Indexing Structures for Files and Physical Database Design

Se inicia diciendo que en el capitulo se van a describir estructuras de acceso auxiliar adicionales llamadas índices, que se utilizan para acelerar la recuperación de records en respuesta a ciertas condiciones de búsqueda. Las estructuras de índice son archivos adicionales en el disco que proporcionan rutas de acceso secundarias, que proporcionan formas alternativas de acceder a los records sin afectar la ubicación física de los records en el archivo de datos primario en el disco. Los tipos de índices más frecuentes se basan en archivos ordenados y utilizan estructuras de datos de árbol para organizar el índice. Los índices también se pueden construir basados en hashing u otras estructuras de datos de búsqueda.

17.1 Types of Single-Level Ordered Indexes:

Un índice ordenado es similar a un índice utilizado en un libro de texto, que enumera los términos importantes al final del libro en orden alfabético junto con una lista de números de página en donde el término aparece en el libro.

Para un archivo con una estructura de registro de muchos campos o atributos, la estructura de acceso de índice se define en un solo campo de un archivo, llamado campo de indexación o atributo e indexación. El índice almacena cada valor del campo de índice junto con una lista de punteros a todos los bloques de disco que contienen registros con ese valor del campo. Los valores en el índice están ordenados para poder hacer una búsqueda binaria en el índice. Si el data file y el archivo de índice son ordenados, y como el archivo de índice es normalmente mucho más pequeño que el data file, buscar en el índice usando una búsqueda binaria es la mejor opción. Los índices con estructura de árbol multinivel implementan una extensión de la idea de una búsqueda binaria que reduce el espacio de búsqueda mediante la partición "two-way" en cada paso de búsqueda a un enfoque de partición "n-ary" que divide el espacio de búsqueda en el archivo "n-ways" en cada etapa.

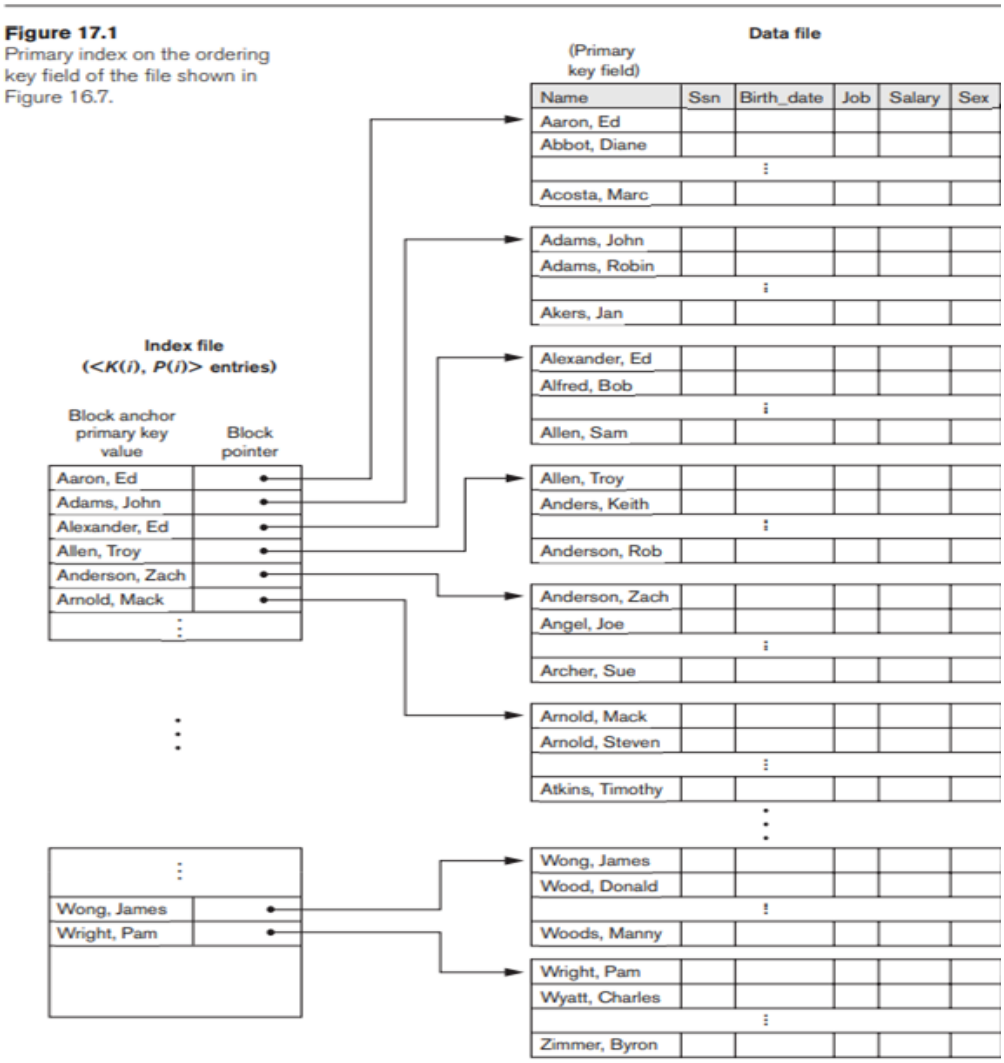
Un índice primario está especificado en el ordering key field de un archivo ordenado de registros, el ordering key field es usado para ordenar físicamente el archivo de registros en el disco, y cada registro tiene un único valor para ese campo. Si el ordering field no es un key field, entonces otro tipo de índice llamado clustering index puede ser usado, y el data file es llamado clustered file. Se debe saber que un archivo solo puede tener un campo de ordenamiento físico, es decir un índice primario o un clustering index. Un tercer tipo de índice, llamado índice secundario, se puede especificar en cualquier campo no ordenado de un archivo. Un data file puede tener varios índices secundarios además de su método de acceso primario.

17.1.1 Primary Indexes:

Un índice primario es un archivo ordenado cuyos registros son de longitud fija con dos campos, y actúa como una estructura de acceso para buscar y acceder eficientemente a los registros de datos en un data file. El primer campo es del mismo tipo de datos que el ordering key field, llamada clave primaria, del data file, y el segundo campo es un puntero a un bloque de disco (una dirección de bloque). Existe una index entry (ó index record) en el archivo de índice para cada bloque en el data file. Cada index entry tiene el valor del campo de la clave principal para el primer registro en un bloque y un puntero a ese bloque como sus dos valores de campo.

La figura 17.1 ilustra este índice primario. El número total de entradas en el índice es el mismo que el número de bloques de disco en el archivo de datos ordenado. El primer registro en cada bloque del archivo de datos (data file) se llama “anchor record of the block”, o simplemente el “block anchor”.

Un dense index tiene una index entry para cada valor clave de búsqueda (para cada registro) en el archivo de datos. Un sparse index (ó nondense), tiene index entries solo para algunos de los valores de búsqueda. Un sparse index tiene menos entradas que el número de registros en el archivo. Por lo tanto, un índice primario es un nondense index (sparse), ya que incluye una entrada para cada bloque de disco del archivo de datos y las claves de su anchor record en lugar de para cada valor de búsqueda (o cada registro).



17.1.2 Clustering Indexes:

Si los registros de archivos se ordenan físicamente en un campo sin clave, que no tiene un valor distinto para cada registro, ese campo se llama clustering field y el archivo de datos se llama clustered file. Podemos crear un tipo diferente de índice, llamado clustering index, para acelerar la recuperación de todos los registros que tienen el mismo valor para el clustering field. Esto difiere de un índice primario, que requiere que el ordering field del archivo de datos tenga un valor distinto para cada registro.

Un clustering index también es un ordered file con dos campos; el primer campo es del mismo tipo que el clustering field del archivo de datos, y el segundo campo es un puntero de bloque de disco.

La figura 17.2 muestra un ejemplo. Tenga en cuenta que la inserción y eliminación de registros sigue causando problemas porque los registros de datos están ordenados físicamente. Para aliviar el problema de la inserción, es común reservar un bloque completo para cada valor del clostering field, todos los registros con ese valor se colocan en el bloque, lo que hace que la inserción y eliminación sea relativamente sencilla, la Figura 17.3 muestra este esquema

Un clustering index es otro ejemplo de un nondense index porque tiene una entrada para cada valor distinto del indexing field, que es nonkey por definición y por lo tanto, tiene valores duplicados en lugar de un valor único para cada registro en el archivo.

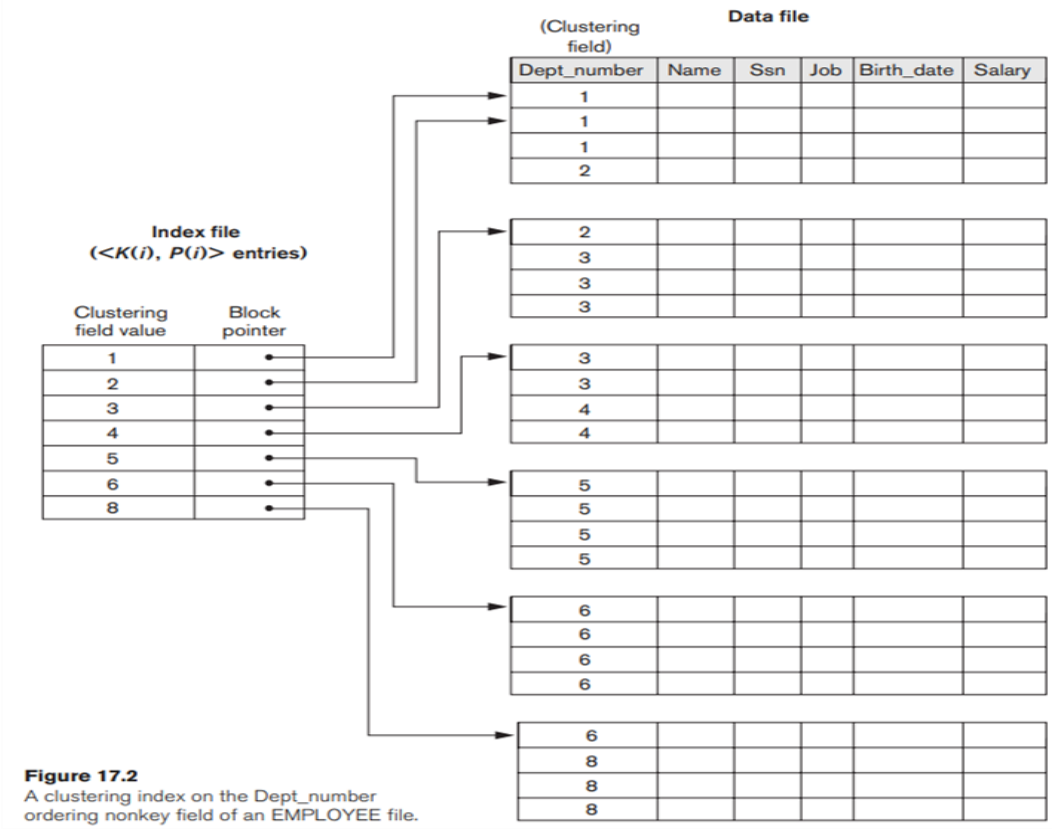
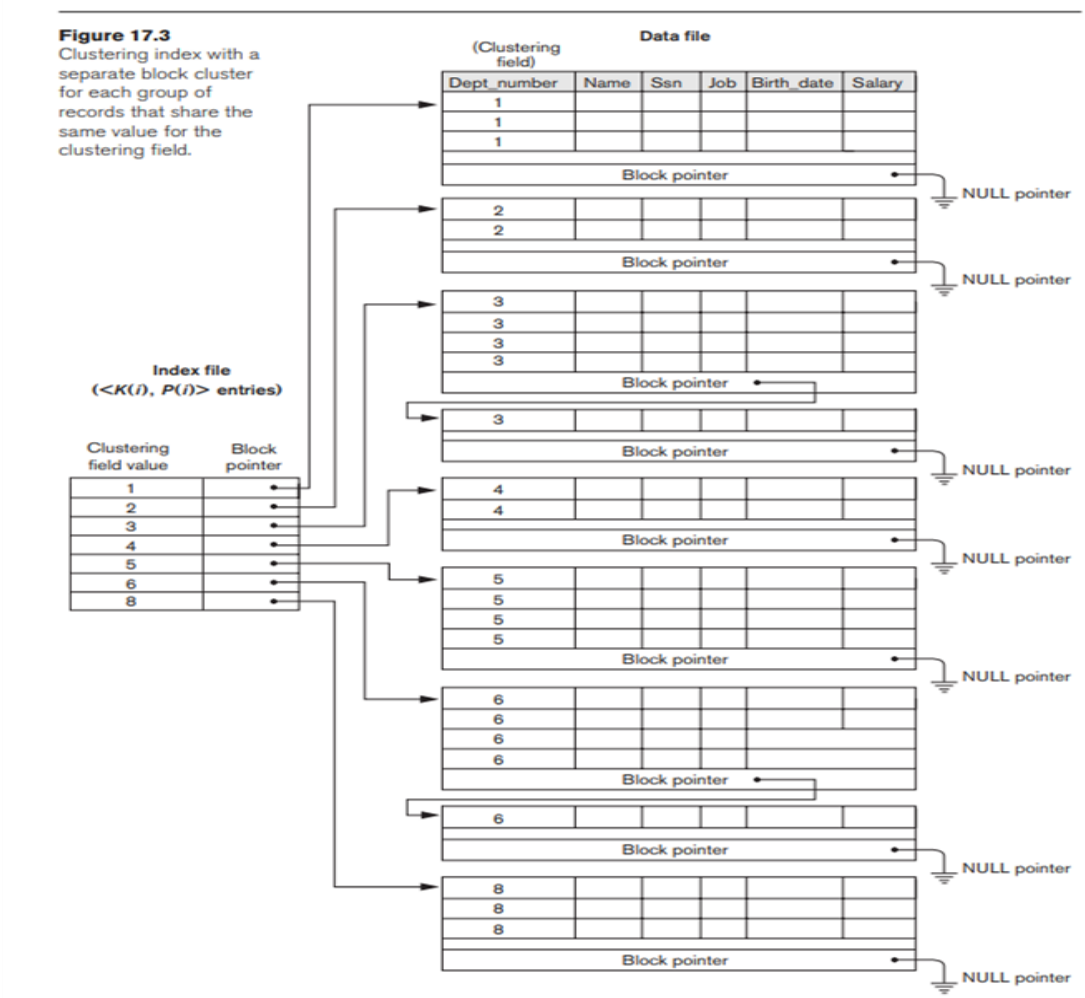


Figure 17.2
A clustering index on the Dept_number ordering nonkey field of an EMPLOYEE file.



Existe cierta similitud entre las Figuras 17.1, 17.2 y 17.3 y las Figuras 16.11 y 16.12. Un índice es algo similar al hashing dinámico (descrito en la Sección 16.8.3) y a las estructuras de directorio utilizadas para el hashing extensible. Se busca en ambos para encontrar un puntero al bloque de datos que contiene el registro deseado. Una diferencia principal es que una búsqueda de índice usa los valores del campo de búsqueda en sí mismo, mientras que una búsqueda de directorio hash usa el valor de hash binario que se calcula aplicando la función hash al campo de búsqueda.

17.1.3 Secondary Indexes:

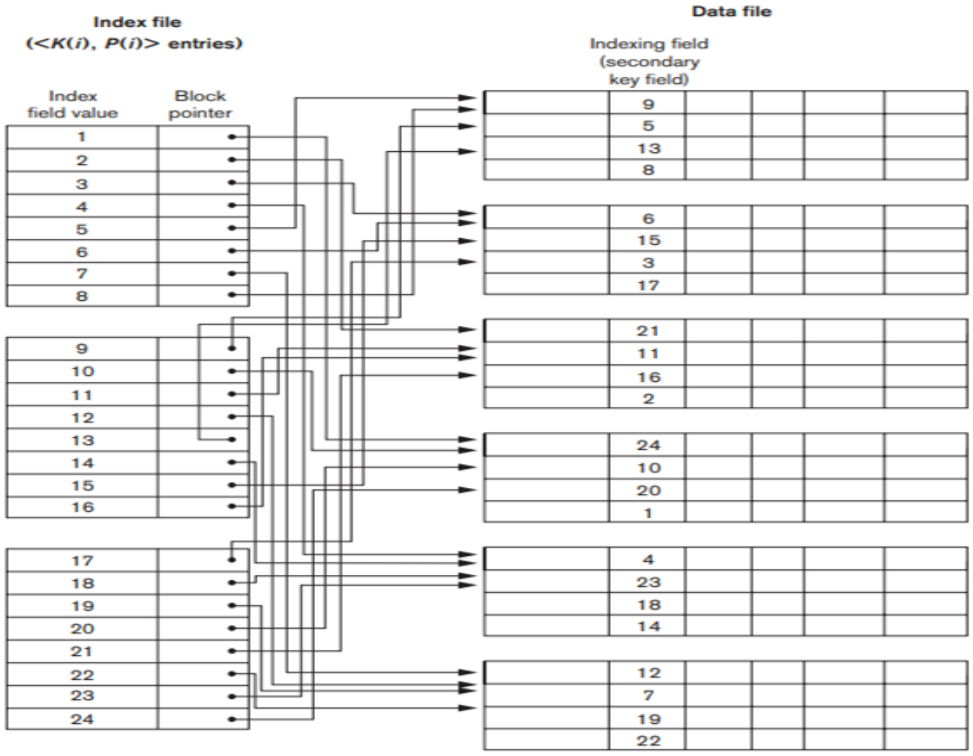
Un índice secundario proporciona un medio secundario para acceder a un archivo de datos para el cual ya existe algún acceso primario. Los registros del archivo de datos pueden ser ordenados, desordenados o hash. El índice secundario se puede crear en un campo que es una clave candidata y tiene un valor único en cada registro, o en un campo sin clave con valores duplicados. El índice es nuevamente un archivo ordenado con dos campos. El primer campo es del mismo tipo de datos que algún campo no ordenado del archivo de datos, el cual es un indexing field. El segundo campo es un puntero de bloque o un puntero de registro.

Se considera una estructura de acceso al índice secundario en un campo clave que tiene un valor distinto para cada registro. Ese campo a veces se llama una clave secundaria, esto correspondería a cualquier atributo de clave ÚNICO o al atributo de clave primaria de una tabla. En este caso, hay una index entry para cada registro en el archivo de datos, que contiene el valor del campo para el registro y un puntero al bloque en el que está almacenado el registro o al registro mismo, por eso, este índice es denso.

Las entradas están ordenadas por valor, para que podamos realizar una búsqueda binaria, como los registros del archivo de datos no está físicamente ordenados por valores del campo de clave secundaria, no podemos usar block anchors. Es por eso que se crea una index entry para cada registro en el archivo de datos, en lugar de para cada bloque, como en el caso de un índice primario. Un índice secundario generalmente necesita más espacio de almacenamiento y más tiempo de búsqueda que un índice primario, debido a su mayor número de entradas.

La figura 17.4 ilustra un índice secundario en el que los punteros en las index entries son punteros de bloque, no punteros de registro.

Figure 17.4
A dense secondary index (with block pointers) on a nonordering key field of a file.



Observe que un índice secundario proporciona un orden lógico en los registros por el indexing field. Si accedemos a los registros en el orden de las entradas en el índice secundario, los obtenemos en el orden del indexing field. Los índices primario y los clustering index suponen que el campo utilizado para ordenar físicamente los registros en el archivo es el mismo que el indexing field.

Table 17.1 Types of Indexes Based on the Properties of the Indexing Field

	Index Field Used for Physical Ordering of the File	Index Field Not Used for Physical Ordering of the File
Indexing field is key	Primary index	Secondary index (Key)
Indexing field is nonkey	Clustering index	Secondary index (NonKey)

Table 17.2 Properties of Index Types

Type of Index	Number of (First-Level) Index Entries	Dense or Nondense (Sparse)	Block Anchoring on the Data File
Primary	Number of blocks in data file	Nondense	Yes
Clustering	Number of distinct index field values	Nondense	Yes/no[a]
Secondary (key)	Number of records in data file	Dense	No
Secondary (nonkey)	Number of records[b] or number of distinct index field values[c]	Dense or Nondense	No

[a]Yes if every distinct value of the ordering field starts a new block; no otherwise.
[b]For option 1.
[c]For options 2 and 3.

17.2 Multilevel Indexes:

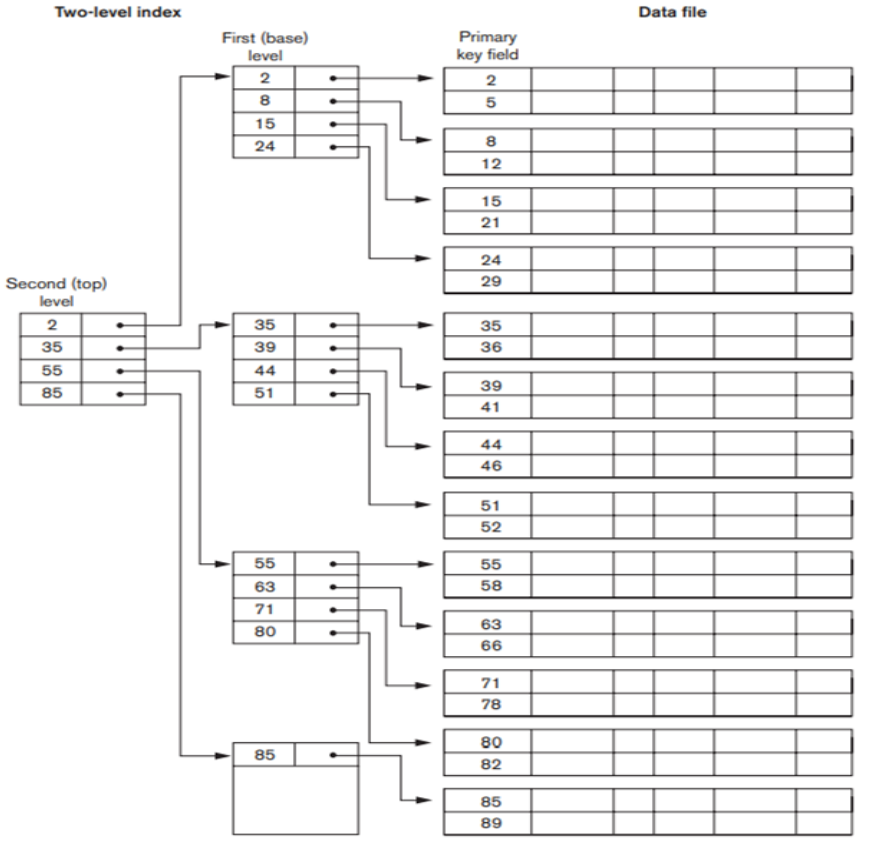
Se aplica una búsqueda binaria al índice para localizar punteros a un bloque de disco o a un registro en el archivo que tiene un valor de campo de índice específico. Una búsqueda binaria requiere aproximadamente accesos de bloque para un índice con bloques b sub i porque cada paso del algoritmo reduce la parte del archivo de índice que continuamos buscando por un factor de 2. Mientras que dividimos el espacio de búsqueda de registros en dos mitades en cada paso durante una búsqueda binaria, la dividimos de n maneras en cada paso de búsqueda usando el índice multinivel. La búsqueda de un índice multinivel requiere aproximadamente accesos de bloque, que es un número sustancialmente menor que para una búsqueda binaria si el abanico es mayor que 2.

Un índice multinivel considera el archivo índice, al que ahora nos referiremos como el primer nivel (o base) de un índice multinivel, como un archivo ordenado. Por lo tanto, al considerar el archivo de índice de primer nivel como un archivo de datos ordenados, podemos crear un índice primario para el primer nivel, este índice al primer nivel se denomina segundo nivel del índice multinivel. Debido a que el segundo nivel es un índice primario, podemos usar block anchors para que el segundo nivel tenga una entrada para cada bloque del primer nivel. El factor de bloqueo para el segundo nivel, y para todos los niveles posteriores, es el mismo que para el índice de primer nivel porque todas las entradas de índice son del mismo tamaño; cada uno tiene un valor de campo y una dirección de bloque.

El tercer nivel, que es un índice primario para el segundo nivel, tiene una entrada para cada bloque de segundo nivel. Podemos repetir el proceso anterior hasta que todas las entradas de

algún nivel de índice t quepan en un solo bloque. Este bloque en el t-avo nivel se denomina top index level. El esquema multinivel descrito aquí se puede usar en cualquier tipo de índice, ya sea primario, de agrupación o secundario, siempre que el índice de primer nivel tenga valores distintos para K (i) y entradas de longitud fija. La figura 17.6 muestra un índice multinivel construido sobre un índice primario.

Figure 17.6
A two-level primary index resembling ISAM (indexed sequential access method) organization.



Tenga en cuenta que también podríamos tener un índice primario multinivel, que no sería denso. Para un índice denso, esto puede determinarse accediendo al primer nivel de índice (sin tener que acceder a un bloque de datos), ya que hay una entrada de índice para cada registro en el archivo.

El algoritmo 17.1 describe el procedimiento de búsqueda de un registro en un archivo de datos que utiliza un índice primario multinivel no denso con niveles t. Nos referimos a la entrada i en el nivel j del índice como <Kj (i), Pj (i)>, y buscamos un registro cuyo valor de clave principal sea K. Suponemos que se ignoran los registros de desbordamiento. Si el registro está en el archivo, debe haber alguna entrada en el nivel 1 con K1 (i) ≤ K <K1 (i + 1) y el registro estará en el bloque del archivo de datos cuya dirección es P1 (i).

Algorithm 17.1. Searching a Nondense Multilevel Primary Index with t Lev

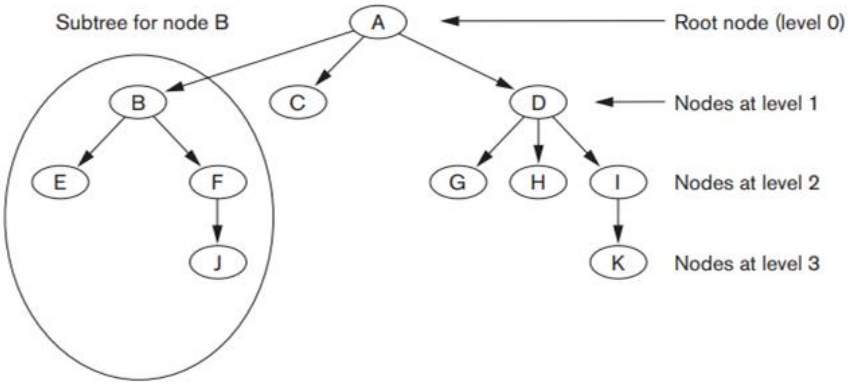
```
(*We assume the index entry to be a block anchor that is the first key per block
p ← address of top-level block of index;
for j ← t step – 1 to 1 do
    begin
        read the index block (at jth index level) whose address is p;
        search block p for entry i such that Kj (i) ≤ K < Kj(i + 1)
    (* if Kj(i)
        is the last entry in the block, it is sufficient to satisfy Kj(i) ≤ K *);
        p ← Pj(i ) (* picks appropriate pointer at jth index level *)
    end;
read the data file block whose address is p;
search block p for record with key = K;
```

Para conservar los beneficios de usar la indexación multinivel mientras se reducen los problemas de inserción y eliminación de índices, los diseñadores adoptaron un índice multinivel denominado índice dinámico multinivel que deja espacio en cada uno de sus bloques para insertar nuevas entradas y utiliza algoritmos de inserción / eliminación apropiados para crear y eliminar nuevos bloques de índice cuando el archivo de datos crece y se reduce. A menudo se implementa mediante el uso de estructuras de datos llamadas árboles B y árboles B +.

17.3 Dynamic Multilevel Indexes Using B-Trees and B+-Trees:

Los árboles B y los árboles B + son casos especiales de la conocida estructura de datos de búsqueda conocida como árbol. Cada nodo en el árbol, a excepción de un nodo especial llamado raíz, tiene un nodo primario y cero o más nodos secundarios. El nodo raíz no tiene padre. El nivel de un nodo siempre es uno más que el nivel de su padre, y el nivel del nodo raíz es cero. Una definición recursiva precisa de un subárbol es que consta de un nodo n y los subárboles de todos los nodos secundarios de n.

Figure 17.7
A tree data structure that shows an unbalanced tree.



La figura 17.7 ilustra una estructura de datos de árbol. En esta figura, el nodo raíz es A, y sus nodos secundarios son B, C y D. Los nodos E, J, C, G, H y K son nodos hoja. Dado que los nodos de las hojas están en diferentes niveles del árbol, este árbol se llama desequilibrado.

17.3.1 Search Trees and B-Trees:

Un árbol de búsqueda es un tipo especial de árbol que se utiliza para guiar la búsqueda de un registro, dado el valor de uno de los campos del registro. Al seguir un puntero, restringimos nuestra búsqueda en cada nivel a un subárbol del árbol de búsqueda e ignoramos todos los nodos que no están en este subárbol.

Un árbol de búsqueda es ligeramente diferente de un índice multinivel. Un árbol de búsqueda de orden p es un árbol tal que cada nodo contiene como máximo valores de búsqueda p - 1 y punteros de p en el orden <P1, K1, P2, K2, ..., Pq − 1, Kq − 1, Pq>, donde q ≤ p. Cada P*i* es un puntero a un nodo secundario (o un puntero NULO), y cada K*i* es un valor de búsqueda de algún conjunto ordenado de valores. Se supone que todos los valores de búsqueda son únicos. La figura 17.8 ilustra un nodo en un árbol de búsqueda.

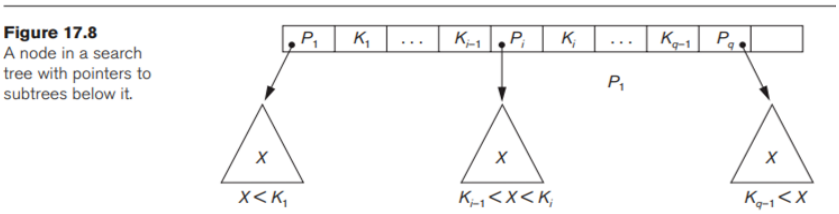


Figure 17.8
A node in a search tree with pointers to subtrees below it.

Podemos usar un árbol de búsqueda como mecanismo para buscar registros almacenados en un archivo de disco. Los valores en el árbol pueden ser los valores de uno de los campos del archivo, llamado campo de búsqueda (que es el mismo que el campo de índice si un índice multinivel guía la búsqueda). Cada valor clave en el árbol está asociado con un puntero al registro en el archivo de datos que tiene ese valor. Alternativamente, el puntero podría estar en el bloque de disco que contiene ese registro. El árbol de búsqueda en sí mismo puede almacenarse en el disco asignando cada nodo del árbol a un bloque de disco. Cuando se inserta un nuevo registro en el archivo, debemos actualizar el árbol de búsqueda insertando una entrada en el árbol que contenga el valor del campo de búsqueda del nuevo registro y un puntero al nuevo registro.

La figura 17.9 ilustra un árbol de búsqueda de orden p = 3 y valores de búsqueda de enteros. Tenga en cuenta que algunos de los punteros Pi en un nodo pueden ser punteros NULL.

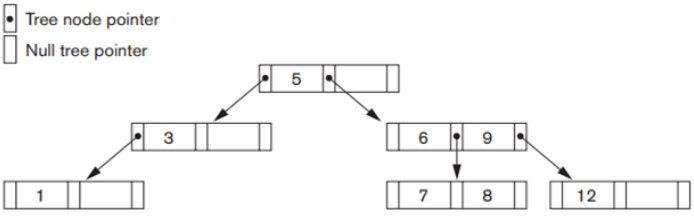


Figure 17.9
A search tree of order $p = 3$.

El árbol de la figura 17.7 no está equilibrado porque tiene nodos hoja en los niveles 1, 2 y 3. Los objetivos para equilibrar un árbol de búsqueda son los siguientes:

■ Para garantizar que los nodos se distribuyan uniformemente, de modo que la profundidad del árbol se minimice para el conjunto de claves dado y que el árbol no se sesgue con algunos nodos que se encuentran en niveles muy profundos

■ Para que la velocidad de búsqueda sea uniforme, de modo que el tiempo promedio para encontrar cualquier clave aleatoria sea aproximadamente el mismo

Minimizar el número de niveles en el árbol es un objetivo, otro objetivo implícito es asegurarse de que el index tree no necesite demasiada reestructuración ya que los registros se insertan y eliminan del archivo principal. El B-tree resuelve ambos problemas al especificar restricciones adicionales en el árbol de búsqueda.

El árbol B tiene restricciones adicionales que aseguran que el árbol siempre esté equilibrado y que el espacio perdido por eliminación, si lo hay, nunca se vuelva excesivo. Sin embargo, los algoritmos de inserción y eliminación se vuelven más complejos para mantener estas restricciones. La mayoría de las inserciones y eliminaciones son procesos simples; se complican solo en circunstancias especiales, es decir, cada vez que intentamos una inserción en un nodo que ya está lleno o una eliminación de un nodo que lo hace menos de la mitad.

La figura 17.10 (b) ilustra un árbol B de orden p = 3. Observe que todos los valores de búsqueda K en el árbol B son únicos porque asumimos que el árbol se usa como estructura de acceso en un campo clave. Si usamos un árbol B en un campo sin clave, debemos cambiar la definición de los punteros de archivo Pri para que apunten a un bloque, o un grupo de bloques, que contenga los punteros a los registros del archivo. Este nivel adicional de indirección es similar a la opción 3, discutida en la Sección 17.1.3, para índices secundarios.

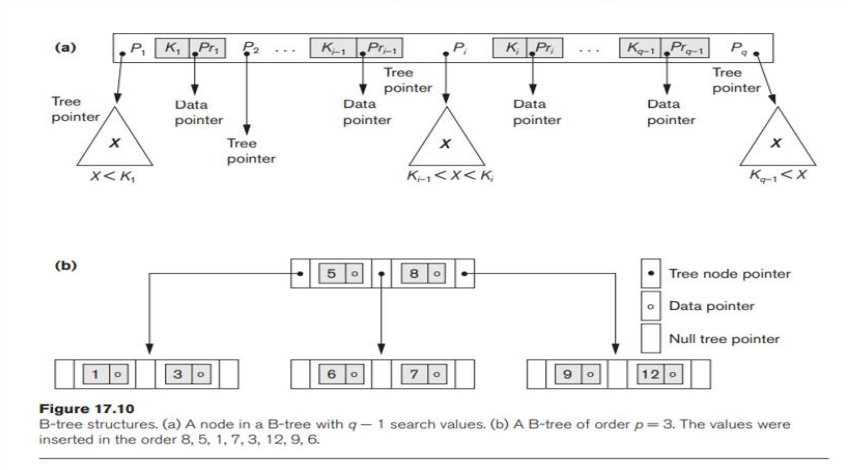


Figure 17.10
B-tree structures. (a) A node in a B-tree with $q - 1$ search values. (b) A B-tree of order $p = 3$. The values were inserted in the order 8, 5, 1, 7, 3, 12, 9, 6.

Un B-tree comienza con un único nodo raíz (que también es un nodo hoja) en el nivel 0. Una vez que el nodo raíz está lleno con valores de clave de búsqueda e intentamos insertar otra entrada en el árbol, el nodo raíz se divide en dos nodos en el nivel 1. Solo el valor medio se mantiene en el nodo raíz, y el resto de los valores se dividen en partes iguales entre los otros dos nodos. Cuando un nodo no raíz está lleno y se inserta una nueva entrada en él, ese nodo se divide en dos nodos en el mismo nivel, y la entrada central se mueve al nodo principal junto con dos punteros a los nuevos nodos divididos. Si el nodo padre está lleno, también se divide. La división puede propagarse hasta el nodo raíz, creando un nuevo nivel si la raíz se divide.

El análisis y la simulación han demostrado que, después de numerosas inserciones y eliminaciones aleatorias en un B-tree, los nodos están llenos aproximadamente en un 69% cuando el número de valores en el árbol se estabiliza. Cada nodo del B-tree puede tener como máximo *p* punteros de árbol, *p - 1* punteros de datos y *p - 1* valores de campo clave de búsqueda, ver Figura 17.10.

17.3.2 B+-Trees:

La mayoría de las implementaciones de un índice dinámico multinivel utilizan una variación de la estructura de datos del B-tree llamada B+-tree. En un B-tree, cada valor del campo de búsqueda aparece una vez en algún nivel del árbol, junto con un puntero de datos. En un B+-tree, los punteros de datos se almacenan solo en los nodos hoja del árbol; por lo tanto, la estructura de los nodos hoja difiere de la estructura de los nodos internos. Los nodos hoja tienen una entrada para cada valor del campo de búsqueda, junto con un puntero de datos al registro (o al bloque que contiene este registro) si el campo de búsqueda es un campo clave. Para un campo de búsqueda sin clave, el puntero apunta a un bloque que contiene punteros a los registros del archivo de datos, creando un nivel adicional de indirección.

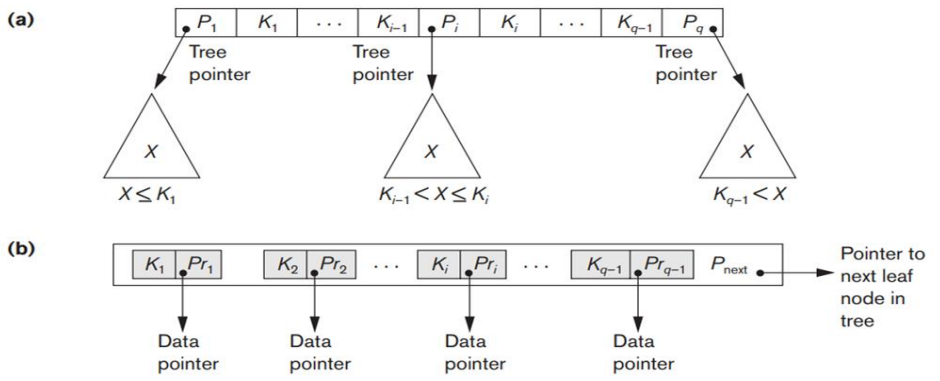


Figure 17.11
The nodes of a B^+-tree. (a) Internal node of a B^+-tree with $q - 1$ search values. (b) Leaf node of a B^+-tree with $q - 1$ search values and $q - 1$ data pointers.

Los nodos hoja del B+-tree generalmente están vinculados para proporcionar acceso ordenado en el campo de búsqueda a los registros. Estos nodos hoja son similares al primer nivel (base) de un índice. Los nodos internos del B+-tree corresponden a los otros niveles de un índice multinivel. Algunos valores de campo de búsqueda de los nodos hoja se repiten en los nodos internos del B+-tree para guiar la búsqueda. La estructura de los nodos internos de un B+-tree de orden *p* se observa en la Figura 17.11 (a). La estructura de los nodos hoja de un B+-tree de orden *p* lo muestra la Figura 17.11 (b).

Los punteros en los nodos internos son punteros de árbol a bloques que son nodos de árbol, mientras que los punteros en nodos de hoja son punteros de datos a los registros o bloques de archivos de datos, excepto el puntero P*next*, que es un puntero de árbol al siguiente nodo de hoja. Al comenzar en el nodo de hoja más a la izquierda, es posible atravesar los nodos de hoja como una lista vinculada, utilizando los punteros de P*next*. Esto proporciona acceso ordenado a los registros de datos en el campo de indexación.

Para el mismo tamaño de bloque (nodo), el orden *p* será mayor para el B+-tree que para el B-tree. Esto puede conducir a menos niveles de B+-tree, mejorando el tiempo de búsqueda. Debido a que las estructuras para los nodos internos y de hoja de un B+-tree son diferentes, el orden *p* puede ser diferente. Usaremos *p* para denotar el orden de los nodos internos y p_{leaf} para denotar el orden de los nodos hoja, que definimos como el número máximo de punteros de datos en un nodo hoja.

El algoritmo 17.2 describe el procedimiento utilizando el B+-tree como estructura de acceso para buscar un registro. El algoritmo 17.3 ilustra el procedimiento para insertar un registro en un archivo con una estructura de acceso B+-tree. Estos algoritmos suponen la existencia de un campo de búsqueda clave, y deben modificarse adecuadamente para el caso de un B+-tree en un campo sin clave.

Algorithm 17.2. Searching for a Record with Search Key Field Value K, Using a B^+-Tree

```
n ← block containing root node of B+-tree;
read block n;
while (n is not a leaf node of the B+-tree) do
    begin
    q ← number of tree pointers in node n;
    if K ≤ n.K1 (*n.Ki refers to the ith search field value in node n*)
        then n ← n.P1 (*n.Pi refers to the ith tree pointer in node n*)
        else if K > n.Kq–1
            then n ← n.Pq
            else begin
                search node n for an entry i such that n.Ki–1 < K ≤ n.Ki;
                n ← n.Pi
                end;
        read block n
        end;
search block n for entry (Ki, Pri) with K = Ki; (* search leaf node *)
if found
    then read data file block with address Pri and retrieve record
    else the record with search field value K is not in the data file;
```

Algorithm 17.3. Inserting a Record with Search Key Field Value K in a B^+-Tree of Order p

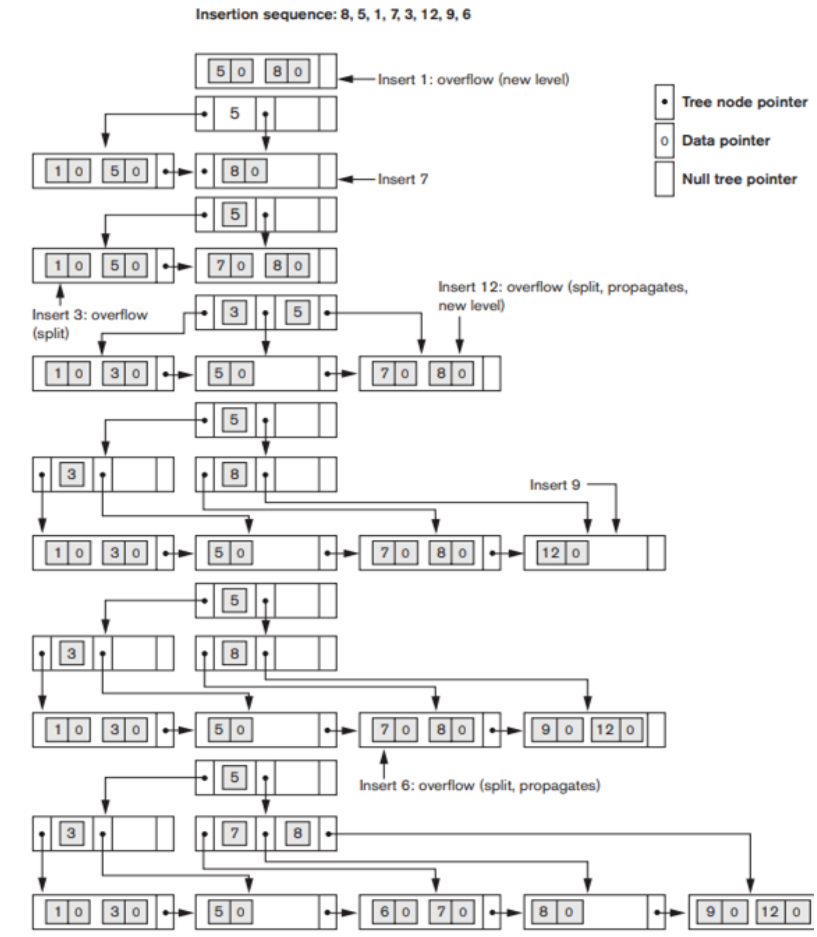
```
n ← block containing root node of B+-tree;
read block n; set stack S to empty;
while (n is not a leaf node of the B+-tree) do
    begin
    push address of n on stack S;
        (*stack S holds parent nodes that are needed in case of split*)
    q ← number of tree pointers in node n;
    if K ≤ n.K1 (*n.Ki refers to the ith search field value in node n*)
        then n ← n.P1 (*n.Pi refers to the ith tree pointer in node n*)
        else if K ← n.Kq–1
            then n ← n.Pq
            else begin
                search node n for an entry i such that n.Ki–1 < K ≤ n.Ki;
                n ← n.Pi
                end;
        read block n
    end;
search block n for entry (Ki,Pri) with K = Ki; (*search leaf node n*)
if found
    then record already in file; cannot insert
    else (*insert entry in B+-tree to point to record*)
        begin
        create entry (K, Pr) where Pr points to the new record;
        if leaf node n is not full
            then insert entry (K, Pr) in correct position in leaf node n
            else begin (*leaf node n is full with pleaf record pointers; is split*)
                copy n to temp (*temp is an oversize leaf node to hold extra entries*);

                new ← remaining entries in temp; K ← Kj ;
                (*now we must move (K, new) and insert in parent internal node;
                  however, if parent is full, split may propagate*)
                finished ← false;
                repeat
                if stack S is empty
                    then (←no parent node; new root node is created for the tree*)
                        begin
                        root ← a new empty internal node for the tree;
                        root ← <n, K, new>; finished ← true;
                        end
                    else begin
                        n ← pop stack S;
                        if internal node n is not full
                            then
                                begin (*parent node not full; no split*)
                                insert (K, new) in correct position in internal node n;
                                finished ← true
                                end
                            else begin (*internal node n is full with p tree pointers;
                                        overflow condition; node is split*)
                            copy n to temp (*temp is an oversize internal node*);
                            insert (K, new) in temp in correct position;
                            (*temp now has p + 1 tree pointers*)
                            new ← a new empty internal node for the tree;
                            j ← ⌊((p + 1)/2⌋ ;
                            n ← entries up to tree pointer Pj in temp;
                            (*n contains <P1, K1, P2, K2, ... , Pj–1, Kj–1, Pj >*)
                            new ← entries from tree pointer Pj+1 in temp;
                            (*new contains < Pj+1, Kj+1, ... , Kp–1, Pp, Kp, Pp+1 >*)
                            K ← Kj
                            (*now we must move (K, new) and insert in
                              parentinternal node*)
                        end
                    end
                until finished
                end;
        end;
```

La figura 17.12 ilustra la inserción de registros en un B+-tree de orden p = 3 y p_{leaf} = 2. Primero, observamos que la raíz es el único nodo en el árbol, por lo que también es un nodo hoja. Tan

pronto como se crea más de un nivel, el árbol se divide en nodos internos y nodos hoja. Observe también que cada valor que aparece en un nodo interno también aparece como el valor más a la derecha en el nivel de hoja del subárbol al que apunta el puntero del árbol a la izquierda del valor.

Figure 17.12
An example of insertion in a B^+-tree with $p = 3$ and $p_{leaf} = 2$.



Cuando un nodo hoja está lleno y se inserta una nueva entrada allí, el nodo se desborda y debe dividirse. Las primeras entradas *j* en el nodo original se mantienen allí, y las entradas restantes se mueven a un nuevo nodo hoja. El valor de búsqueda *j*th se replica en el nodo interno primario y se crea un puntero adicional al nuevo nodo en el nodo primario. Estos deben insertarse en el nodo padre en su secuencia correcta. Si el nodo interno principal está lleno, el nuevo valor también hará que se desborde, por lo que debe dividirse. Un nuevo nodo interno mantendrá las entradas desde P*j* + 1 hasta el final de las entradas en el nodo (ver Algoritmo 17.3). Esta división puede propagarse hacia arriba para crear un nuevo nodo raíz y, por lo tanto, un nuevo nivel para el B+-tree.

La figura 17.13 ilustra la eliminación de un B+-tree. Cuando se elimina una entrada siempre se elimina del nivel de hoja. Si ocurre en un nodo interno, también debe eliminarse de ahí, aquí el valor a su izquierda en el nodo hoja debe reemplazarlo en el nodo interno porque ese valor es ahora la entrada más a la derecha en el subárbol. La eliminación puede causar un desbordamiento al reducir el número de entradas en el nodo hoja por debajo del mínimo requerido. En este caso,

tratamos de encontrar un nodo hoja hermano (un nodo hoja a la izquierda o la derecha) y redistribuir las entradas entre el nodo y su hermano para que ambos estén al menos medio llenos, de lo contrario, el nodo se fusiona con sus hermanos y se reduce el número de nodos hoja. Un método común es intentar redistribuir entradas con el hermano izquierdo, si esto no es posible, se intenta redistribuir con el hermano correcto, si esto tampoco es posible los tres nodos se fusionan en dos nodos hoja. En ese caso el flujo inferior puede propagarse a los nodos internos porque se necesita un puntero de árbol menos y un valor de búsqueda. Esto puede propagar y reducir los niveles de los árboles.

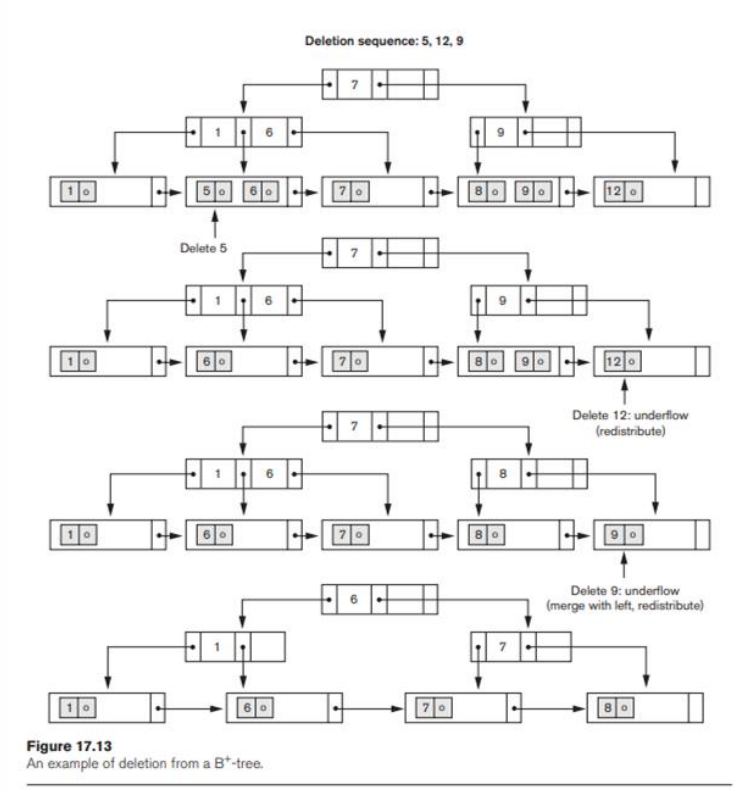


Figure 17.13
An example of deletion from a B^+-tree.

En algunos casos, la restricción 5 en el B-tree, que requiere que cada nodo esté al menos medio lleno, se puede cambiar para requerir que cada nodo esté lleno al menos dos tercios. En este caso, el B-tree ha sido llamado B*-tree. En general, algunos sistemas permiten al usuario elegir un factor de relleno entre 0.5 y 1.0, donde este último significa que los nodos del B-tree deben estar completamente llenos.

17.4 Indexes on Multiple Keys:

En muchas solicitudes de recuperación y actualización, hay varios atributos involucrados. Si se usa con frecuencia una determinada combinación de atributos, es ventajoso configurar una estructura de acceso para proporcionar un acceso eficiente mediante un valor clave que sea una combinación de esos atributos. Por ejemplo, considere un archivo EMPLEADO que contiene los atributos Dno (número de departamento), Edad, Calle, Ciudad, Código postal, Salario y Código de habilidad, con la clave de Ssn (número de Seguro Social). Considere la consulta: haga una lista de los empleados

en el departamento número 4 cuya edad es 59 años. Tenga en cuenta que tanto Dno como Edad son atributos que no son clave, lo que significa que un valor de búsqueda para cualquiera de estos puntos apuntará a múltiples registros. Se pueden considerar las siguientes estrategias de búsqueda alternativas:

1. Suponiendo que Dno tenga un índice, pero Age no, acceda a los registros que tienen Dno = 4 utilizando el índice, y luego seleccione de entre ellos aquellos registros que satisfagan Age = 59.
2. Alternativamente, si Age está indexado pero Dno no, acceda a los registros que tienen Age = 59 utilizando el índice, y luego seleccione entre ellos aquellos registros que satisfagan Dno = 4.
3. Si se han creado índices tanto en Dno como en Age, se pueden usar ambos índices; cada uno proporciona un conjunto de registros o un conjunto de punteros (a bloques o registros). Una intersección de estos conjuntos de registros o punteros produce aquellos registros o punteros que satisfacen ambas condiciones.

Todas estas alternativas finalmente dan el resultado correcto, sin embargo, si el conjunto de registros que cumplen con cada condición individualmente es grande, pero solo unos pocos registros satisfacen la condición combinada, entonces ninguno de los anteriores es una técnica eficiente para la solicitud de búsqueda dada. Existen varias posibilidades que tratarían la combinación <Dno, Age> o <Age, Dno> como una clave de búsqueda compuesta de múltiples atributos. Nos referiremos a las claves que contienen múltiples atributos como claves compuestas.

17.4.1 Ordered Index on Multiple Attributes:

Toda la discusión en este capítulo hasta ahora todavía se aplica si creamos un índice en un campo clave de búsqueda que es una combinación de <Dno, Age>. La clave de búsqueda es un par de valores <4, 59> en el ejemplo anterior. En general, si se crea un índice en los atributos <A1, A2, ..., An>, los valores de la clave de búsqueda son tuplas con n valores: <v1, v2, ..., vn>.

Un orden lexicográfico de estos valores de tupla establece un orden en esta clave de búsqueda compuesta. Para nuestro ejemplo, todas las claves de departamento para el número de departamento 3 preceden a las del número de departamento 4. Por lo tanto, <3, n> precede a <4, m> para cualquier valor de m y n. El ordenamiento lexicográfico funciona de manera similar al ordenamiento de cadenas de caracteres.

17.4.2 Partitioned Hashing:

El partitioned hashing es una extensión del static external hashing que permite el acceso en múltiples claves. Es adecuado solo para comparaciones de igualdad; las consultas de rango no son compatibles. En el partitioned hashing, para una clave que consta de *n* componentes, la función hash está diseñada para producir un resultado con *n* direcciones hash separadas. La bucket address es una concatenación de estas *n* direcciones. Entonces es posible buscar la composite search key requerida buscando los buckets apropiados que coinciden con las partes de la dirección en la que estamos interesados.

Por ejemplo, considere la clave de búsqueda compuesta. Si Dno y Age se combinan en una dirección de 3 bits y 5 bits respectivamente, obtenemos una dirección de depósito de 8 bits.

Suponga que Dno = 4 tiene una dirección hash '100' y Age = 59 tiene una dirección hash '10101'. Luego, para buscar el valor de búsqueda combinado, Dno = 4 y Edad = 59, uno va a la dirección de depósito 100 10101; solo para buscar a todos los empleados con Edad = 59, se buscarán todos los buckets (ocho de ellos) cuyas direcciones sean '000 10101', '001 10101', ... y así sucesivamente. Una ventaja del partitioned hashing es que se puede extender fácilmente a cualquier número de atributos, y el principal inconveniente es que no puede manejar consultas de rango en ninguno de los component attributes. Además, la mayoría de las funciones hash no mantienen registros en orden por la clave que se está haciendo hash. Por lo tanto, acceder a los registros en orden lexicográfico mediante una combinación de atributos como el que se usa como clave no sería sencillo ni eficiente.

17.4.3 Grid Files:

Otra alternativa es organizar el archivo EMPLEADO como un grid file. Si queremos acceder a un archivo con dos claves, digamos Dno y Age como en nuestro ejemplo, podemos construir una grid array con una escala lineal para cada uno de los atributos de búsqueda. La figura 17.14 muestra una grid array para el archivo EMPLOYEE con una escala lineal para Dno y otra para el atributo Age. La grid array que se muestra para este archivo tiene un total de 36 celdas.

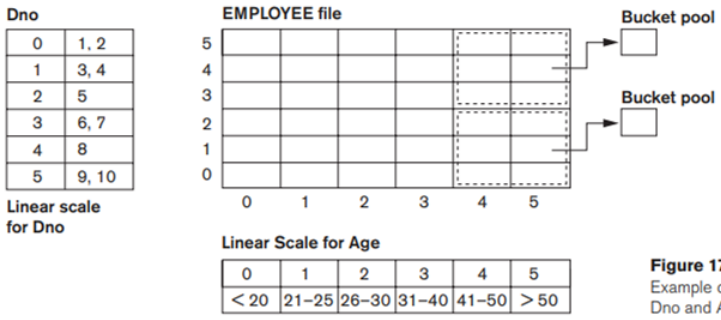


Figure 17.14
Example of a grid array on Dno and Age attributes.

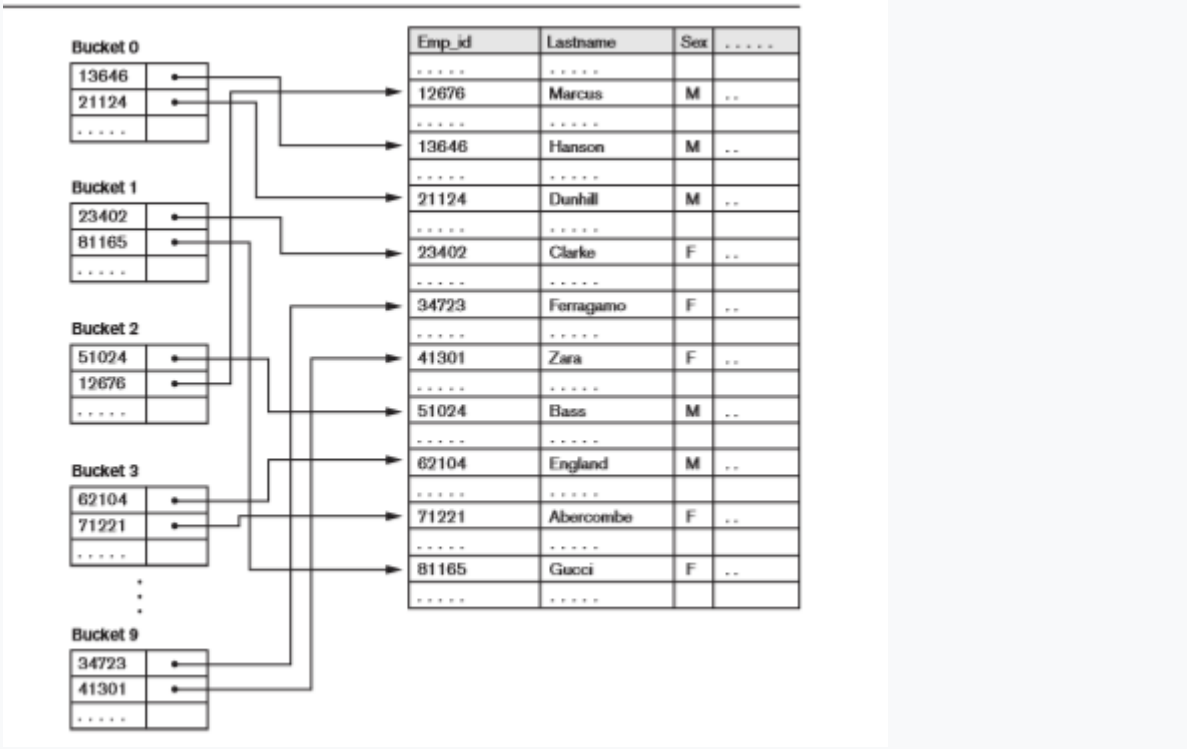
Este método es particularmente útil para consultas de rango que se asignarían a un conjunto de celdas correspondientes a un grupo de valores a lo largo de las escalas lineales. Si una consulta de rango corresponde a una coincidencia en algunas de las grid cells, puede procesarse accediendo exactamente a los buckets de esas grid cells.

La grid array permite así una partición del archivo a lo largo de las dimensiones de los atributos clave de búsqueda y proporciona un acceso por combinaciones de valores a lo largo de esas dimensiones. Los grid files funcionan bien en términos de reducción de tiempo para acceso de clave múltiple. Sin embargo, representan una sobrecarga de espacio en términos de la estructura del grid array. Además, con los archivos dinámicos, una reorganización frecuente del archivo aumenta el costo de mantenimiento.

17.5 Otro tipo de índices

17.5.1 Hash Index: Es una estructura secundaria para acceder a archivos mediante el uso de hash en una clave de búsqueda distinta a la utilizada para la organización del archivo de datos primario. Las entradas del índice son tipo <K,Pr> o <K,P>. Pr es un puntero al registro que contiene la clave o P es un puntero al bloque que contiene el registro para esa clave. Se puede organizar como un archivo hash expandible dinámicamente. la búsqueda

de una entrada usa el algoritmo de búsqueda hash en K. Una vez que se encuentra una entrada, el puntero Pr (o P) se usa para ubicar el registro correspondiente en el archivo de datos.



La imagen muestra un índice hash en el campo Emp_id para un archivo que se ha almacenado como un archivo secuencial ordenado por Nombre. Emp_id se convierte en un número de depósito utilizando una función de resumen: la suma de los dígitos de Emp_id módulo 10. Por ejemplo, para encontrar Emp_id 51024, la función hash da como resultado el número de depósito 2; ese cubo se accede primero. Contiene la entrada de índice <51024, Pr>; el puntero Pr nos lleva al registro real en el archivo.

17.5.2 Bitmap Indexes: se usa para relaciones que contienen una gran cantidad de filas. Crea un índice para una o más columnas y se indexa cada valor o rango de valores en esas columnas. Por lo general, se crea un **Bitmap Index** para aquellas columnas que contienen un número bastante pequeño de valores únicos. Para construir **Bitmap Index** en un conjunto de registros en una relación, los registros deben estar numerados del 0 al n con una identificación que se pueda asignar a una dirección física compuesta por un número de bloque y un registro de desplazamiento dentro del bloque. Se basa en un valor particular de un campo particular y es solo una matriz de bits. Para un campo dado, hay un bitmap index separado (o un vector) mantenido correspondiente a cada valor único en la base de datos.

EMPLOYEE

Row_id	Emp_id	Lname	Sex	Zipcode	Salary_grade
0	51024	Bass	M	94040	..
1	23402	Clarke	F	30022	..
2	62104	England	M	19046	..
3	34723	Ferragamo	F	30022	..
4	81165	Gucci	F	19046	..
5	13646	Hanson	M	19046	..
6	12676	Marcus	M	30022	..
7	41301	Zara	F	94040	..

Bitmap index for Sex

M	F
10100110	01011001

Bitmap index for Zipcode

Zipcode 19046	Zipcode 30022	Zipcode 94040
00101100	01010010	10000001

La imagen muestra la relación EMPLEADO con las columnas Emp_id, Lname, Sex, Zipcode y Salary_grade y un índice de mapa de bits para las columnas Sex y Zipcode. Como ejemplo, si el mapa de bits para Sex = F, los bits para Row_ids 1, 3, 4 y 7 se establecen en 1, y el resto de los bits se establecen en 0.
Para encontrar empleados con Sexo = F y zipcode = 30022, interceptamos los mapas de bits "01011001" y "01010010" que producen Row_ids 1 y 3. Los empleados que no viven en Zipcode = 94040 se obtienen al complementar el vector de bits "10000001" y se obtienen Row_ids 1 a 6. Supongamos que una columna tiene 5 valores distintos y otra tiene 10 valores distintos, se puede considerar que la condición de unión en estos dos tiene una selectividad de 1 / 50 (= 1/5 * 1/10).
Cuando se eliminan los registros, la numeración de filas y el desplazamiento de bitmap index se vuelve costoso. Se puede utilizar otro mapa de bits, denominado **existence bitmap**, para evitar este gasto. Tiene un bit 0 para las filas que se han eliminado, pero todavía están físicamente presentes y un bit para las filas que realmente existen.

Bitmaps for B+-Tree Leaf Nodes (Bimaps para nodos de hoja de árbol B +-):

Bitmaps se pueden usar en los nodos hoja de Índices B + -tree, así como para señalar el conjunto de registros que contienen cada valor específico del campo indexado en el nodo hoja. Cuando el B +-tree se construye en un campo de búsqueda sin clave, el registro de hoja debe contener una lista de punteros de registro junto a cada valor del atributo indexado. Para los frecuentes se puede almacenar un índice de bimaps en lugar de los punteros.

17.5.3 Function-Based Indexing

function-based indexing: crear un índice tal que el valor que resulta de aplicar la misma función en un campo o una colección de campos se convierta en la clave del índice.
CREATE INDEX upper_ix ON Employee (UPPER(Lname)); Esta instrucción crea un function-based indexing (SUPERIOR (Lname)) en la tabla empleado, basada en una

representación en mayúscula de la columna Lname. eje SUPERIOR ('Smith') devolverá "SMITH".
Asegura que el sistema Oracle Database usará el índice en lugar de realizar una exploración completa de la tabla. Ejemplo, la siguiente consulta usará el índice:

```
SELECT First_name, Lname
FROM Employee
WHERE UPPER(Lname)= "SMITH".
```

Sin el function-based index, una base de datos Oracle podría realizar un escaneo completo de la tabla.
Eje 2) La tabla empleado tiene dos campos salario y comisión_pct y se crea un index para sumar el salario y la comisión.

```
CREATE INDEX income_ix
ON Employee(Salary + (Salary*Commission_pct));
```

La siguiente consulta usa el índice income_ix, aunque los campos salario y comisión_pct se producen en el orden inverso en la consulta en comparación con la definición del índice.

```
SELECT First_name, Lname
FROM Employee
WHERE ((Salary*Commission_pct) + Salary ) > 15000;
```

Eje 3) crea un único function-based index **basado en funciones en la tabla PEDIDOS evita que un cliente se aproveche de una identificación de promoción más de una vez. Crea un índice compuesto en los campos Customer_id y Promotion_id juntos,crea un índice Customer_id único.**

```
CREATE UNIQUE INDEX promo_ix ON Orders
(CASE WHEN Promotion_id = 2 THEN Customer_id ELSE NULL END,
CASE WHEN Promotion_id = 2 THEN Promotion_id ELSE NULL END);
```

al usar la instrucción CASE, el objetivo es eliminar del índice cualquier fila donde Promotion_id no sea igual a 2. La base de datos no se almacena en el indice B + -tree cualquier fila donde todas las claves son NULL

17.6 Some General Issues Concerning Indexing

17.6.1 Logical versus Physical Indexes: supusimos que las entradas de índice <K, Pr> (o <K, P>) siempre incluyen un puntero físico Pr (o P) que especifica la dirección de registro físico en el disco como un número de bloque y desplazamiento (esto es **physical index**). tiene la desventaja de que el puntero debe cambiarse si el registro se mueve a otra ubicación del disco
logical index: se usa para remedia esta situación. Las entradas de índice tienen la forma <K, Kp>. Cada entrada tiene un valor K para el campo de indexación secundario que coincide con el valor Kp del campo utilizado para la organización del archivo primario. Al buscar el índice secundario en el valor de K, un programa puede localizar el valor correspondiente de Kp y usarlo para acceder al registro a través de la organización de archivos primaria, usando un índice primario si está disponible. Se usan cuando se espera que las direcciones de registros físicos cambien con frecuencia

17.6.2 Index Creation: La forma general de este comando es:

CREATE [UNIQUE] INDEX <index name>

ON <table name> (<column name> [<order>] { , <column name> [<order>] })
[CLUSTER] ;
Los ketwords Unique y cluster (se usa cuando el índice a crear también debe ordenar los registros del archivo de datos en el atributo de indexación) son opcionales.
especificar CLUSTER en un atributo clave (único) crearía alguna variación de un índice primario, mientras que en un atributo no clave (no único) crearía alguna variación de un índice de agrupación. El valor para <orden> puede ser ASC (ascendente) (esta por defecto)o DESC (descendente). Eje
CREATE INDEX DnoIndex
ON EMPLOYEE (Dno)
CLUSTER ;

Index Creation Process:

En muchos sistemas, un índice no es una parte integral del archivo de datos, pero puede crearse y descartarse dinámicamente. Es por eso que a menudo se llama una *access structure. Cuando accedemos a un campo con frecuencia podemos solicitar al DBMS que cree un índice en ese campo como se muestra arriba para el noIndex. Por lo general, se crea un índice secundario para evitar el orden físico de los registros en el archivo de datos en el disco. Su ventaja es que pueden crearse en conjunto con prácticamente cualquier organización de registros primaria.* P*odría usarse para complementar otros métodos de acceso primario.*
bulk loading*: Proceso de inserción de una gran cantidad de entradas en el índice.*
Indexing of Strings: Strings puede ser de longitud variable y los strings pueden ser demasiado largos limitando el despliegue. Si se va a construir un índice B + -tree con un strings como clave de búsqueda, puede haber un número desigual de claves por nodo de índice y el despliegue puede variar. Algunos nodos pueden verse obligados a dividirse cuando se llenan.
The technique of prefix: solo almacena el prefijo de la clave de búsqueda adecuada para distinguir las claves que se separan y se dirigen al subárbol.

17.6.3 Tuning Indexes:

La elección inicial de índices puede tener que revisarse por las siguientes razones
1. Ciertas consultas pueden tardar demasiado en ejecutarse por falta de un índice.
2. Ciertos índices pueden no ser utilizados en absoluto.
3. Ciertos índices pueden sufrir demasiadas actualizaciones porque el índice está en un atributo que sufre cambios frecuentes.
La mayoría de los DBMS tienen una función de comando o rastreo, que el DBA puede usar para pedirle al sistema que muestre cómo se ejecutó una consulta: qué operaciones se realizaron en qué orden y qué estructuras de acceso secundario (índices) se usaron.
El objetivo del tunnig es evaluar dinámicamente los requisitos, para reorganizar los índices y las organizaciones de archivos para obtener el mejor rendimiento general.
rebuilding the index: eliminar o crear índices y cambiar de un índice no agrupado a uno agrupado y viceversa, puede mejorar el rendimiento.

17.6.4 Additional Issues Related to Storage of Relations and Indexes

Using an Index for Managing Constraints and Duplicates:

Es común usar un índice para imponer una restricción clave en un atributo. Mientras busca en el índice para insertar un nuevo registro, es sencillo verificar si hay otro registro con la misma clave, si es así puede ser rechazada.
Si se crea un índice en un campo sin clave, se producen duplicados, El manejo de estos duplicados es un problema que los proveedores de productos DBMS tienen que enfrentar y afecta el almacenamiento de datos, así como la creación y administración de índices, si hay claves duplicadas los registros de estas están en un mismo bloque o en varios donde son posibles los datos duplicados. Algunos sistemas agregan un id de fila al registro para que tengan identificadores únicos.
Inverted Files and Other Access Methods: fully inserted file: archivo que tiene un índice secundario en cada uno de sus campos, porque todos los index son secundarios. Se insertan nuevos registros al final del archivo; por lo tanto, es un archivo desordenado. Los índices generalmente se implementan como B+-trees por lo que se actualizan dinámicamente para reflejar la inserción o eliminación de registros.

virtual storage access method (**VSAM**) (IBM method): Similar a una Estructura de acceso B + –tree

Using Indexing Hints in Queries: hints suggest: sugieren el uso de un índice para mejorar la ejecución de una consulta. Aparecen como un comentario especial. Eje, para recuperar la SSA, el salario y el número de departamento para los empleados que trabajan en números de departamento con no menos de 10.
SELECT /*+ INDEX (EMPLOYEE emp_dno_index) */ Emp_ssn, Salary, Dno
FROM EMPLOYEE
WHERE Dno < 10;
Incluye una pista para usar un índice válido llamado emp_dno_index (índice sobre la relación EMPLEADO en Dno)

Column-Based Storage of Relations: Para los warehouses que son bases de datos de solo lectura, el almacenamiento basado en columnas ofrece ventajas. Los RDBMS de almacenamiento de columnas consideran almacenar cada columna de datos individualmente y ofrecen ventajas de rendimiento en las siguientes áreas:

1. Particionar verticalmente la tabla columna por columna, de modo que se pueda construir una tabla de dos columnas para cada atributo y, por lo tanto, solo se pueda acceder a las columnas necesarias
2. Usar índices en columnas y unir índices en varias tablas para responder consultas sin tener que acceder a las tablas de datos
3. Uso de vistas materializadas para admitir consultas en varias columnas.

Column-wise storage: Permite una libertad adicional en la creación de índices, tales como los índices de mapa de bits**.**
Physical Database Design in Relational Databases

17.7.1 Factors That Influence Physical Database Design: El objetivo no es solo crear la estructura adecuada de los datos en el almacenamiento, sino que también garantice un buen rendimiento. Factores:

A. Analyzing the Database Queries and Transactions: Antes de emprender el diseño de la DB física, debemos tener una buena idea del uso previsto de la BD definiendo en un alto nivel las consultas y transacciones que se esperan ejecutar en la base de datos. Para cada consulta de recuperación, se necesitaría la siguiente información sobre la consulta;
1. Los archivos (relaciones) a los que accederá la consulta
2. Los atributos en los que se especifican las condiciones de selección para la consulta
3. Si la condición de selección es una condición de igualdad, desigualdad o rango
4. Los atributos en los que se especifican condiciones de unión o condiciones para vincular múltiples tablas u objetos para la consulta
5. Los atributos cuyos valores serán recuperados por la consulta

Para cada operación de actualización o transacción de actualización, la siguiente información sería necesario:

1. Los archivos que se actualizarán.
2. El tipo de operación en cada archivo (insertar, actualizar o eliminar)
3. Los atributos en los que se especifican las condiciones de selección para una eliminación o actualización
4. Los atributos cuyos valores serán cambiados por una operación de actualización

B. Analyzing the Expected Frequency of Invocation of Queries and Transactions: La información de frecuencia, junto con la información del atributo recopilada en cada consulta y transacción, se utiliza para compilar una lista acumulativa de la frecuencia de uso esperada para todas las consultas y transacciones.

C. Analyzing the Time Constraints of Queries and Transactions: Algunas consultas y las transacciones pueden tener restricciones de rendimiento estrictas.

D. Analyzing the Expected Frequencies of Update Operations: Se debe especificar un número mínimo de rutas de acceso para un archivo que se actualiza con frecuencia, porque la actualización de las rutas de acceso mismas relentiza las operaciones de actualización.

E. Analyzing the Uniqueness Constraints on Attributes. Las rutas de acceso deben especificarse en todos los atributos clave candidatos que sean la clave principal de un archivo o atributos únicos. La existencia de un índice hace que sea suficiente buscar solo el índice cuando se verifica esta restricción de unicidad

17.7.2 Physical Database Design Decisions

Design Decisions about Indexing: los atributos cuyos valores son obligatorios en condiciones de igualdad o rango son aquellas que son claves o que participan en condiciones de unión.
Las decisiones de diseño físico para la indexación se dividen en las siguientes categorías:

1. **Whether to index an attribute:** Para crear un índice en un atributo clave (única) o debe haber alguna consulta que use ese atributo en una condición de selección o en una condición de unión

2. **What attribute or attributes to index on:** Se puede construir un índice en un solo atributo o en más de un atributo si es un índice compuesto
3. **Whether to set up a clustered index:** la decisión sobre cuál debería ser el índice primario o de agrupamiento depende de si es necesario mantener la tabla ordenada en ese atributo.
4. **Whether to use a hash index over a tree index:** Tree index admite consultas de igualdad y rango en el atributo. Los índices hash funcionan bien con condiciones de igualdad, pero no admite consultas de rango.
5. **Whether to use dynamic hashing for the file:** Para archivos que son muy volátiles: es decir, aquellos que crecen y se reducen continuamente