Capítulo 12: Indexación y asociación

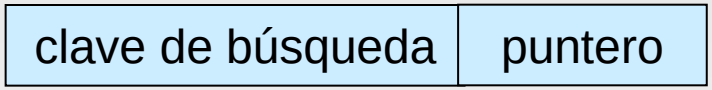
Fundamentos de Bases de datos, 5ª Edición.

Capítulo 12: Indexación y asociación

- Conceptos básicos
- Índices ordenados
- Archivos de índices de árbol B+
- Archivos de índices de árbol B
- Asociación estática
- Asociación dinámica
- Comparación entre indexación ordenada y asociación
- Definición de índices en SQL
- Acceso multiclave

Conceptos básicos

- Mecanismos de indexación empleados para acelerar el acceso a los datos deseados.
  - Por ejemplo, el catálogo de autores en una biblioteca
- **Clave de búsqueda** – atributo, del conjunto de atributos, empleado para buscar registros en un archivo.
- Un **archivo de índices** consta de registros (denominados **entradas de índice**) de la forma



- Los archivos de índices generalmente son más pequeños que el archivo original
- Dos clases básicas de índices:
  - **Índices ordenados:** las claves de búsqueda se almacenan de forma ordenada
  - **Índices asociativos:** las claves de búsqueda están distribuidas uniformemente en “cajones”, empleando una “función de asociación”.

Métricas de evaluación de índices

- Tipos de acceso soportados eficientemente. Por ejemplo,
  - registros con un valor concreto en el atributo
  - o registros con un valor de atributo que se encuentra en un determinado rango de valores.
- Tiempo de acceso
- Tiempo de inserción
- Tiempo de borrado
- Costes de espacio

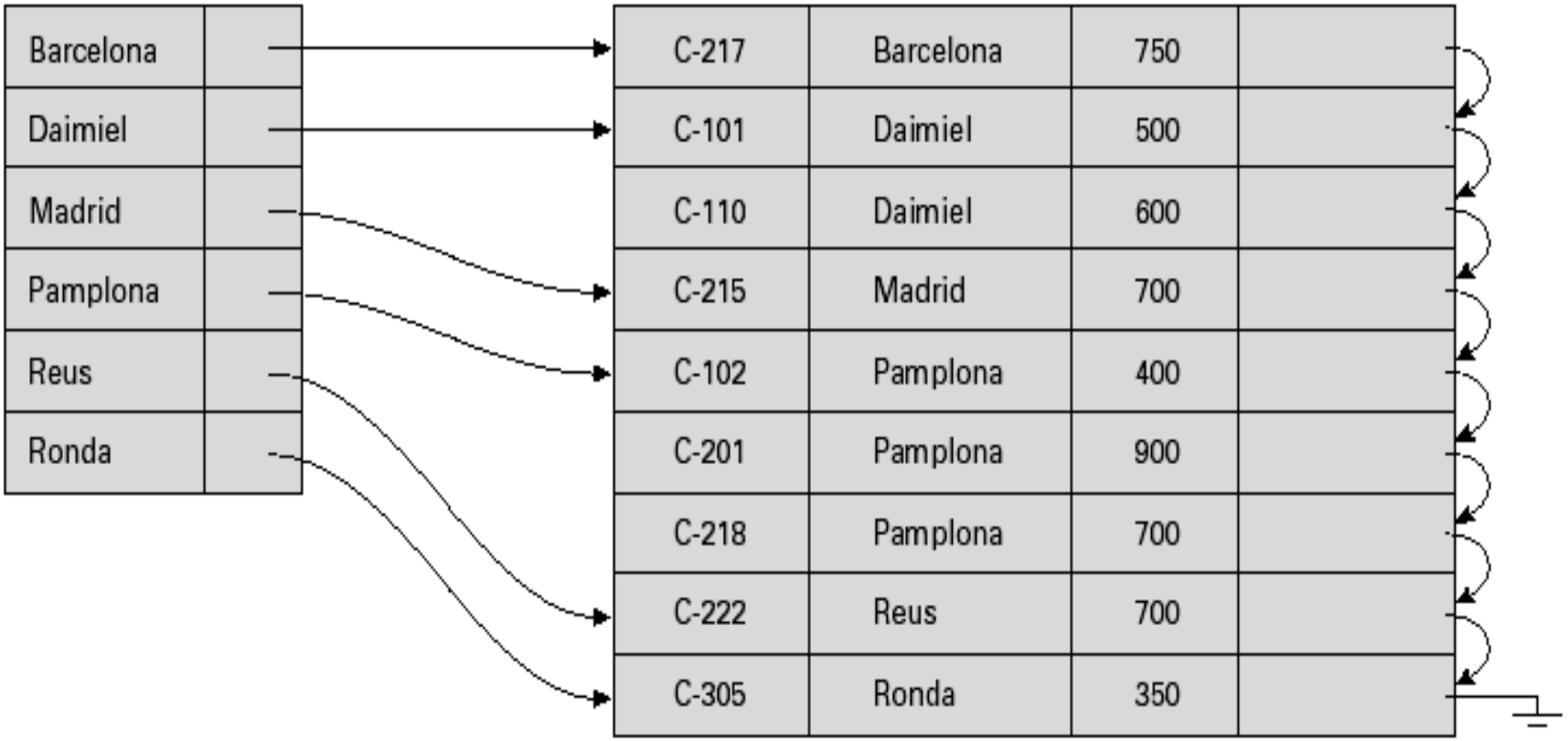
Índices ordenados

Técnicas de indexado evaluadas en base a:

- En un **índice ordenado**, las entradas de índices se almacenan ordenadas sobre el valor de la clave de búsqueda. Por ejemplo, el catálogo de autores en una biblioteca.
- **Índice primario:** en un archivo ordenado secuencialmente, el índice cuya clave de búsqueda determina el orden secuencial del archivo.
  - También denominado **índice con agrupación**
  - La clave de búsqueda de un índice primario es generalmente, pero no necesariamente, la clave primaria.
- **Índice secundario**: un índice cuya clave de búsqueda determina un orden diferente del orden secuencial del archivo. También llamado índice sin agrupación**.**
- Archivo secuencial indexado: archivo secuencial ordenado con un índice primario.

Archivos de índice denso

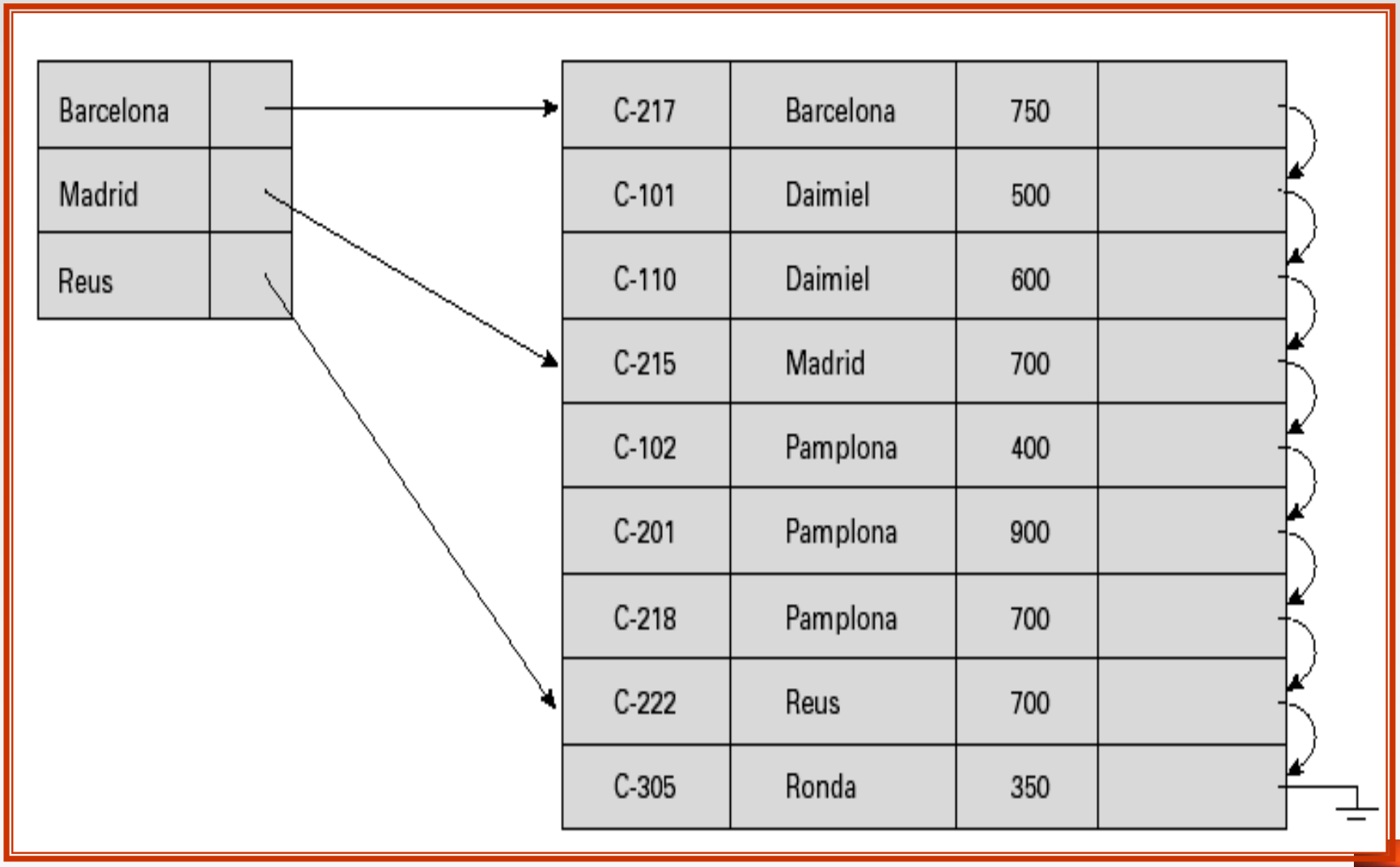
- Índice denso — Registro del índice que aparece por cada valor de la clave de búsqueda del archivo.



Archivos de índice disperso

- Índice disperso: contiene registros del índice, sólo para algunos valores de la clave de búsqueda.
  - Aplicable cuando los registros están ordenados secuencialmente sobre la clave de búsqueda
- Para localizar un registro con valor *K* de la clave de búsqueda:
  - Encontrar el registro del índice con mayor valor de clave de búsqueda < *K*
  - Búsqueda secuencial del archivo, empezando por el registro al que apunta el registro del índice
- Menos espacio y menores costes de mantenimiento para las inserciones y los borrados.
- Generalmente más lento, para localizar registros, que el índice denso.
- Buen equilibrio: índice disperso con una entrada del índice por cada bloque en el archivo, correspondiente al menor valor de la clave de búsqueda en el bloque.

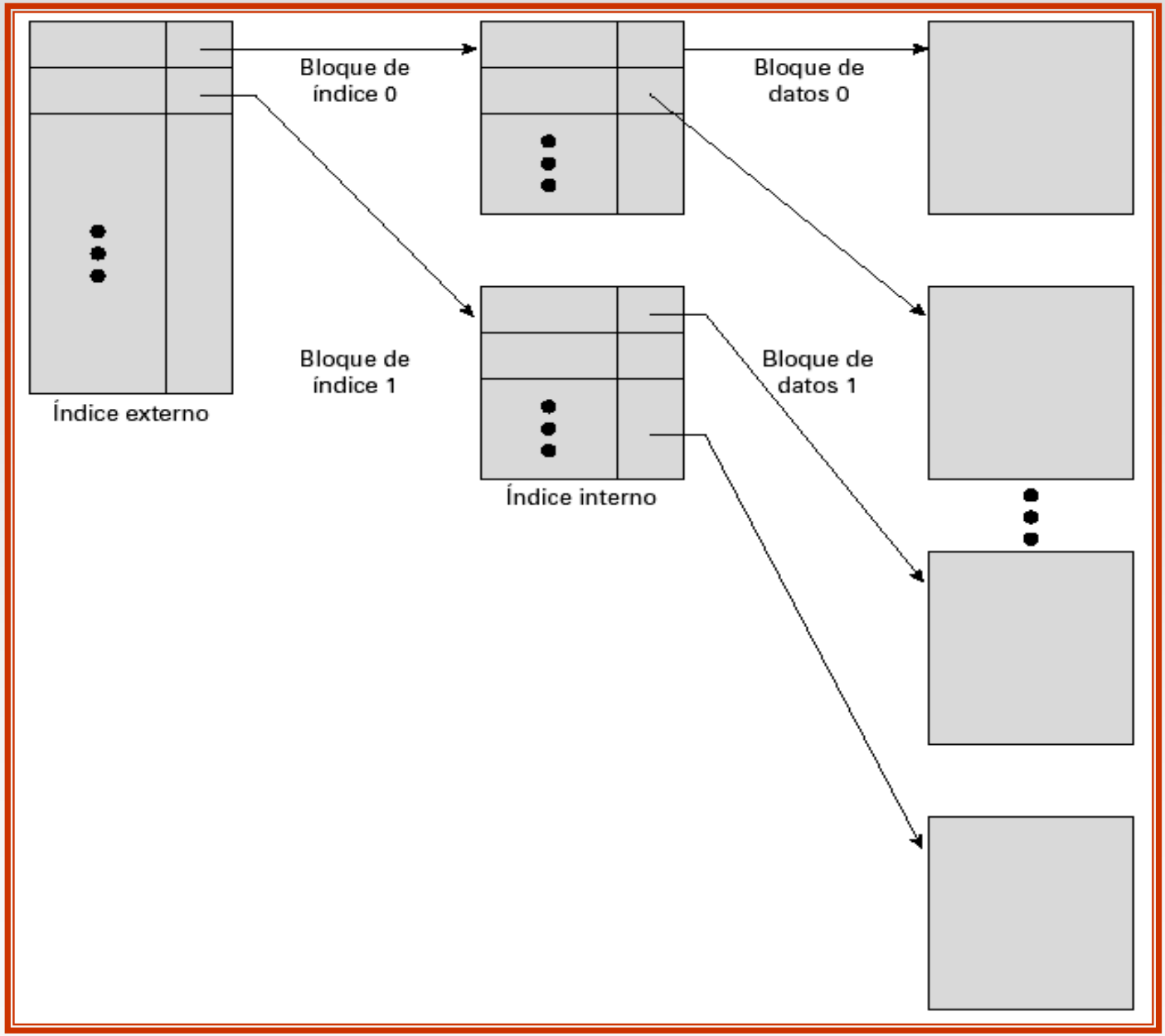
Ejemplo de archivos de índice disperso



Índice multinivel

- Si el índice primario no encaja en memoria, el acceso se hace costoso.
- Para reducir el número de accesos a disco para los registros del índice, tratar de mantener el índice primario sobre disco como un archivo secuencial y construir un índice disperso en él.
  - índice externo – un índice disperso de índice primario
  - índice interno – el archivo del índice primario
- Si incluso el índice externo es demasiado grande para encajar en memoria principal, aún se puede crear otro nivel de índice, etcétera.
- Al insertar o borrar del archivo, se deben actualizar los índices a todos los niveles.

Índice multinivel (Cont.)



Actualización del índice: Borrado

- Si el registro borrado era el único registro del archivo con su valor de clave de búsqueda concreto, la clave de búsqueda se borra también del índice.
- Borrado del índice de un solo nivel:
  - Índices densos – el borrado de la clave de búsqueda es similar al borrado del registro del archivo.
  - Índices dispersos –
    - si existe una entrada para la clave de búsqueda en el índice se borra, reemplazando la entrada en el índice con el siguiente valor de la clave de búsqueda en el archivo (ordenado por clave de búsqueda).
    - Si el valor de la siguiente clave de búsqueda tiene una entrada del índice, se borra la entrada en vez de reemplazarla.

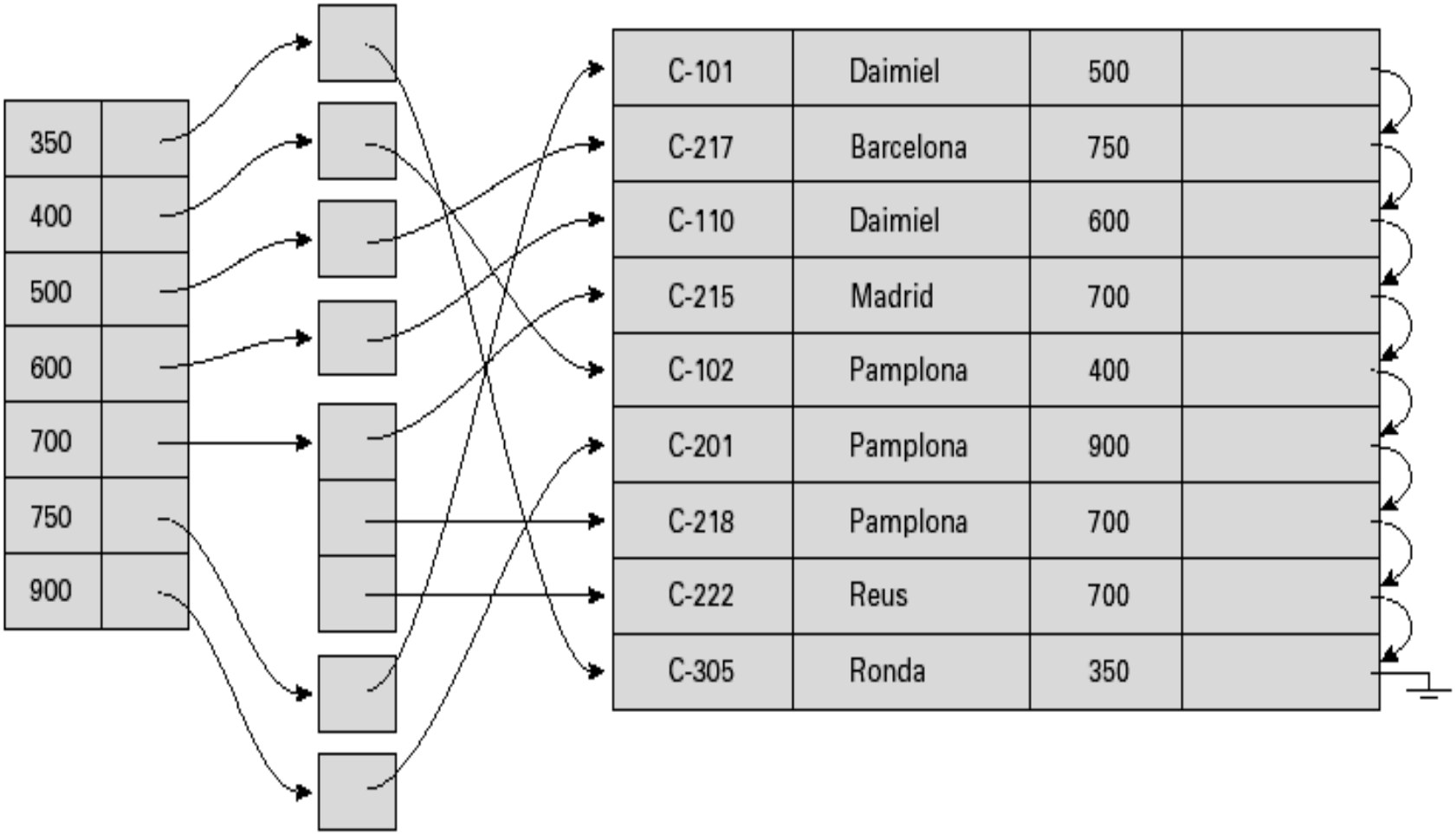
Actualización del índice: Inserción

- Inserción de índices de un solo nivel:
  - Realizar una búsqueda empleando el valor de la clave de búsqueda que aparece en el registro a insertar.
  - Índices densos – si el valor de la clave de búsqueda no aparece en el índice, insertarlo.
  - Índices dispersos – si el índice almacena una entrada por cada bloque del archivo, no es necesario hacer ningún cambio al índice, a menos que se cree un nuevo bloque.
    - Si se crea un nuevo bloque, se inserta en el índice el primer valor de la clave de búsqueda en el nuevo bloque.
- Los algoritmos de inserciones multinivel (así como en el borrado) son simples extensiones de los algoritmos de un solo nivel

Índices secundarios

- Frecuentemente, se quieren encontrar todos los registros cuyos valores en un cierto campo (que no es la clave de búsqueda del índice primario) cumplen alguna condición.
  - Ejemplo 1: En la base de datos *cuenta*, almacenada secuencialmente por número de cuenta, se pueden encontrar fácilmente todas las cuentas de una determinada oficina
  - Ejemplo 2: como antes, pero donde se quieren encontrar todas las cuentas con un determinado saldo o rango de saldos
- Se puede tener un índice secundario con un registro del índice por cada valor de la clave de búsqueda;
  - el registro del índice apunta a un cajón que contiene punteros a todos los registros actuales, con ese valor particular de clave de búsqueda.

Índice secundario sobre el campo *saldo* de *cuenta*



Índices primario y secundario

- Los índices secundarios han de ser densos.
- Los índices ofrecen importantes ventajas en la búsqueda de registros.
- Cuando se modifica un archivo, se debe actualizar cada índice del archivo; Actualizar los índices implica sobrecargas en la modificación de la base de datos.
- La búsqueda secuencial empleando índices primarios es eficiente, pero utilizando un índice secundario es costosa
  - cada acceso de registro puede coger un nuevo bloque del disco

Archivos de índice de árbol B+

Los índices de árbol B^+ son una alternativa a los archivos secuenciales indexados.

- Inconvenientes de los archivos secuenciales indexados: el rendimiento baja cuando el archivo crece, dado que se crean muchos bloques de desbordamiento. Es necesario reorganizar periódicamente todo el archivo.
- Ventajas de los archivos de índice de árbol B^+: se reorganiza automáticamente por sí mismo con pequeños cambios locales, a pesar de las inserciones y los borrados. No es necesario reorganizar todo el archivo para mantener el rendimiento.
- Inconvenientes de los árboles B^+: inserciones extras, sobrecarga de borrados y costes de espacio.
- Las ventajas de los árboles B^+ superan a los inconvenientes, por lo que se emplean ampliamente.

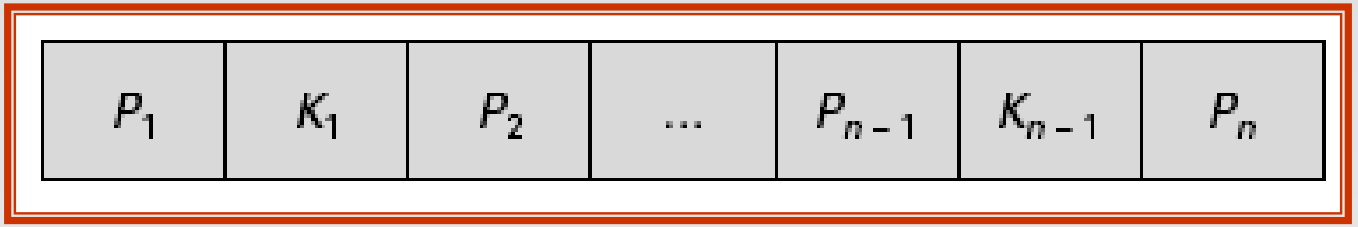
Archivos de índice de árbol B+ (Cont.)

Un árbol B^+ es un árbol con raíz que satisface las siguientes propiedades:

- Todos los caminos, desde la raíz a las hojas, tienen la misma longitud
- Cada nodo que no es ni raíz ni hoja, tiene entre [n/2] y n hijos.
- Un nodo hoja tiene entre [(n–1)/2] y n–1 valores
- Casos especiales:
  - Si la raíz no es una hoja, tiene al menos 2 hijos.
  - Si la raíz es una hoja (es decir, no hay otros nodos en el árbol), puede tener entre 0 y (n–1) valores.

Estructura de nodos del árbol B+

- Nodo típico



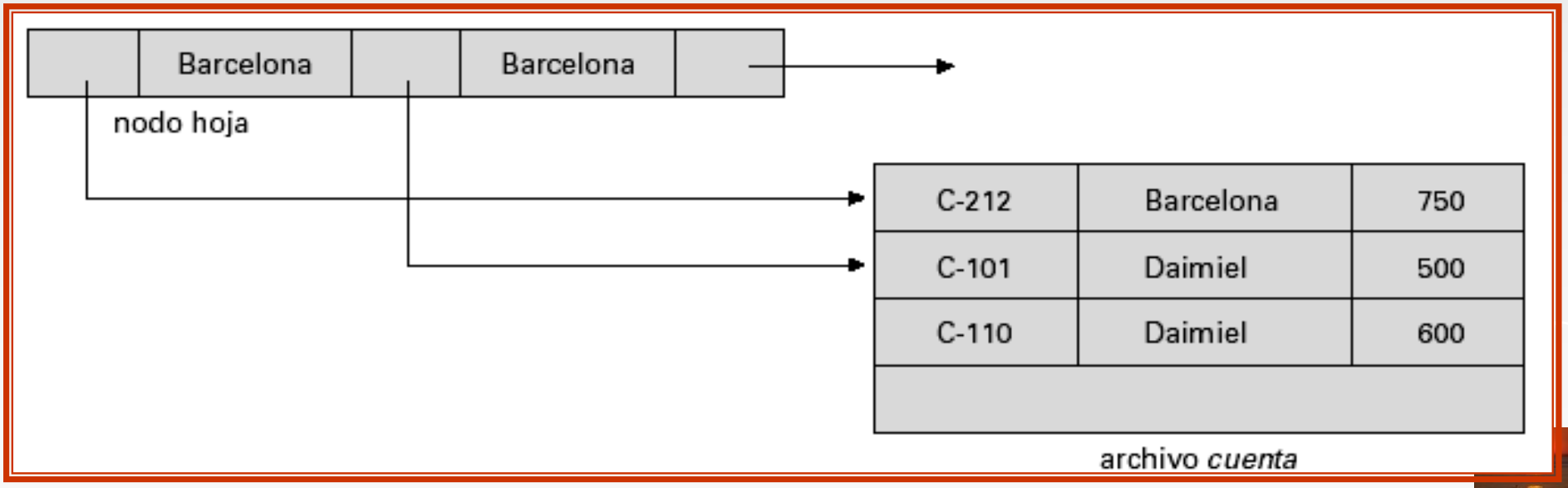
  - K_i son los valores de la clave de búsqueda
  - P_i son los punteros a los hijos (para nodos no hoja) o a los registros o cajones de registros (para nodos hoja).
- En un nodo las claves de búsqueda están ordenadas

$$K_1 < K_2 < K_3 < \ldots < K_{n-1}$$

Nodos hoja en árboles B^+

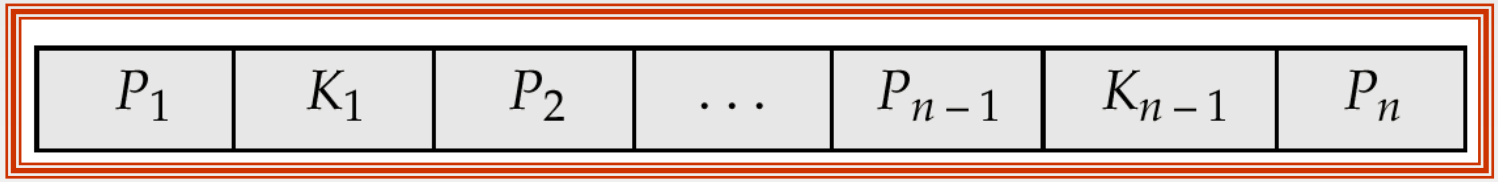
Propiedades de un nodo hoja:

- Para $i = 1, 2, \ldots, n-1$, el puntero P_i apunta a un registro del archivo con valor de clave de búsqueda K_i, o a un cajón de punteros a los registros del archivo, cada registro con valor de clave de búsqueda K_i. Sólo es necesaria una estructura de cajones si la clave de búsqueda no es una clave primaria.
- Si L_i, L_j son nodos hoja e $i < j$, los valores de clave de búsqueda de L_i son menores que los de L_j
- P_n apunta al siguiente nodo hoja, ordenado por clave de búsqueda

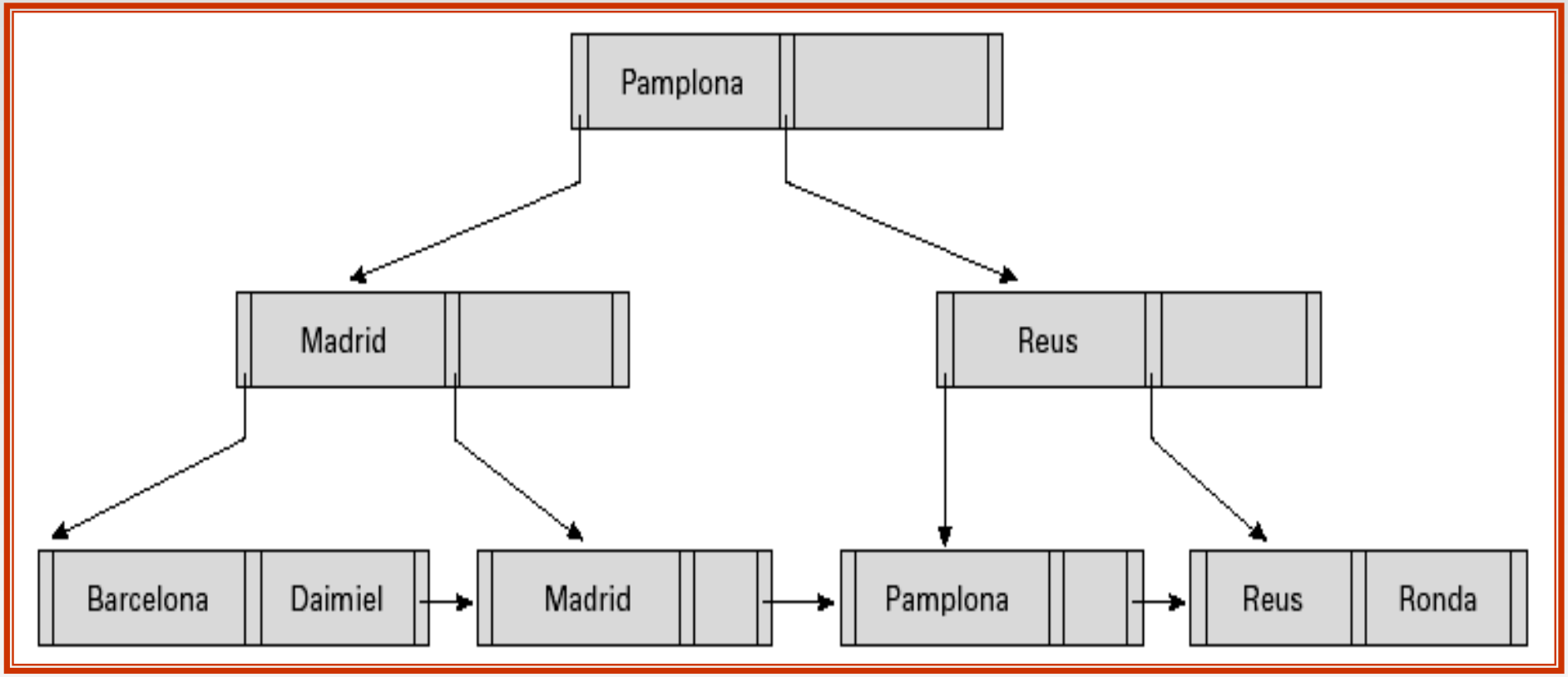


Nodos que no son hoja en árboles B^+

- Los nodos que no son hoja forman un índice disperso multinivel sobre los nodos hoja. Para un nodo no hoja con *m* punteros:
  - Todas las claves de búsqueda en el subárbol al que apunta P_1 son menores que K_1
  - Para $2 \leq i \leq n - 1$, todas las claves de búsqueda en el subárbol al que apunta P_i, tienen valores mayores o iguales que K_{i-1} y menores que K_{m-1}

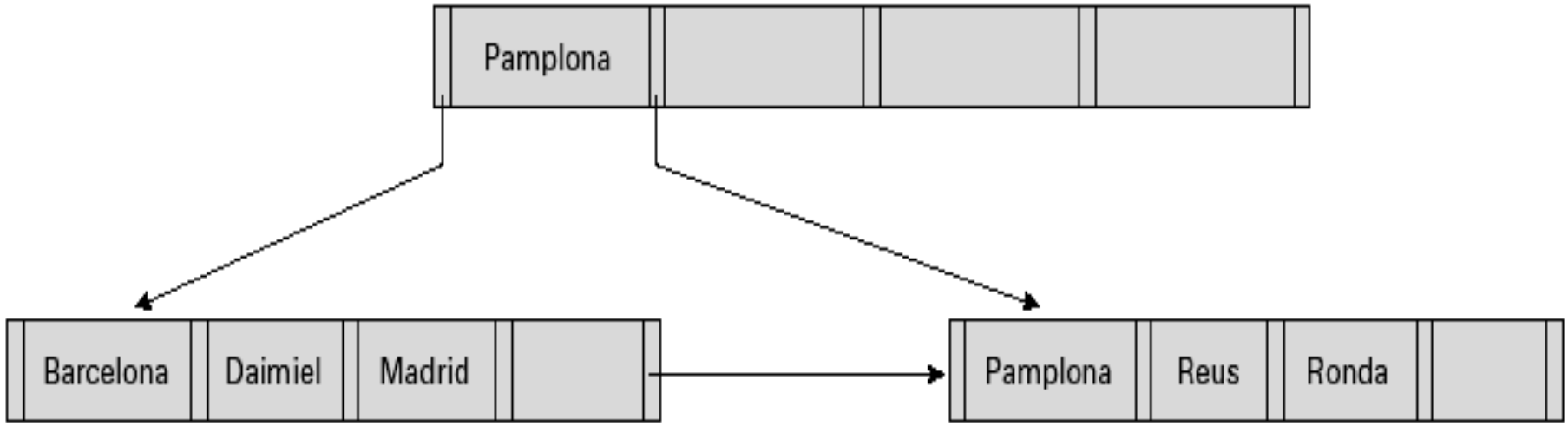


Ejemplo de un árbol B^+



Árbol B^+ del archivo *cuenta* (n = 3)

Ejemplo de árbol B+



Árbol B+ del archivo *cuenta* (*n* - 5)

- Los nodos hoja deben tener entre 2 y 4 valores ($\lceil (n-1)/2 \rceil$ y $n-1$, con $n = 5$).
- Otros nodos que no son hoja, excepto la raíz, deben tener entre 3 y 5 hijos ($\lceil n/2 \rceil$ y n con $n = 5$).
- La raíz debe tener al menos 2 hijos.

Observaciones sobre los árboles B^+

- Dado que las conexiones entre nodos se hacen mediante punteros, los bloques “lógicamente” cercanos no necesitan estar “físicamente” próximos.
- Los niveles que no son hoja del árbol B^+ forman una jerarquía de índices dispersos.
- El árbol B^+ contiene un número relativamente pequeño de niveles (logarítmicos en el tamaño del archivo principal), por lo que las búsquedas se pueden realizar de forma eficiente.
- Las inserciones y los borrados sobre el archivo principal se pueden gestionar eficientemente, del mismo modo que el índice se puede reconstruir de forma logarítmica (como se verá).

Consultas sobre árboles B^+

- Encontrar todos los registros con valor de la clave de búsqueda *k.*
  1. Empezar con el nodo raíz
     1. Examinar el nodo para el menor valor de clave de búsqueda > *k.*
     2. Si existe un valor así, suponer que es K_j. Entonces, siguen P_i al nodo hijo
     3. De lo contrario, $k \geq K_{m-1}$, donde hay *m* punteros en el nodo. Entonces, siguen P_m al nodo hijo.
  2. Si el nodo alcanzado siguiendo el puntero anterior no es un nodo hoja, repetir el procedimiento anterior sobre el nodo.
  3. De lo contrario al alcanza un nodo hoja.
     1. Si para algún *i*, la clave K_i = k, seguir el puntero P_i hasta el registro o cajón deseado.
     2. De lo contrario no existe ningún registro con valor de clave de búsqueda *k*.

Consultas sobre árboles B^+ (Cont.)

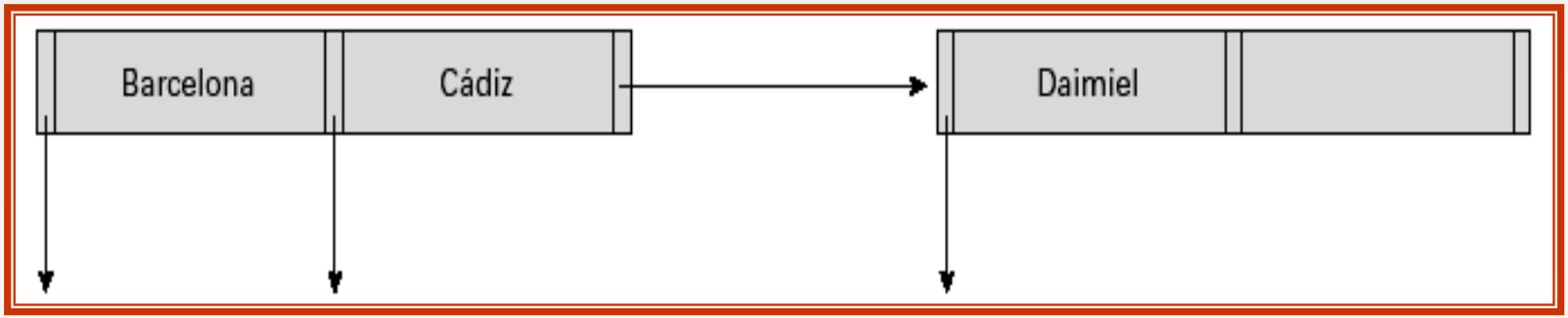
- Al procesar una consulta un camino recorre el árbol, desde la raíz hasta algún nodo hoja.
- Si hay *K* valores de clave de búsqueda en el archivo, el camino no es mayor que $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$.
- Un nodo es generalmente del mismo tamaño que un bloque de disco, típicamente 4 kb, y *n* suele estar entorno a 100 (40 bytes por entrada de índice).
- Con 1 millón de valores de clave de búsqueda y $n = 100$, como máximo $log_{50}(1.000.000) = 4$ nodos son accedidos en una búsqueda.
- Contrasta esto con un libre binario equilibrado con 1 millón de valores de clave de búsqueda — en una búsqueda se accede a alrededor de 20 nodos
  - ¡la diferencia anterior es importante, dado que los accesos a los nodos pueden necesitar una E/S de disco, con un coste estimado de entorno a 20 milisegundos!

Actualizaciones sobre árboles B+: Inserción

- Encontrar el nodo hoja en que aparecería el valor de la clave de búsqueda
- Si el valor de la clave de búsqueda ya está en el nodo hoja, se añade el registro al archivo y, si es necesario, se inserta un puntero en el cajón.
- Si el valor de la clave de búsqueda no está allí, se añade el registro al archivo principal y, si es necesario, se crea un cajón. Entonces:
  - Si hay espacio en el nodo hoja, insertar el par (valor de la clave, puntero)
  - De lo contrario, dividir el nodo (además de la nueva entrada, valor de la clave y puntero) como se trata en la siguiente transparencia.

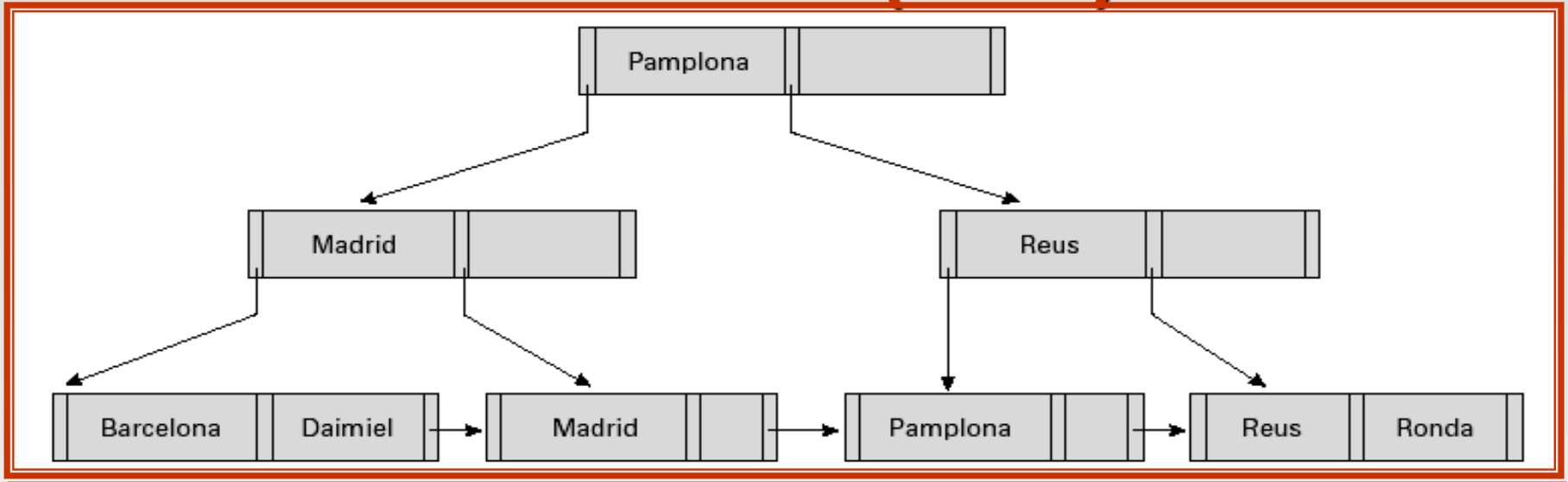
Actualizaciones sobre árboles B+: Inserción (Cont.)

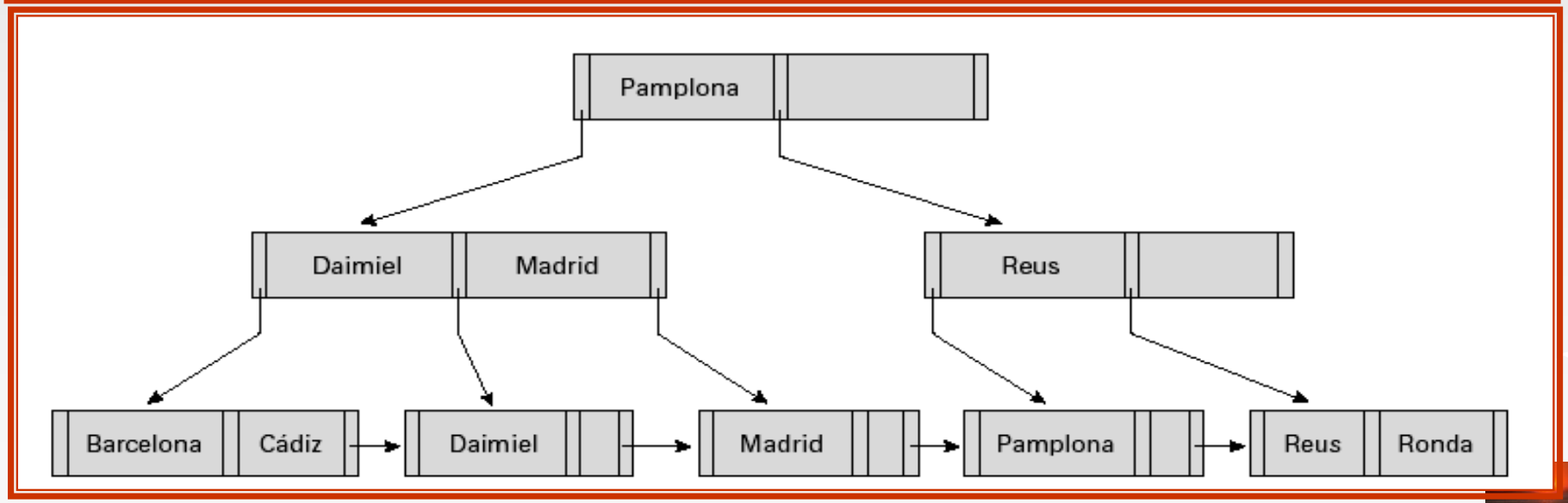
- División de un nodo:
  - Tomar los *n* pares (valor de la clave, puntero) de forma ordenada (incluyendo el que se está insertando). Situar el primer $\lceil n/2 \rceil$ en el nodo original y los el resto en un nodo nuevo.
  - sea *p* el nuevo nodo, y sea *k* el menor valor de la clave en *p.* Insertar (*k, p*) en el padre del nodo que se está dividiendo. Si el padre está lleno, dividirlo y propagar la división hacia arriba.
- La división de los nodos se lleva a cabo hacia arriba, hasta encontrar un nodo que no esté lleno. En el peor de los casos, el nodo raíz se puede dividir incrementando la altura del árbol en 1.



Resultado de dividir un nodo que contiene Barcelona y Daimiel al insertar Cádiz

Actualizaciones sobre árboles B+: Inserción (Cont.)





Árbol B^+ antes y después de insertar “Cádiz”

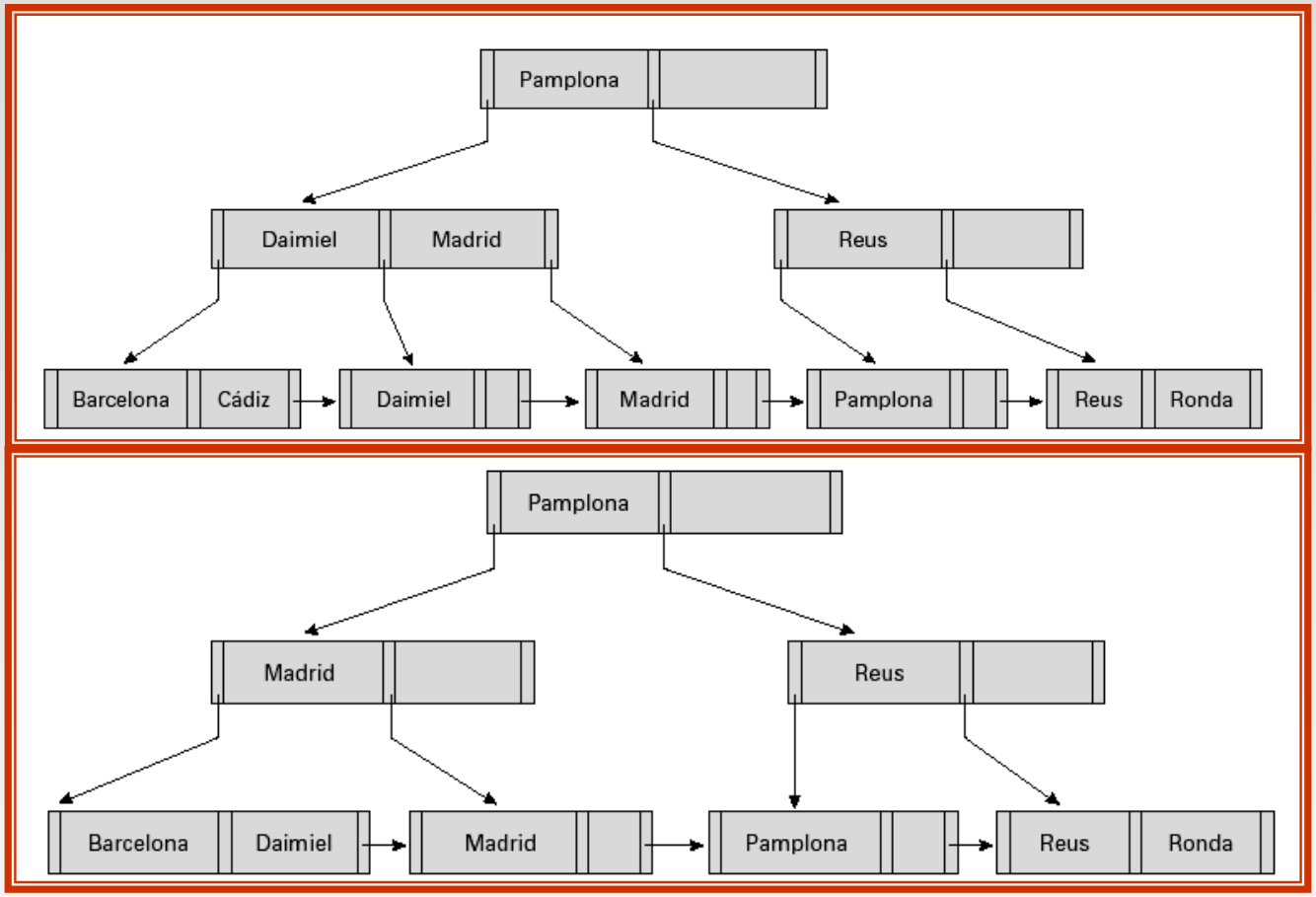
Actualizaciones sobre árboles B+: Borrado

- Encontrar el registro a borrar y eliminarlo del archivo principal y del cajón (si está presente)
- Eliminar (valor de la clave de búsqueda y puntero) del nodo hoja si no hay ningún cajón, o si se ha quedado vacío
- Si el nodo tiene demasiadas pocas entradas debido a la eliminación y las entradas en el nodo y un hermano encajan en un solo nodo, entonces
  - Insertar todos los valores de claves de búsqueda de los dos nodos en un solo nodo (el de la izquierda) y borrar el otro.
  - Borrar el par (K_{i-1}, P_i), donde P_i es el puntero al nodo borrado, desde su padre, empleando recursivamente el procedimiento anterior.

Actualizaciones sobre árboles B+: Borrado

- De lo contrario, si el nodo tiene demasiadas pocas entradas debido a la eliminación, y las entradas en el nodo y un hermano encajan en un solo nodo, entonces
  - Redistribuir los punteros entre el nodo y un hermano, de manera que ambos tengan más que el número mínimo de entradas.
  - Actualizar el correspondiente valor de la clave de búsqueda en el padre del nodo.
- Los borrados de nodos pueden propagarse en cascada hacia arriba, hasta encontrar un nodo que tenga $\lceil n/2 \rceil$ o más punteros. Si el nodo raíz tiene después del borrado un solo puntero, se borra y el hijo único se convierte en la raíz.

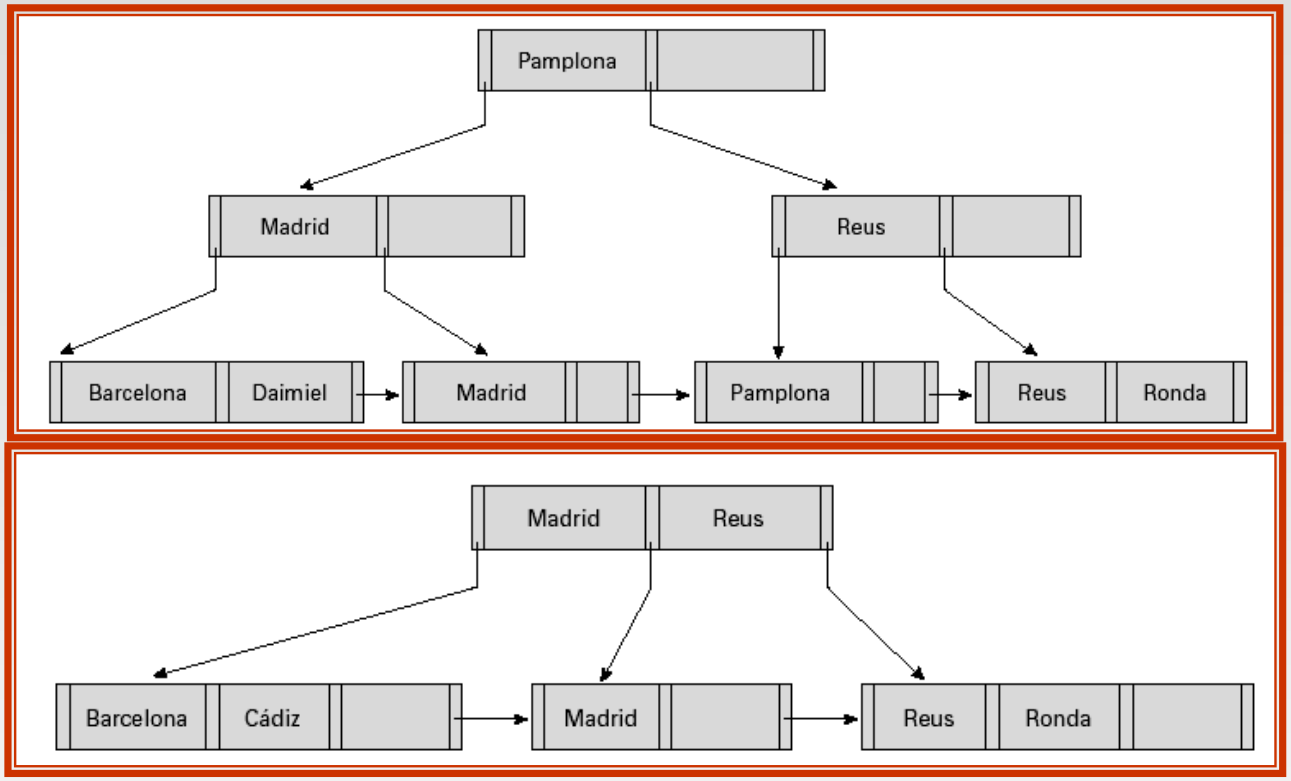
Ejemplos de borrado de árbol B+



Antes y después de borrar “Daimiel”

- La eliminación del nodo hoja conteniendo “Daimiel” no resultó en su padre, al tener demasiados pocos punteros. Así, los borrados en cascada se detienen al borrar el padre del nodo hoja.

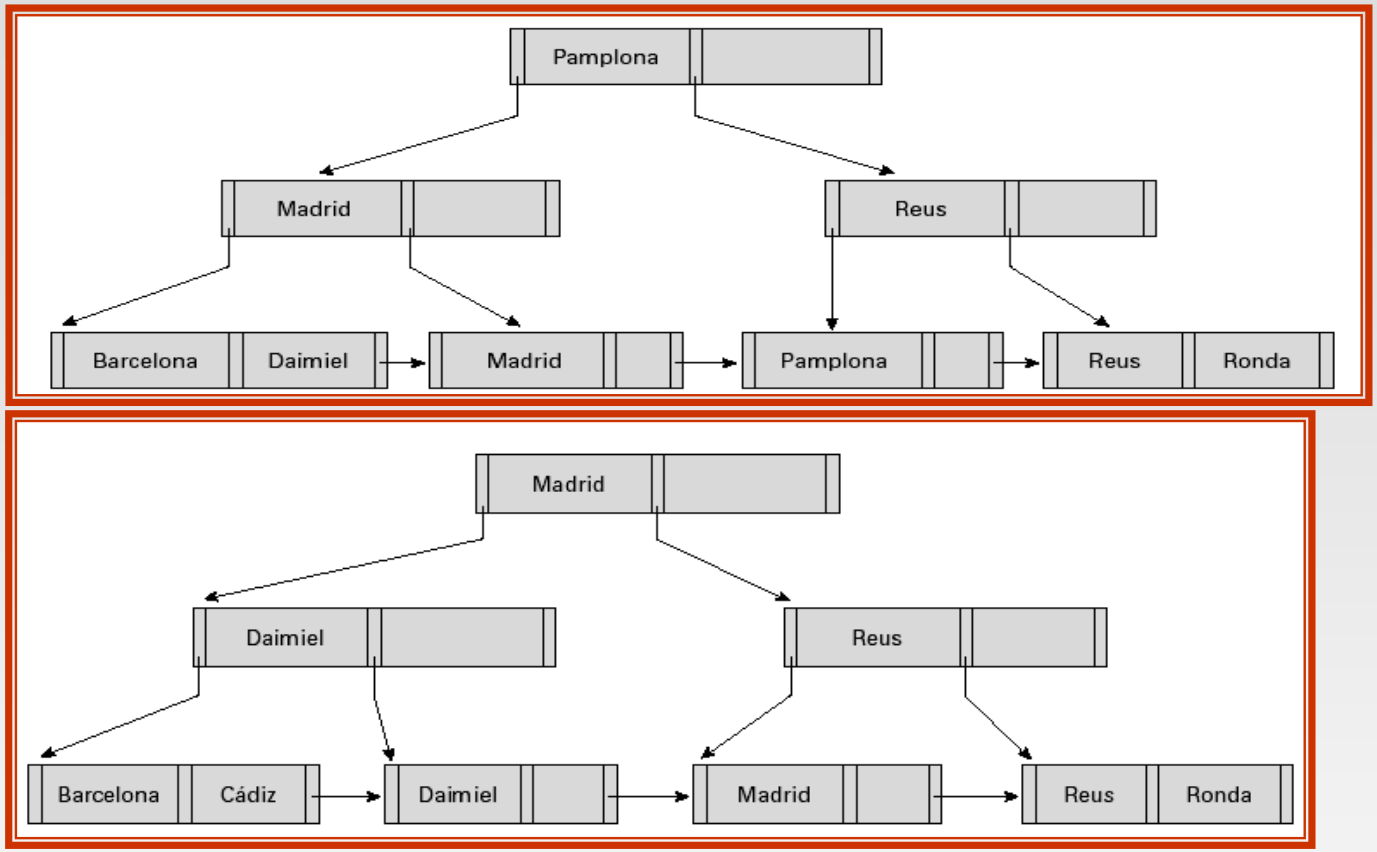
Ejemplos de borrado de árbol B+ (Cont.)



Borrado de “Pamplona” a partir del resultado del ejemplo anterior

- El nodo con “Pamplona” se vuelve infraocupado (en realidad vacío, en este caso especial) y se mezcla con su hermano.
- Como consecuencia, el padre del nodo “Pamplona” se hizo infraocupado y se mezcló con su hermano (y se borró una entrada desde su padre)
- Entonces el nodo raíz sólo tenía un hijo y se borró, convirtiéndose su hijo en el nuevo nodo raíz

Ejemplo de borrado de árbol B+ (Cont.)



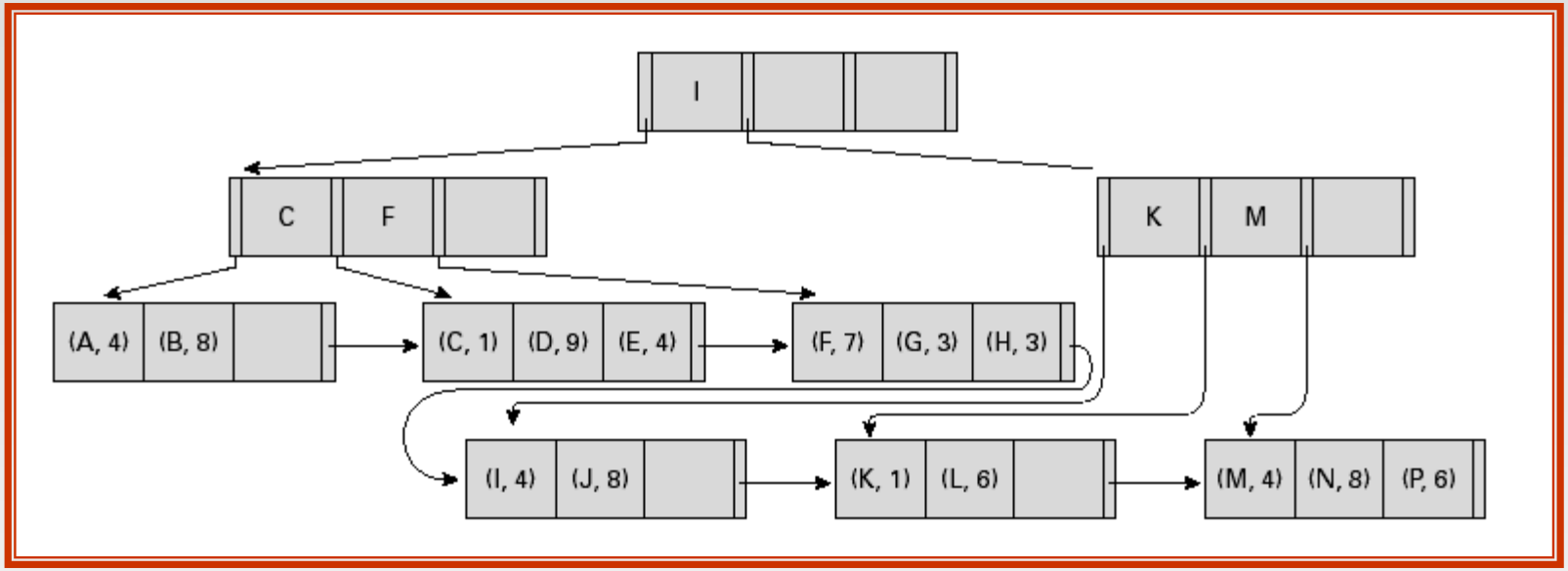
Antes y después del borrado de “Pamplona” en el ejemplo anterior

- El padre de la hoja que contiene Navacerrada se hizo infraocupado, tomando prestado un puntero desde su hermano izquierdo
- Como consecuencia, cambia el valor de la clave de búsqueda en el padre del padre

Organización de archivos de árbol B+

- El problema de la degradación de los archivos de índices se soluciona empleando índices de árbol B^+. El problema de la degradación de los archivos de datos se soluciona empleando organización de archivos de árbol B^+.
- Los nodos hoja en una organización de archivos de árbol B^+ almacenan registros, en vez de punteros.
- Dado que los registros son mas 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 no hoja.
- Los nodos hoja todavía se solicitan para estar medio llenos.
- La inserción y el borrado se gestionan de la misma manera que la inserción y el borrado de las entradas en un índice de árbol B^+.

Organización de archivos de árbol B+ (Cont.)

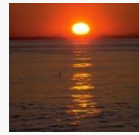


Ejemplo de organización de archivos de árbol B^+

- Es importante la buena utilización del espacio, dado que los registros emplean más espacio que los punteros.
- Para mejorar la utilización del espacio implicar más nodos hermanos en la redistribución, durante las divisiones y las mezclas
  - Implicando a 2 hermanos en la redistribución (para evitar donde sea posible, división / mezcla) se consigue que al memos se tengan entradas en cada nodo $\lfloor 2n/3 \rfloor$

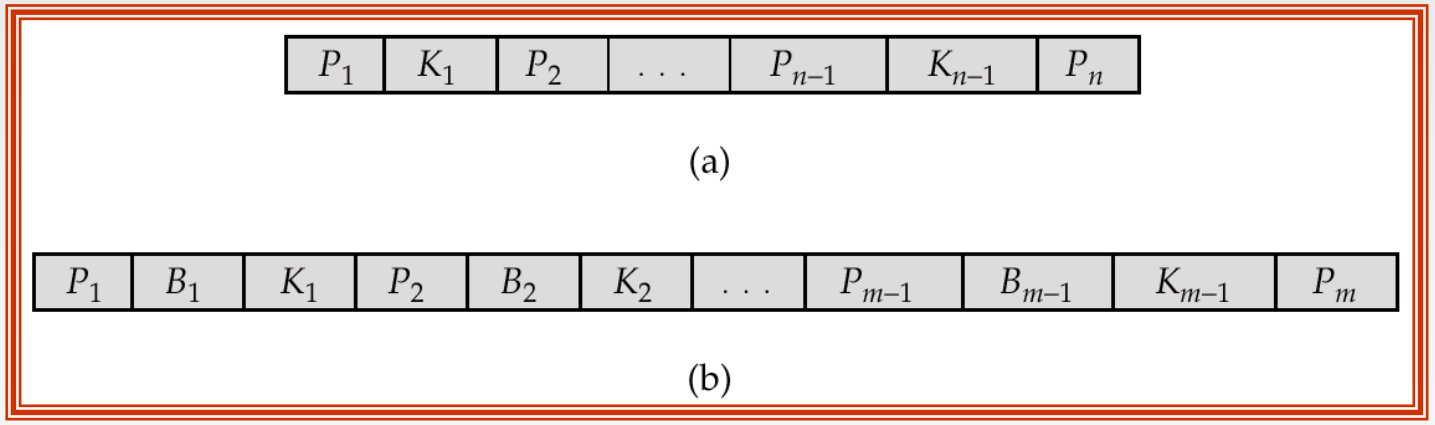
Índices sobre cadenas de caracteres

- Cadenas de longitud variable como clave
  - Diferentes grados de salida
  - Usar la utilización del espacio como criterio para distribuir y no el numero máximo de entradas que puede contener
- Compresión del prefijo
  - Los valores de las claves en los nodos internos pueden ser sólo prefijos de la clave completa
    - Disponer de caracteres suficientes que permitan distinguir entradas del árbol como distintas por su clave.
      - Ej. “Silas” y “Silberschatz” se puedes separar por “Silb”



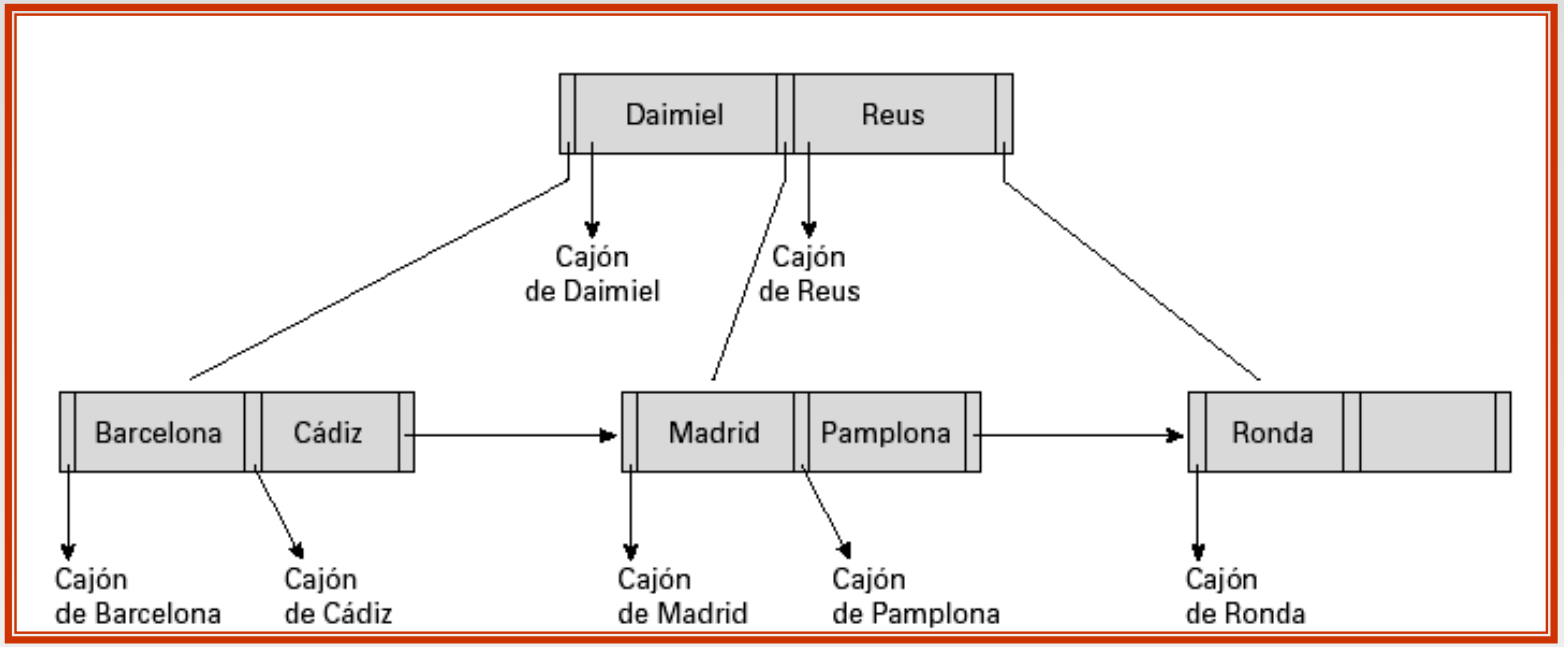
Archivos de índice de árbol B

- Similar al árbol B+, pero los árboles B permiten que los valores de la clave de búsqueda sólo aparezcan una vez; eliminan el almacenamiento redundante de las claves de búsqueda.
- Las claves de búsqueda en los nodos no hoja no aparecen en ningún otro sitio que no sea el árbol B; se debe incluir un campo de puntero adicional para cada clave de búsqueda en un nodo no hoja.
- Nodo de hoja de árbol genérico

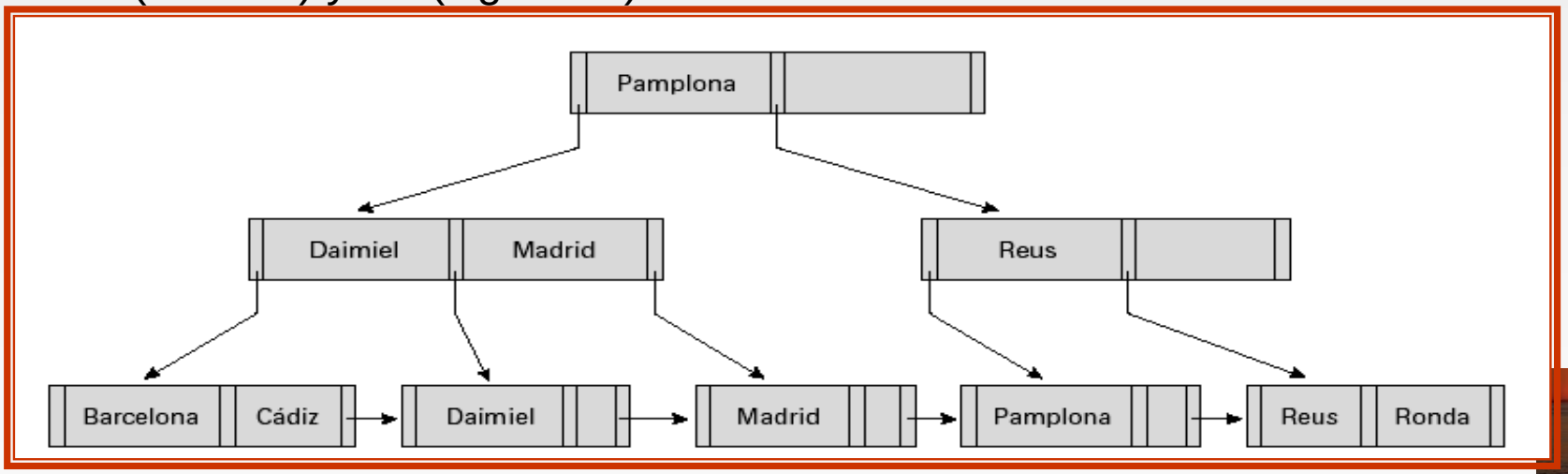


- Nodo no hoja – punteros B_i son el cajón o los punteros del registro del archivo.

Ejemplo de archivos de índice de árbol B



Árbol B (anterior) y B+ (siguiente) sobre el mismo dato



Archivos de índice de árbol B (cont.)

- Ventajas de los índices de árbol B:
  - Pueden usar menos nodos árbol que un correspondiente árbol B^+..
  - Algunas veces es posible encontrar el valor de la clave de búsqueda antes de alcanzar el nodo hoja.
- Inconvenientes de los índices de árbol B:
  - Sólo se encuentran previamente pequeñas fracciones, de entre todos los valores de las claves de búsqueda
  - Los nodos no hoja son más grandes, por lo que se reduce el grado de salida. De este modo, los árboles B tienen generalmente mayor profundidad que los correspondientes árboles B^+
  - La inserción y el borrado son más complicados que en los árboles B^+
  - La implementación es más dura que los árboles B^+.
- Generalmente, las ventajas de los árboles B no superan los inconvenientes.

Acceso multiclave

- Emplear múltiples índices para ciertos tipos de consultas.
- Ejemplo:

  select *número-cuenta*

  from *cuenta*

  where *nombre-sucursal* = “Navacerrada” **and** *saldo* = 1.000
- Estrategias posibles para el procesamiento de consultas empleando índices sobre atributos simples:
  1. Usar el índice sobre *nombre-sucursal* para encontrar cuentas con saldos de $1.000; probar con *nombre-sucursal* = “Navacerrada”.
  2. Usar el índice sobre *saldo* para encontrar cuentas con saldos de $1.000; probar con *nombre-sucursal* = “Navacerrada”.
  3. Emplear el índice de *nombre-sucursal* para encontrar punteros a todos los registros que pertenecen a la sucursal de Navacerrada. Análogamente, emplear el índice sobre *saldo*. Tomar la inserción de los dos conjuntos de punteros obtenidos.

Índices sobre varias claves

- Las **claves de búsqueda compuestas** son claves de búsqueda que contienen más de un atributo
  - Por ejemplo (*nombre_sucursal, saldo*)
- Orden lexicográfico: $(a_1, a_2) < (b_1, b_2)$ si
  - $a_1 < a_2$, o bien
  - $a_1 = a_2$ y $b_1 < b_2$

Índices sobre múltiples atributos

Supóngase que tenemos un índice con la clave de búsqueda combinada (*nombre-sucursal, saldo*).

- Con la cláusula **where**
  where *nombre-sucursal* = “Navacerrada” **and** *saldo* = 1.000
  se puede utilizar el índice (*nombre-sucursal, saldo*) para tomar sólo los registros que cumplan ambas condiciones.
  - Emplear índices independientes es menos eficiente — se pueden tomar muchos registros (o punteros) que sólo cumplen una de las condiciones.
- También se puede manejar eficientemente
  where *nombre-sucursal* = “Navacerrada” **and** *saldo* < 1.000
- Pero no se puede gestionar eficientemente
  where *nombre-sucursal* < “Navacerrada” **and** *saldo* = 1.000
  - Se pueden tomar muchos registros que cumplen la primera condición, pero no la segunda.

Claves de búsqueda no únicas

- Alternativas:
  - Disponer los cajones en bloques separados (mala idea)
  - Lista de punteros a tuplas desde cada clave
    - Código extra para manejar listas largas
    - El borrado de una tupla puede resultar costoso
    - Baja sobrecarga de espacio, no tiene coste adicional en las consultas
  - Hacer que la clave sea única añadiéndole un identificador de registro
    - Sobrecarga de almacenamiento en las claves
    - Código más simple para la inserción/borrado
    - Ampliamente utilizado

Otros temas

- Indices de cobertura
  - Almacenan los valores de algunos atributos junto con los punteros a los registros, lo que permite responder consultas utilizando sólo el índice.
    - Particularmente útiles para los índices secundarios
      - ¿Por qué?
  - Pueden almacenar atributos extra sólo en las hojas
- Indices secundarios y reubicación de registros
  - Si un registro se reubica, hay que actualizar todos los índices secundarios que almacenan punteros a dichos registros.
  - La división de nodos en una organización de archivo de árbol B^+ se vuelve muy costoso
  - Solución: utilizar claves primarias de búsqueda en lugar de punteros en los índices secundarios.
    - Navegación extra de los índices primarios para localizar un registro
      - Mayor coste de las consultas, pero la división de nodos es más sencilla
    - Añadir un id de registro si las claves de búsqueda de los índices primarios no son únicos

Asociación estática

- Un **cajón** es una unidad de almacenamiento que contiene uno o más registros (generalmente un cajón es un bloque de disco).
- En una **organización de archivos asociativa** se obtiene el cajón de un registro directamente desde su valor de clave de búsqueda, empleando una **función de asociación.**
- La función de asociación *h* es una función desde el conjunto de todos los valores de claves de búsqueda *K*, hasta el conjunto de todas las direcciones de cajones *B.*
- La función de asociación se emplea para localizar registros para accesos, inserciones y borrados.
- Los registros con diferentes valores de claves de búsqueda pueden asociarse al mismo cajón; de este modo, para localizar un registro, se ha de recorrer secuencialmente el cajón entero.

Ejemplo de organización de archivos asociativa (Cont.)

La organización de archivos asociativa del archivo *cuenta*, utilizando *nombre_sucursal* como clave (véase la figura de la siguiente transparencia).

- Hay 10 cajones,
- La representación binaria del carácter *i* ésimo se asume que sea el entero *i.*
- La función de asociación devuelve la suma de las representaciones binarias de los caracteres módulo 10
  - Por ejemplo, h(Navacerrada) = 5 h(Ronda) = 3 h(Galapagar) = 3

Ejemplo de organización de archivos asociativa

Cajón 0

Cajón 1

Cajón 2

Cajón 3

C-217	Barcelona	750
C-101	Daimiel	500

Cajón 4

C-222	Reus	700

Cajón 5

C-102	Pamplona	400
C-201	Pamplona	900
C-218	Pamplona	700

Cajón 6

Cajón 7

C-215	Madrid	700

Cajón 8

C-305	Ronda	350
C-110	Daimiel	600

Cajón 9

La organización de archivos asociativa del archivo *cuenta*, utilizando *nombre-sucursal* como clave (para obtener más detalles, véase la transparencia anterior).

Funciones de asociación

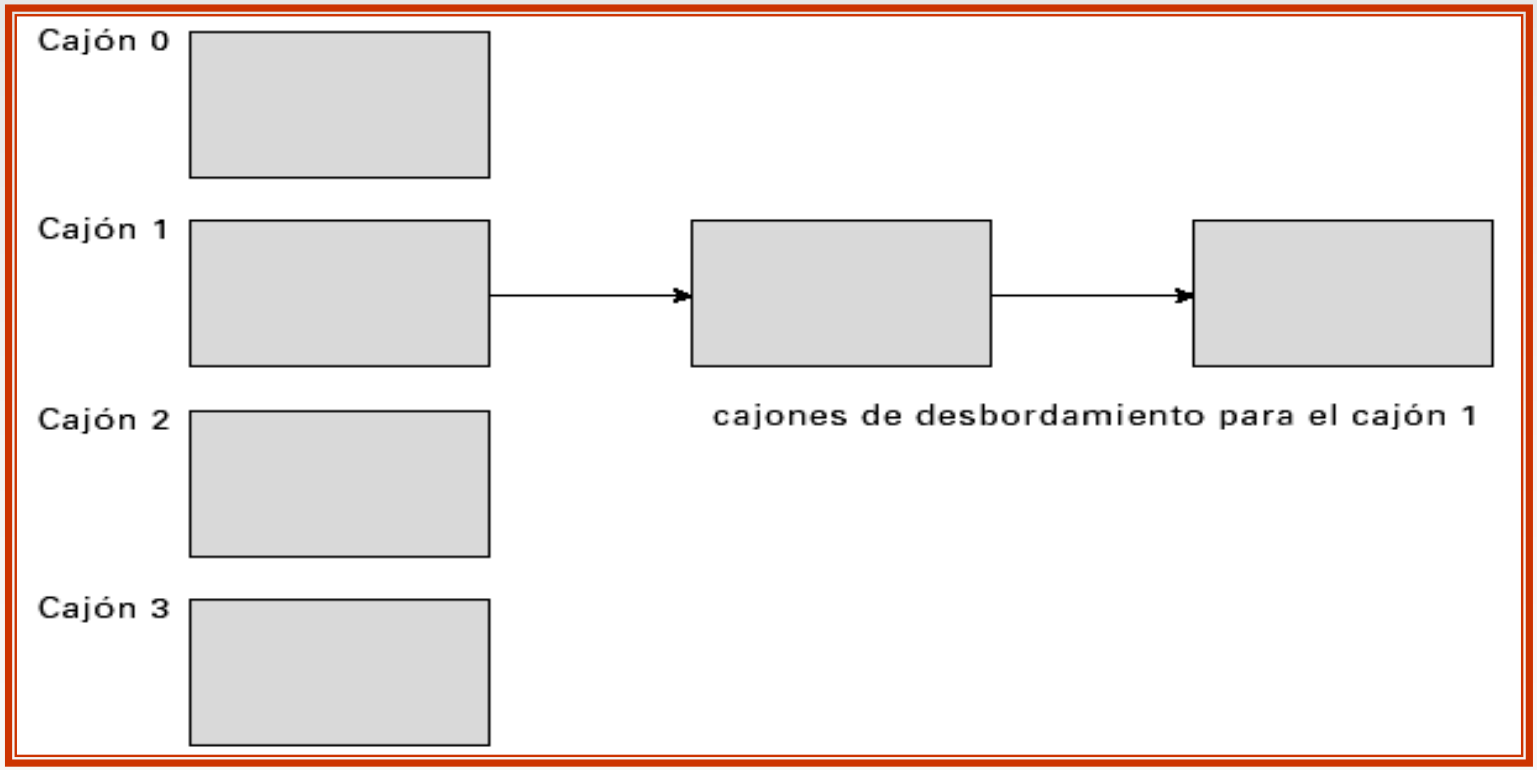
- La peor función de asociación asocia todos los valores de las claves de búsqueda al mismo cajón; esto hace que el tiempo de acceso sea proporcional al número de valores de claves de búsqueda en el archivo.
- Una función de asociación ideal es **uniforme**, es decir, cada cajón se asigna al mismo número de valores de claves de búsqueda, desde el conjunto de *todos* los valores posibles.
- La función de asociación ideal es **random**; así cada cajón tendrá asignado el mismo número de registros, independientemente de la *distribución* real de los valores de las claves de búsqueda en el archivo.
- Las funciones de asociación típicas realizan cálculos sobre la representación binaria interna de la clave de búsqueda.
  - Por ejemplo, para una clave de búsqueda de secuencia de caracteres, se podrían añadir las representaciones binarias de todos los caracteres en la secuencia y se podría devolver la suma del número de cajones.

Gestión de desbordamiento de cajones

- El desbordamiento de cajones puede producirse por
  - Insuficientes cajones
  - Desviación en la distribución de los registros. Esto puede tener lugar por dos razones:
    - múltiples registros tienen el mismo valor de clave de búsqueda
    - la función de asociación elegida produce una distribución no uniforme de los valores de las claves
- Aunque se puede reducir la probabilidad de desbordamiento de cajones, no se puede eliminar y se gestiona empleando *cajones de desbordamiento*.

Gestión de desbordamiento de cajones (Cont.)

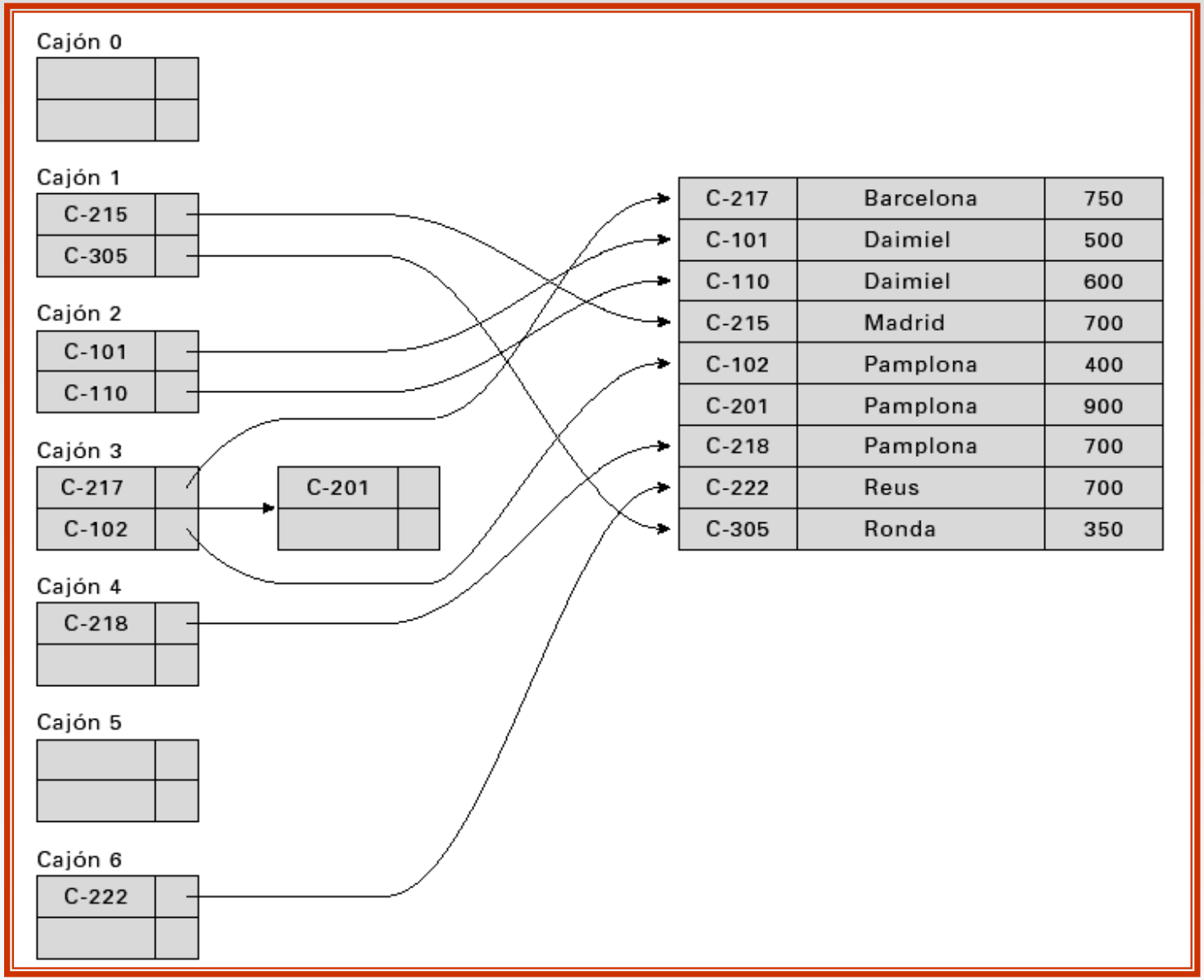
- Cadena de desbordamiento – los cajones de desbordamiento de un determinado cajón se encadenan juntos en una lista enlazada.
- El esquema anterior se denomina asociación cerrada.
  - Una alternativa, denominada asociación abierta, que no emplea cajones de desbordamiento, no es adecuada para aplicaciones de bases de datos.



Índices asociativos

- La asociación no sólo se puede emplear en la organización de archivos, sino también para la creación de estructuras de índices.
- Un **índice asociativo** organiza las claves de búsqueda, con sus punteros de registros asociados, en una estructura de archivos asociativa.
- En su sentido estricto, los índices asociativos son siempre índices secundarios
  - si el propio archivo está organizado empleando asociación, no es necesario un índice asociativo primario independiente de él, que emplee la misma clave de búsqueda.
  - Sin embargo, se emplea el término índice asociativo para referirse, tanto a las estructuras de índices secundarios, como a los archivos organizados en asociación.

Ejemplo de índice asociativo



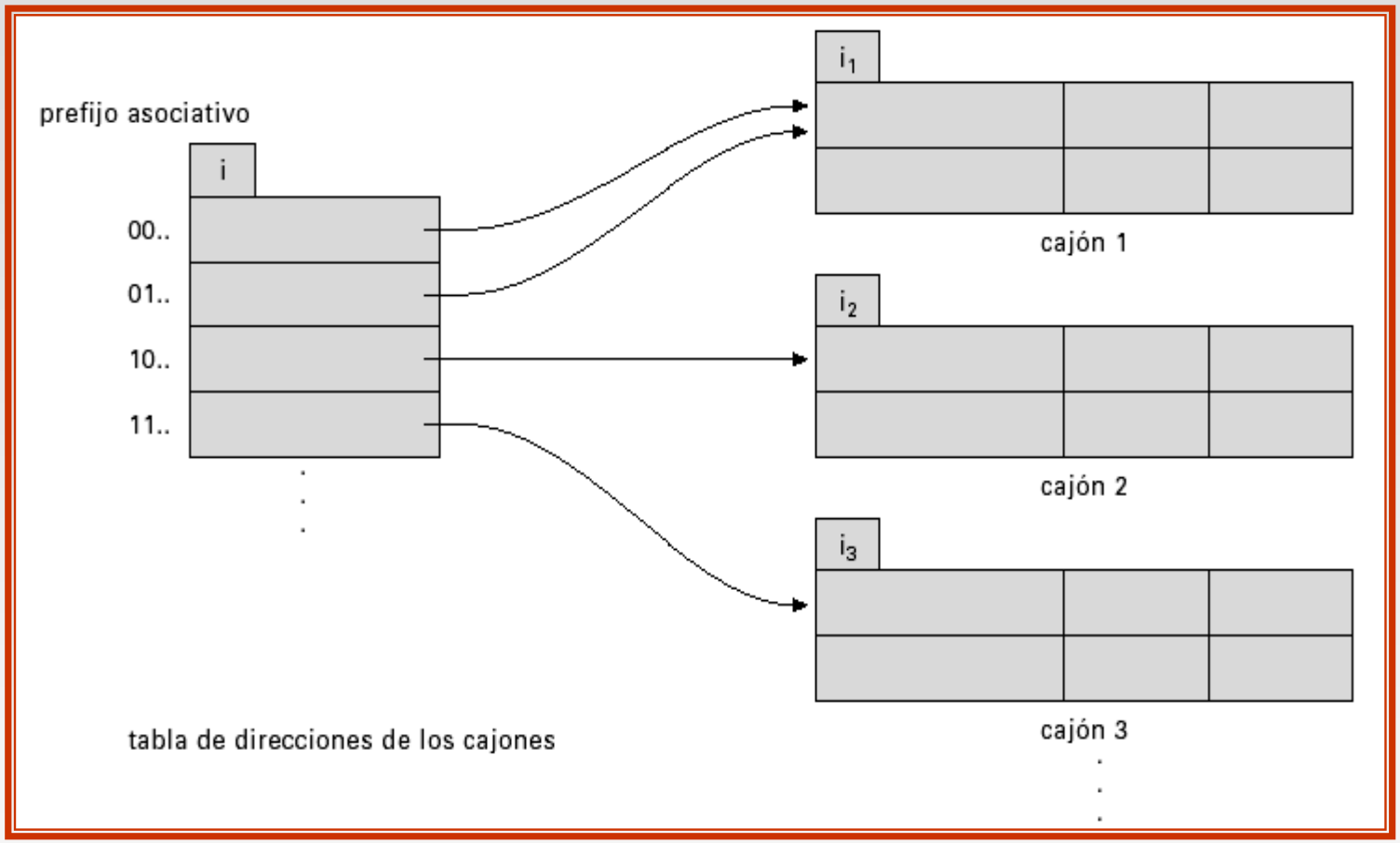
Deficiencias de la asociación estática

- En la asociación estática la función *h* asocia valores de claves de búsqueda a un determinado conjunto *B*, de direcciones de cajones.
  - Las bases de datos crecen con el tiempo. Si el número inicial de cajones es demasiado pequeño, disminuirá el rendimiento debido a los muchos desbordamientos.
  - Si se anticipa el tamaño del archivo en algún momento del futuro, y en consecuencia el número de cajones asignados, un aumento significativo de espacio se desperdiciará inicialmente.
  - Si disminuye la base de datos, nuevamente se desperdiciará espacio.
  - Una opción es la reorganización periódica del archivo con una nueva función de asociación, pero es muy costoso.
- Estos problemas se pueden evitar empleando técnicas que permitan que el número de cajones se modifique dinámicamente.

Asociación dinámica

- Buena para las bases de datos que aumentan y disminuyen de tamaño
- Permite modificar dinámicamente la función de asociación
- **Asociación extensible** – una forma de asociación dinámica
  - La función de asociación genera valores en un amplio rango — generalmente *b*-bit enteros, con $b = 32$.
  - En cualquier momento se emplea sólo un prefijo de la función de asociación, para indexar en una tabla de direcciones de cajones.
  - Sea la longitud del prefijo *i* bits, donde $0 \leq i \leq 32$.
  - El tamaño de la tabla de direcciones de los cajones es = 2^{i}. Inicialmente $i = 0$
  - El valor de *i* crece y disminuye según lo hace el tamaño de la base de datos.
  - Múltiples entradas en la tabla de direcciones de los cajones pueden apuntar a un cajón.
  - Así, el número real de cajones es $< 2^{i}$
    - El número de cajones también cambia dinámicamente debido a las agrupaciones y divisiones de los cajones.

Estructura de asociación extensible general



En esta estructura, $i_2 = i_3 = i$, mientras que $i_1 = i - 1$ (para obtener más detalles, véase la siguiente transparencia)

Uso de la estructura de asociación extensible

- Cada cajón *j* almacena un valor i_j; todas las entradas que apuntan al mismo cajón tienen los mismos valores sobre los primeros i_j bits.
- Para localizar el cajón que contiene la clave de búsqueda K_j:
  1. Calcular $h(K_j) = X$
  2. Emplear los primeros *j* bits de orden superior de *X* como un desplazamiento en la tabla de direcciones de los cajones, y seguir el puntero al cajón apropiado
- Para insertar un registro con valor de clave de búsqueda K_j
  - seguir el mismo procedimiento de búsqueda y localizar, por ejemplo, el cajón *j*.
  - Si hay espacio en el cajón *j* insertar un registro en él.
  - De lo contrario se debe dividir el cajón y reintentar la inserción (siguiente transparencia).
    - En algunos casos, en su lugar, se emplean cajones de desbordamiento (se verá en breve)

Actualizaciones en la estructura de asociación extensible

Para dividir un cajón *j* al insertar un registro con el valor de clave de búsqueda K_j:

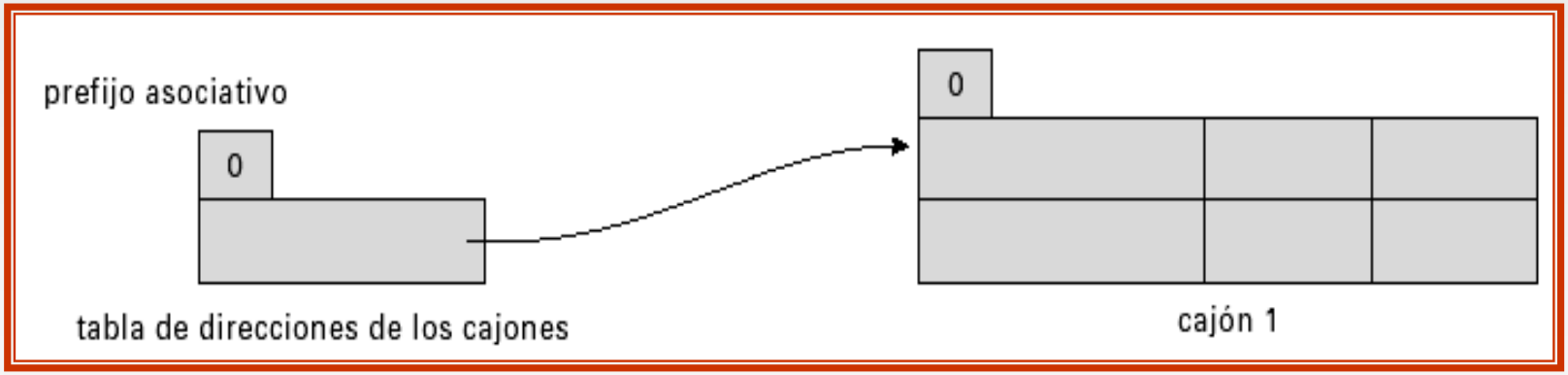
- Si $i > i_j$ (más de un puntero al cajón *j*)
  - asignar un nuevo cajón *z* y definir i_j y i_z para el viejo i_j -+ 1.
  - hacer que la segunda mitad de las entradas de la tabla de direcciones de los cajones que apuntan a *j*, lo hagan a *z*.
  - eliminar y volver a insertar cada registro en el cajón *j*.
  - recalcular el nuevo cajón para K_j e insertar un registro en el cajón (si el cajón está todavía lleno, son necesarias aún más divisiones)
- Si $i = i_j$ (sólo un puntero al cajón *j*)
  - incrementar *i* y doblar el tamaño de la tabla de direcciones de los cajones.
  - sustituir cada entrada en la tabla por dos entradas que apunten al mismo cajón.
  - recalcular la nueva entrada de la tabla de direcciones de los cajones para K_j
    Ahora $i > i_j$ por lo que emplear el primero de los casos anteriores.

Actualizaciones en la estructura de asociación extensible (Cont.)

- Cuando se inserta un valor, si el cajón se llena después de varias divisiones (es decir, *i* alcanza algún límite *b*) crear un cajón de desbordamiento en vez de dividir aún más la tabla de entradas del cajón.
- Para borrar un valor de la clave,
  - localizarlo en su cajón y eliminarlo.
  - El propio cajón se puede eliminar si se queda vacío (con actualizaciones apropiadas a la tabla de direcciones de los cajones).
  - Se pueden hacer agrupaciones de cajones (sólo se pueden agrupar con un cajón “compañero” que tenga el mismo valor de i_j y el mismo prefijo i_j –1,si está presente)
  - Es también posible disminuir el tamaño de la tabla de direcciones de los cajones
    - Nota: reducir el tamaño de la tabla de direcciones de los cajones es una operación costosa y sólo se debería hacer si el número de cajones se hace mucho más pequeño que el tamaño de la tabla

Uso de la estructura de asociación extensible: Ejemplo

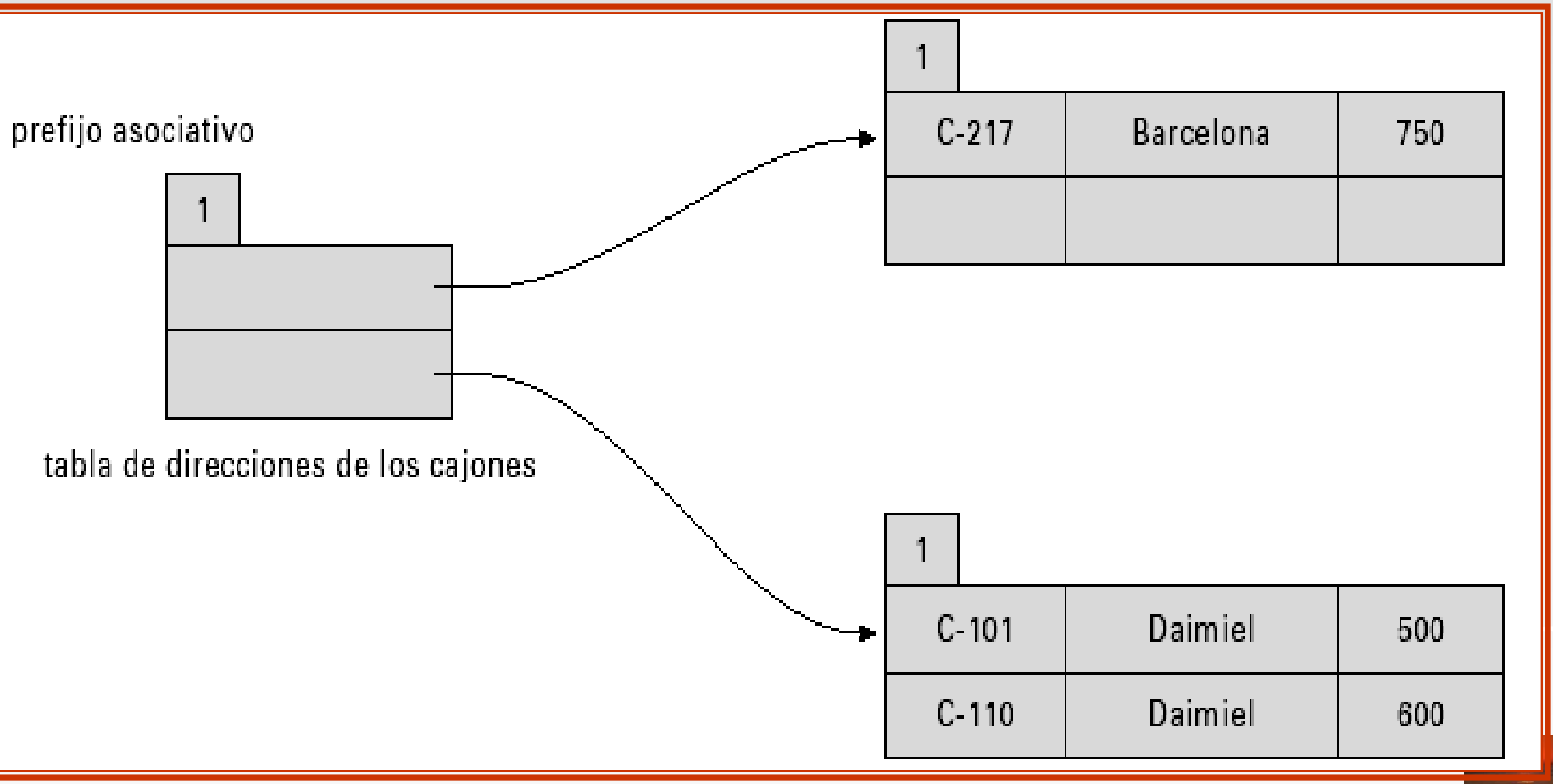
nombre-sucursal	*h(nombre-sucursal)*
Barcelona	0010 1101 1111 1011 0010 1100 0011 0000
Daimiel	1010 0011 1010 0000 1100 0110 1001 1111
Madrid	1100 0111 1110 1101 1011 1111 0011 1010
Pamplona	1111 0001 0010 0100 1001 0011 0110 1101
Reus	0011 0101 1010 0110 1100 1001 1110 1011
Ronda	1101 1000 0011 1111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



Estructura asociativa inicial, tamaño del cajón = 2

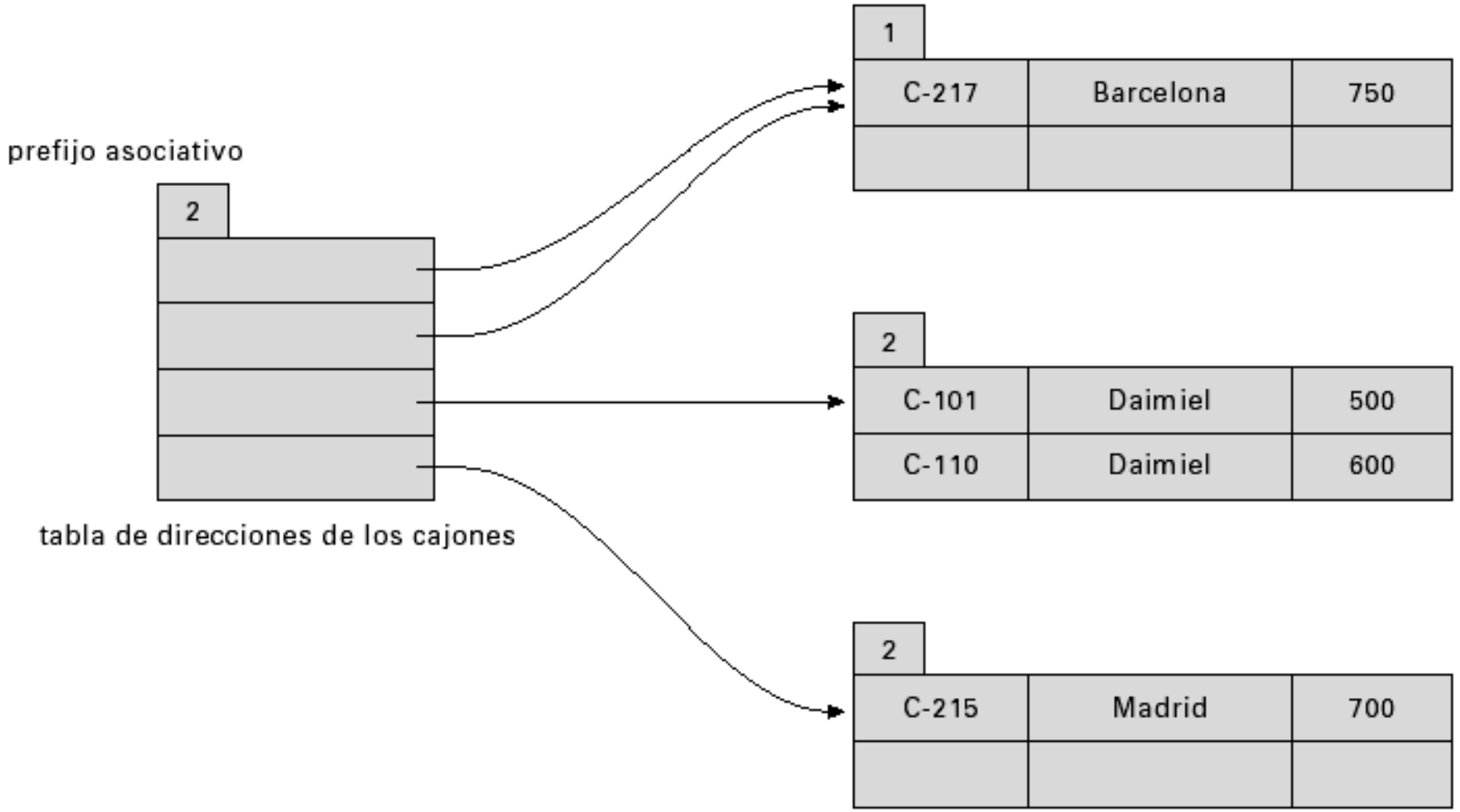
Ejemplo (Cont.)

- Estructura de asociación después de la inserción de un registro “Barcelona” y dos registros “Daimiel”

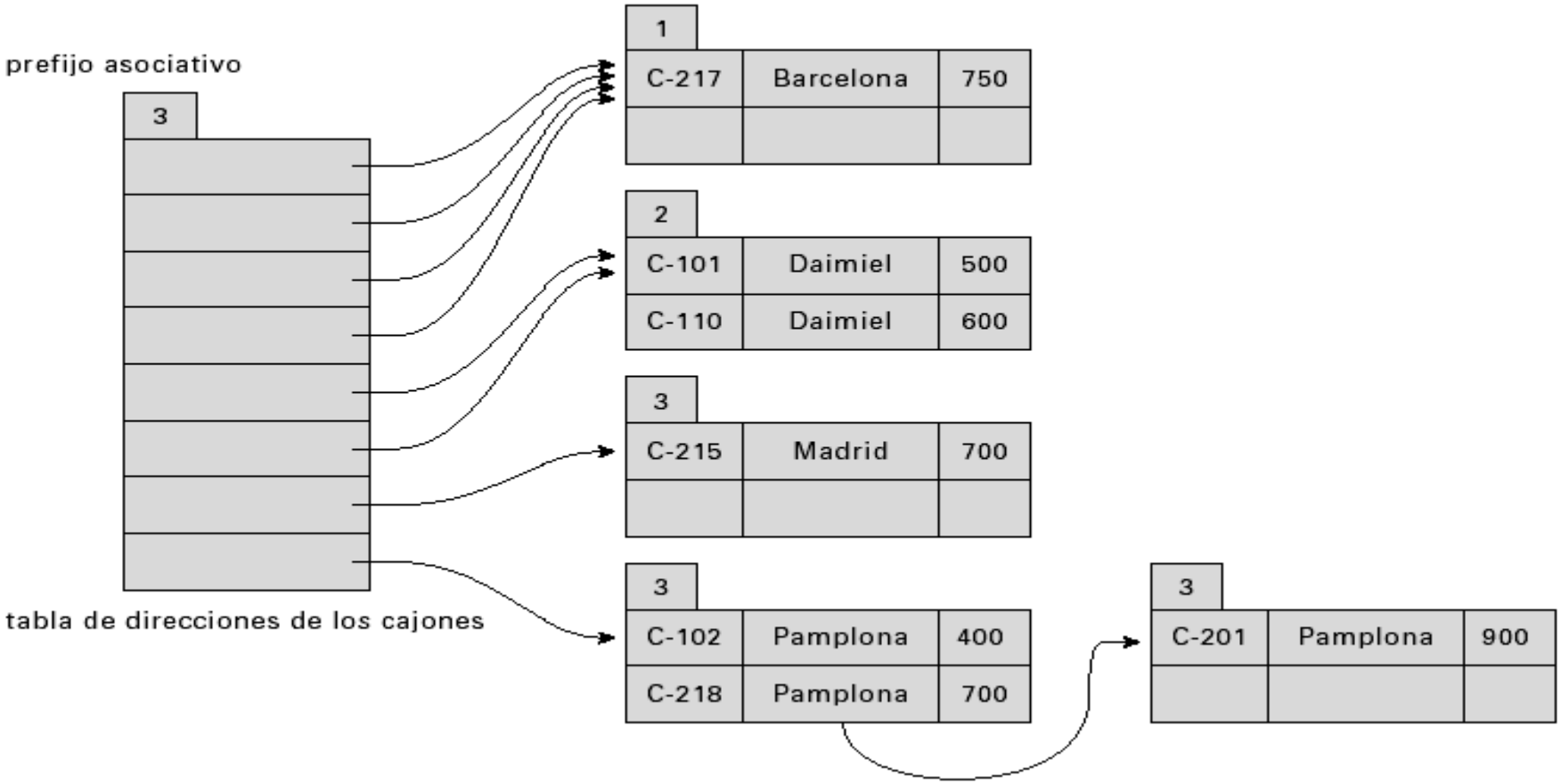


Ejemplo (Cont.)

Estructura de asociación después de la inserción del registro “Madrid”.



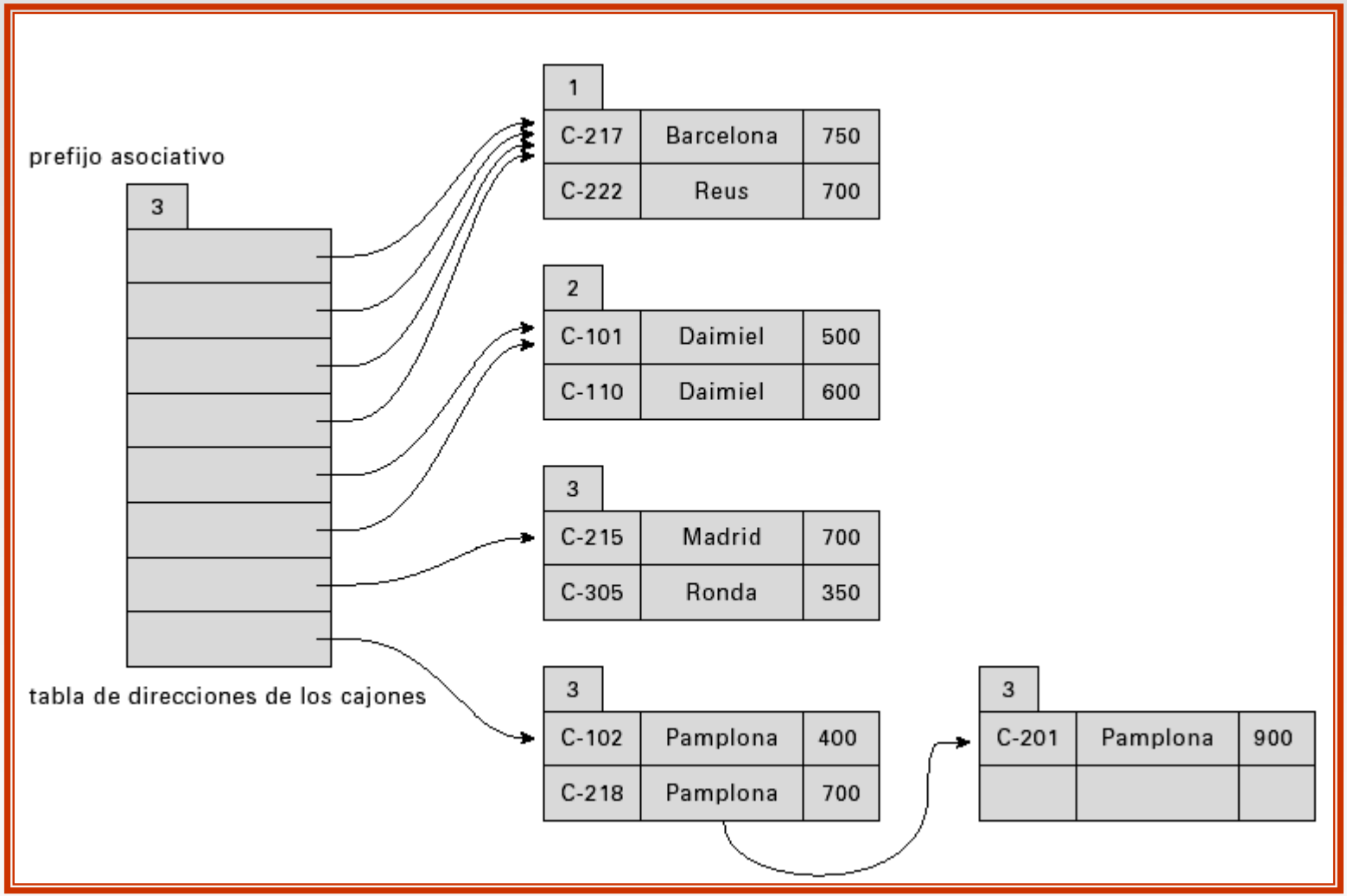
Ejemplo (Cont.)



Estructura de asociación después de la inserción de tres registros “Pamplona”.

Ejemplo (Cont.)

- Estructura de asociación después de la inserción de los registros “Reus” y “Ronda”



Asociación extensible vs otros esquemas

- Ventajas de la asociación extensible:
  - Las prestaciones de la asociación no disminuyen con el crecimiento del archivo
  - Costes de espacio mínimos
- Inconvenientes de la asociación extensible
  - Nivel extra de falta de dirección para encontrar el registro deseado
  - La tabla de direcciones de los cajones puede hacerse muy grande (más que la memoria)
    - ¡Necesita una estructura de árbol para localizar el registro deseado en la estructura!
  - El cambio del tamaño de la tabla de direcciones de cajones es una operación costosa
- La **asociación lineal** es un mecanismo alternativo que evita estos inconvenientes, al posible precio de más desbordamientos de cajones

Comparación entre indexación ordenada y asociación

- Coste de una reorganización periódica
- Frecuencia relativa de inserciones y borrados
- ¿Es deseable optimizar el tiempo de acceso medio, a costa del tiempo de acceso del peor de los casos?
- Tipo esperado de consultas:
  - La asociación es generalmente mejor para recuperar registros que tienen un valor determinado de la clave.
  - Si las consultas de rangos son comunes, es preferible que los índices estén ordenados

Índices de mapas de bits

- Los índices de mapas de bits son un tipo especial de índice, diseñado para consultas eficientes sobre claves múltiples
- Los registros en una relación se asume que se numeran secuencialmente desde, por ejemplo, 0
  - Dado un número *n* debe ser fácil recuperar el registro *n*
    - Particularmente fácil si los registros son de tamaño fijo
- Aplicable sobre atributos que toman un número relativamente pequeño de valores distintos
  - Por ejemplo, sexo, país, provincia, …
  - Por ejemplo, nivel de ingresos (ingresos descompuestos en un pequeño número de niveles tales como 0-9.999, 10.000-19.999, 20.000-50.000, 50.000- infinito)
- Un mapa de bits es simplemente un array de bits

Índices de mapas de bits (cont.)

- En su forma más simple, un índice de mapa de bits sobre un atributo tiene una mapa de bits por cada valor del atributo
  - El mapa de bits tiene tantos bits como registros
  - En un mapa de bits para el valor *v*, el bit para un registro es 1 si el registro tiene el valor *v* para el atributo, de lo contrario es 0

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

Índices de mapas de bits (cont.)

- Los índices de mapas de bits son útiles para consultas sobre atributos múltiples
  - no son particularmente útiles para consultas de atributos simples
- Las consultas son respondidas empleando operaciones de mapas de bits
  - Inserción (and)
  - Unión (or)
  - Complementariedad (not)
- Cada operación toma dos mapas de bits del mismo tamaño y aplica la operación sobre los correspondientes bits, para obtener el mapa de bits resultante
  - Por ejemplo 100110 AND 110011 = 100010

    100110 OR 110011 = 110111

    NOT 100110 = 011001
  - Hombres con nivel de ingresos: 10010 AND 10100 = 10000
    - Entonces se pueden recuperar las tuplas solicitadas.
    - Contar el número de tuplas equivalentes es incluso más rápido

Índices de mapas de bits (cont.)

- Los índices de mapas de bits son generalmente muy pequeños, comparados con el tamaño de la relación
  - Por ejemplo, si el registro es 100 bytes, el espacio para un solo mapa de bits es 1/800 del empleado por la relación.
    - Si el número de valores distintos de los atributos es 8, el mapa de bits es sólo el 1% del tamaño de la relación
- Es necesario gestionar adecuadamente los borrados
  - Mapa de bits existente para notar si hay un registro válido en una ubicación de registros
  - Necesario para la complementariedad
    - not(*A=v*): *(NOT mapadebits-A-v) AND mapadebitsexistente*
- Se deberían mantener mapas de bits para todos los valores, incluso para los valores nulos
  - Para manejar correctamente el SQL, semánticos nulos para NOT(*A=v*):
    - las intersecciones anteriores resultarán con (NOT *mapadebits-A-Null*)

Implantación eficiente de las operaciones de los mapas de bits

- Los mapas de bits están empaquetados en palabras; una sola palabra and (una instrucción de CPU básica) calcula and de 32 o 64 bits al mismo tiempo
  - Por ejemplo, 1 millón de mapas de bits se pueden añadir con sólo 31.250 instrucciones
- Contar el número de 1s se puede hacer rápidamente mediante un truco:
  - Emplear cada byte para indexar en un array precalculado de 256 elementos cada uno, almacenando el recuento de 1s en la representación binaria
    - Se pueden emplear pares de bytes para acelerar aún más, con un mayor coste de memoria
  - Sumar los contadores recuperados
- Se pueden emplear los mapas de bits en lugar de las listas de id-tupla en los niveles de las hojas de los árboles B^+, para valores que tienen un gran número de registros equivalentes
  - Merece la pena si >1/64 de los registros tienen ese valor, asumiendo un id-tupla de 64 bits
  - Las técnicas anteriores reúnen los beneficios de los mapas de bits y los índices de árboles B^+.

Definición de índice en SQL

- Crear un índice

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

  Por ejemplo: **create index** *índice_sucursal* **on** *sucursal(nombre_sucursal)*

- Emplear **create unique index** para, indirectamente, especificar e imponer la condición de que la clave de búsqueda sea una clave candidata.
  - No se requiere realmente si el SQL **unique** soporta restricciones de integridad
- Para borrar un índice

  drop index <nombre-índice>

Fin del capítulo