árbol B^+ y es por lo tanto bajo. Debido a la velocidad de las operaciones en los árboles B^+, estas estructuras de índice se usan frecuentemente al implementar las bases de datos.

12.3.4. Organización de archivos con árboles B^+

Como se mencionó en el Apartado 12.3, el inconveniente de la organización de archivos secuenciales de índices es la degradación del rendimiento según crece el archivo: con el crecimiento, un porcentaje mayor de registros índice y registros reales se desaprovechan y se almacenan en bloques de desbordamiento. Se resuelve la degradación de las búsquedas en el índice mediante el uso de índices de árbol B^+ en el archivo. También se soluciona el problema de la degradación al almacenar los registros reales utilizando el nivel de hoja del árbol B^+ para almacenar los registros reales en los bloques. En estas estructuras, la estructura del árbol B^+ se usa no sólo como un índice, sino también como un organizador de los registros dentro del archivo. En la **organización de archivo con árboles B^+**, los nodos hoja del árbol almacenan registros, en lugar de almacenar punteros a registros. En la Figura 12.18 se muestra un ejemplo de la organización de un archivo con un árbol B^+. Ya que los registros son normalmente más grandes que los punteros, el número máximo de registros que se pueden almacenar en un nodo hoja es menor que el número de punteros en un nodo interno. Sin embargo, todavía se requiere que los nodos hoja estén llenos al menos hasta la mitad.

La inserción y borrado de registros en una organización de archivo con árboles B^+ se trata del mismo modo que la inserción y borrado de entradas en un índice de árbol B^+. Cuando se inserta un registro con un valor de clave v, se localiza el bloque que debería contener ese registro mediante la búsqueda en el árbol B^+ de la mayor clave que sea $\leq v$. Si el bloque localizado tiene bastante espacio libre para el registro, se almacena el registro en el bloque. De no ser así, como en una inserción en un árbol B^+, se divide el bloque en dos y se redistribuyen sus registros (en el orden de la clave

```
procedure borrar (valor V, puntero P)
    encontrar el nodo hoja L que contiene (V, P)
    borrar_entrada(L, V, P)
end procedure

procedure borrar_entrada (nodo L, valor V, puntero P)
    borrar (V, P) de L
    if (L es la raíz and L tiene sólo un hijo)
    then hacer del hijo de L la nueva raíz del árbol y borrar L
    else if (L tiene muy pocos valores/punteros) then begin
            Sea L' el anterior o el siguiente hijo de padre(L)
            Sea V' el valor enter los punteros L y L' en padre(L)
            if las entradas en L y L' caben en un solo nodo)
                then begin /* Fundir nodos */
                    if (L es un predecesor de L') then intercambiar_variables(L, L')
                    if (L no es una hoja)
                        then concatenar V' y todos los punteros y valores de L a L'
                        else concatenar todos los pares (Kj, Pj) de L a L'; hacer L'.Pn = L.Pn

                    borrar_entrada(padre(L), V', L); borrar el nodo L
                end
            else begin /* Redistribución : prestar una entrada de L'*/
                if (L' es un predecesor de L) then begin
                    if (L es un nodo interno) then begin
                        sea m tal que L'.Pm es el último puntero en L'
                        suprimir (L'.Km-1, L'.Pm) de L'
                        insertar (L'.Pm, V') como el primer puntero y valor en L, desplazando otros punteros y valores a la derecha
                        reemplazar V' en padre(L) por L'.Km-1
                    end
                    else begin
                        sea m tal que (L'.Pm, L'.Km) es el último par puntero/valor en L'
                        suprimir (L'.Pm, L'.Km) de L'
                        insertar (L'.Pm, L'.Km) como el primer puntero y valor en L, desplazando otros punteros y valores a la derecha
                        reemplazar V' en padre(L) por L'.Km
                    end
                end
                else ... caso simétrico al then ...
            end
        end
end procedure
```

FIGURA 12.17. Borrado de elementos de un árbol B^+.

del árbol B^+) creando espacio para el nuevo registro. Esta división se propaga hacia arriba en el árbol B^+ de la manera usual. Cuando se borra un registro, primero se elimina del bloque que lo contiene. Si como resultado un bloque B llega a ocupar menos que la mitad, los registros en B se redistribuyen con los registros en un bloque B' adyacente. Si se asume que los registros son de tamaño fijo, cada bloque contendrá por lo menos la

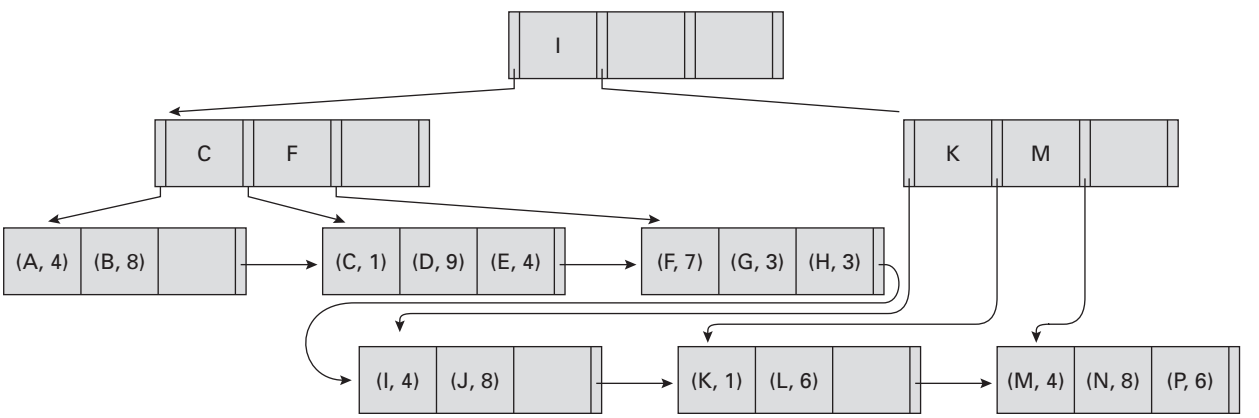

FIGURA 12.18. Organización de archivos con árboles B^+.

mitad de los registros que pueda contener como máximo. Los nodos internos del árbol B^+ se actualizan por tanto de la manera habitual.

Cuando un árbol B^+ se utiliza para la organización de un archivo, la utilización del espacio es particularmente importante, ya que el espacio ocupado por los registros es mucho mayor que el espacio ocupado por las claves y punteros. Se puede mejorar la utilización del espacio en un árbol B^+ implicando a más nodos hermanos en la redistribución durante las divisiones y fusiones. La técnica es aplicable a los nodos hoja y nodos internos y funciona como sigue.

Durante la inserción, si un nodo está lleno se intenta redistribuir algunas de sus entradas en uno de los nodos adyacentes para hacer sitio a una nueva entrada. Si este intento falla porque los nodos adyacentes están llenos, se divide el nodo y las entradas entre uno de los nodos adyacentes y los dos nodos que se obtienen al dividir el nodo original. Puesto que los tres nodos juntos contienen un registro más que puede encajar en dos nodos, cada nodo estará lleno aproximadamente hasta sus dos terceras partes. Para ser más precisos, cada nodo tendrá por lo menos $\lfloor 2n/3 \rfloor$ entradas, donde n es el número máximo de entradas que puede tener un nodo ($\lfloor n \rfloor$ denota el mayor entero menor o igual que x; es decir, eliminamos la parte fraccionaria si la hay).

Durante el borrado de un registro, si la ocupación de un nodo está por debajo de $\lfloor 2n/3 \rfloor$, se intentará tomar prestada una entrada desde uno de sus nodos hermanos. Si ambos nodos hermanos tienen $\lfloor 2n/3 \rfloor$ registros, en lugar de tomar prestada una entrada, se redistribuyen las entradas en el nodo y en los dos nodos hermanos uniformemente entre dos de los nodos y se borra el tercer nodo. Se puede usar este enfoque porque el número total de entradas es $3 \lfloor 2n/3 \rfloor - 1$, lo cual es menor que $2n$. Utilizando tres nodos adyacentes para la redistribución se garantiza que cada nodo pueda tener $\lfloor 3n/4 \rfloor$ entradas. En general, si hay m nodos ($m - 1$ hermanos) implicados en la redistribución se garantiza que cada nodo pueda contener al menos $\lfloor (m-1)n/m \rfloor$ entradas. Sin embargo, el costo de la actualización se vuelve mayor según haya más nodos hermanos involucrados en la redistribución.

12.4. ARCHIVOS CON ÍNDICES DE ÁRBOL B

Los *índices de árbol B* son similares a los índices de árbol B^+. La diferencia principal entre los dos enfoques es que un árbol B elimina el almacenamiento redundante de los valores de la clave búsqueda. En el árbol B^+ de la Figura 12.12, las claves de búsqueda «Daimiel», «Madrid», «Reus» y «Pamplona» aparecen dos veces. Cada valor de clave de búsqueda aparece en algún nodo hoja; algunos se repiten en nodos internos.

Un árbol B permite que los valores de la clave de búsqueda aparezcan solamente una vez. En la Figura 12.19 se muestra un árbol B que representa las mismas claves de búsqueda que el árbol B^+ de la Figura 12.12. Ya que las claves de búsqueda no están repetidas en el árbol B, sería posible almacenar el índice empleando menos nodos del árbol que con el correspondiente índice de árbol B^+. Sin embargo, puesto que las claves de búsqueda que aparecen en los nodos internos no aparecen en ninguna otra parte del árbol B, nos vemos obligados a incluir un campo adicional para un puntero por cada clave de búsqueda de un nodo interno. Estos punteros adicionales apuntan a registros del archivo o a los cajones de la clave de búsqueda asociada.

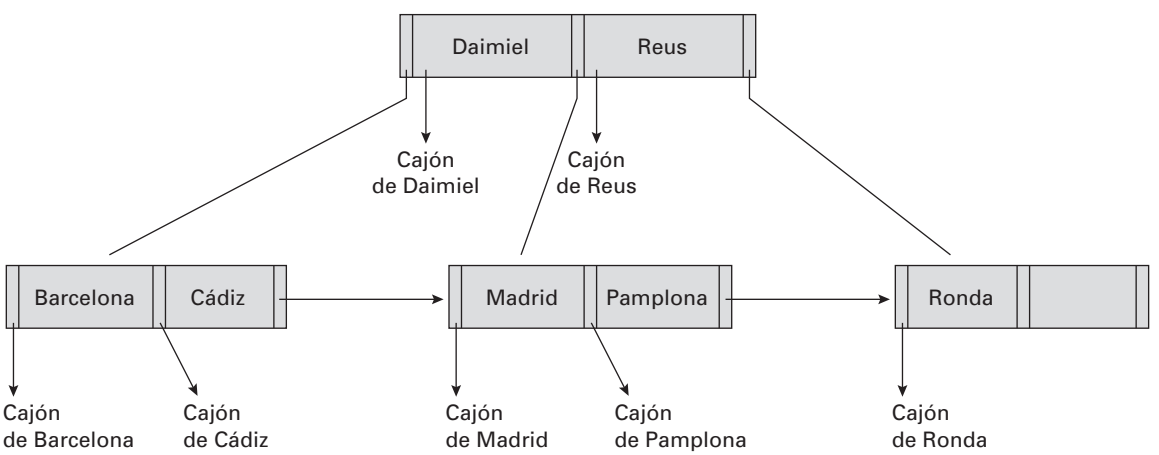

FIGURA 12.19. Árbol B equivalente al árbol B^+ de la Figura 12.12.

Un nodo hoja generalizado de un árbol B aparece en la Figura 12.20a; un nodo interno aparece en la Figura 12.20b. Los nodos hoja son como en los árboles B^+. En los nodos internos los punteros P_i son los punteros del árbol que se utilizan también para los árboles B^+, mientras que los punteros B_i en los nodos internos son punteros a cajones o registros del archivo. En la figura del árbol B generalizado hay $n - 1$ claves en el nodo hoja, mientras que hay $m - 1$ claves en el nodo interno. Esta discrepancia ocurre porque los nodos internos deben incluir los punteros B_i, y de esta manera se reduce el número de claves de búsqueda que pueden contener estos nodos. Claramente, $m < n$, pero la relación exacta entre m y n depende del tamaño relativo de las claves de búsqueda y de los punteros.

El número de nodos accedidos en una búsqueda en un árbol B depende de dónde esté situada la clave de búsqueda. Una búsqueda en un árbol B^+ requiere atravesar un camino desde la raíz del árbol hasta algún nodo hoja. En cambio, algunas veces es posible encontrar en un árbol B el valor deseado antes de alcanzar el nodo hoja. Sin embargo, hay que realizar aproximadamente n accesos según cuántas claves haya almacenadas tanto en el nivel de hoja de un árbol B como en los niveles internos de hoja y, dado que n es normalmente grande, la probabilidad de encontrar ciertos valores pronto es relativamente pequeña. Por otra parte, el hecho de que menos claves de búsqueda aparezcan en los nodos internos del árbol B, comparado con los árboles B^+, implica que un árbol B tiene un grado de salida menor y, por lo tanto, puede que tenga una profundidad mayor que el correspondiente al árbol B^+. Así, la búsqueda en un árbol B es más rápida para algunas claves de búsqueda pero más lenta para otras, aunque en general, el tiempo de la búsqueda es todavía proporcional al logaritmo del número de claves de búsqueda.

El borrado en un árbol B es más complicado. En un árbol B^+ la entrada borrada siempre aparece en una hoja. En un árbol B, la entrada borrada podría aparecer en un nodo interno. El valor apropiado a colocar en su lugar se debe elegir del subárbol del nodo que contiene la entrada borrada. Concretamente, si se borra la clave de búsqueda K_i, la clave de búsqueda más pequeña que aparezca en el subárbol del puntero $P_i + 1$ se debe trasladar al campo ocupado anteriormente por K_i. Será necesario tomar otras medidas si ahora el nodo hoja tuviera pocas entradas. Por el contrario, la inserción en un árbol B es sólo un poco más complicada que la inserción en un árbol B^+.

Las ventajas de espacio que tienen los árboles B son escasas para índices grandes y normalmente no son de mayor importancia los inconvenientes que hemos advertido. De esta manera, muchos implementadores de sistemas de bases de datos aprovechan la sencillez estructural de un árbol B^+. Los detalles de los algoritmos de inserción y borrado para árboles B se estudian en los ejercicios.

12.5. ASOCIACIÓN ESTÁTICA

Un inconveniente de la organización de archivos secuenciales es que hay que acceder a una estructura de índices para localizar los datos o utilizar una búsqueda binaria y, como resultado, más operaciones de E/S. La organización de archivos basada en la técnica de **asociación** (*hashing*) permite evitar el acceso a la estructura de índice. La asociación también proporciona una forma de construir índices. Se estudiarán las organizaciones de archivos e índices basados en asociación en los próximos apartados.

12.5.1. Organización de archivos por asociación

En una **organización de archivos por asociación** se obtiene la dirección del bloque de disco que contiene el registro deseado mediante el cálculo directo de una función sobre el valor de la clave búsqueda del registro. En nuestra descripción de asociación, utilizaremos el término **cajón** (*bucket*) para indicar una unidad de almacenamiento que puede guardar uno o más registros. Un

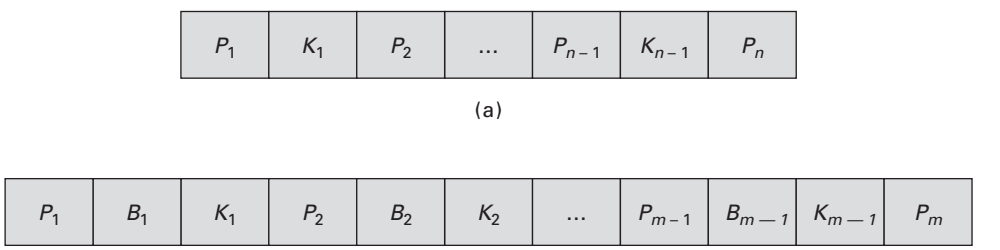

FIGURA 12.20. Nodos típicos de un árbol B. (a) Nodo hoja. (b) Nodo interno.

cajón es normalmente un bloque de disco, aunque también se podría elegir más pequeño o más grande que un bloque de disco.

Formalmente, sea K el conjunto de todos los valores de clave de búsqueda y sea B el conjunto de todas las direcciones de cajón. Una **función de asociación** h es una función de K a B. Sea h una función asociación.

Para insertar un registro con clave de búsqueda K_i, calcularemos $h(K_i)$, lo que proporciona la dirección del cajón para ese registro. Se supone que por ahora hay espacio en el cajón para almacenar el registro. Entonces, el registro se almacena en este cajón.

Para realizar una búsqueda con el valor K_i de la clave de búsqueda, simplemente se calcula $h(K_i)$ y luego se busca el cajón con esa dirección. Supongamos que dos claves de búsqueda, K_5 y K_7, tienen el mismo valor de asociación; esto es, $h(K_5) = h(K_7)$. Si se realiza una búsqueda en K_5, el cajón $h(K_5)$ contendrá registros con valores de la clave de búsqueda K_5 y registros con valores de la clave de búsqueda K_7. Así, hay que comprobar el valor de clave de búsqueda de cada registro en el cajón para verificar que el registro es el que queremos.

El borrado es igual de sencillo. Si el valor de clave de búsqueda del registro a borrar es K_i, se calcula $h(K_i)$, después se busca el correspondiente cajón para ese registro y se borra el registro del cajón.

12.5.1.1. Funciones de asociación

La peor función posible de asociación asigna todos los valores de la clave de búsqueda al mismo cajón. Tal función no es deseable, ya que todos los registros tienen que guardarse en el mismo cajón. Una búsqueda tiene que examinar cada registro hasta encontrar el deseado. Una función de asociación ideal distribuye las claves almacenadas uniformemente a través de los cajones para que cada uno de ellos tenga el mismo número de registros.

Puesto que no se sabe durante la etapa de diseño qué valores de la clave de búsqueda se almacenarán en el archivo, se pretende elegir una función de asociación que asigne los valores de las claves de búsqueda a los cajones de manera que se cumpla lo siguiente:

- Distribución *uniforme*. Esto es, cada cajón tiene asignado el mismo número de valores de la clave de búsqueda dentro del conjunto de *todos* los valores posibles de la clave de búsqueda.
- Distribución *aleatoria*. Esto es, en el caso promedio, cada cajón tendrá casi el mismo número de valores asignados a él, sin tener en cuenta la distribución actual de los valores de la clave de búsqueda. Para ser más exactos, el valor de asociación no será correlativo a ninguna orden exterior visible en los valores de la clave de búsqueda, como por ejemplo el orden alfabético o el orden determinado por la longitud de las claves de búsqueda; la función de asociación tendrá que parecer ser aleatoria.

Como una ilustración de estos principios, vamos a escoger una función de asociación para el archivo *cuenta* utilizando la clave búsqueda *nombre-sucursal*. La función de asociación que se escoja debe tener las propiedades deseadas no sólo en el ejemplo del archivo *cuenta* que se ha estado utilizando, sino también en el archivo *cuenta* de tamaño real para un gran banco con muchas sucursales.

Supóngase que se decide tener 26 cajones y se define una función de asociación que asigna a los nombres que empiezan con la letra i-ésima del alfabeto el i-ésimo cajón. Esta función de asociación tiene la virtud de la simplicidad, pero no logra proporcionar una distribución uniforme, ya que se espera tener más nombres de sucursales que comienzan con letras como «B» y «R» que con «Q» y «X», por citar un ejemplo.

Ahora supóngase que se quiere una función de asociación en la clave de búsqueda *saldo*. Supóngase que el saldo mínimo es 1 y el saldo máximo es 100.000, se utiliza una función de asociación que divide el valor en 10 rangos, 1-10.000, 10.001-20.000, y así sucesivamente. La distribución de los valores de la clave de búsqueda es uniforme (ya que cada cajón tiene el mismo número de valores del *saldo* diferente), pero no es aleatorio. Los registros con saldos entre 1 y 10.000 son más comunes que los registros con saldos entre 90.001 y 100.000. Como resultado, la distribución de los registros no es uniforme: algunos cajones reciben más registros que otros. Si la función tiene una distribución aleatoria, incluso si hubiera estas correlaciones en las claves de búsqueda, la aleatoriedad de la distribución hará que todos los cajones tengan más o menos el mismo número de registros, siempre que cada clave de búsqueda aparezca sólo en una pequeña fracción de registros. (Si una única clave de búsqueda aparece en una gran fracción de registros, el cajón que la contiene probablemente tenga más registros que otros cajones, independientemente de la función de asociación usada.)

Las funciones de asociación típicas realizan cálculos sobre la representación binaria interna de la máquina para los caracteres de la clave de búsqueda. Una función de asociación simple de este tipo, en primer lugar, calcula la suma de las representaciones binarias de los caracteres de una clave y, posteriormente, devuelve la suma módulo al número de cajones. En la Figura 12.21 se muestra la aplicación de este esquema, utilizando 10 cajones, para el archivo *cuenta*, bajo la suposición de que la letra i-ésima del alfabeto está representada por el número entero i.

Las funciones de asociación requieren un diseño cuidadoso. Una mala función de asociación podría provocar que una búsqueda tome un tiempo proporcional al número de claves de búsqueda en el archivo. Una función bien diseñada en un caso medio de búsqueda toma un tiempo constante (pequeño), independiente del número de claves búsqueda en el archivo.

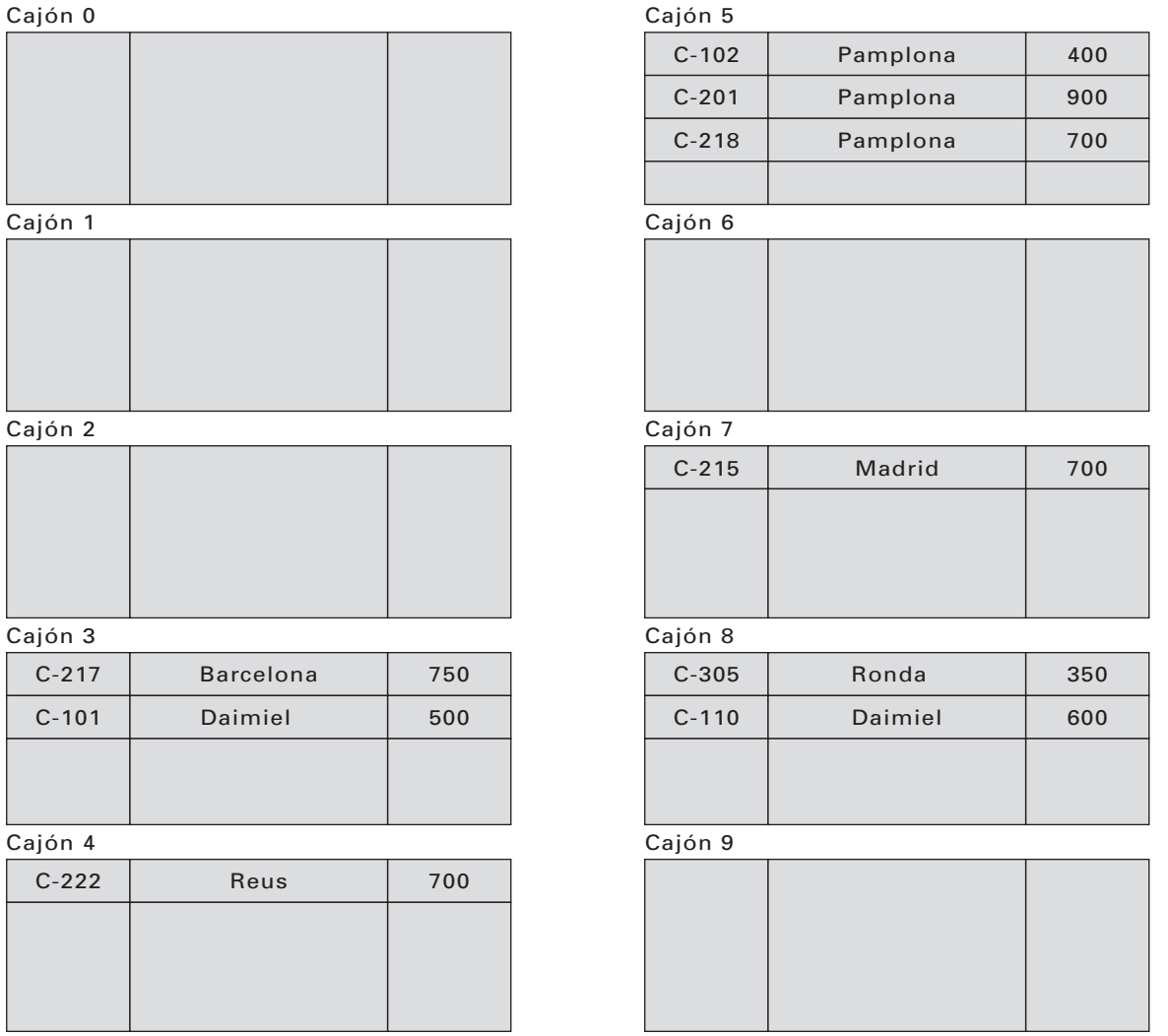

FIGURA 12.21. Organización asociativa del archivo *cuenta* utilizando *nombre-sucursal* como clave.

12.5.1.2. Gestión de desbordamientos de cajones

Hasta ahora se ha asumido que cuando se inserta un registro, el cajón al que se asigna tiene espacio para almacenar el registro. Si el cajón no tiene suficiente espacio, sucede lo que se denomina **desbordamiento de cajones**. Los desbordamientos de cajones pueden ocurrir por varias razones:

- **Cajones insuficientes.** El número de cajones, que se denota con n_B, debe ser escogido tal que $n_C > n_r / f_r$, donde n_r denota el número total de registros que serán almacenados, y f_r denota el número de registros que se colocarán en un cajón. Esta designación, por supuesto, está bajo la suposición de que se conoce el número total de registros cuando se define la función de asociación.
- **Atasco.** Algunos cajones tienen asignados más registros que otros, por lo que un cajón se podría desbordar incluso cuando otros cajones tienen todavía espacio. Esta situación se denomina *atasco* de cajones. El atasco puede ocurrir por dos razones:
  1. Varios registros podrían tener la misma clave de búsqueda.
  2. La función de asociación elegida podría producir una distribución no uniforme de las claves de búsqueda.

Para que la probabilidad de desbordamiento de cajones se reduzca, se escoge el número de cajones igual a $(n_r / f_r)*(1 + d)$, donde d es un factor normalmente con un valor cercano a 0,2. Se pierde algo de espacio: alrededor del 20 por ciento del espacio en los cajones estará vacío. Pero el beneficio es que la probabilidad de desbordamiento se reduce.

A pesar de la asignación de unos pocos más de cajones de los requeridos, el desbordamiento de cajones todavía puede suceder. Trataremos el desbordamiento de cajo-

nes utilizando **cajones de desbordamiento**. Si un registro se tiene que insertar en un cajón *c* y *c* está ya lleno, se proporcionará un cajón de desbordamiento para *c* y el registro se insertará en el cajón de desbordamiento. Si el cajón de desbordamiento está también lleno, se proporcionará otro cajón de desbordamiento, y así sucesivamente. Todos los cajones de desbordamiento de un cajón determinado están encadenados juntos en una lista enlazada, como se muestra en la Figura 12.22. El tratamiento del desbordamiento utilizando una lista enlazada se denomina **cadena de desbordamiento**.

Se debe variar un poco el algoritmo de búsqueda para tratar la cadena de desbordamiento. Como se dijo antes, la función de asociación se emplea sobre la clave de búsqueda para identificar un cajón *c*. Luego se examinan todos los registros en el cajón *c* para ver si coinciden con la clave búsqueda, como antes. Además, si el cajón *c* tiene cajones de desbordamiento, los registros en todos los cajones de desbordamiento de *c* se tienen que examinar también.

La forma de la estructura asociativa que se acaba de describir se denomina algunas veces **asociación cerrada**. Bajo una aproximación alternativa, llamada **asociación abierta**, se fija el conjunto de cajones y no hay cadenas de desbordamiento. Por el contrario, si un cajón está lleno, los registros se insertan en algún otro cajón del conjunto inicial *C* de cajones. Una política es utilizar el siguiente cajón (en orden cíclico) que tenga espacio; esta política se llama *ensayo lineal*. Se utilizan también otras políticas, tales como el cálculo funciones de asociación adicionales. La asociación abierta se emplea en la construcción de tablas de símbolos para compiladores y ensambladores, aunque es preferible la asociación cerrada para los sistemas de bases de datos. La razón es que el borrado bajo la asociación abierta es costoso. Normalmente, los compiladores y ensambladores realizan solamente operaciones de búsqueda e inserción en sus tablas de símbolos. Sin embargo, en un sistema de bases de datos es importante ser capaz de tratar el borrado tan bien como la inserción. Así, la asociatividad abierta es un aspecto de menor importancia en la implementación de bases de datos.

Un inconveniente importante relativo a la forma de asociación que se ha descrito es que la función de asociación se debe elegir cuando se implementa el sistema y no se puede cambiar fácilmente después si el archivo que se está indexando aumenta o disminuye. Ya que la función *h* asigna valores de la clave búsqueda a un conjunto fijo *C* de direcciones de cajón, emplearemos más espacio si *C* fue concebido para manejar el futuro crecimiento del archivo. Si *C* es demasiado pequeño, los cajones contienen registros de una gran variedad de valores de la clave búsqueda, pudiendo originar el desbordamiento del cajón. A medida que el archivo aumenta el rendimiento se degrada. Se estudiará más adelante, en el Apartado 12.6, cómo cambiar dinámicamente el número de cajones y la función de asociación.

12.5.2. Índices asociativos

La asociatividad se puede utilizar no solamente para la organización de archivos sino también para la creación de estructuras de índice. Un **índice asociativo** (*hash index*) organiza las claves de búsqueda, con sus punteros asociados, dentro de una estructura de archivo asociativo. Un índice asociativo se construye como se indica a continuación. Primero se aplica una función de asociación sobre la clave de búsqueda para identificar un cajón, luego se almacenan la clave y los punteros asociados en el cajón (o en los cajones de desbordamiento). En la Figura 12.23 se muestra un índice asociativo secundario en el archivo *cuenta* para la clave de búsqueda *número-cuen-*

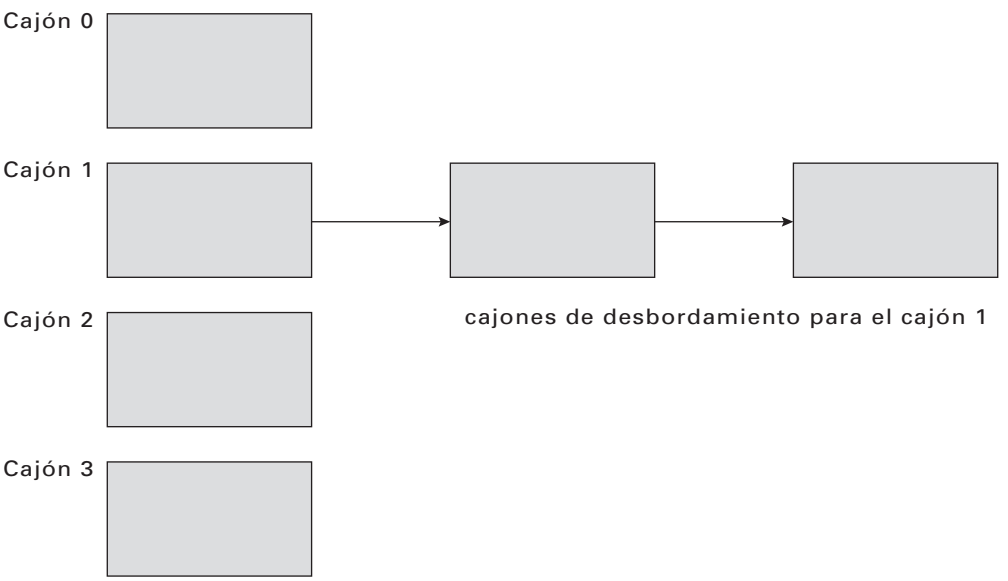

FIGURA 12.22. Cadena de desbordamiento en una estructura asociativa.

ta. La función de asociación utilizada calcula la suma de los dígitos del número de cuenta módulo siete. El índice asociativo tiene siete cajones, cada uno de tamaño dos (los índices realistas tendrían, por supuesto, tamaños de los cajones más grandes). Uno de los cajones tiene tres claves asignadas a él, por lo que tiene un cajón de desbordamiento. En este ejemplo, *número-cuenta* es una clave primaria para *cuenta*, así que cada clave de búsqueda tiene solamente un puntero asociado. En general se pueden asociar múltiples punteros con cada clave.

Se usa el término *índice asociativo* para denotar las estructuras de archivo asociativo, así como los índices secundarios asociativos. Estrictamente hablando, los índices asociativos son sólo estructuras de índices secundarios. Un índice asociativo nunca necesita una estructura de índice primario, ya que si un archivo está organizado utilizando asociatividad, no hay necesidad de una estructura de índice asociativo separada. Sin embargo, ya que la organización de archivos asociativos proporciona el mismo acceso directo a registros que se proporciona con la indexación, se pretende que la organización de un archivo mediante asociación también tenga un índice primario asociativo virtual en él.

12.6. ASOCIACIÓN DINÁMICA

Como se ha visto, la necesidad de fijar el conjunto *C* de direcciones de cajón presenta un problema serio con la técnica de asociación estática vista en el Apartado anterior. La mayoría de las bases de datos crecen con el tiempo. Si se va a utilizar la asociación estática para estas bases de datos, tenemos tres clases de opciones:

1. Elegir una función de asociación basada en el tamaño actual del archivo. Esta opción produci-

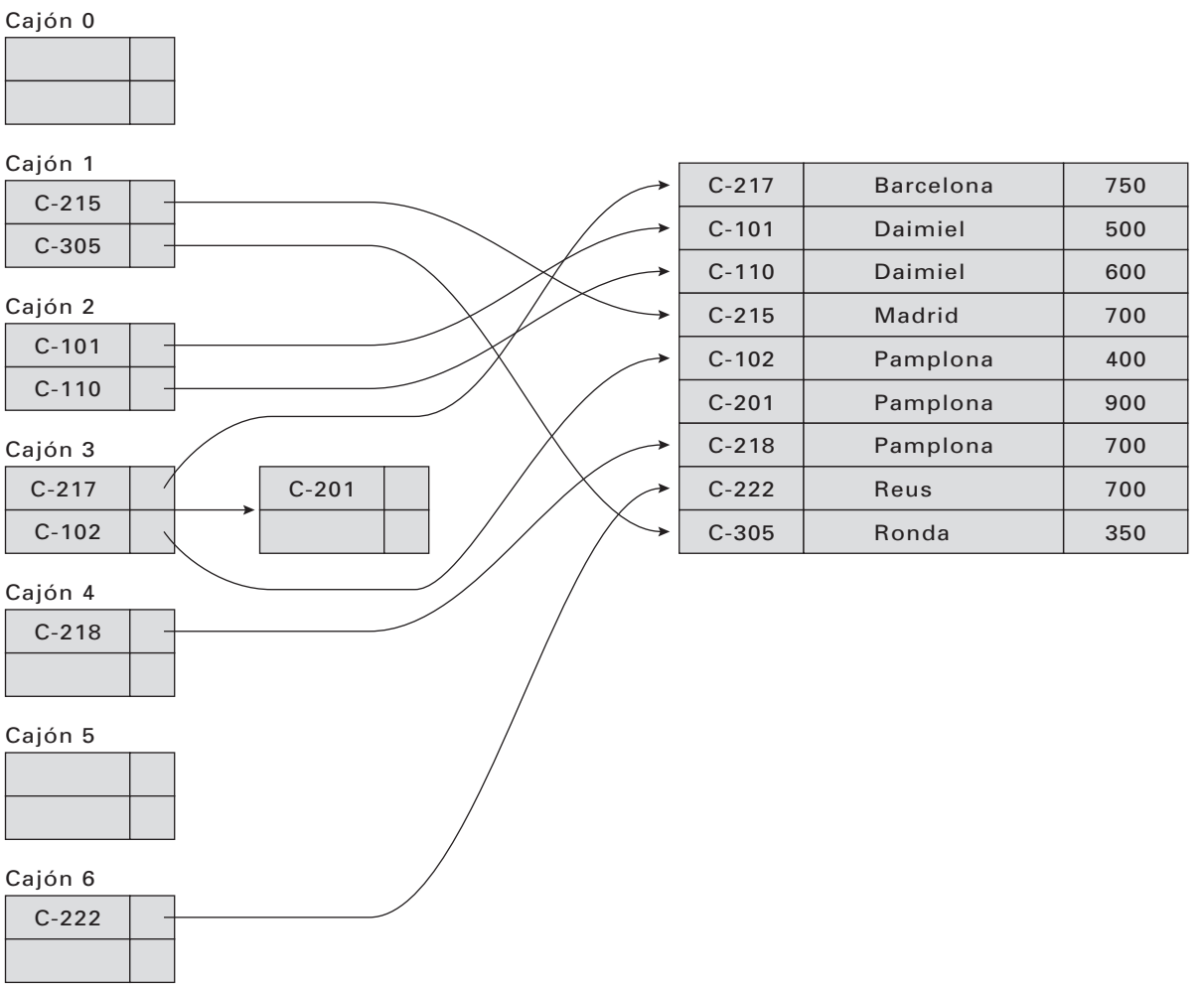

FIGURA 12.23. Índice asociativo de la clave de búsqueda *número-cuenta* del archivo *cuenta*.

rá una degradación del rendimiento a medida que la base de datos crezca.

2. Elegir una función de asociación basada en el tamaño previsto del archivo con relación a un punto determinado del futuro. Aunque se evite la degradación del rendimiento, inicialmente puede que se pierda una cantidad de espacio significante.
3. Reorganizar periódicamente la estructura asociativa en respuesta al crecimiento del archivo. Esta reorganización implica elegir una nueva función de asociación, volviendo a calcular la función de asociación de cada registro en el archivo y generando nuevas asignaciones de los cajones. Esta reorganización es una operación masiva que requiere mucho tiempo. Además, es necesario prohibir el acceso al archivo durante la reorganización.

Algunas técnicas de **asociación dinámica** permiten modificar la función de asociación dinámicamente para acomodarse al aumento o disminución de la base de datos. Describiremos una forma de asociación dinámica, llamada **asociación extensible**. Las notas bibliográficas proporcionan referencias a otras formas de asociatividad dinámica.

12.6.1. Estructura de datos

La asociación extensible hace frente a los cambios del tamaño de la base de datos dividiendo y fusionando los cajones a medida que la base de datos aumenta o disminuye. Como resultado se conserva eficazmente el espacio. Por otra parte, puesto que la reorganización se realiza sobre un cajón cada vez, la degradación del rendimiento resultante es aceptablemente baja.

Con la asociación extensible se elige una función de asociación h con las propiedades deseadas de uniformidad y aleatoriedad. Sin embargo, esta función de asociación genera valores dentro de un rango relativamente amplio, llamado, enteros binarios de b bits. Un valor normal de b es 32.

No se crea un cajón para cada valor de la función de asociación. De hecho, 2^{32} está por encima de 4 billones y no sería razonable crear tantos cajones salvo para las mayores bases de datos. Por el contrario, se crean tantos cajones bajo demanda, esto es, tantos como registros haya insertados en el archivo. Inicialmente no se utiliza el total de b bits del valor de la función de asociación. En cualquier caso, empleamos i bits, donde $0 \leq i \leq b$. Estos i bits son utilizados como desplazamiento en una tabla adicional con las direcciones de los cajones. El valor de i aumenta o disminuye con el tamaño de la base de datos.

En la Figura 12.24 se muestra una estructura general de asociación extensible. La i que aparece en la figura encima de la tabla de direcciones de los cajones indica que se requieren i bits de la función de asociación $h(K)$ para determinar el cajón apropiado para K. Obviamente, este número cambiará a medida que el archivo aumente. Aunque se requieren i bits para encontrar la entrada correcta en la tabla de direcciones de los cajones, algunas entradas consecutivas de la tabla podrían apuntar al mismo cajón. Todas estas entradas tendrán un prefijo común del valor de la función de asociación, aunque la longitud de este prefijo podría ser menor que i. Por lo tanto, se asocia con cada cajón un número entero que proporciona la longitud del prefijo común del valor de la función de asociación. En la Figura 12.24, el entero asociado con el cajón j aparece como i_j. El

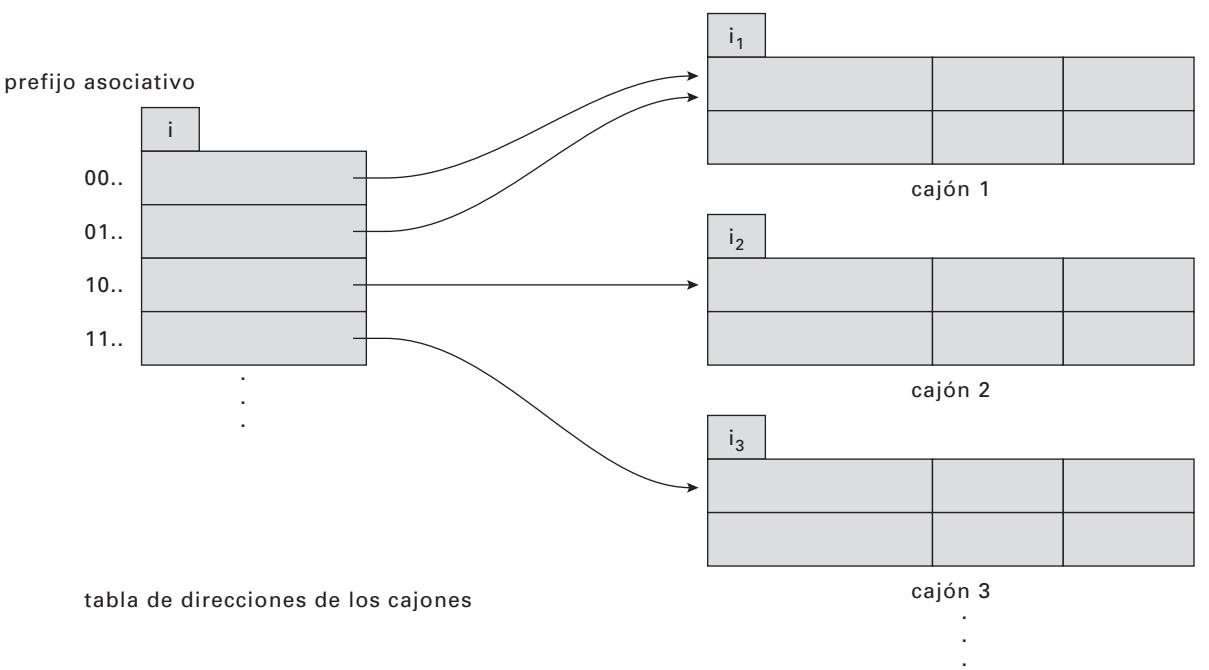

FIGURA 12.24. Estructura asociativa general extensible.

número de entradas en la tabla de direcciones de cajones que apuntan al cajón *j* es:

$$2^{(i - i_j)}$$

12.6.2. Consultas y actualizaciones

Ahora se verá cómo realizar la búsqueda, la inserción y el borrado en una estructura asociativa extensible.

Para localizar el cajón que contiene el valor de la clave de búsqueda K_l, se toman los primeros *i* bits más significativos de $h(K_l)$, se busca la entrada de la tabla que corresponda a esta cadena de bits, y se sigue el puntero del cajón en la entrada de la tabla.

Para insertar un registro con un valor de la clave de búsqueda K_l se sigue el mismo procedimiento de búsqueda que antes, llegando a algún cajón *j*. Si hay sitio en el cajón se inserta el registro en el cajón. Si por el contrario el cajón está lleno, hay que dividir el cajón y redistribuir los registros actuales más uno nuevo. Para dividir el cajón, primero hay que determinar del valor de la función de asociación por si fuera necesario incrementar el número de bits que se están usando.

- Si $i = i_j$, entonces solamente una entrada en la tabla de direcciones de los cajones apunta al cajón *j*. Por tanto, es necesario incrementar el tamaño de la tabla de direcciones de los cajones para incluir los punteros a los dos cajones que resultan de la división del cajón *j*. Esto se hace considerando un bit adicional en el valor de la función de asociación. Luego se incrementa el valor de *i* en uno, duplicando el tamaño de la tabla de direcciones de cajones. Cada entrada se sustituye por dos entradas, cada una de las cuales con el mismo puntero que la entrada original. Ahora dos entradas en la tabla de direcciones de cajones apuntan al cajón *j*. Así pues, se asigna un nuevo cajón (cajón *z*) y hacemos que la segunda entrada apunte al nuevo cajón. Se pone i_j e i_z a *i*. A continuación se vuelve a calcular la función de asociación para cada registro del cajón *j* y, dependiendo de los primeros *i* bits (recuérdese que se ha añadido uno a *i*), se mantiene en el cajón *j* o se coloca en el cajón recién creado.

  Ahora se vuelve a intentar la inserción del nuevo registro. Normalmente el intento tiene éxito. Sin embargo, si todos los registros del cajón *j*, así como el nuevo registro, tienen el mismo prefijo del valor de la función de asociación, será necesario dividir el cajón de nuevo, ya que todos los registros en el cajón *j* y el nuevo registro tienen asignados el mismo cajón. Si la función de asociación se eligió cuidadosamente, es poco probable que una simple inserción provoque que un cajón se divida más de una vez, a menos que haya un gran número de registros con la misma clave de búsqueda. Si todos los registros en el cajón *j* tienen el mismo valor de la clave de búsqueda, ningún número de divisiones servirá. En estos casos se usan cajones de desbordamiento para almacenar los registros, como en la asociación estática.

- Si $i > i_j$, entonces más de una entrada en la tabla de direcciones de cajones apunta al cajón *j*. Así, se puede dividir el cajón *j* sin incrementar el tamaño de la tabla de direcciones. Obsérvese que todas las entradas que apuntan al cajón *j* corresponden a prefijos del valor de la función de asociación que tienen el mismo valor en los i_j bits más a la izquierda. Se asigna un nuevo cajón (cajón *z*) y se pone i_j e i_z al valor que resulta de añadir uno al valor i_j original. A continuación, es necesario ajustar las entradas en la tabla de direcciones de cajones que anteriormente apuntaban al cajón *j*. (Nótese que con el nuevo valor de i_j no todas las entradas corresponden a prefijos del valor de la función de asociación que tienen el mismo valor en los i_j bits más a la izquierda). La primera mitad de todas las entradas se dejan como estaban (apuntando al cajón *j*) y el resto de entradas se ponen apuntando al cajón recién creado (cajón *z*). Por último, como en el caso anterior, se vuelve a calcular la función de asociación para cada registro en el cajón *j* y se colocan o bien en el cajón *j* o bien en el cajón *z* recién creado.

  Luego se vuelve a intentar la inserción. En el caso poco probable de que vuelva a fallar, se aplica uno de los dos casos, $i = i_j$ o $i > i_j$, según sea lo apropiado.

Nótese que en ambos casos sólo se necesita recalcular la función de asociación en los registros del cajón *j*.

Para borrar un registro con valor de la clave de búsqueda K_l se sigue el mismo procedimiento de búsqueda anterior, finalizando en algún cajón, llamémosle *j*. Se borran ambos, el registro del archivo y la clave de búsqueda del cajón. El cajón también se elimina si se queda vacío. Nótese que en este momento, varios cajones se pueden fusionar, reduciendo el tamaño de la tabla de direcciones de cajones a la mitad. El procedimiento para decidir cuándo y cómo fusionar cajones se deja como un ejercicio a realizar. Las condiciones bajo la que la tabla de direcciones de cajones se puede reducir de tamaño también se dejan como ejercicio. A diferencia de la fusión de cajones, el cambio de tamaño de la tabla de direcciones de cajones es una operación muy costosa si la tabla es grande. Por tanto, sólo sería aconsejable reducir el tamaño de la tabla de direcciones de cajones si el número de cajones se reduce considerablemente.

El ejemplo del archivo *cuenta* en la Figura 12.25 ilustra la operación de inserción. Los valores de la función de asociación de 32 bits para *nombre-sucursal* se mues-

C-217	Barcelona	750
C-101	Daimiel	500
C-110	Daimiel	600
C-215	Madrid	700
C-102	Pamplona	400
C-201	Pamplona	900
C-218	Pamplona	700
C-222	Reus	700
C-305	Ronda	350

FIGURA 12.25. Archivo de ejemplo *cuenta*.

tran en la Figura 12.26. Se asume que inicialmente el archivo está vacío, como se muestra en la Figura 12.27. Insertaremos los registros de uno en uno. Para mostrar todas las características de la asociación extensible se empleará una estructura pequeña y se hará la suposición no realista de que un cajón sólo puede contener dos registros.

Vamos a insertar el registro (C-217, Barcelona, 750). La tabla de direcciones de cajones contiene un puntero al único cajón donde se inserta el registro. A continuación, insertamos el registro (C-101, Daimiel, 500). Este registro también se inserta en el único cajón de la estructura.

Cuando se intenta insertar el siguiente registro (C-110, Daimiel, 600), nos encontramos con que el cajón está lleno. Ya que $i = i_0$, es necesario incrementar el número de bits que se toman del valor de la función de asociación. Ahora se utiliza un bit, permitiendo $2^1 = 2$ cajones. Este incremento en el número de bits necesarios duplica el tamaño de la tabla de direcciones de cajones a dos entradas. Se continúa con la división del cajón, colocando en el nuevo cajón aquellos registros cuya clave de búsqueda tiene un valor de la función de asociación que comienza por 1 y dejando el resto de registros en el cajón original. En la Figura 12.28 se muestra el estado de la estructura después de la división.

A continuación insertamos (C-215, Madrid, 700). Ya que el primer bit de h(Madrid) es 1, se tiene que insertar este registro en el cajón apuntado por la entrada «1» en la tabla de direcciones de cajones. Una vez más, nos encontramos con el cajón lleno e $i = i_1$. Se incrementa el número de bits que se usan del valor de la función de asociación a dos. Este incremento en el número de bits necesarios duplica el tamaño de la tabla de direcciones de cajones a cuatro entradas, como se muestra en la Figura 12.29. Como el cajón con el prefijo 0 del valor de la función de asociación de la Figura 12.28 no se dividió, las dos entradas de la tabla de direcciones 00 y 01 apuntan a este cajón.

Para cada registro en el cajón de la Figura 12.28 con prefijo 1 del valor de la función de asociación (el cajón que se dividió) se examinan los dos primeros bits del valor de la función de asociación para determinar qué cajón de la nueva estructura le corresponde.

Proseguimos insertando el registro (C-102, Pamplona, 400), el cual se aloja en el mismo cajón que Madrid. La siguiente inserción, la de (C-201, Pamplona, 900), provoca un desbordamiento en un cajón, produciendo el incremento del número de bits y la duplicación del tamaño de la tabla de direcciones de cajones. La inserción del tercer registro de Pamplona (C-218, Pamplona, 700) produce otro desbordamiento. Sin embargo, este desbordamiento no se puede resolver incrementando el número de bits, ya que los tres registros tienen

nombre-sucursal	*h(nombre-sucursal)*
Barcelona	0010 1101 1111 1011 0010 1100 0011 0000
Daimiel	1010 0011 1010 0000 1100 0110 1001 1111
Madrid	1100 0111 1110 1101 1011 1111 0011 1010
Pamplona	1111 0001 0010 0100 1001 0011 0110 1101
Reus	0011 0101 1010 0110 1100 1001 1110 1011
Ronda	1101 1000 0011 1111 1001 1100 0000 0001

FIGURA 12.26. Función de asociación para *nombre-sucursal*.

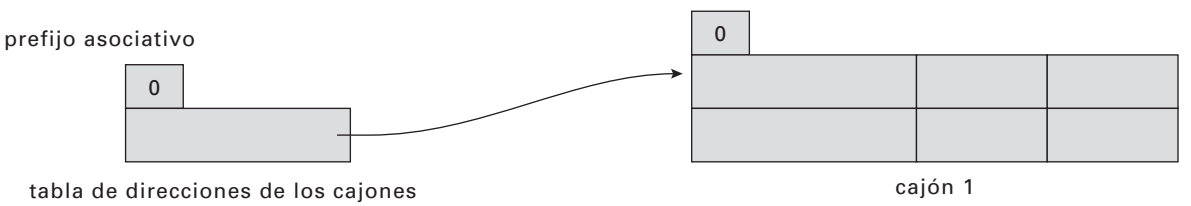

FIGURA 12.27. Estructura asociativa extensible inicial.

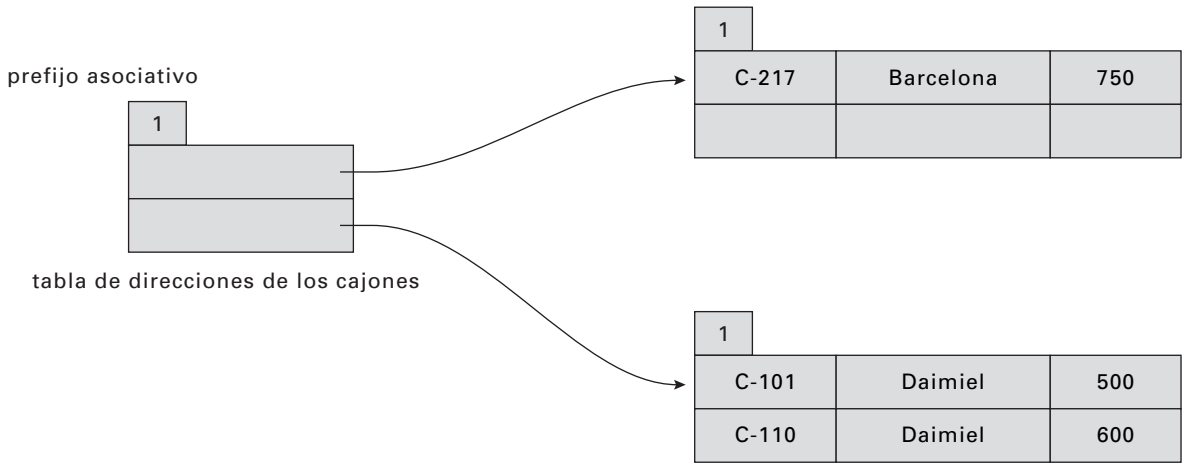

FIGURA 12.28. Estructura asociativa después de tres inserciones.

exactamente el mismo valor de la función de asociación. Por tanto, se utiliza un cajón de desbordamiento, como se muestra en la Figura 12.30.

Se continúa de esta manera hasta que se insertan todos los registros del archivo *cuenta* de la Figura 12.24. La estructura resultante se muestra en la Figura 12.31.

12.6.3. Comparaciones con otros esquemas

Examinemos a continuación las ventajas e inconvenientes de la asociación extensible frente a otros esquemas ya discutidos. La ventaja principal de la asociación extensible es que el rendimiento no se degrada según crece el archivo. Además de esto, el espacio adicional requerido es mínimo. Aunque la tabla de direcciones de cajones provoca un gasto adicional, sólo contiene un puntero por cada valor de la función de asociación con la longitud del prefijo actual. De esta manera el tamaño de la tabla es pequeño. El principal ahorro de espacio de la asociación extensible sobre otras formas de asociación es que no es necesario reservar cajones para un futuro crecimiento; en vez de ello se pueden asignar los cajones de manera dinámica.

Un inconveniente de la asociación extensible es que la búsqueda implica un nivel adicional de indirección,

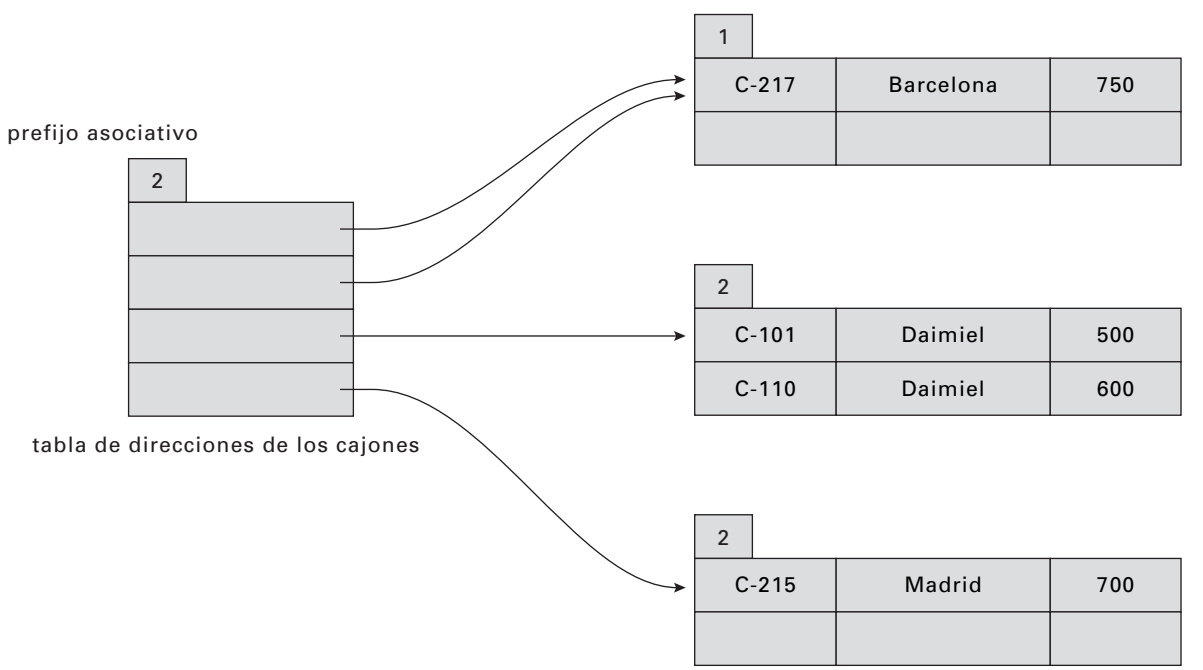

FIGURA 12.29. Estructura asociativa después de cuatro inserciones.

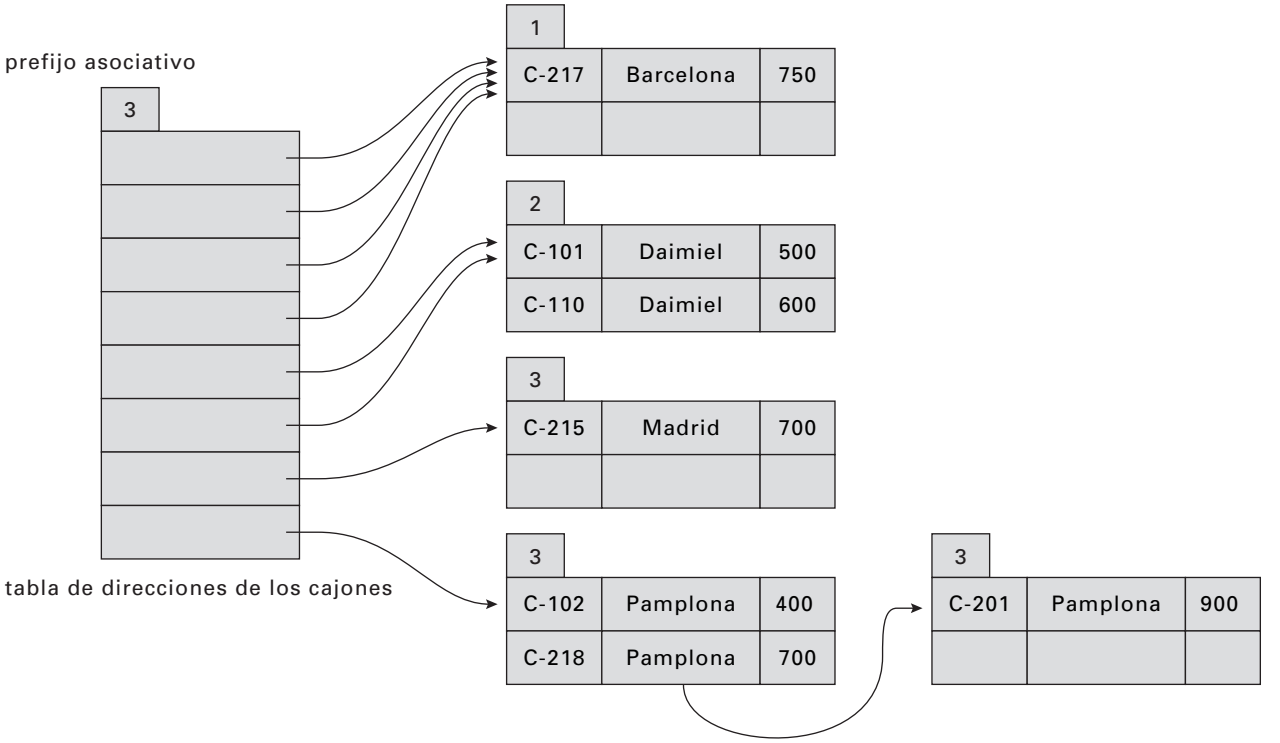

FIGURA 12.30. Estructura asociativa después de siete inserciones.

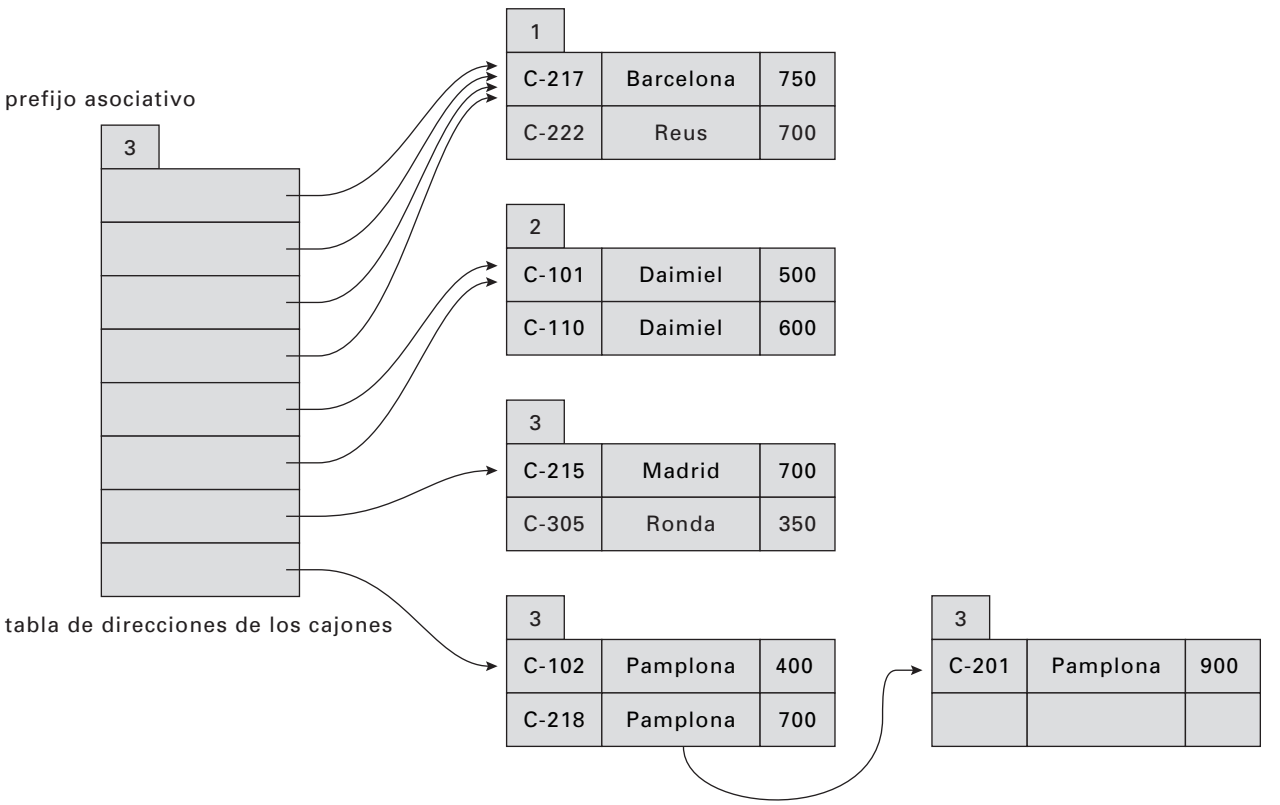

FIGURA 12.31. Estructura asociativa extensible para el archivo *cuenta*.

ya que se debe acceder a la tabla de direcciones de los cajones antes que acceder al propio cajón. Esta referencia extra tiene una repercusión menor en el rendimiento. Aunque las estructuras asociativas que se discutieron con anterioridad no tienen este nivel extra de indirección, pierden su leve ventaja en el rendimiento cuando se llenan.

Por tanto, la asociación extensible se muestra como un técnica muy atractiva teniendo en cuenta que se acepta la complejidad añadida en su implementación. En las notas bibliográficas se proporcionan descripciones más detalladas sobre la implementación de la asociación extensible. Las notas bibliográficas también tienen referencias a otra forma de asociación dinámica llamada **asociación lineal**, la cual evita el nivel extra de indirección asociado con la asociación extensible y el posible coste de más cajones de desbordamiento.

12.7. COMPARACIÓN DE LA INDEXACIÓN ORDENADA Y LA ASOCIACIÓN

Se han visto varios esquemas de indexación ordenada y varios esquemas de asociación. Se pueden organizar los archivos de registros como archivos ordenados, utilizando una organización de índice secuencial o usando organizaciones de árbol B^+. Alternativamente se pueden organizar los archivos usando asociación. Finalmente, podemos organizar los archivos como montículos, donde los registros no están ordenados de ninguna manera en particular.

Cada esquema tiene sus ventajas dependiendo de la situación. Un implementador de un sistema de bases de datos podría proporcionar muchos esquemas, dejando al diseñador de la base de datos la decisión final de qué esquemas utilizar. Sin embargo, esta aproximación requiere que el implementador escriba más código, aumentando así el coste del sistema y el espacio que éste ocupa. Por tanto, la mayoría de los sistemas usan solamente unos pocos o sólo una forma de organización asociativa de archivos o índices asociativos.

Para hacer una sabia elección, el implementador o el diseñador de la base de datos debe considerar los siguientes aspectos:

- ¿Es aceptable el coste de una reorganización periódica del índice o de una estructura asociativa?
- ¿Cuál es la frecuencia relativa de las inserciones y borrados?
- ¿Es deseable mejorar el tiempo medio de acceso a expensas de incrementar el tiempo de acceso en el peor de los casos?
- ¿Qué tipos de consultas se supone que van a realizar los usuarios?

De estos puntos ya se han examinado los tres primeros, comenzando con la revisión de las ventajas relativas de las distintas técnicas de indexación y otra vez en la discusión de las técnicas de asociación. El cuarto punto, el tipo esperado de la consulta, es un aspecto crítico para la elección entre la indexación ordenada o la asociación.

Si la mayoría de las consultas son de la forma:

select $A_1, A_2, ..., A_n$
from r
where $A_i = c$

entonces, para procesar esta consulta, el sistema realizará una búsqueda en un índice ordenado o en una estructura asociativa de un atributo A_i con el valor c. Para este tipo de consultas es preferible un esquema asociativo. Una búsqueda en un índice ordenado requiere un tiempo proporcional al logaritmo del número de valores para A_i en r. Sin embargo, en una estructura asociativa, el tiempo medio de una búsqueda es una constante que no depende del tamaño de la base de datos. La única ventaja de un índice sobre una estructura asociativa con este tipo de consulta es que el tiempo de una búsqueda en el peor de los casos es proporcional al logaritmo del número de valores para A_i en r. Por el contrario, si se utiliza una estructura asociativa, el tiempo de una búsqueda en el peor de los casos es proporcional al número de valores para A_i en r. Sin embargo, al ser poco probable el peor caso de búsqueda (mayor tiempo de búsqueda) con la asociación, es preferible usar en este caso una estructura asociativa.

Las técnicas de índices ordenados son preferibles a las estructuras asociativas en los casos donde la consulta especifica un rango de valores. Estas consultas tienen el siguiente aspecto:

select $A_1, A_2, ..., A_n$
from r
where $A_i \leq c_2$ **and** $A_i \geq c_1$

En otras palabras, la consulta anterior encuentra todos los registros A_i con valores entre c_1 y c_2.

Consideremos cómo procesar esta consulta usando un índice ordenado. Primero se realiza una búsqueda en el valor c_1. Una vez que se ha encontrado un cajón que contiene el valor c_1, se sigue la cadena de punteros en el índice en orden para leer el siguiente cajón y continuamos de esta manera hasta encontrar c_2.

Si en vez de un índice ordenado se tiene una estructura asociativa, se puede realizar una búsqueda en c_1 y localizar el correspondiente cajón, pero en general no es fácil determinar el siguiente cajón que se tiene que examinar. La dificultad surge porque una buena función de asociación asigna valores aleatoriamente a los cajones. Por tanto, no existe la noción del «siguiente cajón en el orden». La razón por la que no se pueden encade-

nar cajones juntos según cierto orden en A_i es que cada cajón tiene asignado muchos valores de la clave de búsqueda. Ya que los valores están diseminados aleatoriamente según la función de asociación, es muy probable que el rango de valores esté esparcido a través de muchos cajones o tal vez en todos. Por esta razón se tienen que leer todos los cajones para encontrar las claves de búsqueda requeridas.

Normalmente se usa la indexación ordenada, a menos que se sepa de antemano que las consultas sobre un rango de valores van a ser poco frecuentes, en cuyo caso se utiliza la asociación. Las organizaciones asociativas son particularmente útiles para archivos temporales creados durante el procesamiento de consultas, siempre que se realicen búsquedas basadas en un valor de la clave, pero no consultas de rangos.

12.8. DEFINICIÓN DE ÍNDICES EN SQL

La norma SQL no proporciona al usuario o administrador de la base de datos ninguna manera de controlar qué índices se crean y se mantienen por el sistema de base de datos. Los índices no se necesitan para la corrección, ya que son estructuras de datos redundantes. Sin embargo, los índices son importantes para el procesamiento eficiente de las transacciones, incluyendo las transacciones de actualización y consulta. Los índices son también importantes para un cumplimiento eficiente de las ligaduras de integridad. Por ejemplo, las implementaciones típicas obligan a declarar una clave (Capítulo 6) mediante la creación de un índice con la clave declarada como la clave de búsqueda del índice.

En principio, un sistema de base de datos puede decidir automáticamente qué índices crear. Sin embargo, debido al coste en espacio de los índices, así como el efecto de los índices en el procesamiento de actualizaciones, no es fácil hacer una elección apropiada automáticamente sobre qué índices mantener. Por este motivo, la mayoría de las implementaciones de SQL proporcionan al programador control sobre la creación y eliminación de índices vía órdenes del lenguaje de definición de datos.

A continuación se ilustrará las sintaxis de estas órdenes. Aunque la sintaxis que se muestra se usa ampliamente y está soportada en muchos sistemas de bases de datos, no es parte de la norma SQL:1999. Las normas SQL (hasta SQL:1999, al menos) no dan soporte al control del esquema físico de la base de datos y aquí nos limitaremos al esquema lógico de la base de datos.

Un índice se crea mediante la orden **create index**, la cual tiene la forma

create index <nombre-índice> **on** <nombre-relación> (<lista-atributos>)

lista-atributos es la lista de atributos de la relación que constituye la clave de búsqueda del índice.

Para definir un índice llamado *índice-s* de la relación *sucursal* con la clave de búsqueda *nombre-sucursal*, se escribe

create index *índice-s* **on** *sucursal* (*nombre-sucursal*)

Si deseamos declarar que la clave de búsqueda es una clave candidata, hay que añadir el atributo **unique** a la definición del índice. Con esto, la orden

create unique index *índice-s* **on** *sucursal* (*nombre-sucursal*)

declara *nombre-sucursal* como una clave candidata de *sucursal*. Si cuando se introduce la orden **create unique index**, *nombre-sucursal* no es una clave candidata, se mostrará un mensaje de error y el intento de crear un índice fallará. Por otro lado, si el intento de crear el índice ha tenido éxito, cualquier intento de insertar una tupla que viole la declaración de clave fallará. Hay que observar que el carácter **unique** es redundante si la declaración **unique** de SQL estándar se soporta en el sistema de base de datos.

El nombre de índice especificado con el índice se necesita para hacer posible la eliminación (*drop*) de índices. La orden **drop index** tiene la forma

drop index <nombre-índice>

12.9. ACCESOS MULTICLAVE

Hasta ahora se ha asumido implícitamente que se utiliza solamente un índice (o tabla asociativa) para procesar una consulta en una relación. Sin embargo, para ciertos tipos de consultas es ventajoso el uso de múltiples índices si éstos existen.

12.9.1. Uso de varios índices de clave única

Asumamos que el archivo *cuenta* tiene dos índices: uno para el *nombre-sucursal* y otro para *saldo*. Consideremos la consulta : «Encontrar todos los números de cuen-

ta de la sucursal Pamplona con saldos igual a 1.000 €». Se escribe

select *número-préstamo*
from *cuenta*
where *nombre-sucursal* = «Pamplona» **and**
saldo = 1000

Hay tres estrategias posibles para procesar esta consulta:

1. Usar el índice en *nombre-sucursal* para encontrar todos los registros pertenecientes a la sucursal de Pamplona. Luego se examinan estos registros para ver si *saldo* = 1.000.
2. Usar el índice en *saldo* para encontrar todos los registros pertenecientes a cuentas con saldos de 1.000 €. Luego se examinan estos registros para ver si *nombre-sucursal* = «Pamplona.»
3. Usar el índice en *nombre-sucursal* para encontrar *punteros* a registros pertenecientes a la sucursal Pamplona. Y también usar el índice en *saldo* para encontrar los punteros a todos los registros pertenecientes a cuentas con un saldo de 1.000. 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 1.000 €.

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

- Hay muchos registros pertenecientes a la sucursal Pamplona.
- Hay muchos registros pertenecientes a cuentas con un saldo de 1.000 €.
- Hay solamente unos cuantos registros pertenecientes a *ambos,* a la sucursal Pamplona y a las cuentas con un saldo de 1.000 €.

Si estas condiciones ocurrieran, se tendrían que examinar un gran número de punteros para producir un resultado pequeño. La estructura de índices denominada «índice de mapas de bits» acelera significativamente la operación de inserción usada en la tercera estrategia. Los índices de mapas de bits se describen en el Apartado 12.9.4.

12.9.2. Índices sobre varias claves

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

El empleo de una estructura de índice ordenado con múltiples atributos tiene algunas deficiencias. Como ilustración considérese la consulta

select *número-préstamo*
from *cuenta*
where *nombre-sucursal* < «Pamplona» **and**
saldo = 1000

Se puede responder a esta consulta usando un índice ordenado con la clave de búsqueda (*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 1.000. 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.

La diferencia entre esta consulta y la anterior es que la condición en *nombre-sucursal* es una condición de comparación, en vez de una condición de igualdad.

Para acelerar el procesamiento en general de consultas con varias claves de búsqueda (las cuales pueden implicar una o más operaciones de comparación) se pueden emplear varias estructuras especiales. Se considerará la estructura de *archivos en retícula* en el Apartado 12.9.3. Hay otra estructura, denominada *árbol R*, que también se puede usar para este propósito. El árbol R es una extensión que se usa fundamentalmente con tipos de datos geográficos y se pospone su descripción hasta el Capítulo 23.

12.9.3. Archivos en retícula

En la Figura 12.32 se muestra una parte de un **archivo en retícula** para las claves de búsqueda *nombre-sucursal* y *saldo* en el archivo *cuenta*. 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 para cada atributo de la clave de búsqueda.

Las claves de búsqueda se asignan a las celdas como se describe a continuación. Cada celda en el *array* en retícula tiene un puntero a un cajón que contiene valores de las claves de búsqueda y punteros a los registros. Sólo se muestran en la figura algunos de los cajones y punteros desde las celdas. Para conservar espacio se permite que varios elementos del *array* puedan apuntar al

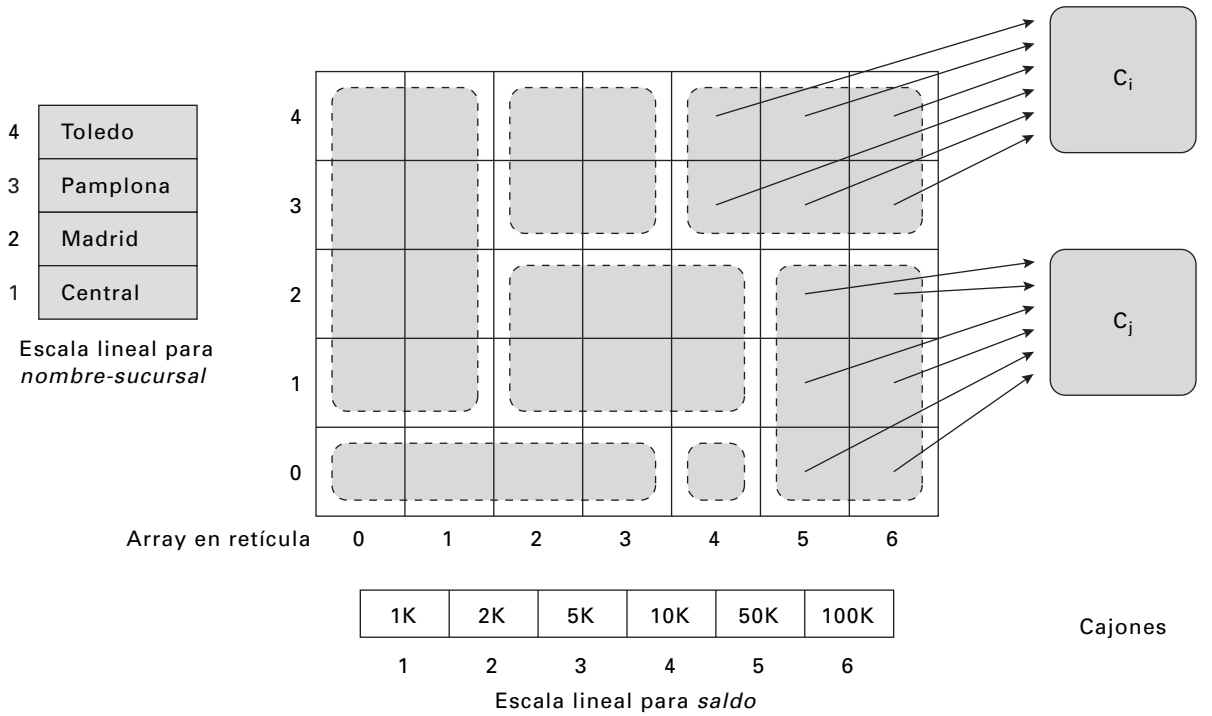

FIGURA 12.32. Archivo de rejilla para las claves *nombre-sucursal* y *saldo* del archivo *cuenta*.

mismo cajón. Los recuadros punteados de la figura señalan las celdas que apuntan al mismo cajón.

Supongamos que se quiere insertar en el índice de archivo en retícula un registro cuyo valor de la clave es («Barcelona», 500.000). Para encontrar la celda asignada a esta clave se localizan por separado la fila y la columna de la celda correspondiente.

Primero se utilizan las escalas lineales en *nombre-sucursal* para localizar la fila de la celda asignada a la clave de búsqueda. Para ello se busca en el *array* para encontrar el menor elemento que es mayor que «Barcelona». En este caso es el primer elemento, así que la fila asignada a la clave de búsqueda es la 0. Si fuera el *i*-ésimo elemento, la clave de búsqueda se asignaría a la fila *i*-1. Si la clave de búsqueda 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 a la clave de búsqueda. En este caso, el saldo 500.000 tiene asignado la columna 6.

Por tanto, el valor de la clave de búsqueda («Barcelona», 500.000) tiene asignado la celda de la fila 0, columna 6. De la misma manera («Daimiel», 60.000) tendría asignada la celda de la fila 1, columna 5. Ambas celdas apuntan al mismo cajón (como se indica con el recuadro punteado), así que en los dos casos los valores de la clave de búsqueda y el puntero al registro están almacenados en el cajón con la etiqueta C_j de la figura.

Para realizar una búsqueda que responda a la consulta de nuestro ejemplo, con la condición de búsqueda

nombre-sucursal < «Pamplona» **and** *saldo* = 1000

buscamos todas las filas con nombres de sucursal menores que «Pamplona», utilizando la escala lineal de *nombre-sucursal*. En este caso, estas filas son la 0, 1 y 2. La fila 3 y posteriores contienen nombres de sucursal mayores o iguales a «Pamplona». De igual modo se obtiene que sólo la columna 1 puede tener un *saldo* de 1.000. En este caso sólo la columna 1 satisface esta condición. 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 todas las entradas de los cajones apuntados por estas tres celdas. En este caso, solamente hay dos cajones, ya que dos de las celdas apuntan al mismo cajón, como se indica con los recuadros punteados de la figura. Los cajones podrían contener algunas claves de búsqueda que no satisfagan la condición requerida, de manera que cada clave de búsqueda del cajón se debe comprobar de nuevo para averiguar si satisface la condición o no. De cualquier modo, solamente hay que examinar un pequeño número de cajones 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 cajón —llamémosle *A*— queda lleno y se tiene que insertar una entrada en él, se asigna un cajón adicional *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*. Las entradas en el cajón *A* y la nueva entrada se redistribuyen entre *A* y *B* basán-

dose en las celdas que tengan asignados. Si sólo una celda apunta al cajón *A*, *B* se convierte en un cajón de desbordamiento de *A*. Para mejorar el rendimiento en esta situación se tiene que reorganizar el archivo en retícula con un *array* en retícula extendido y escalas lineales extendidas. Este proceso es como la expansión de la tabla de direcciones de los cajones en la asociación extensible y se deja a realizar como ejercicio.

Es conceptualmente sencillo extender la aproximación del archivo en retícula a cualquier número de claves de búsqueda. Si se quiere utilizar una estructura con *n* claves hay que construir un *array* en retícula *n*-dimensional con *n* escalas lineales.

La estructura de retícula es adecuada también para consultas que impliquen una sola clave de búsqueda. Considérese esta consulta:

select *
from *cuenta*
where *nombre-sucursal* = «Pamplona»

La escala lineal de *nombre-sucursal* no dice que únicamente satisfacen esta condición las celdas de la fila 3. Como no hay condición según el *saldo*, se examinan todos los cajones apuntados por las celdas en la fila 3 para encontrar las entradas pertenecientes a Pamplona. De este modo, se puede usar un índice de archivo en retícula con dos claves de búsqueda para responder consultas en cada clave, así como para contestar consultas en ambas claves a la vez. Así un simple índice de archivo en retícula puede hacer el papel de tres índices distintos. Si cada índice se mantuviera por separado, los tres juntos ocuparían más espacio y el coste de su actualización sería mayor.

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

12.9.4. Índices de mapas de bits

Los índices de mapas de bits son un tipo de índices especializado para la consulta sencilla sobre varias claves, aunqeu cada índice de mapas de bits se construya para una única clave.

Para que se usen los índices de mapas de bits, los registros de la relación deben estar numerados secuencialmente, comenzando, digamos, en 0. Dado un número *n* es fácil recuperar el registro con número *n*. Esto es particularmente fácil de conseguir si los registros tienen un tamaño fijo y están asignados en bloques consecutivos de un archivo. El número de registro se puede traducir fácilmente en un número de bloque y en un número que identifica el registro dentro del bloque.

Considérese una relación *r* con un atributo *A* que sólo puede valer un número pequeño (por ejemplo, entre 2 y 20). Por ejemplo, la relación *info-cliente* puede tener un campo *sexo*, que puede tomar sólo los valores m (masculino) o f (femenino). Otro ejemplo podría ser el atributo *nivel-ingresos*, donde los ingresos se han dividido en 5 niveles: *L*1: 0 – 9.999 €, *L*2: 10.000 – 19.999 €, *L*3: 20.000 – 39.999 €, *L*4: 40.000 – 74.999 y *L*5: 75.000 – ∞. Aquí, los datos originales pueden tomar muchos valores, pero un analista de datos debe dividir los valores en un número menor de rangos para simplificar el análisis de los datos.

12.9.4.1. Estructura de los índices de mapas de bits

Un **mapa de bits** es un *array* de bits. En su forma más simple, un **índice de mapas de bits** sobre un atributo *A* de la relación *r* consiste en un mapa de bits para cada valor que pueda tomar *A*. Cada mapa de bits tiene tantos bits como el número de registros de la relación. El *í*-ésimo bit del mapa de bits para el valor v_j se establece en 1 si el registro con número *i* tiene el valor v_j para el atributo *A*. El resto de bits del mapa de bits se establecen a 0.

En nuestro ejemplo, hay un mapa de bits para el valor m y otro para f. El *i*-ésimo bit del mapa de bits para m se establece en 1 si el valor *sexo* del registro con número *i* es m. El resto de bits del mapa de bits para m se establecen en 0. Análogamente, el mapa de bits para f tiene el valor 1 para los bits correspondientes a los registros con el valor f para el campo *sexo*; el resto de bits tienen el valor 0. La Figura 12.33 muestra un ejemplo de índices de mapa de bits para la relación *info-cliente*.

Ahora se considerará cuándo son útiles los mapas de bits. La forma más simple de recuperar todos los registros con el valor m (o el valor f) sería simplemente leer

número de registro	*nombre*	*sexo*	*dirección*	*nivel-ingresos*
0	Juan	m	Pamplona	L1
1	Diana	f	Barcelona	L2
2	María	f	Jaén	L1
3	Pedro	m	Barcelona	L4
4	Katzalin	f	Pamplona	L3

Mapas de bits para *sexo*

m	10010
f	01101

Mapas de bits para *nivel-ingresos*

L1	10100
L2	10100
L3	10100
L4	10100
L5	10100

FIGURA 12.33. Índices de mapas de bits para la relación *info-cliente*.

todos los registros de la relación y seleccionar los registros con el valor m (o f, respectivamente). El índice de mapas de bits no ayuda realmente a acelerar esta selección.

De hecho, los índices de mapas de bits son útiles para las selecciones principalmente cuando hay selecciones bajo varias claves. Supóngase que se crea un índice de mapas de bits sobre el atributo *nivel-ingresos*, que se describió antes, además del índice de mapas de bits para *sexo*.

Considérese ahora una consulta que seleccione mujeres con ingresos en el rango 10.000 – 19.999 €. Esta consulta se puede expresar como $\sigma_{sexo=f \wedge nivel\text{-}ingresos=L2}(r)$. Para evaluar esta selección se busca el valor f en los mapas de bits de *sexo* y el valor *L*2 en los mapas de bits de *nivel-ingresos*, y se realiza la **intersección** (conjunción lógica) de los dos mapas de bits. En otras palabras, se calcula un nuevo mapa de bits donde el bit *i* tiene el valor 1 si el *i*-ésimo bit de los dos mapas de bits es 1, y tiene el valor 0 en caso contrario. En el ejemplo de la Figura 12.33, la intersección del mapa de bits para *sexo* = f (01101) y el mapa de bits para *nivel-ingresos* = *L*2 (01000) da como resultado el mapa de bits 01000.

Dado que el primer atributo puede tomar dos valores y el segundo cinco, se esperaría en media que entre 1 y 10 registros satisfarían la condición combinada de los dos atributos. Si hay más condiciones, la fracción de registros que satisfacen todas las condiciones será probablemente muy pequeña. El sistema puede calcular entonces el resultado de la consulta buscando todos los bits con valor 1 en el mapa de bits resultado de la intersección, y recuperando los registros correspondientes. Si la fracción es grande, la exploración de la relación completa seguiría siendo la alternativa menos costosa.

Otro uso importante de los mapas de bits es contar el número de tuplas que satisfacen una selección dada. Tales consultas son importantes para el análisis de datos. Por ejemplo, si deseamos determinar cuántas mujeres tienen un nivel de ingresos *L*2, se calcula la intersección de los dos mapas de bits y después se cuenta el número de bits que son 1 en la intersección. Así se puede obtener el resultado del mapa de bits sin acceder a la relación.

Los índices de mapas de bits son generalmente muy pequeños comparados con el tamaño real de la relación. Los registros tienen generalmente de decenas a centenas de bytes, mientras que un único bit representa a un registro en el mapa de bits. Así, el espacio ocupado por un único mapa de bits es usualmente menor que el 1 por ciento del espacio ocupado por la relación. Por ejemplo, si el tamaño de registro de una relación dada es 100 bytes, entonces el espacio ocupado por un único mapa de bits sería la octava parte del 1 por ciento del espacio ocupado por la relación. Si un atributo *A* de la relación puede tomar sólo uno de ocho valores, el índice de mapas de bits consistiría en 8 mapas de bits, que juntos ocupan sólo el 1 por ciento del tamaño de la relación.

El borrado de registros crea huecos en la secuencia de registros, ya que el desplazamiento de registros (o de los números de registro) para rellenar los huecos sería excesivamente costoso. Para reconocer los registros borrados se puede almacenar un **mapa de bits de existencia** en el que el bit *i* es 0 si el registro *i* no existe, y 1 en caso contrario. Se verá la necesidad de la existencia de los mapas de bits en el Apartado 12.9.4.2. La inserción de registros no afecta a la secuencia de numeración de otros registros. Por tanto, se puede insertar tanto añadiendo registros al final del archivo como reemplazando los registros borrados.

12.9.4.2. Implementación eficiente de las operaciones de mapas de bits

Se puede calcular fácilmente la intersección de dos mapas de bits usando un bucle **for**: la iteración *i*-ésima calcula la conjunción de los *i*-ésimos bits de los dos mapas de bits. Se puede acelerar considerablemente el cálculo de la intersección usando las instrucciones de bits **and** soportadas por la mayoría de los conjuntos de instrucciones de las computadoras. Una *palabra* consiste generalmente de 32 o 64 bits, dependiendo de la arquitectura de la computadora. Una instrucción de bits **and** toma dos palabras como entrada y devuelve una palabra en que cada bit es la conjunción lógica de los bits en las posiciones correspondientes de las palabras de entrada. Lo que es importante observar es que una única instrucción de bits **and** puede calcular la intersección de 32 o 64 bits *a la vez*.

Si una relación tuviese un millón de registros, cada mapa de bits contendría un millón de bits, o, equivalentemente, 128 Kbytes. Sólo se necesitan 31.250 instrucciones para calcular la intersección de dos mapas de bits para la relación, asumiendo un tamaño de palabra de 32 bits. Así, el cálculo de las intersecciones de mapas de bits es una operación extremadamente rápida.

Al igual que la intersección de mapas de bits es útil para calcular la conjunción de dos condiciones, la unión de mapas de bits es útil para calcular la disyunción de dos condiciones. El procedimiento para la unión de mapas de bits es exactamente igual que para la intersección, excepto en que se usa la instrucción de bits **or** en lugar de **and**.

La operación complemento se puede usar para calcular un predicado que incluya la negación de una condición, como **not**(*nivel-ingresos* = *L*1). El complemento de un mapa de bits se genera complementando cada bit del mapa de bits (el complemento de 1 es 0 y el complemento de 0 es 1). Puede parecer que **not**(*nivel-ingresos* = *L*1) se puede implementar simplemente calculando el complemento del mapa de bits para el nivel de ingresos *L*1. Sin embargo, si se han borrado algunos registros, el cálculo del complemento del mapa de bits no es suficiente. Los bits que correspondan a esos registros serían 0 en el mapa de bits original, pero serían 1 en el complemento, aunque el registro no exista. Tam-

bién aparece un problema similar cuando el valor de un atributo es *nulo*. Por ejemplo, si el valor de *nivel-ingresos* es nulo, el bit sería 0 en el mapa de bits original para el valor *L*1 y 1 en el complementado.

Para asegurarse de que los bits correspondientes a registros borrados se establezcan a 0 en el resultado, el mapa de bits complementado se debe intersectar con el mapa de bits de existencia para desactivar los bits de los registros borrados. Análogamente, para manejar los valores nulos, el mapa de bits complementado también se debe intersectar con el complemento del mapa de bits para el valor *nulo*[1].

El recuento del número de bits que son 1 en el mapa de bits se puede hacer fácilmente con una técnica inteligente. Se puede mantener un *array* de 256 entradas, donde la *i*-ésima entrada almacene el número de bits que son 1 en la representación binaria de *i*. Se establece el recuento inicial a 0. Se toma cada byte del mapa de bits, se usa para indexar en el *array* y se añade el recuento inicial al recuento total. El número de operaciones de suma sería la octava parte del número de tuplas, y así el proceso de recuento sería muy eficiente. Un gran *array* (que use 2^{16}=65.536 entradas), indexado por pares de bytes, daría incluso mayores aceleraciones pero a un mayor coste de almacenamiento.

12.9.4.3. Mapas de bits y árboles B⁺

Los mapas de bits se pueden combinar con los índices normales de árboles B^+ para las relaciones donde unos pocos valores de atributo son extremadamente comunes, y otros valores también aparecen, pero con mucha menor frecuencia. En una hoja de un índice de un árbol B^+, para cada valor se mantendría normalmente una lista de todos los registros con ese valor para el atributo indexado. Cada elemento de la lista sería un identificador de registro, consistiendo en al menos 32 bits, y usualmente más. Para un valor que aparece en muchos registros, se almacena un mapa de bits en lugar de una lista de registros.

Supónganse que un valor particular v_i aparece en la dieciseisava parte de los registros de una relación, y también que los registros tienen un número de 64 bits que los identifica. El mapa de bits necesita sólo 1 bit por registro, o *N* en total. En cambio, la representación de lista necesita 64 bits por registro en el que aparezca el valor, o 64 * *N*/16 = 4*N* bits. Así, el mapa de bits es preferible para la representación de la lista de registros para el valor v_i. En el ejemplo (con un identificador de registro de 64 bits), si menos de 1 de cada 64 registros tiene un valor particular, es preferible la representación de lista de registros para la identificación de registros con ese valor, ya que usa menos bits que la representación con mapas de bits. Si más de 1 de cada 64 registros tiene un valor particular, la representación de mapas de bits es preferible.

Así, los mapas de bits se pueden usar como un mecanismo de almacenamiento comprimido en los nodos hoja de los árboles B^+, para los valores que aparecen muy frecuentemente.

12.10. RESUMEN

- Muchas consultas solamente hacen referencia a una pequeña proporción de los registros de un archivo. Para reducir el gasto adicional en la búsqueda de estos registros se pueden construir *índices* para los archivos almacenados en la base de datos.
- Los archivos secuenciales indexados son unos de los esquemas de índice más antiguos usados en los sistemas de bases de datos. Para permitir una rápida recuperación de los registros según el orden de la clave de búsqueda, los registros se almacenan consecutivamente y los que no siguen el orden se encadenan entre sí. Para permitir un acceso aleatorio, se emplean estructuras índice.
- Hay dos tipos de índices que se pueden utilizar: los índices densos y los índices dispersos. Los índices densos contienen una entrada por cada valor de la clave de búsqueda, mientras que los índices dispersos contienen entradas sólo para algunos de esos valores.
- Si el orden de una clave de búsqueda se corresponde con el orden secuencial del archivo, un índice sobre la clave de búsqueda se conoce como *índice primario*. Los otros índices son los *índices secundarios*. Los índices secundarios mejoran el rendimiento de las consultas que utilizan otras claves de búsqueda aparte de la primaria. Sin embargo, éstas implican un gasto adicional en la modificación de la base de datos.
- El inconveniente principal de la organización del archivo secuencial indexado es que el rendimiento disminuye según crece el archivo. Para superar esta deficiencia se puede usar un *índice de árbol B⁺*.
- Un índice de árbol B^+ tiene la forma de un árbol *equilibrado*, en el cual cada camino de la raíz a las hojas del árbol tiene la misma longitud. La altura de un árbol

[1] La gestión de predicados tales como **is unknown** causaría aún más complicaciones, que requerirían en general el uso de un mapa de bits extra para determinar los resultados de las operaciones que son desconocidos.

B^+ es proporcional al logaritmo en base *N* del número de registros de la relación, donde cada nodo interno almacena *N* punteros; el valor de *N* está usualmente entre 50 y 100. Los árboles B^+ son más cortos que otras estructuras de árboles binarios equilibrados como los árboles AVL y, por tanto, necesitan menos accesos a disco para localizar los registros.

- Las búsquedas en un índice de árbol B^+ son directas y eficientes. Sin embargo, la inserción y el borrado son algo más complicados pero eficientes. El número de operaciones que se necesitan para la inserción y borrado en un árbol B^+ es proporcional al logaritmo en base *N* del número de registros de la relación, donde cada nodo interno almacena *N* punteros.
- Se pueden utilizar los árboles B^+ tanto para indexar un archivo con registros, como para organizar los registros de un archivo.
- Los índices de árbol B son similares a los índices de árbol B^+. La mayor ventaja de un árbol B es que el árbol B elimina el almacenamiento redundante de los valores de la clave de búsqueda. Los inconvenientes principales son la complejidad y el reducido grado de salida para un tamaño de nodo dado. En la práctica, los índices de árbol B^+ están casi generalmente mejor considerados que los índices de árbol B.
- Las organizaciones de archivos secuenciales necesitan una estructura de índice para localizar los datos. Los archivos con organizaciones basadas en asociación, en cambio, permiten encontrar la dirección de un elemento de datos directamente mediante el cálculo de una función con el valor de la clave de búsqueda del registro deseado. Ya que no se sabe en tiempo de diseño la manera precisa en la cual los valores de la clave de búsqueda se van a almacenar en el archivo, una buena función de asociación a elegir es la que distribuya los valores de la clave de búsqueda a los cajones de una manera uniforme y aleatoria.
- La *asociación estática* utiliza una función de asociación en la que el conjunto de direcciones de cajones está fijado. Estas funciones de asociación no se pueden adaptar fácilmente a las bases de datos que tengan un crecimiento significativo con el tiempo. Hay varias *técnicas de asociación dinámica* que permiten que la función de asociación cambie. Un ejemplo es la *asociación extensible,* que trata los cambios de tamaño de la base datos mediante la división y fusión de cajones según crezca o disminuya la base de datos.
- También se puede utilizar la asociación para crear índices secundarios; tales índices se llaman *índices asociativos*. Por motivos de notación se asume que las organizaciones de archivos asociativos tienen un índice asociativo implícito en la clave de búsqueda usada para la asociación.
- Los índices ordenados con árboles B^+ y con índices asociativos se pueden usar para la selección basada en condiciones de igualdad que involucren varios atributos. Cuando hay varios atributos en una condición de selección se pueden intersectar los identificadores de los registros recuperados con los diferentes índices.
- Los archivos en retícula proporcionan un medio general de indexación con múltiples atributos.
- Los índices de mapas de bits proporcionan una representación muy compacta para la indexación de atributos con muy pocos valores distintos. Las operaciones de intersección son extremadamente rápidas en los mapas de bits, haciéndolos ideales para el soporte de consultas con varios atributos.

TÉRMINOS DE REPASO

- Acceso con varias claves
- Árbol equilibrado
- Archivos en retícula
- Archivo secuencial indexado
- Asociación cerrada
- Asociación dinámica
- Asociación estática
- Asociación extensible
- Atasco
- Cajón
- Desbordamiento de cajones
- Espacio adicional
- Exploración secuencial
- Función de asociación
- Índice con agrupación
- Índice sin agrupación
- Índice de árbol B
- Índice de árbol B^+
- Índice asociativo
- Índice denso
- Índice disperso
- Índice de mapas de bits
- Índice multinivel
- Índice ordenado
- Índice primario
- Índice secundario

- Índices sobre varias claves
- Operaciones de mapas de bits
  - Intersección
  - Unión
  - Complemento
  - Mapa de bits de existencia
- Organización de archivos con árboles B^+
- Organización de archivos con asociación
- Registro/entrada del índice
- Tiempo de acceso
- Tiempo de borrado
- Tiempo de inserción
- Tipos de acceso

EJERCICIOS

12.1. ¿Cuándo es preferible utilizar un índice denso en vez de un índice disperso? Razónese la respuesta.

12.2. Dado que los índices agilizan el procesamiento de consultas, ¿por qué no deberían de mantenerse en varias claves de búsqueda? Enumérense tantas razones como sea posible.

12.3. ¿Cuál es la diferencia entre un índice primario y un índice secundario?

12.4. ¿Es posible en general tener dos índices primarios en la misma relación para dos claves de búsqueda diferentes? Razónese la respuesta.

12.5. Constrúyase un árbol B^+ con el siguiente conjunto de valores de la clave:

(2, 3, 5, 7, 11, 17, 19, 23, 29, 31)

Asúmase que el árbol está inicialmente vacío y que se añaden los valores en orden ascendente. Constrúyanse árboles B^+ para los casos en los que el número de punteros que caben en un nodo son:

a. cuatro
b. seis
c. ocho

12.6. Para cada árbol B^+ del Ejercicio 12.5 muéstrense los pasos involucrados en las siguientes consultas:

a. Encontrar los registros con un valor de la clave de búsqueda de 11.
b. Encontrar los registros con un valor de la clave de búsqueda entre 7 y 17, ambos inclusive.

12.7. Para cada árbol B^+ del Ejercicio 12.5 muéstrese el aspecto del árbol después de cada una de las siguientes operaciones:

a. Insertar 9.
b. Insertar 10.
c. Insertar 8.
d. Borrar 23.
e. Borrar 19.

12.8. Considérese el esquema modificado de redistribución para árboles B^+ descrito en la página 295. ¿Cuál es la altura esperada del árbol en función de n?

12.9. Repítase el Ejercicio 12.5 para un árbol B.

12.10. Explíquense las diferencias entre la asociación abierta y la cerrada. Coméntense los beneficios de cada técnica en aplicaciones de bases de datos.

12.11. ¿Cuáles son las causas del desbordamiento de cajones en un archivo con una organización asociativa? ¿Qué se puede hacer para reducir la aparición del desbordamiento de cajones?

12.12. Supóngase que se está usando la asociación extensible en un archivo que contiene registros con los siguientes valores de la clave de búsqueda:

2, 3, 5, 7, 11, 17, 19, 23, 29, 31.

Muéstrese la estructura asociativa extensible para este archivo si la función de asociación es $h(x) = x \bmod 8$ y los cajones pueden contener hasta tres registros.

12.13. Muéstrese cómo cambia la estructura asociativa extensible del Ejercicio 12.12 como resultado de realizar los siguientes pasos:

a. Borrar 12.
b. Borrar 31.
c. Insertar 1.
d. Insertar 15.

12.14. Dese un pseudocódigo para el borrado de entradas de una estructura asociativa extensible, incluyendo detalles del momento y forma de fusionar cajones. No se debe considerar la reducción del tamaño de la tabla de direcciones de cajones.

12.15. Sugiérase una forma eficaz de comprobar si la tabla de direcciones de cajones en una asociación extensible se puede reducir en tamaño almacenando un recuento extra con la tabla de direcciones de cajones. Dense detalles de cómo se debería mantener el recuento cuando se dividen, fusionan o borran los cajones.

(Nota: la reducción del tamaño de la tabla de direcciones de cajones es una operación costosa y las inserciones subsecuentes pueden causar que la tabla vuelva a crecer. Por tanto, es mejor no reducir el tamaño tan pronto como se pueda, sino solamente si el número de entradas de índice es pequeño en comparación con el tamaño de la tabla de direcciones de cajones).

12.16. ¿Por qué una estructura asociativa no es la mejor elección para una clave de búsqueda en la que son frecuentes las consultas de rangos?

12.17. Considérese un archivo en retícula en el cual se desea evitar el desbordamiento de cajones por razones de rendimiento. En los casos en los que sería necesario un cajón de desbordamiento, en su lugar se reorganiza el archivo. Preséntese un algoritmo para esta reorganización.

12.18. Considérese la relación *cuenta* mostrada en la Figura 12.25.

a. Constrúyase un índice de mapa de bits sobre los atributos *nombre-sucursal* y *saldo*, dividiendo *saldo* en cuatro rangos: menores que 250, entre 250 y menor que 500, entre 500 y menor que 750, y 750 o mayor.

b. Considérese una consulta que solicite todas las cuentas de Daimiel con un saldo de 500 o más. Descríbanse los pasos para responder a la consulta y muéstrense los mapas de bits finales e intermedios construidos para responder la consulta.

12.19. Muéstrese la forma de calcular mapas de existencia a partir de otros mapas de bits. Asegúrese de que la técnica funciona incluso con valores nulos, usando un mapa de bits para el valor *nulo*.

12.20. ¿Cómo afecta el cifrado de datos a los esquemas de índices? En particular, ¿como afectaría a los esquemas que intentan almacenar los datos ordenados?

NOTAS BIBLIOGRÁFICAS

En Cormen et al. [1990] se pueden encontrar discusiones acerca de las estructuras básicas utilizadas en la indexación y asociación. Los índices de árbol B se introdujeron primero en Bayer [1972] y en Bayer y McCreight [1972]. Los árboles B^+ se discuten en Comer [1979], Bayer y Unterauer [1977] y Knuth [1973]. Las notas bibliográficas del Capítulo 16 proporcionan referencias a la investigación sobre los accesos concurrentes y las actualizaciones en los árboles B^+. Gray y Reuter [1993] proporcionan una buena descripción de los resultados en la implementación de árboles B^+.

Se han propuesto varias estructuras alternativas de árboles y basadas en árboles. Los *tries* son unos árboles cuya estructura está basada en los «dígitos» de las claves (por ejemplo, el índice de muescas de un diccionario, con una entrada para cada letra). Estos árboles podrían no estar equilibrados en el sentido que lo están los árboles B^+. Los *tries* se discuten en Ramesh et al. [1989], Orestein [1982], Litwin [1981] y Fredkin [1960]. Otros trabajos relacionados son los árboles B digitales de Lomet [1981].

Knuth [1973] analiza un gran número de técnicas de asociación distintas. Existen varias técnicas de asociación dinámica. Fagin et al. [1979] introduce la asociación extensible. La asociación lineal se introduce en Litwin [1978] y Litwin [1980]; en Larson [1982] se presentó un análisis del rendimiento de este esquema. Ellis [1987] examina la concurrencia con la asociación lineal. Larson [1988] presenta una variante de la asociación lineal. Otro esquema, llamado asociación dinámica se propone en Larson [1978]. Una alternativa propuesta en Ramakrishna y Larson [1989] permite la recuperación en un solo acceso a disco al precio de una gran sobrecarga en una pequeña fracción de las modificaciones de la base de datos. La asociación dividida es una extensión de la asociación para varios atributos, y se trata en Rivest [1976], Burkhard [1976] y Burkhard [1979].

La estructura de archivo en retícula aparece en Nievergelt et al [1984] y en Hinrichs [1985]. Los índices de mapas de bits y las variantes denominadas índices por capas de bits e índices de proyección se describen en O'Neil y Quass [1997]. Se introdujeron por primera vez en el gestor de archivos Model 204 de IBM sobre la plataforma AS 400. Proporcionan grandes ganancias de velocidad en ciertos tipos de consultas y se encuentran implementadas actualmente en la mayoría de sistemas de bases de datos. La investigación reciente en índices de mapas de bits incluye Wu y Buchmann [1998], Chan y Ioannidis [1998], Chan y Ioannidis [1999] y Johnson [1999a].