Organización física de los datos.[1]

6.1 El modelo físico de los datos.

La Base de Datos física es una colección de registros, cada uno consistente de uno ó más campos, cuyos valores son tipos elementales tales como enteros, reales, una cadena de caracteres de longitud fija o inclusive apuntadores; en algunas ocasiones otros tipos de datos, como las cadenas de caracteres de longitud variable, se tratarán como tipos elementales.

Archivo.
Para el modelo relacional una tupla equivale a un registro. Usaremos el término archivo para una colección de registros con el mismo formato. Así por ejemplo, un archivo es una adecuada representación física para una relación.

Almacenamiento de dos niveles.
El medio de almacenamiento en el que registros y archivos residen puede ser un gran arreglo de bytes numerados secuencialmente y que usan un disco como dispositivo físico.
Cuando la cantidad de datos en nuestra B. de D. es grande, el espacio de almacenamiento no puede ser únicamente la memoria principal; por el contrario, para hacer operativo al sistema, los datos deben ser movidos a memoria principal desde el disco y viceversa; esta transferencia en ambos sentidos es lenta, comparada con únicamente leer datos de memoria principal.

Bloques.
El medio de almacenamiento secundario se subdivide en bloques de algún tamaño, esta división es la unidad básica de transferencia de información entre memoria principal y memoria secundaria.
Es muy común que los registros sean mucho menores que los bloques, tal que frecuentemente muchos registros se almacenan en un solo bloque. Nuestro propósito es acceder el menor número posible de bloques cuando necesitemos leer varios registros de un archivo.

El costo de acceder la Base de Datos.
Definimos al número de **bloques accesados** como la unidad de costo para las operaciones sobre bloques físicos, bien sea para leer o escribir. Asumimos que es esta la operación más costosa, consideramos insignificante la computación que debe realizarse con los datos del bloque.
En realidad, no en toda ocasión que necesitemos un registro, requerimos leer el bloque que lo contiene desde el almacenamiento secundario; el Sistema Operativo ó el mismo D.B.M.S. (DataBase Management System) por medio de buffers en memoria principal, guarda copias de ellos mientras exista espacio suficiente. Pero no es posible predecir con certeza, cuando un bloque requerido estará o no en memoria principal, pues depende de factores que están por fuera de nuestro control, tal como otros trabajos que corran simultáneamente en el sistema u otras operaciones realizadas por otros usuarios sobre la B. de. Datos. Recíprocamente, el tiempo de acceso a disco para leer un bloque en particular, depende del último acceso realizado, pues cuenta el tiempo para mover

[1] Tomado y traducido de : Ullman, Jeffrey D. **Database and Knowledge-Base Systems.** Vol I. Cap. 6. pp: 294-331. Computer Science Press. 1988.
Traducido por : John Freddy Duitama Muñoz. Profesor U.de.A.

las cabezas de un cilindro a otro y el tiempo que toma el disco para rotar de una posición angular a otra. Algunos sistemas que manipulan gran cantidad de datos necesitan tomar en cuenta la secuencia exacta de acceso a los registros, para diseñar su disposición en el disco de una manera acorde con los requerimientos; pero esta situación es válida solo para operaciones de datos muy limitadas, comparadas con los lenguajes de manipulación de datos vistos por ejemplo en el modelo relacional. Por ello, no se analizará el acceso a los bloques a este nivel de detalle.
En resumen, asumimos como una probabilidad fija, la necesidad de usar un bloque que requiere transferencia de información entre memoria secundaria y memoria principal. También supondremos que el costo de acceso no depende de los accesos previos, lo que significa que es constante el costo de acceder cualquier bloque de nuestro archivo.

Apuntadores.

Un apuntador a un registro *r* es un dato suficiente para localizar dicho registro rápidamente. Un tipo obvio de apuntador , pero no el único, es una dirección absoluta en memoria virtual ó en disco al comienzo de *r*.
Sin embargo, las direcciones absolutas no son recomendables, pues para mover *r* dentro de un bloque ó dentro un grupo de bloques, sería necesario ubicar todos los apuntadores a *r* y modificarlos. Por ello se prefiere usar un par de apuntadores (*b*,*k*), donde *b* es el bloque donde se encuentra *r* y *k* es el valor clave para *r*; esto es , el valor del campo ó campos que sirven como clave en el archivo al que pertenece. Luego, para hallar un registro *r* en un bloque dado es suficiente con saber que:
1. Todos los registros en el bloque *b* tienen el mismo formato que el registro *r*; además, ningún otro registro puede coincidir en la clave con *r*.
2. El comienzo de todos los registros en el bloque puede ser hallado y
3. Cada registro en el bloque *b* puede ser descompuesto en sus campos, dado el comienzo del registro.

Registros Marcados (pinned).

Son los registros que tiene apuntadores a ellos desde lugares no conocidos. Cuando no tienen apuntadores se pueden mover libremente dentro del bloque o aún de bloque a bloque; si los registros están marcados, podemos solo moverlos dentro del mismo bloque si utilizamos como esquema el par (b,k) como apuntadores ; si usamos direcciones absolutas no podemos hacer ningún movimiento.
Otra restricción que enfrentan los registros marcados es que no pueden ser borrados completamente a menos que fuese posible borrar todos los apuntadores que los referencian. Si hay un apuntador *p* al registro *r* y un tiempo después es borrado *r*, podría posteriormente ubicarse un registro *r′* en el sitio que ocupaba *r*. Si el apuntador p es usado, puedo localizar una información que no corresponde realmente al registro buscado; pues es posible que *r′* tenga la misma clave que *r*, sin que se viole el principio de clave única, porque el registro *r* ya no existe en el archivo. Este problema es conocido como el de los apuntadores suspendidos y se evita adicionando a cada registro un bit , que valdrá uno si el registro es borrado, cero en los demás casos.

Organización de los registro.

Cuando acomodamos los campos de un registro, debemos hacerlo de tal manera que puedan ser fácilmente accesados. Si todos los campos tienen longitud fija, solo tenemos que definir su orden de almacenamiento; cada campo comienza en un número fijo de bytes, llamados *desplazamiento* (offset), después del comienzo del

registro; así, para todo registro con formato conocido, podemos hallar un campo apoyándonos en este desplazamiento.
Pueden existir dentro de cada bloque varios bytes, no dedicados a almacenar datos, pero requeridos para información de control, por ejemplo:
1. Algunos bytes nos dicen cual es el formato del registro. Si almacenamos registros de varios tipos de relaciones en el mismo bloque, será necesario almacenar un código indicando a que relación pertenece cada registro. Alternativamente, podemos almacenar sólo un tipo de relación en cada bloque; en este caso necesitamos sólo un indicador por bloque del tipo de información almacenada allí.
2. Uno ó varios bytes nos dicen qué largo es el registro. Si el registro es de longitud fija, el tipo nos indicará implícitamente este dato.
3. Un byte para indicar si el registro ha sido o no borrado.
4. Un bit de "usado / no usado", que puede ocupar un byte independiente o compartir el byte que informa si ha sido borrado el registro. Esta información es necesaria cuando los bloques son divididos en áreas, para almacenar en cada una un registro de longitud fija; necesitamos entonces saber cuando ellas efectivamente almacenan un valor.
5. Espacio malgastado. Podría no usarse cierta direcciones de disco, pues forzamos a que cada registro comience sólo en direcciones de cierta característica; p.e. muchas maquinas operan más eficientemente en enteros si son valores divisibles por 4. Asumiremos esta posibilidad aquí.

Ejemplo 6.1.

Supongamos un registro del tipo *NUMERO* que consiste de los campos:
1. Campo *Número*: Entero, que sirve de clave. Siempre contiene un número positivo.
2. Campo *Nombre*: Un byte indicando la primera letra del nombre en inglés para el número indicado.
3. Campo *Cuadrado*: Guarda el cuadrado de *número*. En este ejemplo igualmente de tipo entero.

Con la premisa que cada entero ocupa 4 bytes, el espacio total ocupado por los tres campos serán 9 bytes. Adicionaremos otro byte al comienzo del registro para almacenar la información de usado / no usado y de registro borrado, lo llamaremos el byte *INFO*.

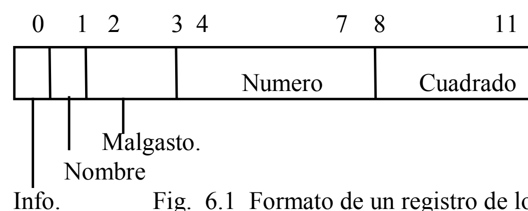


Fig. 6.1 Formato de un registro de longitud fija.

Vamos a suponer que cada campo entero debe comenzar en una dirección que es múltiplo de cuatro. Por ello, haremos un uso más eficiente del espacio, si escogemos el orden mostrado en la figura 6.1 para almacenar los campos, tendremos dos bytes de malgasto después de nombre, lo que nos lleva a utilizar 12 bytes por registro.

Registros de longitud variable.

Contiene campos cuya longitud puede variar; las estrategias de formato para estos registros son en general las dos siguientes:
1. Permitir que cada campo de longitud variable use al comienzo un contador que nos indique cuantos bytes requiere, o
2. Colocar al comienzo de cada registro apuntadores a cada campo de longitud variable; también un apuntador al final del último de tales campos. Es necesario

entonces almacenar primero todos los campos de longitud fija y a continuación los campos de longitud variable.
El esquema uno usa menos espacio, pero consume más tiempo para localizar un campo que esté después de los de longitud variable; pues solo puede calcular el desplazamiento de un campo si examina todos los campos de longitud variable previos. Usaremos en el siguiente ejemplo el esquema 1. El esquema 2 lo usaremos más adelante, no solo para almacenar campos dentro de registros sino también para almacenar los registros dentro de los bloques.

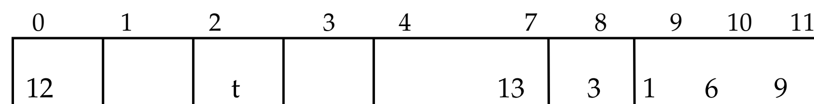
Ejemplo 6.2:

Consideremos el ejemplo 6.1., pero el campo *cuadrado* será una cadena de caracteres incluyendo los dígitos del número al cuadrado. Los bytes de este campo estarán precedidos por un byte que indica el número de bytes usados por *cuadrado*; de esta manera el string podrá contener entre 0 y 255 bytes. El formato de un registro sería entonces:

1. El byte 0 guarda la longitud completa del registro, incluyendo los campos de longitud variable. Así *cuadrado* será mucho menor que 255 bytes en longitud, pues el registro completo no puede tampoco superar este tamaño.
2. El byte 1 guarda los bits de *INFO*.
3. El byte 2 guarda al campo *nombre.*
4. El byte 3 es malgastado.
5. Los bytes del 4 al 7 almacenan al campo *número*.
6. El byte 8 guarda la longitud del campo *cuadrado*.
7. El byte 9 y siguientes guardan el valor de *cuadrado*, como una cadena de caracteres.

La figura 6.2 muestra el contenido de dos registros con este formato.

(a)Registro para el número 2.



Info Malgastado.

(b) Registro para el número 13.

Fig. 6.2. Registros de longitud variable.

Formato de los bloques.

Así como podemos localizar los campos dentro de un registro, debe ser posible localizar los registros dentro de un bloque. De la misma manera como cada registro requiere de un espacio para información de control, los bloques requieren de espacio extra para propósitos especiales. Por ejemplo, los bloques con frecuencia tienen en localizaciones fijas, apuntadores que los enlazan dentro de una lista de bloques.
Si los enteros y los apuntadores requieren de un byte específico para iniciar su almacenamiento, como asumimos en el ejemplo 6.1., entonces debemos ser cuidadosos al colocar los registros dentro del bloque. Si muchas variaciones son posibles, el esquema más simple es asumir que el desplazamiento de enteros y apuntadores

dentro de un registro es siempre divisible por cuatro, lo que implica que el registro deba empezar dentro de un bloque con un desplazamiento también divisible por cuatro. Los bloques a su vez comienzan en un byte que es potencia de dos, lo que implica que la dirección (el primer byte) de cada bloque también será divisible por cuatro, y así, todos los campos que requieren ser alineados arrancan con bytes divisibles por cuatro.
Si un bloque almacena registros de longitud fija, entonces tendremos únicamente que partir el bloque en tantas áreas, cada una guardando un registro, como se ajusten en el bloque. Si hay un campo especial que pertenezca al bloque, tal como un apuntador entre bloques, entonces ese espacio debe ser separado en un sitio fijo. El resto del espacio no usado, ni en registros ni como apuntador del bloque se considera malgastado.
Ejemplo 6.3.:
Asumimos bloques de longitud 64 bytes, tamaño mas pequeño que el real, pero que nos permite facilitar los cálculos. Suponga entonces registros de longitud fija como el ejemplo 6.1., y además un apuntador de 4 bytes de longitud, para usarse como enlace entre bloques. En la fig. 6.3 vemos la disposición de los registros sin pérdida de espacio.

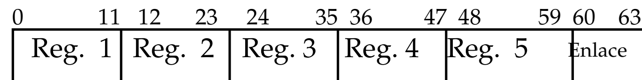


Fig. 6.3 Formato de bloque con registros de longitud fija.

Se asume que el bit usado/no usado aparece en cada registro, así que podemos hallar un área para insertar un registro si existe un espacio en el boque. Este esquema consume tiempo, pues se debe visitar cada registro para verificar si está usado. Alternativamente, podemos colocar este bit para cada área de registro en uno ó varios byte al comienzo del bloque; facilitando la búsqueda de espacio vacío para insertar nuevos registros.
Ejemplo 6.4.
Para el formato del ejemplo 6.3. , podemos separar un área alternativa para el bit de usado / no usado, en lugar de hacerlo en cada registro. Dada la restricción de alineación de los campos en direcciones múltiplos de 4, la longitud del registro debe mantenerse en 12 bytes; además, aunque en esta señalización sólo ocupemos el byte cero del bloque, solo pueden empezar los registros desde el byte 4. Como adicionalmente se requieren cuatro bytes para el enlace, entonces solo pueden almacenarse cuatro registros por bloque en los bytes 4-15,16-27,28-39 y 40-51. Los bytes 1-3 y 52-59 son malgastados.

Bloques con registros de longitud variable.
Si acomodamos en un bloque sin campos de ayuda, registros de longitud variable, todavía es posible localizar el inicio de todos los registros. Asumimos que el primero arranca desde el byte cero, hallamos la longitud de este, incrementamos su valor hasta el siguiente múltiplo de cuatro y hemos localizado el inicio del segundo registro; y así sucesivamente para los demás. Este es un método costoso.
Una estrategia más eficiente es colocar al comienzo del bloque un directorio, que consiste de un arreglo de apuntadores a los varios registros que contenga. Estos apuntadores son realmente desplazamientos en el bloque; pueden existir varias maneras de representarlos, por ejemplo:
1. Comenzar el directorio con un byte que indica cuantos registros hay.

2. Usar un número fijo de campos al comienzo del bloque para apuntar a los registros. Los campos no ocupados, por existir pocos registros en el bloque,se llenan con ceros; en este caso no representarían el desplazamiento.
3. Usar un número variable de campos para apuntar a los registros, con el último campo a usar marcado con cero, actúa como una marca de fin en la lista de apuntadores.

Ejemplo 6.5.:
La fig. 6.4. muestra un bloque que contiene registros de longitud variable en el formato del ejemplo 6.2. El esquema para manejar los apuntadores puede ser el 2 ó el 3, pues únicamente el último de los campos del directorio contiene un cero. Los tres registros mostrados almacenan los números 2,13 y 100 con longitud de 10, 12 y 14 respectivamente. Note que los bytes 26-27 se malgastan, pues necesitamos empezar el segundo registro con un desplazamiento múltiplo de cuatro. Los bytes 54-59 se pierden, pues no existe un registro con el formato propuesto que se acomode allí.

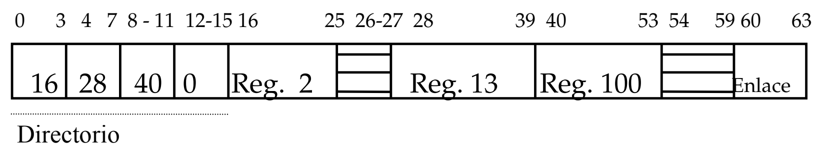


Fig. 6.4. Formato de un bloque con registros de long. variable.

Es posible ser más económicos al almacenar el desplazamiento de los registros; en lugar de utilizar cuatro bytes, ocupar solo uno; al ser el tamaño de bloque pequeño, los desplazamientos son números entre 0 y 63. En realidad, aún en bloques de longitud 1024, valor más común, podríamos también almacenar el desplazamiento en un solo byte; asumiendo que este valor debe ser divisible por cuatro y almacenandolo dividido entre cuatro.

Registros semi-marcados.
Otra razón para adoptar el esquema de la fig. 6.4. es que permite “desmarcar” registros marcados. Si existe, desde fuera del bloque, un apuntador a un registro *r*, lo dirigimos hacia el campo en el directorio que almacena el desplazamiento del registro. Podemos entonces mover *r* en el bloque y cambiar el desplazamiento en el directorio, sin desfasar la referencia del apuntador externo.
Cuando se almacenan registros de longitud variable, se presenta con frecuencia la necesidad de mover estos registros dentro del bloque. Por ejemplo, los datos de un registro pueden crecer o decrecer, producto de actualizaciones. Lo que hace necesario separar espacio para desplazar todos los registros a la derecha o consolidar espacio moviéndolos a la izquierda. El número de registros de un campo puede cambiar producto de nuevas inserciones, provocando la necesidad de crear nuevos campos en el directorio del bloque y desplazar todos los registros para separar espacio.
Otra ventaja del esquema de la figura 6.4. es que permite borrar registros marcados, para ello movemos, asumiendo que hay espacio, el bit de “borrado” desde el registro al directorio del bloque. Entonces, si deseamos borrar un registro *r*, será posible reutilizar el espacio; al marcar en el directorio el registro como borrado, si alguien vía el apuntador externo requiere tal registro, será posible detectar que no existe allí. Obviamente la entrada al directorio se dedicará permanentemente al registro borrado, pero es preferible esta opción a mantener ocupado el espacio del registro, si es de gran tamaño.

6.2. La organización del Montón. (heap).

Vamos a considerar las estructuras de datos que son útiles para los índices primarios, esto es, para las estructuras que determinan la localización de los registros dentro del archivo. Generalmente el índice primario está basado en la clave del archivo y la localización del registro es determinada por este valor. Por medio de su clave entonces un registro puede ser hallado rápidamente.

Como una base para medir desempeño se asumirá la organización más simple del montón, donde los registros, sin guardar ningún orden especial, son almacenados en bloques; a su vez los bloques no tienen ningún tipo de ordenamiento. Asumiremos que hay disponibles apuntadores a todos los bloques del archivo y que los encontramos en memoria principal. Si son demasiados los apuntadores se almacenan en disco (igualmente en bloques) y se leen a memoria cuando sea necesario.
Asumiremos cuatro operaciones básicas :
1. Consulta : Dado un valor "clave", hallas el (los) registro(s) con ese valor clave. Hablamos de clave sin la restricción de unicidad.
2. Inserción: Agregar un registro al archivo. Asumimos que sabemos que el registro no existe ya en el archivo ó que no estaremos pendientes de tal evento. Si deseamos evitar duplicidad de registros la inserción va precedida de una consulta.
3. Borrado : Borrar un registro del archivo. Asumimos que no conocemos de antemano la existencia o no del registro, por lo que previamente requerimos de una consulta.
4. Modificación : Cambiar los valores de uno o más campos de un registro. A fin de modificar un registro, debemos primero consultarlo.

Eficiencia del montón.

Suponga n registros a almacenar y que R es el número de registros que se acomodan en un bloque. Si los registros están marcados y los registros borrados no permiten reutilizar el espacio, entonces n será el número de registros que han existido en el tiempo; de otra manera, n es el número de registros que hay en ese momento en el archivo. Si los registros son de longitud variable, entones R será, en lugar de un número exacto, el promedio de registros que se acomodan en un bloque. Luego, el número mínimo de bloques necesarios para almacenar un archivo es:

$$\lceil n/R \rceil$$

Como normalmente n es mucho mayor que R, entonces $\cong n/R$.

Recordemos que *el costo de las operaciones como insertar, borrar ó consultar está medido en el número de bloques que deben ser recuperados ó almacenados entre memoria secundaria y principal*. Asumiremos, por uniformidad entre todas las estructuras de datos usadas, que inicialmente todo el archivo usado está en memoria secundaria. Para obtener un registro del montón, dada una clave, debemos recuperar $n/2R$ bloques en promedio, hasta hallar el registro; si no hay registros con la clave buscada entonces debemos recuperar todos los n/R bloques.
Para insertar un nuevo registro, tenemos que recuperar el último bloque del montón que suponemos tiene espacio libre. Si el último no tiene más espacio, debemos separar un nuevo bloque. A continuación debemos escribir el bloque a disco. Así, la inserción toma dos accesos al bloque, una para leer y otra para escribir.

El borrado requiere hallar el registro, p.e. ejecutar una consulta y luego rescribir el bloque, para un total de $(n/2R) + 1$ accesos en promedio, cuando el registro es hallado y n/R accesos cuando no lo es. Si los registros son "marcados", entonces el proceso de borrado consiste en marcar el bit de borrado en 1. Si los registros no son marcados se tiene la opción de liberar el espacio que ocupa el registro.
Para el borrado desde archivos con registros de longitud fija, se puede mover un registro del último bloque del archivo al área del registro borrado. Con suerte, se puede liberar el último bloque si el trasladado es su último registro. En general, compactar un archivo es un modo de minimizar el número de bloques utilizados por este, lo que a su vez reduce el costo de futuras consultas y borrados.
También se pueden compactar la información si los registros son de longitud variable. Si usamos un formato como el de la fig. 6.4., podemos mover los registros dentro del bloque, asegurándonos que los apuntadores a esos registros, que se encuentran al comienzo del bloque, continúen apuntando a ellos. Si creamos suficiente espacio en un bloque podemos entonces mover un registro desde el último de ellos. Sin embargo, si los registros varían en longitud, es prudente guardar espacio libre en cada bloque, de tal manera que cuando registros de ese bloque crezcan, sea poco probable el tener que moverlos a otro por falta de espacio.
Finalmente, las modificaciones toman un tiempo similar a los borrados. Necesitamos n/2R accesos a bloques para encontrar el registro, seguido de una escritura del bloque conteniendo el registro modificado. Si los registros son de longitud variable, puede desearse o requerirse leer y escribir unos bloques más para consolidar registros (si el registro modificado es más corto que el original), o hallar otro bloque en el que se pueda acomodar el registro modificado, si este crece.

Ejemplo 6.6. :
Suponga un archivo de 1.000.000 de registros de 200 bytes cada uno. Suponga todos los bloques de 2^{12} = 4096 bytes de longitud. Entonces R = 20, por lo que podemos acomodar 20 registros por bloque. Así, una consulta exitosa toma n/2R =25.000 accesos a bloques y una no exitosa toma 50.000 accesos. Estimando la recuperación de un bloque desde disco en .01 segundo, una consulta exitosa toma cuatro minutos. El tiempo de modificaciones y borrados es esencialmente el mismo que para consultas. Solo la insercion, que asume que no requiere buscar antes en el archivo, es la Unica "rapida", esto es .01 segundo.
El directorio de bloques toma una significativa cantidad de espacio, quizás tanto que no sea plausible mantenerlo en memoria principal. Suponga las direcciones de bloque de 4 bytes, necesitamos 20.000 bytes o 50 bloques para almacenar las direcciones de todos los bloques.

6.3. Archivos aleatorios.

La idea básica de esta organización es dividir los registros del archivo de acuerdo al valor de su clave, para asociarlos a distintas áreas de almacenamiento (buckets). Para cada archivo almacenado de esta manera, hay una *función hash* $h(v)$, que toma como argumento un valor de clave y produce un entero en el rango de 0 a $B-1$, donde B es el número de áreas de almacenamiento usados para ese archivo; $h(v)$ produce un entero con el número de área de almacenamiento para v. No requerimos que v sea un valor único en el archivo, aunque por razones de eficiencia es deseable que no existan muchos registros con el mismo valor de clave.
Cada área de almacenamiento consiste de un número, presumiblemente pequeño, de bloques y los bloques de cada área están organizados como un montón. Hay un arreglo de apuntadores indizados desde 0 a $B-1$, que llamamos el directorio de áreas de

almacenamiento. La entrada para *i* en el directorio de áreas, es un apuntador al primer bloque para el área *i* , lo llamamos apuntador al encabezado del área. Todos los bloques del área *i* están ligados en una lista a través de apuntadores, un apuntador nulo señala o el último bloque de la lista, cuando está llena el área, o al encabezado si al momento está vacía. Lo común para B es que sea lo suficientemente pequeña para hacer posible que el directorio de áreas pueda residir completamente en memoria principal; si no es el caso, el directorio ocupa tantos bloques como requiera y cada uno será llevado a memoria principal cuando sea necesario.

Funciones hash.

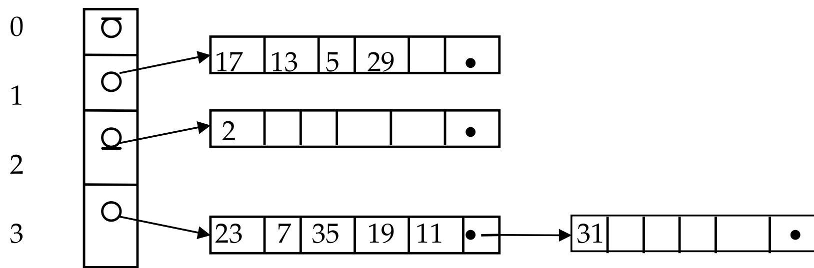
Hay muchas clases de funciones que pueden utilizarse como *h*. Lo esencial es el rango entre 0 y B-1 y que la función disperse las claves; esto es, todos sus valores tengan igual probabilidad de ocurrir.
Una función hash simple convierte cada clave en un entero y toma el residuo de ese entero módulo B. Si las claves son enteras, simplemente $h(v) = v \bmod B$. Si la clave es una cadena de caracteres, se divide la cadena en grupos de caracteres, quizás uno ó dos por grupo, se trata a los bit representando cada grupo de caracteres como un entero y se suman.
Si la clave consiste de varios campos, convierte el valor de cada campo a un entero, por un método como los mencionados arriba, toma su suma y divide el resultado por B. Existen otras alternativas para producir números aleatorios que son fáciles de obtener. (véase Knuth 1973)[2].

Ejemplo 6.7.

En la figura 6.5. vemos un archivo de registros de *números,* con el formato del ejemplo 6.1., organizado como un archivo disperso en cuatro áreas, B = 4. Asumimos los parámetros del ejemplo 6.3., esto es, registros de 20 bytes de longitud, con hasta cinco de ellos y un apuntador enlace ubicados por bloque. Hemos almacenado un conjunto de números primos usando la función hash $h(v) = v \bmod 4$.

Almacenar claves que son números primos es una de las pocas cosas tontas que uno puede hacer con una función hash, que escoge el área de almacenamiento tomando el residuo de dividir la clave entre el número de áreas. Nunca quedará un primo en el área cero, excepto para B mismo si es un número primo. En la figura, nunca habrá un primo para el área cero y solo el primo 2 pertenece al área 2. Así los demás registros se acomodarán en las áreas 1 y 3.

[2] Knuth, D. E. *The Art of Computer Programming*, Vol 3, *Sorting and Searching*, Addison-Wesley, Reading Mass.

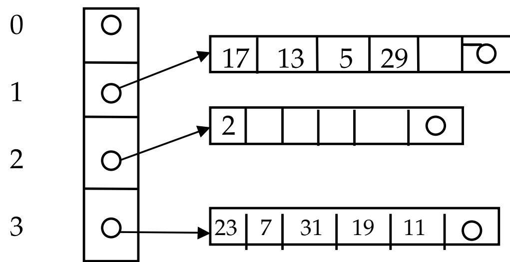


Directorio
de área de
almacenamiento.

Fig. 6.5. organización de archivos aleatorios.

<u>Operaciones en archivos aleatorios</u>.
<u>Para consultar</u> un registro con valor clave v, computamos h(v) , hallamos el encabezado del área de almacenamiento para ese valor calculado y examinamos la lista de bloques en ella como si fuese un montón. Si el registro deseado no es hallado, no hay lugar a examinar nuevas áreas, pues los registros con ese valor hash no pueden estar en otro lugar. El proceso de búsqueda no depende de que exista sólo un registro con clave v, si hay más de uno debemos escoger entre buscar los demás en la misma área ó solo tomar el primero.
<u>Para insertar</u> un registro de clave *v*, calculamos *h(v)* y ubicamos tal área; Presumiblemente, sólo el último bloque del área tiene espacio, debemos localizarlo entonces. Podríamos buscar desde el header del área a través de la lista de bloques, pero hay dos situaciones en las que no es necesario:
1. La inserción fue precedida por una búsqueda en el área de almacenamiento para chequear que no existiera tal registro. En ese caso, se está ya en el último bloque del área.
2. La estructura de datos (el directorio) de la figura 6.5. es mejorada por un arreglo de apuntadores al último bloque de cada área. Entonces puedo hallar el bloque deseado directamente.
Después de hallar el último bloque , si hay espacio insertamos el registro allí; si no lo hay, debemos obtener otro bloque y enlazarlo al final de la lista para el área *h(v).*
<u>Los borrados</u> se efectúan similarmente. Hallamos el registro a ser borrado con una consulta. Si los registros no están marcados, tenemos la opción de compactar los bloques del área, como se hizo con el montón de la sección previa. Hacerlo puede reducir el número de bloques necesarios para esta área, reduciendo el promedio de bloques accesados en futuras operaciones.
<u>Ejemplo 6.8</u>.
Descubrimos que 35 no es un número primo, tal que vamos a borrarlo de la estructura que aparece en la fig. 6.5. Calculamos h(35) = 35 mod 4 = 3; miramos en el área 3 y hallamos el registro para el 35 en el primer bloque en la lista. Asumiendo registros no marcados, vamos al último (el segundo) bloque del área y movemos el único registro de éste al tercer campo del primer bloque. En este caso el segundo bloque queda vacío, por ello lo removemos del área 3, dejando la situación como aparece en la figura 6.6.

Finalmente para <u>las actualizaciones,</u> primero se lleva a cabo una consulta y luego se cambian el o los campos necesarios. Si los registros son de longitud variable, existe la posibilidad de que tengan que ser movidos entre bloques del área hash, como se discutió en las conexiones con el montón.



Directorio de área.

Fig. 6.6 Efectos de borrar 35.

Eficiencia del hashing.

El punto central es que un archivo aleatorio con B áreas se comporta como si fuese un montón de aproximadamente 1/B de tamaño; luego, podemos agilizar las operaciones por un factor B que deseemos. Las consideraciones límite son:

1. Cada área debe tener por lo menos un bloque, así que sin importar qué grande sea B, requerimos de mínimo un acceso para cada consulta.

2. Debemos almacenar el directorio de áreas, bien sea en memoria principal ó en bloques del almacenamiento secundario. Haciendo B demasiado grande, forzamos el uso del almacenamiento secundario e incrementamos el número de bloques accesados por una operación. Para recuperar un bloque necesitamos del encabezado del área.

Si tenemos un archivo de n registros, de los que se ajustan R por bloque y usamos una organización aleatoria con B áreas, cuyos encabezados se almacenan en memoria principal, requerimos en promedio:

a. $\lceil n/2BR \rceil$ accesos para una consulta exitosa o para el borrado ó modificación de un registro existente.

b. $\lceil n / BR \rceil$ accesos para una consulta no exitosa, el chequeo de un registro que no existe en el archivo (durante el intento de borrar un registro que no existe ó durante el chequeo previo a una inserción.)

La razón de esta relación es que en promedio cada área almacena n/B registros y aplicamos el análisis para el montón de la sección previa. Debe tenerse presente que este estimativo asume registros aleatorios en nuestro archivo; si la función hash no distribuye los registros uniformemente entre todas las áreas ó si por mala suerte nuestro archivo tiene una colección de registros atípica, entonces el promedio de accesos por operación podría elevarse considerablemente.

A los cálculos anteriores hay que agregar uno más si el directorio de áreas no está en memoria principal y otro adicional para operaciones que requieren modificar y escribir alguno de los bloques del área. Si los registros son de longitud variable, y puede ser necesario mover registros entre bloques del área, entonces una fracción de acceso por operación deberá ser agregada.

Ejemplo 6.9. :

Consideremos el archivo con n = 1.000.000 y R = 20 discutido en el ejemplo 6.6. Si escogemos B = 1000 entonces el área promedio tiene n / B = 1000 registros que deberían estar distribuidos sobre n / BR = 50 bloques. Con la premisa de que las direcciones de los bloques requieren de 4 bytes, el directorio de áreas requiere de 4.000 bytes y puede ser mantenido fácilmente en memoria principal. Una operación que requiera examinar un área completa, tal como la consulta de un registro que no exista,

requiere 50 accesos a bloques más una adicional si requiere escribir en un bloque, p.e., si la operación es una inserción precedida de un chequeo de no existencia previa del registro. Operaciones que en promedio solo recorran la mitad del área se espera requieran entre 25 y 26 accesos. Usando el estimado previo de .01 segundos por acceso, cualquier operación puede realizarse en menos de un segundo.

6.4 Archivos indizados.

Llamados frecuentemente *isam*, *Indexed sequential access method*. Asumimos que cada clave pertenece a un único registro del archivo y no como en el hashing que determina un pequeño número de estos. (Aunque en la práctica puede ocurrir que equivale a varios registros).
Para esta representación se requiere clasificar los registros del archivo por su valor clave;

Clasificación de claves.
En principio, sin importar el dominio de los valores para un campo, debe ser posible comparar valores del mismo dominio y por ello debe ser posible clasificarlos. El almacenar los archivos como cadenas de bits, que pueden ordenarse si las tratamos como enteros y usamos los criterios de ordenamiento de los números, es lo que hace posible su clasificación. Los dominios usuales como las cadenas de caracteres, los enteros y los reales tienen ya un ordenamiento establecido. Para los enteros y los reales tenemos un ordenamiento numérico, para las cadenas de caracteres tenemos ordenamiento lexicográfico, definido por $X_1X_2 \ldots X_k < Y_1Y_2 \ldots Y_m$, donde las X's y las Y's representan caracteres, si y solo si :
1. k < m and $X_1 \ldots X_k = Y_1 \ldots Y_k$ o
2. Para algún i <= Min(k,m) , tenemos X1 = Y1, X2 = Y2,... $X_{i-1} = Y_{i-1}$ y el codigo binario para Xi en menor numéricamente que el código binario para Yi.
Si tenemos claves de más de un campo, podemos ubicar arbitrariamente estos campos dentro de la clave y a continuación proceder a ordenarlas. Los registros se clasifican por el primer campo, de lo que resulta una secuencia ordenada de grupos (cluster); cada grupo consiste de registros con el mismo valor en el primer campo. A continuación cada grupo se ordena por el segundo campo clave, lo que conforma nuevos grupos de registros coincidentes en los valores de sus dos primeros campos clave. Estos últimos grupos se ordenan por el tercer campo y así sucesivamente. Este tipo de ordenamiento es una generalización del ordenamiento lexicográfico para cadenas de caracteres, pero en este caso asumimos dominios arbitrarios.

Acceso de archivos clasificados.
Si deseamos mantener un archivo de registros organizados por su valor clave, podemos tomar ventaja del orden conocido para, dado un valor clave, hallar rápidamente el registro. Dos casos muy familiares para el lector son la búsqueda en un diccionario y en el directorio telefónico; en ambos casos, cada página tiene en la esquina superior izquierda la primera palabra ó nombre de la página. Rastreando esta primera palabra podemos obtener la página en la que nuestra palabra (si es un diccionario) ó teléfono (si es un directorio) se encuentra. Esta estrategia es mucho mejor que mirar toda entrada en todas las páginas. Excepto para una pagina, la que debemos rastrear completamente, cuando requerimos mirar solo una entrada en ella.
Para ayudar a accesar un archivo clasificado, que llamamos archivo principal, creamos un segundo archivo llamado (esparcido) *índice*, consistente de pares:

(<valor clave>,<dirección del bloque>)

Para cada bloque *b* del archivo principal hay un registro *(v,b)* en el archivo de índices; *v* es una clave de un valor tan bajo como cualquier otro valor clave en *b*, pero más alto

que cualquier valor clave en el bloque que precede a *b*. Con frecuencia escogemos a *v* como la menor clave almacenada en el bloque *b* ó por lo menos menor que cualquier clave que en el tiempo permanezca en dicho bloque. También es conveniente usar $v = -\infty$ si *b* es el primer bloque, donde $-\infty$ es un valor menor que cualquier clave del archivo principal.
El primer campo del par *(v,b)* es la clave para el archivo de índices; a su vez, este archivo de índices está clasificado por su valor clave.
En un sentido, un archivo de índices es otro archivo ordenado por un valor clave, pero con la ventaja que sus registros nunca están marcados por apuntadores almacenados en otro sitio. Visto desde otro ángulo, hay una diferencia importante entre el archivo *índice* y los archivos discutidos hasta ahora; adicionalmente a realizar inserciones, borrados ó modificaciones al valor clave de tal archivo, debemos obtener respuesta a preguntas de la forma: Dado un valor clave *v1* para el archivo principal, hallar el registro *(v2,b)* en el archivo *índice*, tal que $v2 <= v1$ y entonces :
1. *(v2,b)* es el último registro en el índice ó
2. El siguiente registro *(v3,b')* cumple que $v1 < v3$.
Se dice que *v2 cubre a v1*.
En otras palabras, cómo hallar el bloque *b* del archivo principal que contiene un registro con valor clave *v1*, asumiendo que se garantiza mantener ordenado al archivo de índices.
Búsqueda de un índice.
Asumamos que un archivo de índices es almacenado en su propia colección de bloques y que debemos hallar al registro *(v2,b)*, tal que *v2* cubre un valor *v1* dado. La estrategia más simple es usar una búsqueda lineal; recorrer el índice desde el comienzo, mirar cada registro hasta encontrar uno que cubra a *v1.* Este método la mayoría de las veces no es recomendable, excepto para archivos índice pequeños, que por ejemplo se acomoden en memoria principal. En promedio, la mitad de los bloques del índice deberán ser mirados, antes de encontrar el *v2* requerido. Esta estrategia de búsqueda lineal en un archivo índice es superior a una búsqueda lineal en el archivo principal; Si este último tiene R registros por bloque, el archivo índice tiene solo *1/R* parte de registros que el principal. Además, los registros del índice son mucho más pequeños que los registros del principal, lo que permite ubicar más registros por bloque.
Búsqueda binaria.
Una mejor estrategia es usar la búsqueda binaria para hallar la clave en el archivo índice. Suponga que $B_1,...,B_n$ son los bloques del archivo índice (no del archivo principal) y $v_1,...,v_n$ son las primeras claves halladas en $B_1...B_n$ respectivamente. Para Buscar el bloque del archivo principal donde un registro de clave *v* puede ser hallado, en primer lugar recuperamos el bloque $b_{[n/2]}$ del archivo de índices y comparamos *v* con la primera clave *w* de tal bloque.

- Si $v < w$ repetimos el proceso, como si el índice solo estuviese en los bloques B_1 hasta $B_{[n/2]-1.}$
- Si $v >= w$ repetimos el proceso como si el índice solo estuviese en los bloques $B_{[n/2]}$ hasta B_n.

En un momento determinado sólo un bloque permanece disponible, entonce usamos la búsqueda lineal en él para hallar el valor clave en el índice que cubre a *v*. Allí encontramos un apuntador al bloque B del archivo principal asociado con este valor clave y si existe un registro con ese valor clave, estará en tal bloque B.
En cada paso de la búsqueda dividimos por dos el número de bloques, luego en a lo sumo $\lceil \log_2(n+1) \rceil$ pasos ejecutamos la búsqueda del bloque índice. Así la búsqueda binaria en un archivo índice requiere que cerca de $\log_2 n$ bloques sean traidos a memoria principal. Una vez localizado el índice, sabemos exactamente cuál bloque del

archivo principal debe ser examinado y quizás reescrito según la operación ejecutada sobre el archivo. El número total de bloques leidos es cerca de $2 + \log_2 n$, cifra no muy costosa, como veremos.

Ejemplo 6.11.

Consideremos el archivo hipotético del ejemplo 6.6 cuyo tamaño es 1.000.000 de registros. Asumimos cada bloque de tamaño 4.096 bytes y registros de 200 bytes de longitud. Como la longitud de la clave importa para nuestros cálculos, asumimos que el o los campos claves utilizan 20 bytes. R = 20, es decir, 20 registros se acomodan por bloque en el archivo principal, luego este utiliza 50.000 bloques. También necesitamos saber el número de registros por bloque en el archivo índice.

Un registro índice usa 20 bytes por clave y 4 bytes para el apuntador a un bloque del archivo principal. Luego, 170 registros de 24 bytes puedo acomodar por bloque, pero significa que no estamos dejando espacio para caracteres de control y espacio no usado; supongamos entonces 150 registros por bloque para el archivo índice. Requerimos 50.000/150= 334 bloques para dicho archivo. Esto es, n = 334 en nuestro ejemplo.

La búsqueda lineal requiere en promedio, cerca de 168 bloques de índices accesados para una búsqueda exitosa y adicionalmente dos accesos de lectura y escritura a un bloque en el archivo principal. Por otro lado, si usamos búsqueda binaria, leer y escribir un bloque del archivo principal requiere $2 + \log_2 334 \cong 11$ bloques accesados. Comparativamente, una organización aleatoria requiere sólo 3 accesos en promedio, asumiendo que usamos tantas áreas como bloques tiene el archivo principal (un acceso para leer el directorio de áreas y dos accesos para leer y escribir un único bloque del área).

Sin embargo, hay algunas ventajas de usar una organización clasificada en lugar de archivos aleatorios. Para hallar la respuesta a la búsqueda de los registros cuya clave esté en un rango de valores, será necesario examinar casi todos las áreas de la tabla hash, si el rango fuera de tal naturaleza que la tabla hash ofrezca poca ayuda. Por otro lado, la organización clasificada nos permite examinar, casi que exclusivamente, los bloques del índice y del archivo principal que contengan registros relevantes para nuestra búsqueda. El único trabajo extra que tenemos que hacer al usar la búsqueda binaria, se da mientras se obtiene el primer bloque relevante de los índices y el mirar algún registro por fuera del rango deseado en el primero y el último bloque leídos tanto en los índices como en el archivo principal.

Búsqueda por interpolación.

Este método puede ser superior que la búsqueda binaria, pero se apoya en el conocimiento estadístico de la distribución esperada de los valores clave y en el que sea altamente confiable esta distribución. Por ejemplo, si buscamos a John Smith en el directorio telefónico, no abrimos el directorio en la mitad, sino en un 75% de sus páginas, apoyados en nuestro conocimiento aproximado de donde hallar la 'S'; si quedamos ubicados en la 'T' , nos retornamos quizás un 5%, de ninguna manera hasta la mitad como se hace con una búsqueda binaria.

En general, supongamos que tenemos un algoritmo, que dado un valor clave *v1*, nos dice en que fracción del camino entre los valores claves *v2* y *v3* puedo encontrar a *v1*. Llamamos a esta fracción *f(v1,v2,v3).* Si un índice ó parte de él se encuentra entre los bloques $B_1,\ldots B_n$, Sea *v2* el primer valor clave en B_1 y *v3* el último valor clave en B_n. Buscamos al bloque B_i , donde $i = \lceil n\, f(vi,v2,v3) \rceil$ para comparar su primer valor clave v_i con v1. Entonces, como en la búsqueda binaria, repetimos el proceso en $B_1,\ldots B_{i-1}$ ó en $B_i,\ldots B_n$, dependiendo de cual tramo podría contener a v1, hasta que solo quede un bloque por rastrear.

Puede demostrarse que si conocemos la distribución de las claves, podemos esperar examinar cerca de $1 + \log_2 \log_2 n$ bloques del archivo de índices antes de encontrar a v1 [Yao and Yao 1976][3]. Si agregamos a este número dos accesos para lectura y escritura en el archivo principal, necesitamos $3 + \log_2 \log_2 n$. Bajo este supuesto y las premisas del ejemplo 6.11, requerimos 6 accesos para el mismo problema (para la búsqueda binaria necesitamos 11 accesos).

Operaciones en un archivo clasificado con registros no marcados.
Vamos a considerar como llevar a cabo las operaciones de consulta, inserción, borrado y actualización en un archivo clasificado y con registros que no están marcados por apuntadores a localizaciones fijas. Estas cuatro operaciones requieren inserciones, borrados y modificaciones en el archivo *índice*, tal que es importante tener presente que este último es también un archivo clasificado que no tiene registros marcados. Así que al describir la operación en el archivo principal, podemos invocar la misma operación para que se ejecute en el archivo índice. Se asume que el lector sabe como implementar estas operaciones en el último caso. Note que como el archivo índice no tiene índice, la estrategia de búsqueda puede ser alguna de las ya vista (binaria, interpolación, etc.).
El archivo original ya clasificado, es almacenado en una secuencia de bloques $B_1, B_2, \ldots B_k$, con los registros de cada bloque ordenados; los registros de B_i preceden en el ordenamiento a los de B_{i+1}, para $i=1,2,\ldots,k-1$. Asumimos, además, que algunos bytes del comienzo del bloque, para archivos con registros de longitud fija, son usados con información de control (usado/no usado) sobre cada área destinada para registros; si los registros son de longitud variable, entonces el comienzo de cada bloque nos dice, el desplazamiento donde comienza cada registro o donde hay espacio no usado.

Inicialización.
En primer lugar clasificamos los registros del archivo inicial y los distribuimos entre los bloques. Como los archivos tienden a crecer, hallamos una distribución conveniente para los registros iniciales, de tal modo que dejen libre una pequeña fracción, dígase un 20%, del total del espacio para cada bloque.
El segundo paso del proceso de inicialización es crear el archivo de índices, examinando el primer registro de cada bloque en el archivo principal. La clave de cada registro se acompaña con la dirección del bloque que la almacena en el archivo principal. Una excepción útil es reemplazar el valor clave del primer registro del archivo índice con $-\alpha$, un valor menor que cualquier valor clave. Cuando distribuimos los registros índice en los bloques, podríamos desear de nuevo dejar libre una pequeña fracción del espacio disponible, porque cuando nos veamos obligados a aumentar el número de bloques del archivo principal también se incrementaran el número de registros en el archivo índice.
El paso final es crear un directorio con la dirección de los bloques del índice. Con frecuencia es lo suficientemente pequeño para ser cargado en memoria principal. Si este no es el caso, el directorio ocupará tantos bloques como sea necesario y serán movidos a memoria principal ó fuera de ella cada vez que sea necesario.

Consulta.
Suponga que necesitamos hallar el registro del archivo principal con valor clave v_1. Examinamos al archivo índice para hallar el v_2 que cubre a v_1. El archivo índice que

[3]"*The complexity of searching a random ordered table*", Proc. Seventeenth Annual IEEE Symp. on Foundations of Computer Science, pp. 173-177.

contiene a v_2 también contiene un apuntador a un bloque del archivo principal donde hallaremos, si existe, el valor clave v_1.
La búsqueda sobre el archivo índice con clave v_2 que cubre a v_1 puede llevarse a cabo usando alguna de las técnicas vistas anteriormente. Dependiendo del conocimiento de mis datos.

Modificación.
Para modificar un registro con valor clave v_1 uso el procedimiento de consulta anterior. Si el campo modificado es la clave, trato la operación como una inserción y un borrado. De otra manera, sobrescribo el registro encontrado.

Inserción.
Para insertar un registro con un valor clave v_1, uso el procedimiento de consulta ya visto y obtengo el bloque B_i del archivo principal, en el que un registro con tal valor clave debe ser almacenado. Colocamos el nuevo registro en el bloque B_i encontrado; para no perder el ordenamiento dentro del bloque, desplazo a la derecha los registros con valor clave mayor que v_1 y de esta manera separo espacio para el nuevo registro[4].
Si el bloque B_i tiene por lo menos un área disponible donde acomodar al nuevo registro, la operación se da por concluida. Sin embargo, Si B_i está completamente lleno, el nuevo registro no tendrá espacio para almacenarse y debemos seguir alguna estrategia para crear un nuevo bloque. En la siguiente sección se examinará la estrategia seguida por los árboles "B-tree", en la que B_i es dividido en dos bloques vacíos en un 50%. Otra alternativa es examinar B_{i+1}; podemos hallar este bloque, si existe, apoyándonos en el archivo índice (El apuntador lo encuentro en el registro del índice que está a continuación del registro con clave v_2). Si B_{i+1} tiene un área para registros vacía, movemos el registro que excede en B_i (el último en el bloque) como primer registro del bloque B_{i+1} y desplazamos los siguientes, hasta que se cope la primera área reservada para registros que se encuentre vacía. Se cambia en el encabezado del bloque la información de espacio usado / no, usado de tal manera que coincida con la nueva situación y se modifica la clave del archivo índice que corresponde al bloque B_{i+1}, para que refleje el nuevo valor clave en el primer registro.
Si B_{i+1} tiene muchas áreas para registro vacías, podemos desplazar varios registros desde B_i hasta que que se iguale la cantidad de espacio vacío en cada uno; esta operación no incrementa el número de bloques leídos, en realidad, requiere de muy poca computación extra.
Si B_{i+1} no existe, porque $i = k$ o B_{i+1} existe pero está lleno, podríamos considerar obtener espacio desde B_{i-1} y proceder de manera similar. Si este bloque también se encuentra lleno ó no existe, debemos obtener un nuevo bloque, que seguirá a B_i en el orden. Dividir los registros de Bi entre este y el nuevo bloque. Luego insertar un registro para el nuevo bloque en el archivo índice, usando la misma estrategia de inserción que para insertar un registro en el archivo principal.
Borrado.
Como en una inserción existe una variedad de estrategias y en la próxima sección se discutirá una que no permite que los bloques estén copados por debajo de la mitad del espacio disponible. Acá mencionaremos solo la estrategia más simple, apropiada si se realizan pocos borrados. Para borrar un registro con un valor clave v_1, primero obtenemos el bloque donde se halla. Movemos los registros ubicado a su derecha[3] ,

[4]Asumimos que el costo más significativo es el acceso a disco. Los demas cálculos y desplazamiento no pesan en el costo total.
[3] Este paso no es esencial. Si se escoge no dejar espacio intermedio, podemos llevar en el encabezado la cuenta de las áreas para registro llenas, en lugar del control de bits usados/no usados para cada área. El

para que ocupen el área dejada por el registro borrado y ajustamos la información de bits usados / no usados del encabezado. Si el bloque queda completamente vacío, lo retornamos al sistema de archivos y borramos el registro para ese bloque en el archivo índice.
Ejemplo 6.12.
Suponga que tenemos un archivo de registros de *números*; inicialmente nuestro archivo consiste de la siguiente lista aleatoria de datos, generada arrancando en 2 y calculando su cuadrado repetidamente para tomar su residuo del módulo con 101.:

2,4,5,16,25,37,54,56,68,79,80,88

Podemos acomodar 5 registros por bloque, pero ubicamos inicialmente únicamente 4 y así dejamos espacio para futuras expansiones. La disposición inicial es mostrada en la fig. 6.7. Cada uno de los tres bloques del archivo principal tiene un área de registro vacía y 4 bytes de espacio malgastado al final, un byte podría ser ocupado con información de los bits usado / no usado para las cinco áreas de registros en el bloque. Un bloque en el archivo índice tiene tres registros $(-\alpha, b_1)$, $(25,b_2)$ y $(68,b_3)$, donde b_1,b_2 y b_3 son las direcciones de tres bloques del archivo principal.
El directorio para los bloques del índice no es mostrado, pero debe contener la dirección de los bloques del índice.
Ahora, consideremos que los siguientes cuatro números aleatorios a ser insertados son 19,58,31 y 52. Colocamos 19 en el primer bloque, y como es mayor que los números ya almacenados allí lo ubicamos al final, lo que evita desplazamiento a la derecha de información. Igual ocurre para el número 58, que se acomoda en el segundo bloque.
La tercera inserción, 31, también pertenece al segundo bloque y debe ubicarse como segundo registro dentro de este, después del 25. Debemos entonces desplazar a la derecha 37,54,56 y 58 para hacer espacio. Sin embargo, no hay espacio para un sexto registro y debemos entonces hallar sitio para ubicarlo. En este caso, el tercer bloque tiene espacio, así que debemos desplazar 58 como primer registro en el tercer bloque y desplazar a la derecha los cuatro registros ya almacenados. Como 58 pasa a ser la clave de más bajo valor en el tercer bloque, debemos cambiar al registro índice para ese bloque. Las anteriores modificaciones son mostradas en la figura 6.8.
La inserción final es el 52, que pertenece al bloque 2 después del 37. De nuevo no hay espacio en el bloque, pero ahora, los bloques anterior y siguiente están llenos. Entonces, dividimos el segundo bloque en dos, cada uno de los cuales queda con tres registros: 25,31 y 37 en el primero; 52,54 y 56 en el segundo. El primero de estos bloques puede ser identificado como el bloque que está siendo dividido, pues el registro $(25,b_2)$ del archivo índice coincide con él. El segundo bloque requiere de un nuevo registro con clave 52 en el archivo índice, deberá ser insertado en el sitio apropiado dentro del índice. La estructura resultante es mostrada en la fig. 6.9.

Archivos clasificados con registros marcados[4] .

lector debe tener de nuevo en mente que si los registros son marcados se tiene aún la opción de mover los registros hacia áreas de registro que hallan sido borradas.
[4] Tema no traducido.

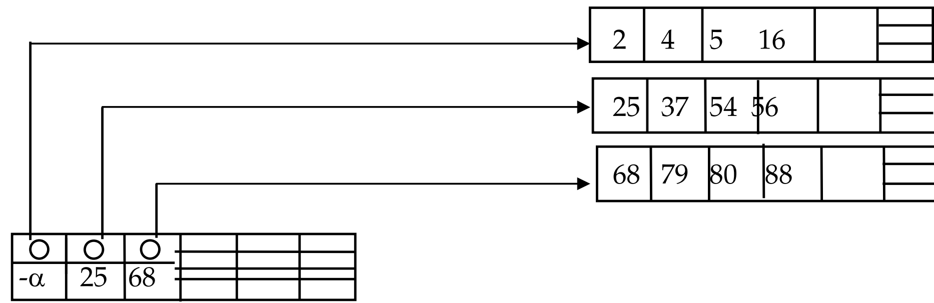


Fig. 6.7. Archivo índice inicial.

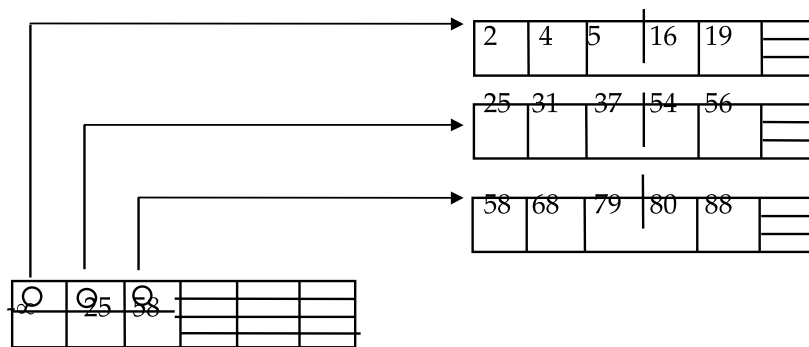


Fig. 6.8 Despues de insertar 19,58,31.

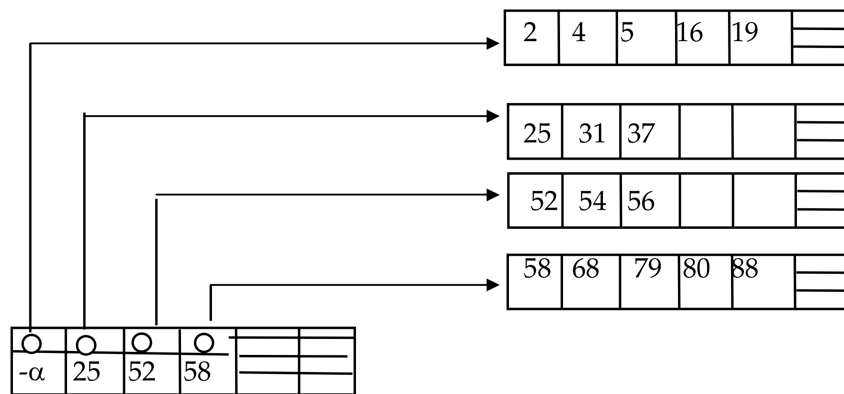


Fig. 6.9 Después de insertar 52.

6.5. B-trees.

Un índice no es más que un archivo con registros no marcados, no hay razón por la que no se pueda tener un índice de un índice, un índice de lo anterior y así sucesivamente, hasta obtener un archivo que se acomode en un bloque, como sugiere la figura 6.12. En realidad, tal organización puede ser considerablemente más eficiente que un archivo de un solo nivel de índices. En la estructura de la figura 6.12., el archivo principal es clasificado por su valor clave. El índice de primer nivel consiste de pares

(v,b), donde *v* es el apuntador al bloque *B* del archivo principal y *v* es la primera clave del bloque *B*. Naturalmente, el índice está también clasificado por valor clave. El segundo nivel del índice tiene pares *(v,b)* donde *b* apunta al primer nivel del bloque índice y *v* es su valor clave, y así sucesivamente.
Hay muchas formas de estructurar índices multinivel y ellas son colectivamente llamadas como B-tree (árboles balanceados). La estructura particular sugerida en la figura 6.12. almacena al archivo principal como parte del árbol B-tree y asume que el archivo principal tiene registros no marcados. Este método no hace un uso eficiente del espacio y lo presentamos de esta manera solo por simplicidad. Un aprovechamiento superior, que ahorra espacio en la mayoría de las situaciones y que es también propio para almacenar registros marcados, se describe en la sección 6.6. Allí se almacena el archivo principal de una manera muy compacta, en bloques con registros sin ningún orden en particular, como un montón; luego las hojas del árbol B-tree contienen en lugar del registro apuntadores al sitio donde se encuentre.

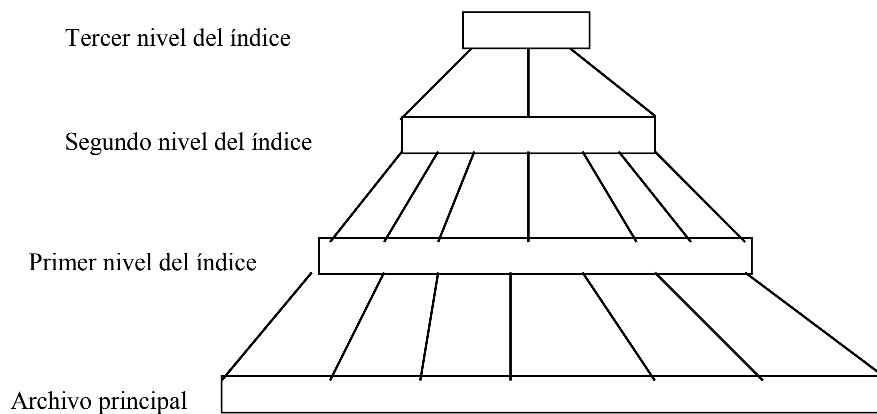


Fig. 6.12. Indice multinivel

Para insertar y borrar en un B-tree, podríamos usar la misma estrategia descrita en la sección previa, aplicando las operaciones de inserción y borrado a los nodos (bloques) del árbol a todos los niveles. Esta estrategia debe resultar en nodos que contienen entre uno y el máximo número de registros posibles. En lugar de lo anterior, los B-tree son usualmente definidos para usar una particular estrategia de inserción y borrado que asegura que ningún nodo, excepto posiblemente la raiz, esten llenos menos de la mitad de su capacidad. Por conveniencia asumimos que el número de registros índice en un bloque es un entero impar entre 2d - 1 >= 3. y que el número de registros por bloque en el archivo principal es también un entero impar entre 2e -1 >= 3.

Antes de proceder, debemos observar una diferencia más entre los B-trees y la jerarquía de índices sugerida en la figura 6.12. En un bloque índice del B-tree, para ahorrar espacio, el valor clave del primer registro es omitido. Durante la consulta, todos los valores claves menores que el valor en el segundo registro del bloque se consideran cubiertos por el primer valor clave.

Consulta.
Para obtener un registro con valor clave *v*, hallamos un camino desde la raíz del B-tree hasta alguna de sus hojas, donde deseamos encontrar tal registro, si existe. El camino comienza en la raiz. Suponga que después de un tiempo de búsqueda alcanzamos el

nodo(bloque) *B*. Si *B* es una hoja (lo que podemos saber si conocemos el número actual de niveles del árbol) entonces simplemente examinamos en el bloque un registro con valor clave *v*.
Si B no es una hoja, es un bloque índice. Determinamos cual valor clave en el bloque B cubre a *v*. Recordemos que el primer registro de B no almacena su valor clave, y el valor no almacenado se presume cubre cualquier valor menor que el valor clave del segundo registro (p. e : Asumimos la clave no almacenada como -α). En el registro B que cubre a *v* hay un apuntador a un bloque *B'*. En el camino que estamos construyendo *B'*sigue a *B* y repetimos el anterior paso, esta vez ubicados sobre el bloque *B'*;.
Como el valor clave en un registro *i* del bloque B es la menor clave que cualquier hoja que descienda desde el *i-esimo* hijo de B y los registros del archivo principal están clasificados por su valor clave, es facil chequear que B' es el único hijo de B en el que un registro con clave *v* puede encontrarse. Esta afirmación también es válida para *i*= 1, así no existan valores claves en el primer registro de B. Esto es, si *v* es menor que la clave en el segundo registro, entonces un registro del archivo principal con clave *v no* podría ser un descendiente del segundo o posteriores hijos de B.
Modificación.
Así como en otras organizaciones discutidas, una modificación de un campo clave es realmente un borrado y una inserción, mientras una modificación que no cambie el valor clave es una consulta seguida de una reescritura del registro involucrado.
Inserción.
Para insertar un registro con valor clave *v*, aplicamos el procedimiento de consulta para hallar el bloque B al que pertenece el registro. Si hay menos que 2*e*-1 registros por bloque, simplemente insertamos el nuevo registro sin perder el ordenamiento del bloque. Se puede mostrar que el nuevo registro nunca puede ser el primero en el bloque B, a menos que B sea la hoja de más a la izquierda. Así, nunca es necesario modificar un valor clave en un ancestro de B.
Si ya hay 2*e* -1 registros en el bloque B, creamos un nuevo bloque B_1; dividimos los registros de B y los insertamos en dos grupos de *e* registros cada uno. Los primeros *e* registros van al bloque B y los restantes van al bloque B_1.
Ahora, Sea P el padre del bloque B. Recordemos que el procedimiento de consulta halla el camino desde la raíz a B, tal que P es ya conocido. Aplicamos el procedimiento de inserción recursivamente, con la constante *d* en lugar de *e*, para insertar un registro para B_1 a la derecha del registro de B en el bloque índice P. Note que si hay muchos ancestros del bloque B que tienen el máximo de 2d - 1 registros, el efecto de insertar un registro en B puede agitar hacia arriba el árbol. Sin embargo, solo serán afectados los ancestros de B. Si la inserción se propaga hasta la raíz, ésta es dividida y creamos una nueva con dos hijos. Esta es la única situación en la que un bloque índice puede tener menos de *d* registros.
Ejemplo 6.15.

Borrado.
Si deseamos borrar un registro con valor clave *v*, usamos el procedimiento de consulta para hallar el camino desde la raíz al bloque B que contiene al registro. Si después del borrado el bloque tiene *e* ó más registros, la mayoría de las veces el proceso termina. Sin embargo, si el registro borrado era el primero del bloque B, entonces debemos ir a su padre y cambiar el valor clave en el registro usado para tal bloque, de tal manera que coincida con el nuevo valor clave . Si B es el primer hijo de su padre, el padre no tiene valor clave para B, tal que debemos ir al padre del padre, al padre de este último y así sucesivamente hasta hallar un ancestro A_1 de B tal que A_1 no sea el primer hijo de su padre A_2. Entonces la nueva clave más baja de B queda en el registro de A_2 que

apunta a A_1. De esta manera, todo registro (v_1,p_1) en todo bloque índice tiene el valor clave v_1 igual al más bajo de todos esos valores claves del archivo original hallados entre las hojas que son descendientes del bloque apuntado por p_1.[5]
Si después del borrado, el bloque B tiene *e*-1 registros, buscamos el bloque B_1 que tenga el mismo padre que B y que resida a continuación de éste por la izquierda ó por la derecha. Si B_1 tiene más de *e* registros, distribuimos los registros de B y B_1 tan igualmente como sea posible, manteniendo el ordenamiento. A continuación, modificamos los valores claves de B y/o B_1 en el padre de B, y si es necesario propagamos los cambios a sus ancestros. Si B_1 tiene solo *e* registros, entonces combinamos *e* con e_1, lo que da como resultado 2e -1 registros, y en el padre de B modificamos el registro para B_1 (lo que puede a su vez provocar cambios en algunos ancestros de B) y borramos el registro para B. El borrado de este registro requiere del uso recursivo del procedimiento de borrado, con la constante *d* en lugar de *e*.
Si un borrado se propaga por todo el camino desde el hijo hasta la raíz, podemos finalizar combinando solo dos hijos de la raíz. En este caso, el nodo formado de la combinación de dos hijos llega a ser la raíz y la vieja raíz es borrada. Esta es una de las situaciones en la que el número de niveles del árbol decrece.
Ejemplo 6.16.

Análisis de tiempo en las operaciones sobre un B-tree.
Suponga que tenemos un archivo con n registros organizados en un B-tree con parámetros *d* y *e*. El árbol no tendrá más de *n/e* hojas, no más de *n/de* padres de hojas, $n/d^2 e$ padres de padres de hojas y así sucesivamente. Si hay *i* nodos en el camino desde la raíz a las hojas, entonces $n >= d^{i-1}e$, o de otro modo habría menos de un nodo a nivel de la raíz, lo que es imposible. Sigue que

$$i <= 1 + \log_d (n/e).$$

Para ejecutar una consulta bastan *i* bloques leídos. Para una inserción, borrado ó modificación, usualmente solo un bloque (la hoja almacenando al registro involucrado) requiere ser escrito; aunque en casos extremos, cerca de *i* lecturas y escrituras adicionales pueden ser necesarias. Es muy difícil un análisis exacto de la probabilidad de hallar bloques con demasiados registros en una inserción ó pocos registros en un borrado. Sin embargo, no es difícil mostrar que aún para $e = d = 2$, el número esperado de lecturas y escrituras extras (un exceso de *i* lecturas para hallar la hoja y una escritura para almacenarla) es una fracción propia. Omitiremos esta fracción y estimaremos el número de lecturas y escrituras como $2 + \log_d (n/e)$. Aunque esta cifra es conservadora, porque en promedio muchos bloques tendrán más que el mínimo número de registros y por lo tanto la altura del árbol puede ser menor que $1 + \log_d (n/e)$.
Ejemplo 6.17.
Reconsideremos el archivo tomado en el ejemplo 6.11 que discutimos para un índice de un solo nivel. Los registros se asumen de un longitud de 200 bytes, si deseamos un número impar en bloques de 4096 bytes, debemos escoger $2e-1 = 19$; luego $e = 10$. Asumimos en el ejemplo 6.11. que las claves eran de 20 bytes de longitud y que los apuntadores tomaban 4 bytes. Dado que omitimos la primera clave, podemos acomodar 171 registros índice por bloque de 4096 bytes, pues 170 x 20 + 171 x 4 = 4084. Así, d = 86. El número esperado de bloques accesados por operación es :

$$2 + \text{Log}_d (n/e) = 2 + \text{Log}_{86} (1.000.000/10) < 5 .$$

[5] Esta propiedad no es esencial y podríamos pasar por alto las modificaciones en las claves de los bloques índice. Entonces v_1 debería ser un límite inferior para las claves descendientes del bloque apuntado por p_1. Los descendientes a la izquierda de ese bloque tendrían todavía claves menores que v_1 , por lo que continúan siendo útiles para hallar registros en el B-tree.

Esta cifra es mayor que la mejor del método aleatorio (cerca de 3 lecturas /escrituras), pero es superior a los métodos que usan un solo nivel de índices, excepto quizás en aquellas situaciones en las que la búsqueda por interpolación puede ser ejecutada. El B-tree comparte con los métodos de la sección 6.4. la ventaja sobre el método aleatorio de permitir al archivo ser listado o buscado de una manera ordenada.

6.6. Archivos con un índice denso.

En los esquemas discutidos hasta ahora, disperso, aleatorio y B-tree, la mayoría de los bloques del archivo principal están llenos solo parcialmente. Por ejemplo, una estructura aleatoria con un adecuado número de áreas tendrá solo uno ó dos bloques en la mayoría de las áreas y al menos uno de ellos estará parcialmente lleno. En el esquema B-tree discutido en la sección previa, todos los bloques hoja están entre la mitad y completamente llenos, con un promedio por bloque de alrededor tres cuartos de lleno.
En contraste, el montón, mencionado en la sección 6.2., guarda todos los bloques del archivo principal completamente llenos, lo que ahorra una significativa cantidad de espacio[6]. El problema con usar un montón, es por supuesto, que dado su valor clave debemos tener un modo eficiente de hallar un registro. Para lograrlo necesitamos otro archivo, llamado un índice denso, que consiste de registros (v,p) para cada valor clave v en el archivo principal, donde p es un apuntador al registro del archivo principal que contiene al valor clave v. La estructura del índice denso puede ser alguna de las discutidas en las secciones 6.3 a 6.5; pues lo que requerimos es, dado un valor v , poder hallar rápidamente al registro (v,p). Note que un índice denso almacena un apuntador a todo registro del archivo principal, mientras que el índice esparcido, discutido previamente, almacena claves para un subconjunto de registros del archivo principal, normalmente solo aquellos que están de primeros en cada bloque.
Para consultar, modificar o borrar un registro del archivo principal, dada su clave, ejecutamos una consulta en el archivo del índice denso. Con la clave obtenida allí, sabemos que bloque del archivo principal es necesario leer para obtener el registro deseado. Podemos entonces leer el bloque del archivo principal. Si el registro va a ser modificado, debemos efectuar los cambios y rescribir el bloque al almacenamiento secundario. Son necesarios dos accesos más a bloques (uno para lectura y otro para escritura) para la operación correspondiente en el archivo del índice denso.
Si vamos a borrar el registro, de nuevo rescribimos el bloque y también borramos el registro con tal valor clave en el índice denso. Esta operación toma dos accesos más que una consulta y borrado desde el índice denso.
Para insertar un registro r, ubicamos tal registro al final del archivo principal y luego insertamos un apuntador a r junto con su valor clave en el archivo del índice denso. De nuevo esta operación toma dos accesos más que hacer una inserción en el índice denso.
Podría parecer que un archivo con un índice denso requiere siempre de dos accesos más que si usaramos para el archivo principal cualquier otra organización distinta (aleatoria, indexado o B-tree). Sin embargo, hay dos factores que van en dirección opuesta y que justifican el uso de índices densos en algunas situaciones.
1. Los registros del archivo principal pueden estar marcados, pero los del índice denso no necesitan estarlo; de tal manera que podemos usar una organización más simple y eficiente en el índice denso que en el archivo principal.

[6] En el caso de registros marcados no puede llevarse a cabo borrado físico, solo pueden marcarse como borrados. Sin embargo, este costo extra de espacio se presenta con todas las estructuras de almacenamiento para registros marcados.

2. Si los registros del archivo principal son grandes, el número total de bloques usados en el índice denso puede ser mucho más pequeño que los que deberían ser usados para un índice esparcido o un B-tree. Similarmente, el número de áreas o el promedio de bloques por área puede ser más pequeño, si el acceso aleatorio es usado en el índice denso que si fuese usado en el archivo principal.

Ejemplo 6.18.

Consideremos el archivo discutido en el ejemplo 6.17., donde usamos un B-tree con d = 86 y e = 10, en un archivo de n = 1.000.000 de registros. Como los registros de un índice denso son del mismo tamaño que los registros de los nodos interiores de un B-tree, si usamos esta misma organización para el índice denso, podemos tomar d = 86 y e = 85.[7] Así, el número típico de accesos para buscar el índice denso es 2 + $\log_{86}$ (1.000.000/85) o ligeramente mayor que 4. Para esto debemos agregar dos accesos al archivo principal, tal que el índice denso más la organización B-tree toma uno ó dos más accesos a bloques que la organización B-tree simple.

Hay, sin embargo, factores compensatorios en favor del índice denso. Podemos copar completamente los bloques del archivo principal si usamos esta organización, mientras que en la organización B-tree simple , los bloques hoja, que contienen al archivo principal, están copados entre la mitad y totalmente. Así, podemos ahorrar cerca del 25% de espacio de almacenamiento para el archivo principal. El espacio usado para las hojas del B-tree en el índice denso es solo el 12% del espacio del archivo principal, pues los registros índice tienen 24 bytes de longitud mientras que los registros del archivo principal tienen 200 bytes. Así, todavía tenemos un ahorro neto de aproximadamente un 13% de espacio. Quizás, de mas importancia, si el archivo principal tiene registros marcados, no podríamos usar la organización B-tree descrita en la sección 6.5.

Métodos para registros no marcados.

Otro uso para los índices densos es el de un sitio para recibir apuntadores a registros. Esto es, un apuntador al registro *r* del archivo principal puede mejor apuntar al registro del índice denso que apunta a *r*. La desventaja estriba en que para seguir un apuntador a *r* debemos seguir un apuntador extra desde el índice denso al archivo principal. La compensación está en que ahora los registros del archivo principal no están marcados (aunque los registros del archivo índice si lo están). Cuando deseemos mover un registro del archivo principal, solo tenemos que cambiar el apuntador en el índice denso que apunta al registro movido. Podemos, además, usar un método más compacto de almacenamiento para el archivo principal y el almacenamiento ahorrado podría ser mayor que cubrir el costo del índice denso. Por ejemplo, si el archivo principal es no marcado, podemos reutilizar las áreas de registros borrados.

Otra técnica para manejar archivos no marcados es usar el valor clave de los registros en lugar de apuntadores. Esto es, en lugar de almacenar la dirección de un registro *r*, almacenamos el valor de su clave, y para hallar a *r* , realizamos una búsqueda estándar dado el valor clave. El sistema de Base de Datos IMS (IBM 1978), por ejemplo, hace uso de esta técnica. De esta manera, tanto el archivo denso como el archivo principal pueden ser no marcados. La desventaja de esta implementación de apuntadores es que para seguir a un "apuntador" al archivo principal, debemos buscar por el valor clave del registro en el índice denso, ó en cualquier estructura usada para acceder al archivo principal, lo que probablemente tomará varios accesos a bloques. En comparación, deberíamos necesitar solo un acceso a bloque si pudiésemos ir

[7] Técnicamente, los bloques hoja del B-tree usados para un índice denso debe tener el valor clave en todos los registros, incluyendo el primero. Esta diferencia significa que podemos acomodar solo 170 registros en los bloques hoja y por ello debemos tomar e = 85.

directamente al registro del archivo principal o dos accesos si vamos directamente al registro en el índice denso y luego al registro en el archivo principal.

Resumen.
La fig. 6.16. lista los 4 tipos de organización para archivos que permiten, dado un valor, consultar, modificar, insertar y borrar un registro. En el análisis de tiempo, tomamos n como el número de registros en el archivo principal y para uniformidad con los B-tree, asumimos que los registros del archivo principal están almacenados e por bloque en promedio y los registros del archivo índice pueden acomodarse d por bloque en promedio.

Organización	**Tiempo por operación**	**Ventajas y desventajas**	**Problemas con registros marcados**.
Aleatoria.	≈ 3 si hay un bloque en promedio por área	El más veloz de todos los métodos. Si el archivo crece, se hace más lento el accceso, pues se aumenta tamaño del área. No permite fácil acceso ordenado de los registros.	Debe buscar áreas con espacio libre durante la inserción o permitir más bloques por área que el óptimo.
Isam	≈ 2 + log (n /de) para búsqueda binaria. ≈ 3 + log log (n/de) si el calculo de la dirección es posible. (interpolación)	Rápido acceso si uso interpolación. Puedo recorrer ordenadamente los registros.	Idem al anterior.
B-tree	≈ 2 + $\log_d$ (n/e)	Rápido acceso. Los registros pueden recorrerse ordenadamente. Los bloques tienden a no estar almacenados de una manera muy compacta.	Use B-tree como índice denso.
Indice Denso	<= 2 + tiempo para operación en el índice denso.	Frecuentemente más lento en uno o dos accesos a bloques que si el mismo método de acceso usado para el archivo índice fuese usado para el archivo principal. Puede ahorrar espacio.	Ninguno.

Fig. 6.16. Resumen de métodos de acceso.