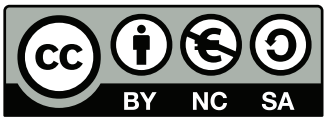
Fundamentos de Bases de Datos

LibreIM

Doble Grado de Informática y Matemáticas

Universidad de Granada

libreim.github.io/apuntesDGIIM



Fundamentos de Bases de Datos

LibreIM

Doble Grado de Informática y Matemáticas

Universidad de Granada

libreim.github.io/apuntesDGIIM

Índice

1 Tema 4. Nivel Interno

1.1 Introducción

1.2 Importancia de la organización de los datos

Sabemos que una **BD** sirve para almacenar de forma permanente grandes cantidades de datos con el propósito principal de gestionar los datos y su almacenamiento de forma eficiente. Esto afecta a su **organización lógica** de los datos y a su **organización física**.

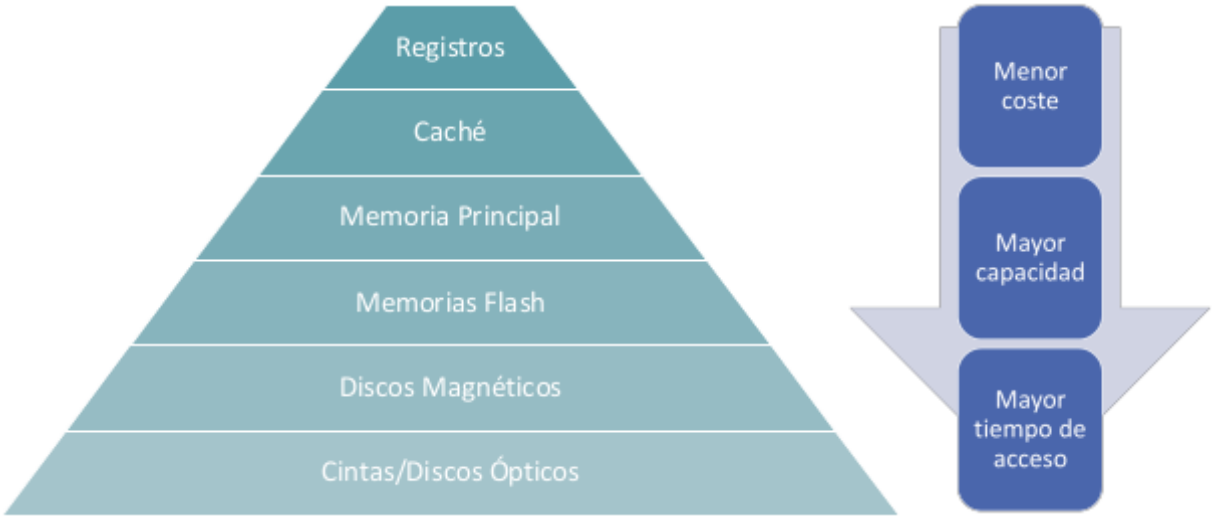
1.2.1 Nivel interno

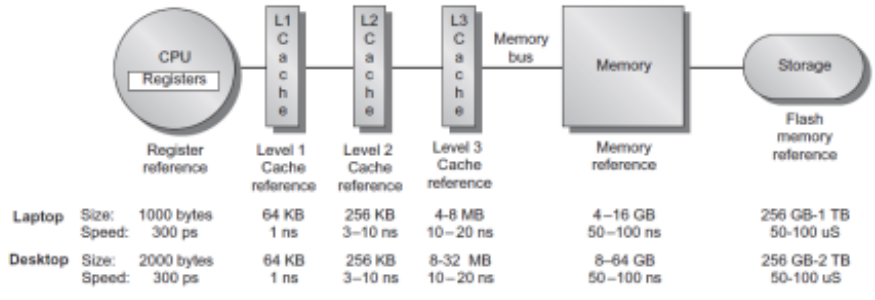
El **nivel interno** expresa en última instancia, las operaciones sobre los datos (creación, alteración y recuperación) en términos de actuación sobre unidades mínimas de almacenamiento denominadas **páginas o bloques** de base de datos. Está implementado en el **SGBD** y provee al administrador mecanismos para optimizar el acceso y almacenamiento de datos.

1.2.2 Nivel físico

El **nivel físico** está implementado en el SO y proporciona al SGBD una capa de abstracción sobre el hardware. Este nivel accede a los medios de almacenamiento mediante llamadas a los servicios del sistema de archivos proporcionado por el SO.

1.2.3 Jerarquía de memoria





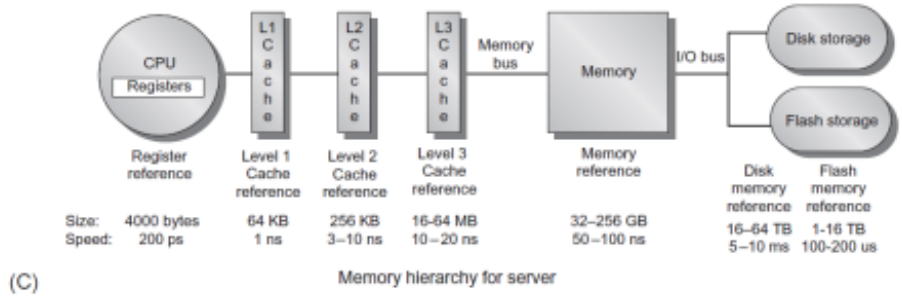


Figura tomada de John. L. Hennessy, David A. Patterson: Computer Architecture. A Quantitative Approach, 6th Edition, Morgan Kaufmann, 2017.

1.2.4 Dispositivos de almacenamiento

Existen diferentes dispositivos de almacenamiento con diferentes características. En orden creciente de coste, capacidad y tiempo de acceso los más importantes son: Registros, Caché, Memoria Principal, Discos Magnéticos, Discos ópticos.

1.2.5 Memoria principal

La **memoria principal** es el dispositivo de almacenamiento primario de los ordenadores. Hace trabajos de caché de la porción de la BD que ha sido usada más recientemente. Es el elemento de almacenamiento intermedio que ubica de forma temporal los datos afectados por las operaciones.

El **nivel interno** debe optimizar su uso:

- Es **rápida**: acelera el procesamiento.
- Es **cara**: hay que optimizar su uso.

Es **volátil**, su información se pierde cuando se apaga el sistema (o se cae). El nivel interno debe garantizar que la información contenida tenga un respaldo en almacenamiento secundario para que no se pierda.

Se utilizan distintos niveles de caché para acelerar el acceso a los datos.

1.2.6 Disco duro

El **disco duro** es el dispositivo de almacenamiento más usado en BD. Son conjuntos de discos magnéticos de dos caras donde cada cara tiene un conjunto de pistas concéntricas (cilindro: la misma pista de todas las caras) que se dividen en sectores con la misma capacidad de almacenamiento (bloque). Un bloque se puede localizar en un cilindro, la superficie de disco o un sector. Los parámetros son: capacidad, tiempo medio de acceso, RPM y velocidad sostenida de lectura/escritura.

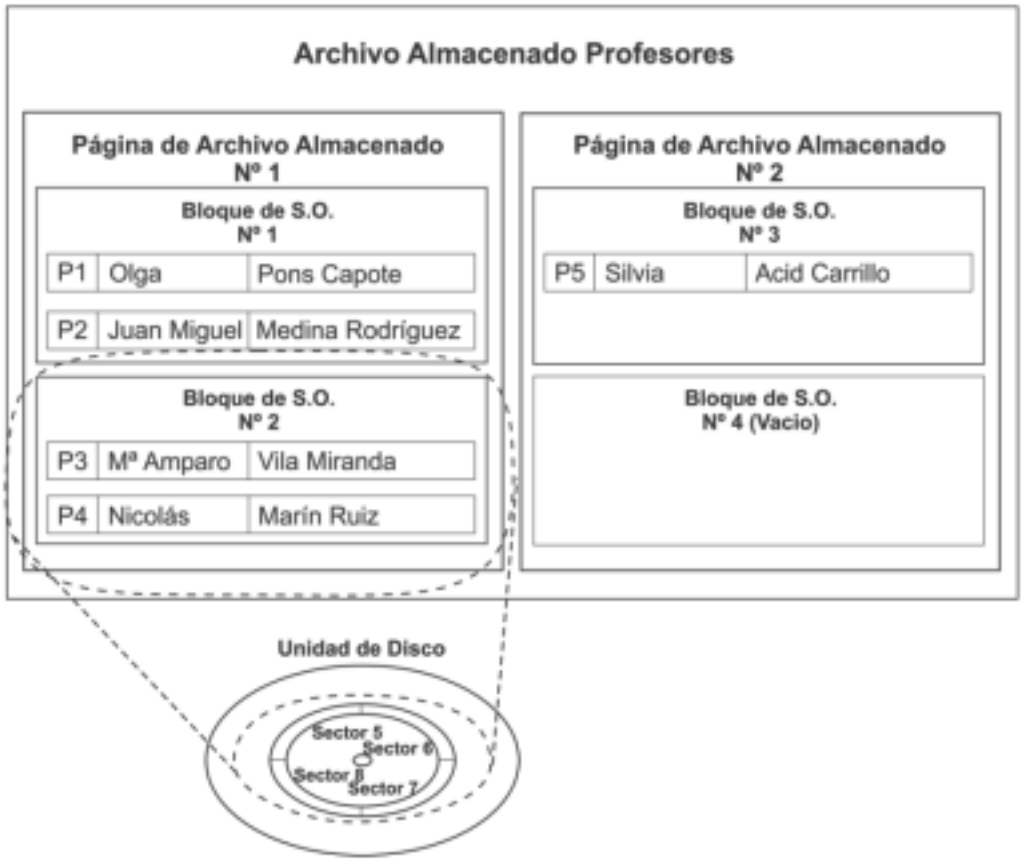
Con respecto a las medidas de rendimiento, cada dispositivo tiene:

- **Tiempo medio de acceso (*ta*)**: tiempo medio transcurrido entre una instrucción y la obtención de la información.
- **Tiempo medio de búsqueda (*tb*)**: tiempo medio de posicionamiento en pista.
- **Tiempo de latencia rotacional (*tl*)**: tiempo medio de posicionamiento en sector.

$$ta = tb + tl$$

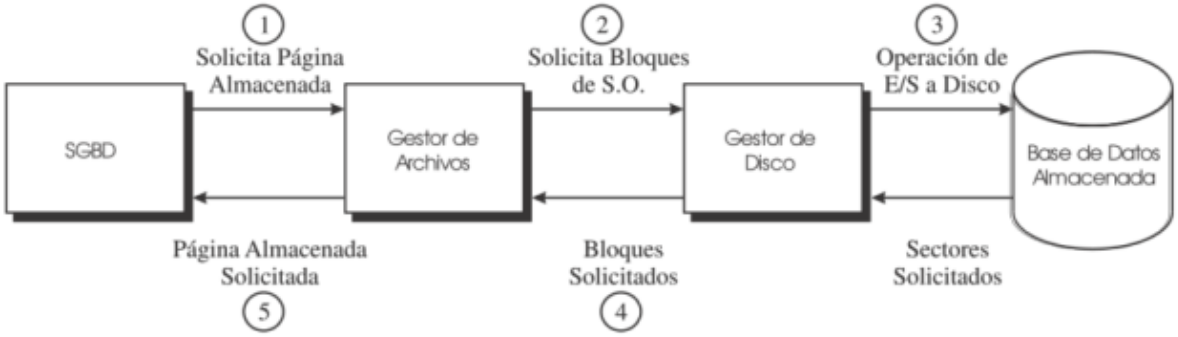
- **Tiempo medio entre fallos (*MTBF*)**.

1.3 Método de acceso a la BD almacenada

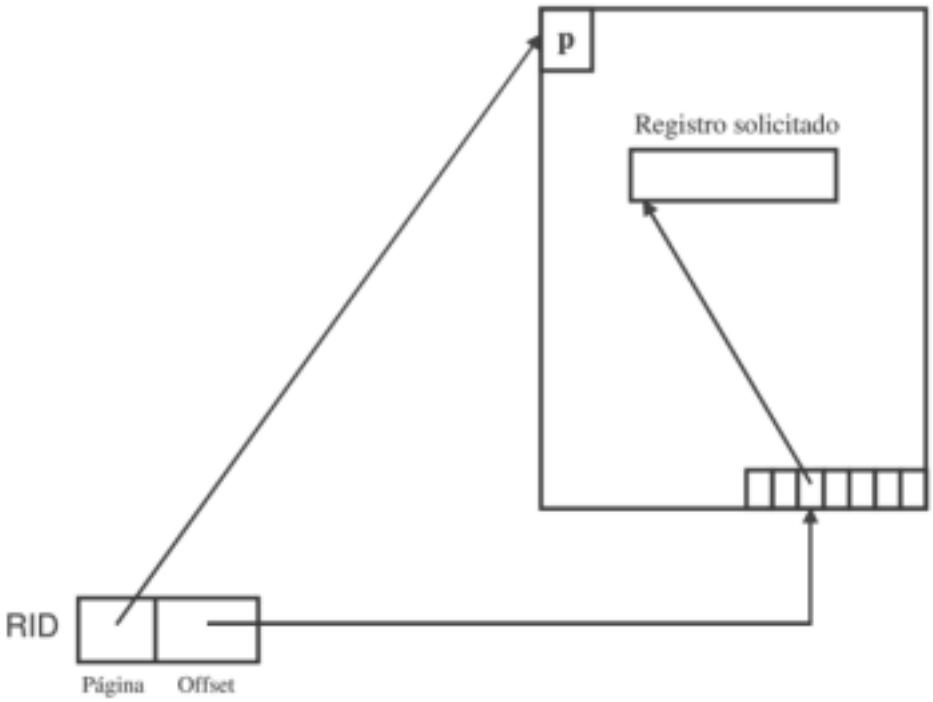


Vamos a ver cómo se transforma un registro almacenado en una representación física en el almacenamiento secundario. Si tenemos un bloque de la BD en disco, para obtenerlo se siguen una serie de pasos:

1. El SGBD solicita la página almacenada al gestor de archivos.
2. El gestor de archivos solicita los bloques de SO al gestor de disco.
3. El gestor de disco hace una operación de E/S al disco de la base de datos almacenada.
4. El disco devuelve de la base de datos almacenada los sectores solicitados al gestor de disco.
5. El gestor de disco devuelve los bloques solicitados al gestor de archivos.
6. El gestor de archivos manda la página almacenada solicitada al SGBD.



En este proceso, para que el gestor de almacenamiento pueda localizar la página de BD adecuada, se utiliza el *Record IDentifier* (RID).



Cada registro almacenado tiene:

- **Cabecera**: número y tipo de columnas que lo integran.
- **Datos**: contenido de las columnas.

Las **páginas o bloques** de la BD deben tener un tamaño múltiplo de las páginas del SO (mínima unidad de E/S), de forma que para recuperar un registro almacenado hay que determinar en qué página de la BD está y entonces recuperar los bloques de disco que la integran. Así, hay que **organizar** la estructura de almacenamiento y los métodos de acceso, de forma que se optimice la interacción con los dispositivos de almacenamiento secundario. Además, deben **minimizarse** las operaciones de E/S al almacenamiento secundario.

1.3.1 El gestor de disco del SO

Normalmente, el SGBD interactúa con la BD almacenada mediante el **gestor de disco** del SO. El gestor de disco organiza los datos en conjuntos de bloques o archivos de SO y gestiona el espacio libre en disco. Una BD puede valerse de uno o varios de estos archivos para almacenar su contenido.

Sus **funciones** fundamentales son:

- Crear un nuevo archivo de SO.
- Eliminar archivos de SO existentes.
- Añadir un bloque nuevo al conjunto de bloques c.
- Eliminar el bloque b del conjunto de bloques c.
- Devolver el bloque b del conjunto de bloques c.
- Reemplazar el bloque b dentro d el conjunto de bloques c.

1.3.2 El gestor de archivos del SGBD

El **gestor de archivos del SGBD** es un componente del SGBD que se encarga de:

- Hacer la **transformación** entre:
    - Campos, registros y archivos almacenados
    - y
    - bloques y conjuntos de bloques (que pueda entender el gestor de disco).
- **Organizar** los datos de manera que se minimice el tiempo de recuperación y las E/S a disco.

Las **funciones** básicas son:

- Crear un nuevo archivo almacenado: asociar al archivo un conjunto de páginas o bloques de la BD.
- Eliminar un archivo almacenado.
- Recuperar el registro almacenado `r` del archivo almacenado `a`. Normalmente, el SGBD proporciona el RID. Sólo hay que obtener en memoria la página que contiene el registro para extraerlo.
- Añadir un nuevo registro almacenado al archivo almacenado `a`. Hay que localizar la página de BD más apropiada de las pertenecientes al archivo almacenado. Si no se pudiera, se solicita una nueva página. Se devuelve al SGBD el RID nuevo.

- Eliminar el registro `r` del archivo almacenado `a`. Hay que recuperar la página de BD que contiene dicho registro y marcar el espacio ocupado por el registro en dicha página como disponible.
- Actualizar el registro r en el archivo almacenado `a`. Recupera la página de la BD que contiene el registro que se desea actualizar. Trata de sustituir la información. Si no puede, se intenta ubicar en otra página.

1.4 Representación de la BD en el nivel interno

La BD se representa de formas diferentes en los distintos niveles de arquitectura del SGBD. Esta representación en el nivel interno no tiene por qué coincidir con la representación en el nivel conceptual, y un conjunto de registros del mismo tipo no tiene por qué ser un fichero.

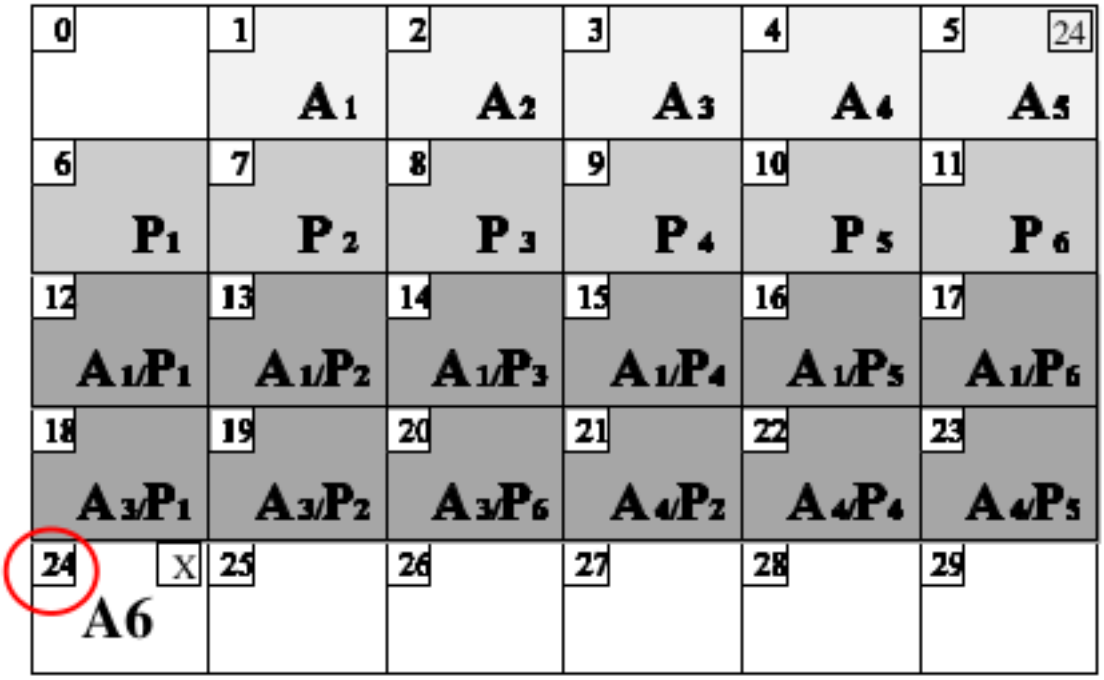
El nivel interno debe traducir las estructuras del nivel conceptual a otras estructuras intermedias cercanas al almacenamiento real de los datos (nivel físico).

Si la BD en el nivel interno tiene al conjunto de páginas en las que se ubican los registros, tenemos:

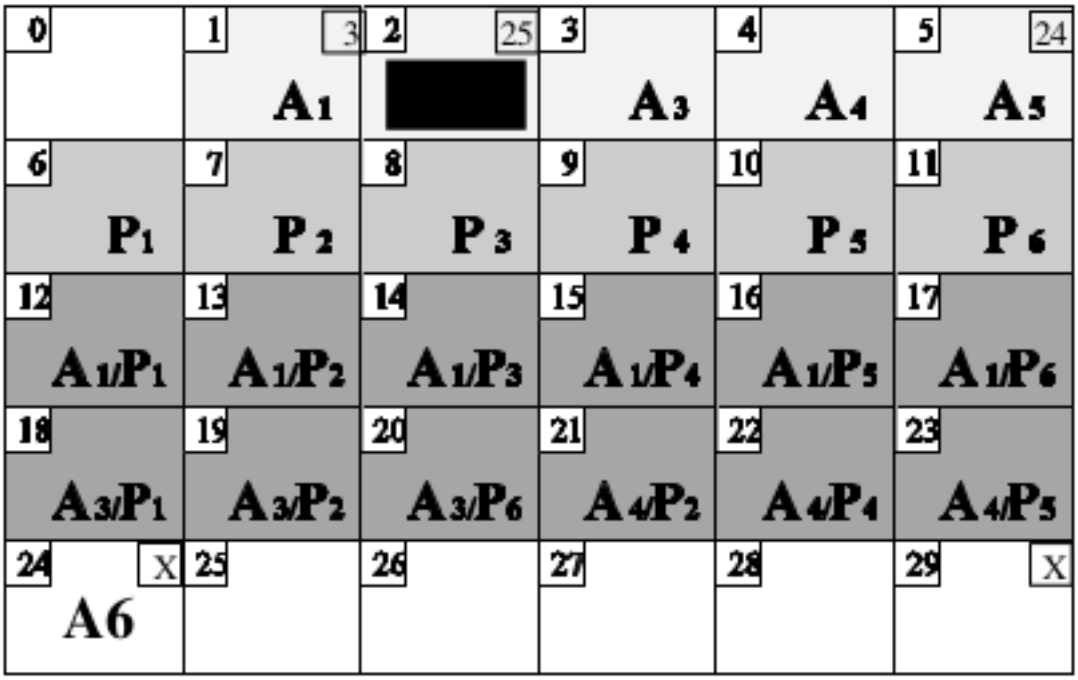
- **Agrupamiento Intra-Archivo**: se ubican en la misma página registros del mismo tipo. Es el más frecuente.
- **Agrupamiento Inter-Archivo**: se ubican en la misma página registros de distinto tipo. Debe existir relación entre ellos (por ejemplo, entidades fuerte-débil).

Ejemplo.

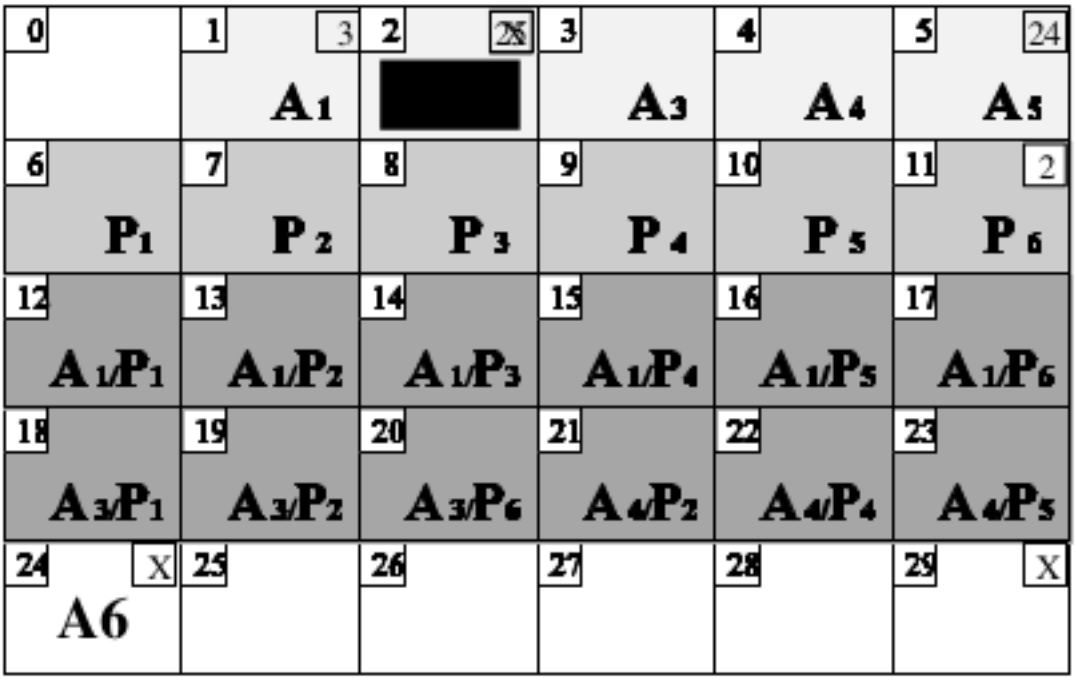
- Se inserta una nueva asignatura con código A6.
    - Se localiza la primera página libre (la 24).
    - Se inserta el registro correspondiente.
    - Se añade esta página al conjunto de páginas de asignaturas.



- Se borra la asignatura con código A2.
    - La página que contiene a esta asignatura pasa al conjunto de páginas libres.
    - Se reorganiza la lista correspondiente a Asignaturas.

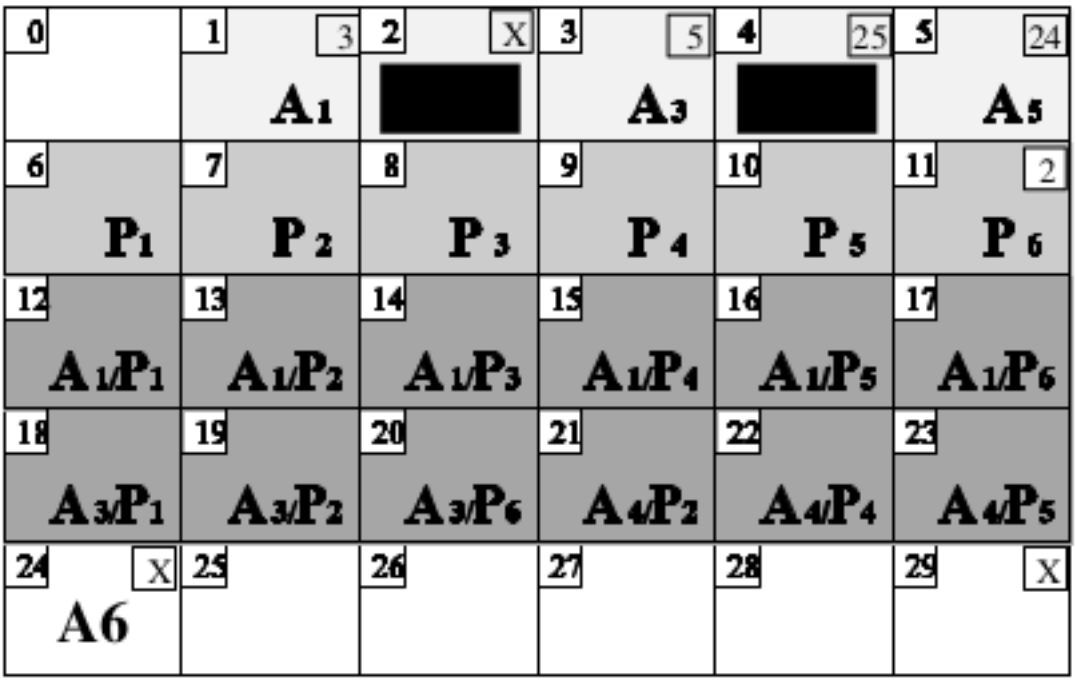


- Se introduce un nuevo profesor con código P7.
    - Se ubica en la primera página libre disponible (la segunda).
    - Se ajustan las cadenas de punteros.

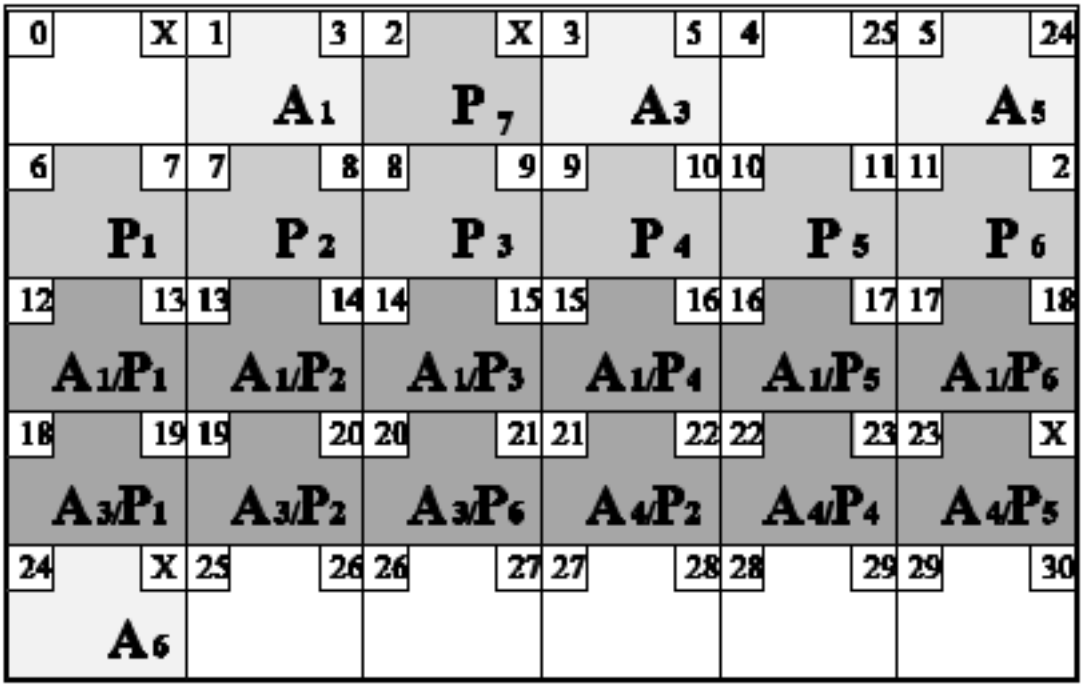


- Se borra A4.

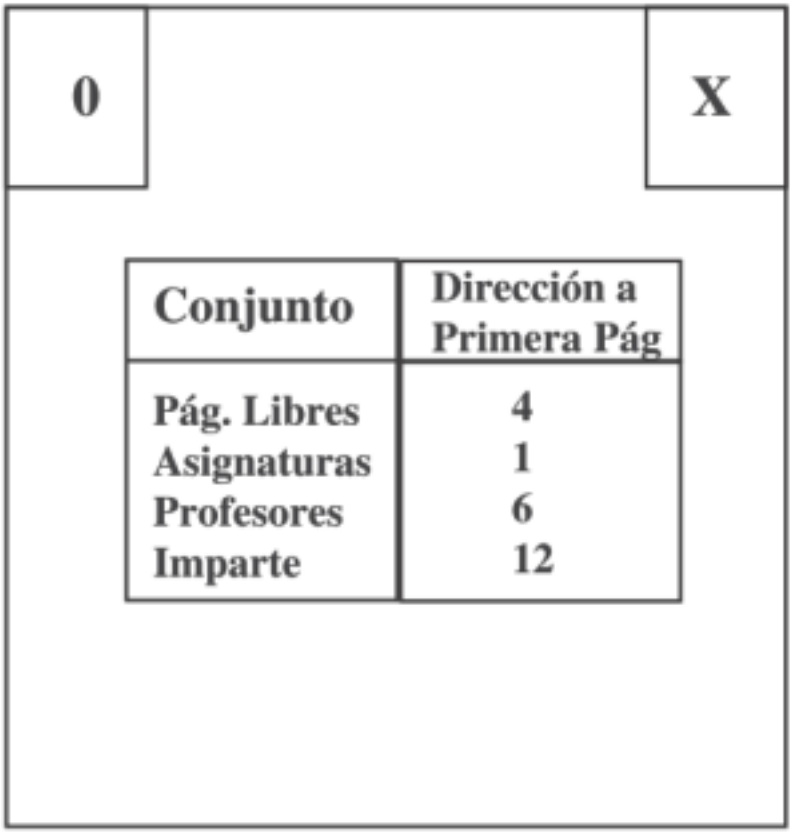
  - Su página pasa al conjunto de páginas libres.
  - Se reorganiza la cadena de punteros de las Asignaturas.



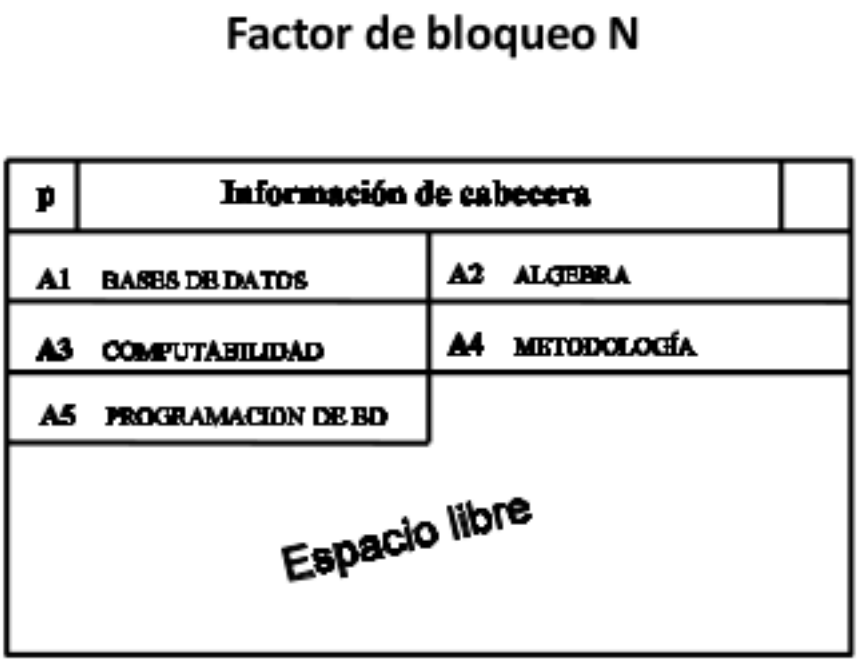
Punteros para el recorrido secuencial lógico:



Índice en la página 0



Conjunto	Dirección a Primera Pág
Pág. Libres	4
Asignaturas	1
Profesores	6
Imparte	12



En realidad, las páginas contienen más de un registro

La organización descrita es un ejemplo general. Cada SGBD comercial utiliza su variante concreta, aunque la idea subyacente es la misma.

No existe una relación directa fichero-almacenado/fichero-físico, ya que todos los conjuntos de páginas irán almacenados, con toda probabilidad, en uno o varios ficheros físicos.

1.5 Organización y métodos de acceso

> ¿Son los índices mejores que los hash? Ninguna de las organizaciones presentadas es mejor en términos absolutos.

Cuando queremos **acceder** a los datos de una BD, nuestro objetivo es minimizar el número de acceso a disco. Por ello, necesitamos minimizar la cantidad da páginas de una BD involucradas en una operación de BD.

Ninguna de las organizaciones presentadas es mejor en términos absolutos. Existen varias organizaciones usadas para las BD.

Los criterios básicos para medir la **calidad** de una organización son:

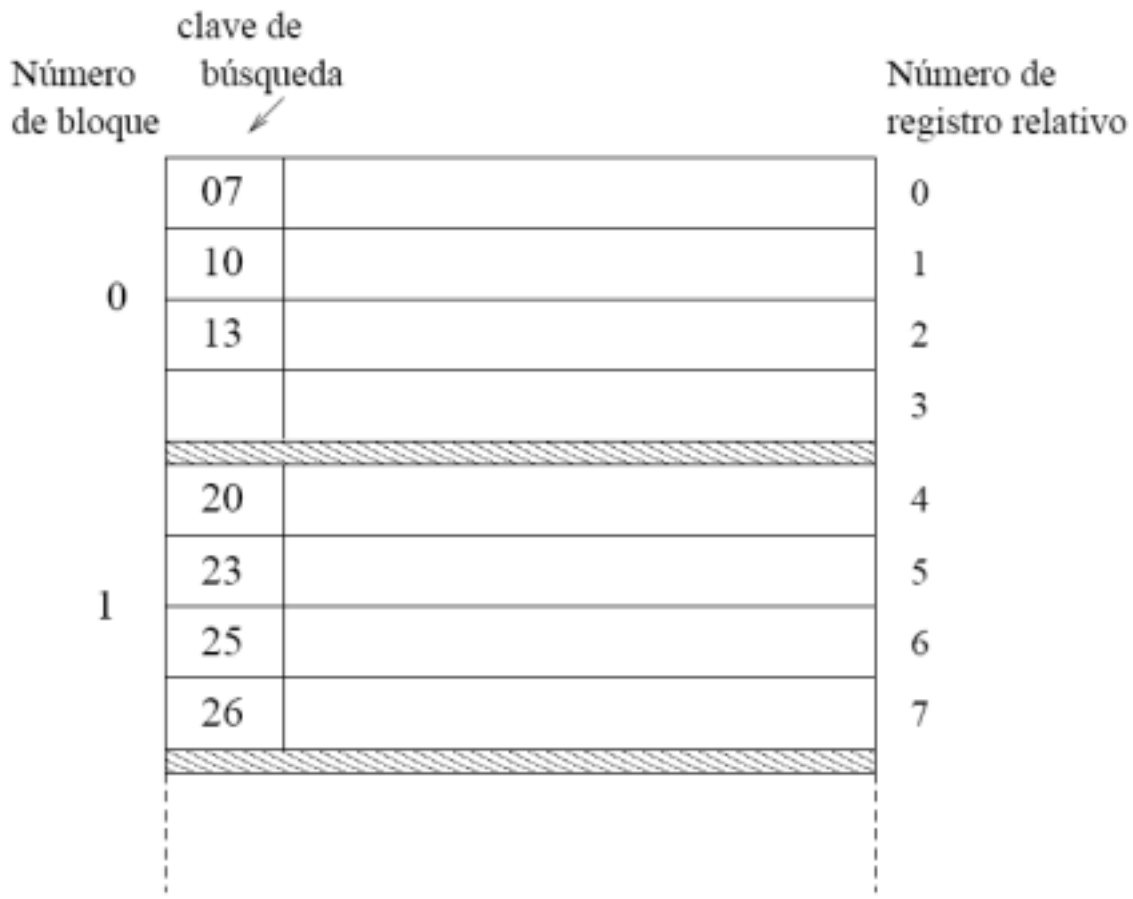
- **Tiempo de acceso** a los datos requeridos.
- **Porcentaje de memoria ocupada** por los datos requeridos con respecto a las páginas de BD que los contienen.

Trabajaremos a **dos niveles**:

- **Organización** de registros de datos a nivel de almacenamiento
- **Adición** de estructuras complementarias para acelerar el acceso a dichos registros.

1.6 Organización secuencial

Existe un **fichero de acceso secuencial** donde los registros están almacenados consecutivamente y ordenados por una clave (**clave física**). Para acceder a un registro debemos pasar por los registros que le preceden.



Ejemplo. Mostrar la relación completa de departamentos. La consulta se resolvería rápidamente si los departamentos están almacenados conjuntamente en bloques contiguos de un fichero. Sin embargo, ¿qué pasa si queremos plantear consultas por valor de clave o por rango de valores?

El primer caso implica:

- Recorrer uno tras otro cada uno de los registros.
- En el caso peor (no encontrarse dicho departamento o ser el último de la lista) supone recorrer de forma completa el fichero.
- Búsqueda es $O(N)$.

El segundo caso tiene un tratamiento muy parecido:

- Se realiza la búsqueda por valor de clave de la cota inferior del intervalo.
- Se continúa hasta alcanzar la cota superior. Si están ordenados por el valor de la clave.

Si queremos:

- **Insertar un registro**: implica buscar el bloque que le corresponde, insertarlo si hay sitio y si no o se crea un nuevo bloque o se crea uno se crea un bloque de desbordamiento. Es recomendable dejar espacio vacío entre bloques para evitar problemas de reorganización.
- **Borrar un registro**: implica buscar el registro y borrarlo. Además, puede implicar una reorganización local de los registros de un bloque.

En resumen, las dos operaciones suponen:

- Escritura del bloque del registro que se inserta o borra.
- Creación o liberación de bloques de datos en el fichero secuencial.
- Creación o liberación de bloques de desbordamiento.
- Reorganización de registros entre bloques contiguos, lo que implica la escritura de los bloques implicados en el desplazamiento.

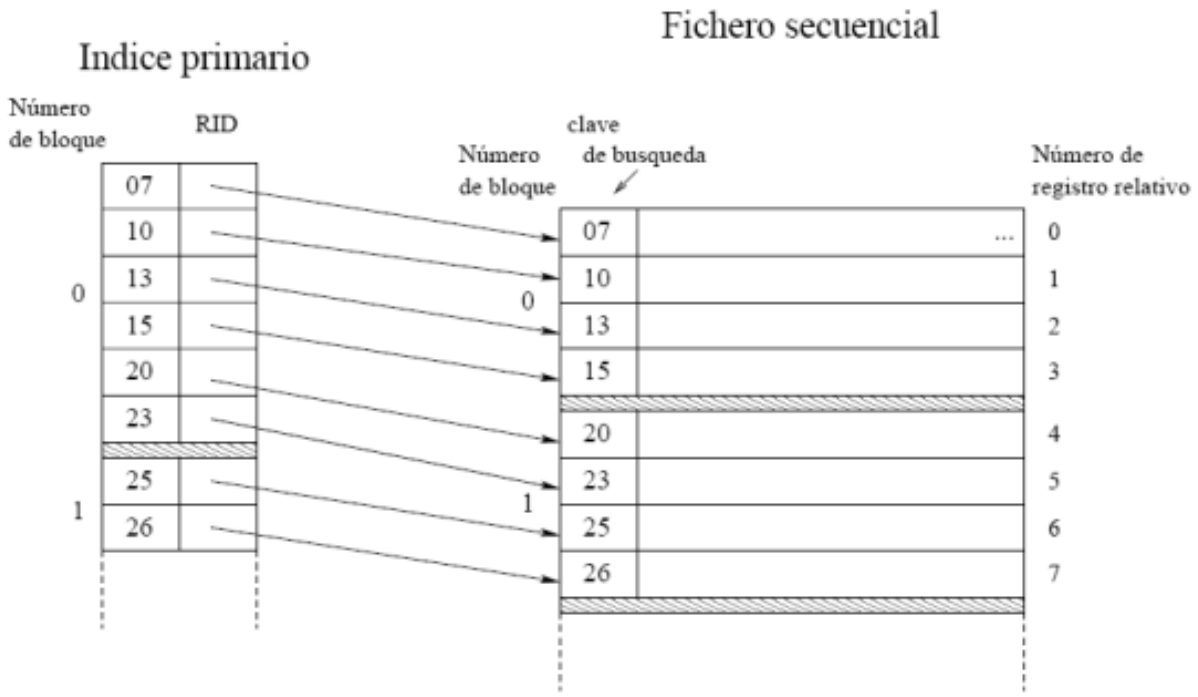
Es por ello que esta forma tiene grandes inconvenientes que son subsanables, al menos en parte, mediante el uso de estructuras adicionales que nos permitan acelerar la localización de los datos y disminuir el número de bloques de disco transferidos. Las técnicas más populares son la **indexación** (índices) y el **acceso directo**.

1.7 Indexación

Pretende disminuir el tiempo de acceso a los datos utilizando una **clave de búsqueda**. Es similar a la idea de un índice en un libro. Existen muchas formas de llevar a cabo esta idea.

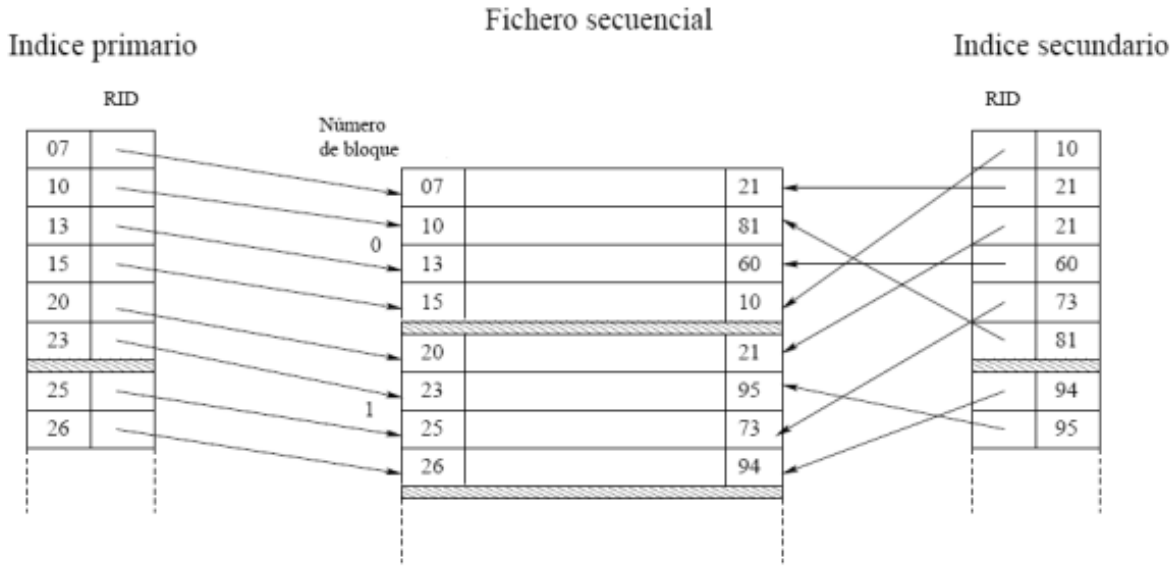
1.8 Fichero indexados

Partiendo de un fichero secuencial sobre el que disponemos una estructura adicional (fichero índice), cuyos resgistros tienen una clave de búsqueda (campo clave) y un campo de referencia con los RIDs de los registros. Son registros más pequeños que los del fichero de datos, aunque hay el mismo número de registros en ambos.



- **Índice primario**: la clave de búsqueda es el mismo campo clave (clave física) por el que se ordena el fichero secuencial de datos.

- **Índices secundarios**: se construyen sobre otros campos que no sean la clave física del fichero de datos.



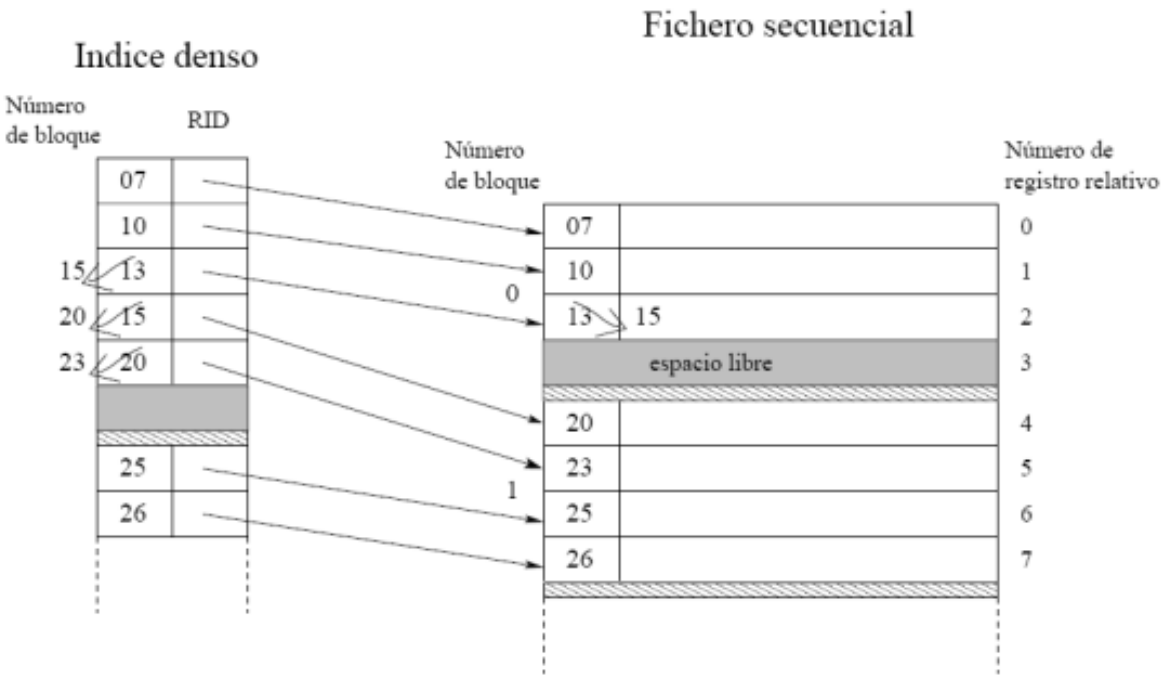
Para **consultar** un valor, podemos consultar:

- **Por valor de la clave**: sobre el índice localizamos la clave (recorrido secuencial), obtenemos el RID del registro y vamos a disco para recuperar el bloque donde está el registro señalado por el RID. La búsqueda en el índice es más rápida.
- **Por rango de valores**: se busca en el índice por valor de la clave de la cota inferior y se recorren las entradas del índice que están en el intervalo, recuperando los registros correspondientes gracias a su RID.

Para **insertar un nuevo registro** se hace lo mismo que en el fichero secuencial, y hay que actualizar el índice (inserción en un fichero secuencial).

Para **borrar un registro** se borra un registro en el fichero de datos y se borra una entrada en el índice.

Ejemplo. Borrado del registro de clave 13.



Se puede montar un índice sobre más de un campo de un registro. La clave es la concatenación de los campos indicados.

Hay que tener cuidado: si hay un índice sobre nombre-alumno y DNI. Es útil para consultas que involucran: Nombre; Nombre y DNI. No es útil para consultas sobre el DNI.

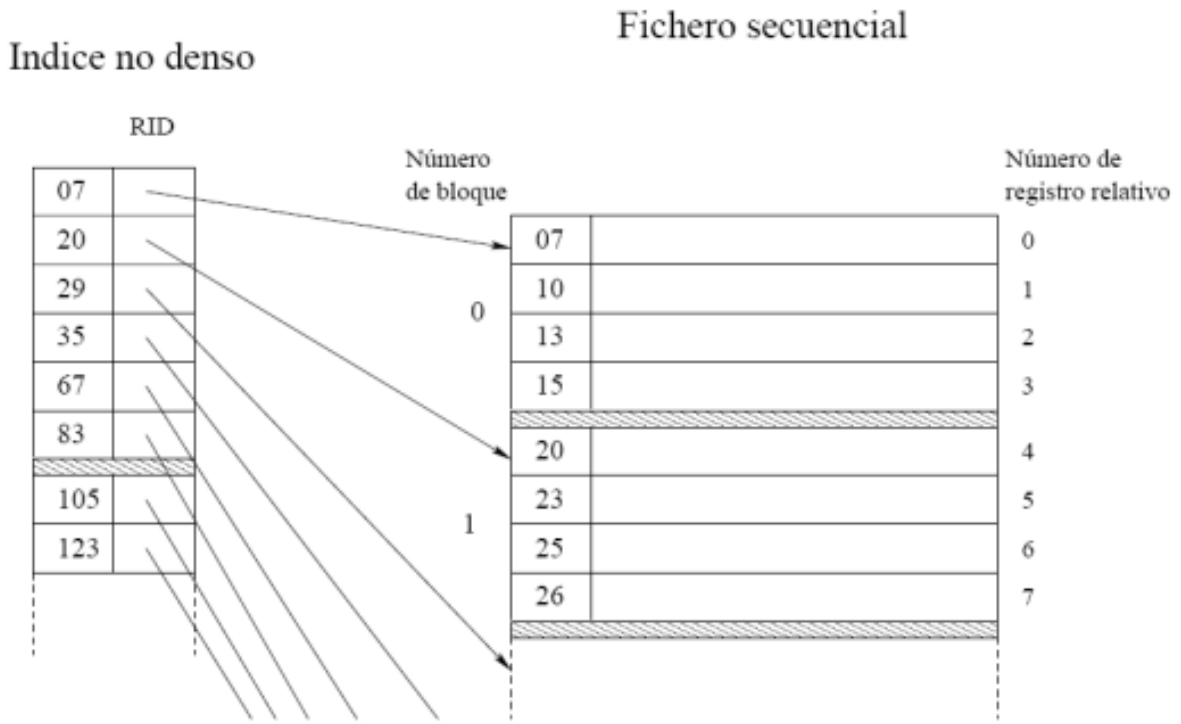
Como conclusión, estos índices aceleran el acceso a los datos pero ralentizan las otras operaciones, por lo que hay que mantener estos índices.

Por tanto, hay que considerar la conveniencia de crear cada índice: frecuencia de las consultas y frecuencia de las operaciones de mantenimiento de los datos.

1.9 Índices no densos

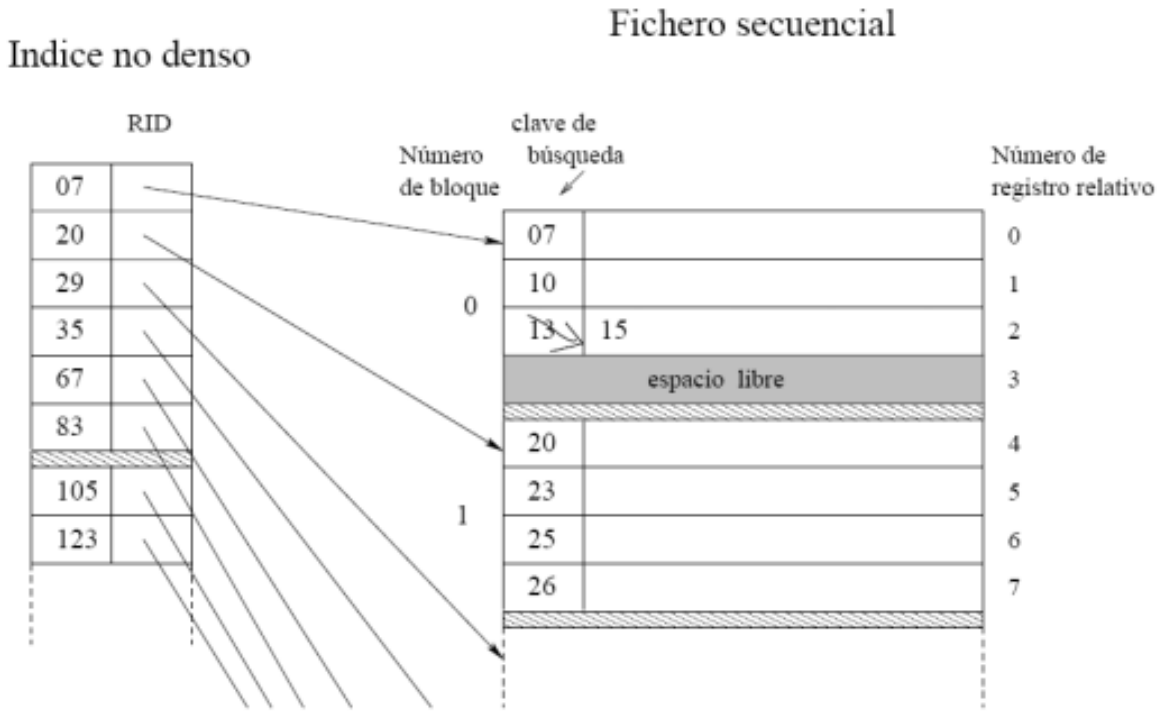
Lo ideal es mantener el índice en memoria principal. Pero en realidad, los índices suelen ser muy grandes porque contienen todos los registros del fichero que indexan. Son densos.

Por tanto, para reducir el tamaño aparecen los **índices no densos**, que son registros que se componen por la clave de búsqueda y la dirección de comienzo del bloque donde puede estar el registro deseado. El número de registros se reduce al número de bloques del fichero de datos. El acceso secuencial al índice no denso se acelera.



La búsqueda ahora es diferente, pues una vez se encuentra el bloque donde podría estar el registro hay que cargarlo en memoria y hacer búsqueda secuencial en ese bloque. No tiene costes en términos de acceso a disco. Además, no se garantiza encontrar el registro hasta consultar el bloque de datos leído.

Los índices no densos sólo se pueden definir sobre la clave física. El mantenimiento de un índice no denso es menos costoso: la inserción y borrado son menos frecuentes pues solo ocurren cuando la operación afecta al valor que representa al bloque completo.

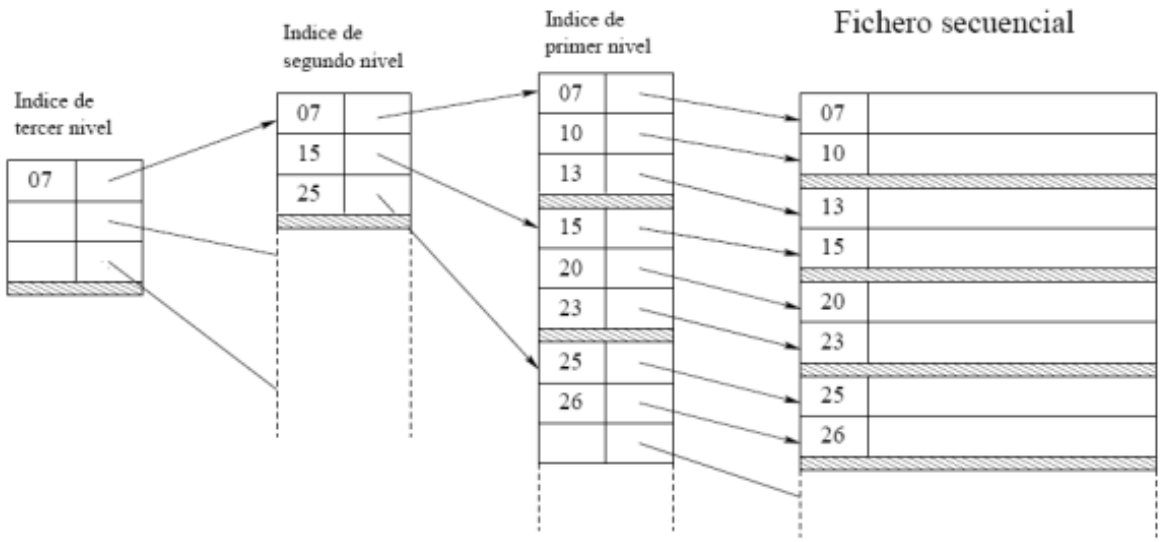


1.10 Índices jerárquicos

Estos índices quieren disminuir el tiempo necesario para recorrer el índice en busca de un registro. Para ello, crearán **índices multinivel**, formados por índices construidos sobre índices, luego hay varios niveles en el acceso a los datos. Un índice multinivel está formado por:

- Un **índice de primer nivel** sobre el fichero de datos, puede ser denso o no dependiendo de la clave.
- **Otros índices**, no densos, construidos sucesivamente unos sobre otros.

El **tamaño de bloque** se establece para optimizar las operaciones de acceso al disco físico. Así se reduce el número de accesos a disco para encontrar un registro pero es más difícil mantener los índices. En el peor caso, el número de accesos pueden ser tantos como niveles.

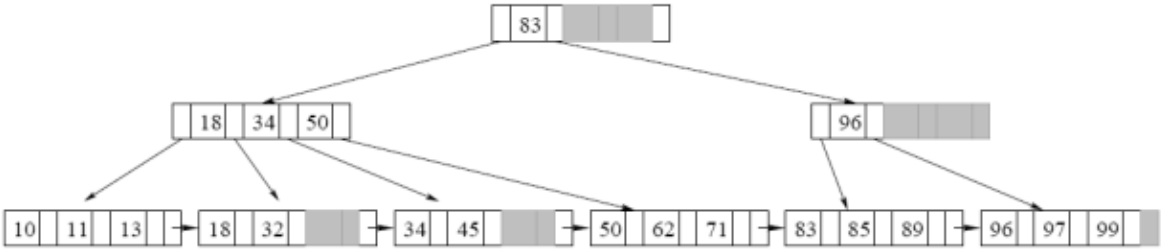


1.11 Árboles B+

Fueron propuestos en 1972 por Bayer y McCreight. Los **Árboles B+** son una generalización de los árboles binarios balanceados en la que los nodos pueden tener más de 2 hijos. Los valores de la clave se encuentran almacenados en los nodos hoja.

Un árbol B+ de orden M (máximo número de hijos que puede tener cada nodo) es un árbol con la siguiente estructura:

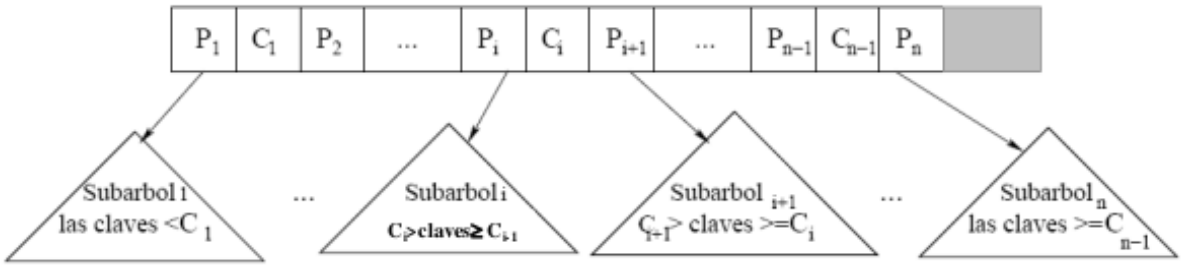
- Nodo de nivel superior: raíz del árbol
- Nodos del nivel inferior: hojas.
- Cada nodo distinto de las hojas tiene como máximo M hijos y como mínimo $(M+1)/2$ hijos.
- Todos los nodos hoja aparecen al mismo nivel.
- Las claves contenidas en cada nodo nos guiarán hasta el siguiente nodo del nivel inmediatamente inferior.
- Un nodo no hoja con n hijos contiene:
    - $n-1$ valores de clave almacenados.
    - n punteros P_i que apuntan a un nodo hijo.



Se pone como restricción que los valores de clave C_i están ordenados dentro del nodo y que los valores x del subárbol apuntado por P_i cumplen:

$$C_{i-1} \leq x < C_i$$

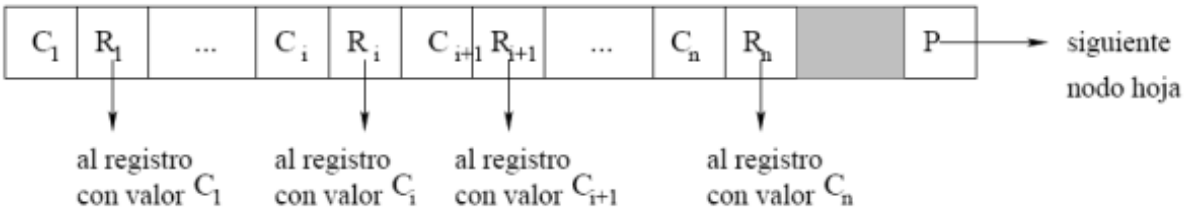
Excepto para $i = 1$ (donde $x < C_1$) e $i = n$ (donde $x \geq C_{n-1}$).



Los **nodos hoja** tienen una estructura diferente:

- Parejas clave - RID.
- Punteros al siguiente nodo hoja.
- Algunas variantes también tienen punteros al nodo hoja anterior.

La **lista concatenada de nodos hoja** (conjunto secuencia) es útil para hacer consultas por intervalos.



Las **claves** aparecerán ordenadas en cada nodo y todas deben ser menores que las del siguiente nodo en el conjunto secuencia.

Los **nodos hoja** se encuentran en el mismo nivel: árbol equilibrado y todos los caminos desde la raíz a un ondo hoja tienen la misma longitud.

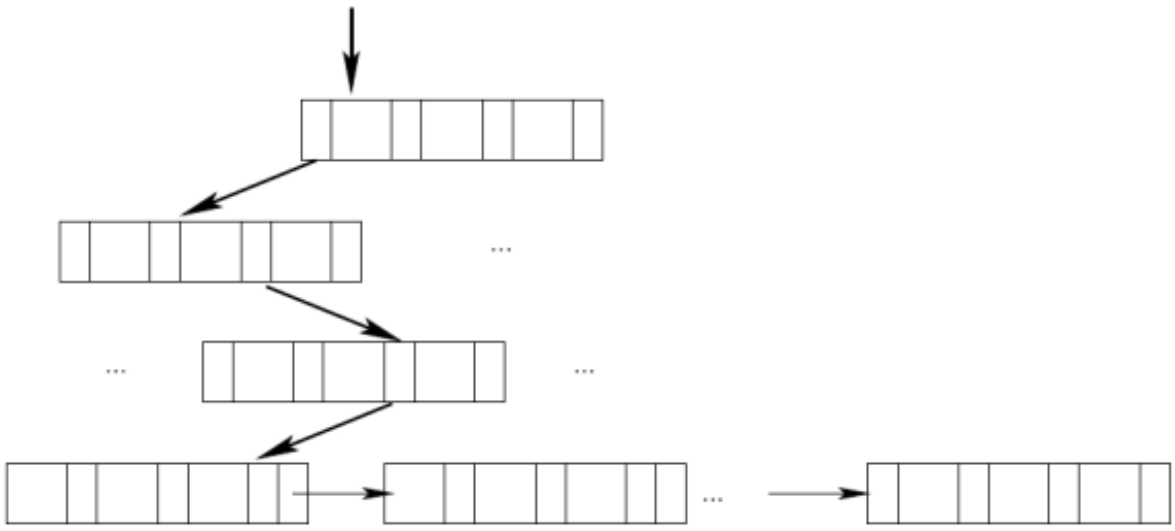
Los nodos deberán estar rellenos como mínimo hasta la mitad. ???

Nodo	**Restricción**	**Min**	**Max**	**Ejemplo** $M = 5$
Raíz (cuando no es nodo único)	Número de hijos	2	M	2-5
Interno	Número de hijos	$\lceil M/2 \rceil$	M	3-5
Hoja	Número de claves	$\lfloor M/2 \rfloor$	M-1	2-4

- **Consulta**: para **localizar** un registro, navegamos desde la raíz, bajando niveles y buscando el registro en el nodo hoja y recuperando el registro del fichero de datos gracias al RID.

  Para **consultar** en un rango se localiza el nodo hoja con el valor inferior y se recorren los demás nodos hoja hasta encontrar el superior, recuperando los registros pertinentes del fichero de datos.

- **Inserción y borrado**: los algoritmos utilizados deben garantizar que el árbol siga siendo equilibrado.



1.12 Árboles B

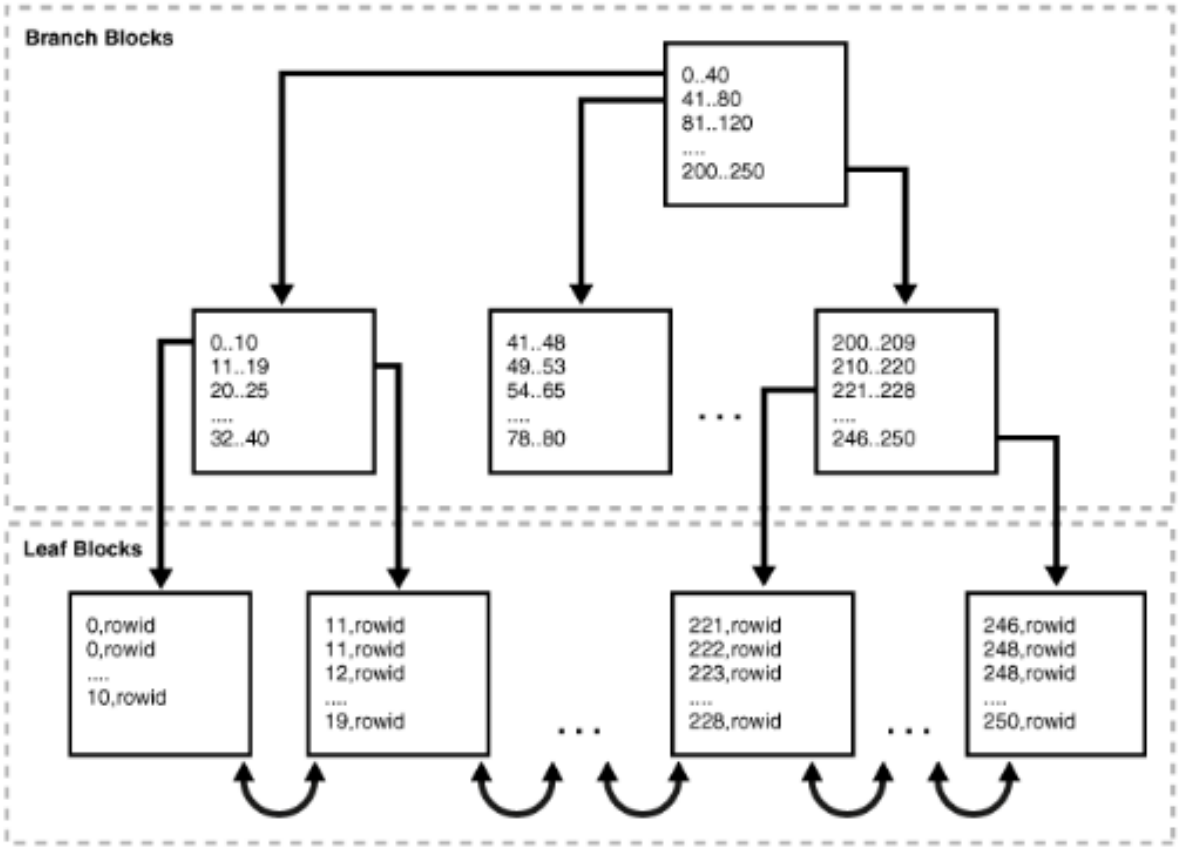
Los **árboles-B** (B-Tree) son una variante de B+Tree en la que no se almacenan todos los valores de la clave en los nodos hoja, sino que algunos valores se van almacenando en los nodos intermedios conforme se crea el árbol.

1.13 Árboles B+ en BD

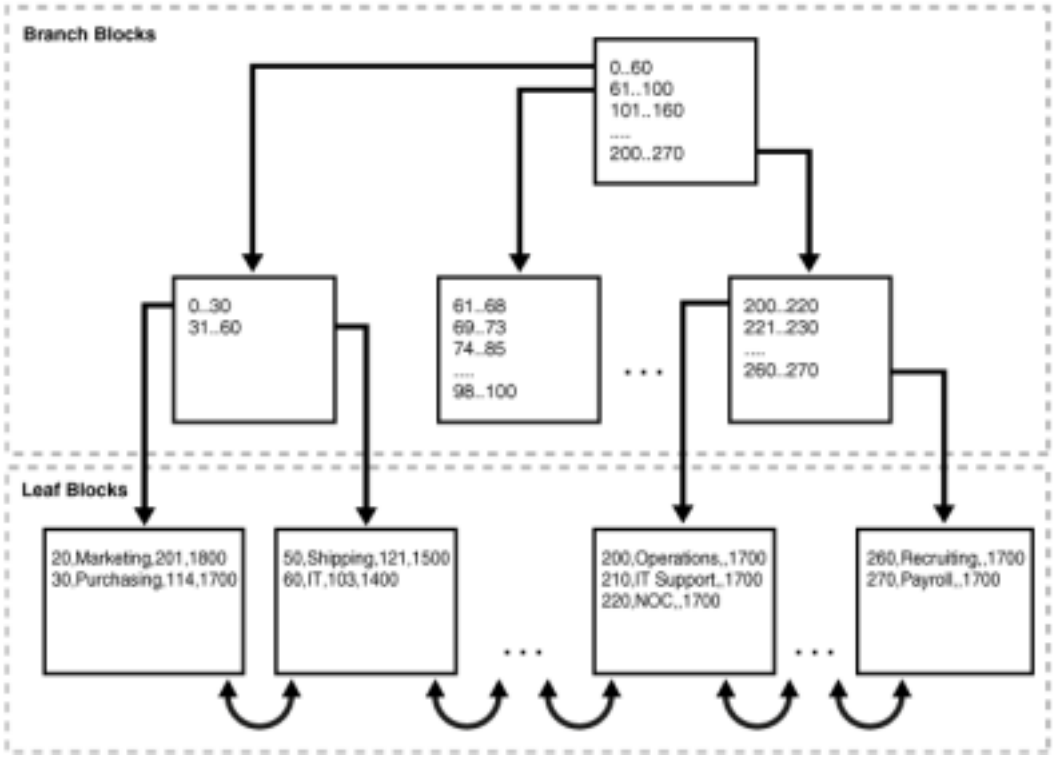
Los **árboles B+ en BD** son variaciones del árbol B+ de orden elevado en la que se procura que cada nodo tenga casi el mismo almacenamiento que un bloque de datos. Esto reduce los acceso a disco que suelen ser los que determinan el rendimiento de las búsquedas en BD.

En los nodos **intermedios** solo están los rangos de valores de la clave y los punteros a los nodos hijo correspondientes.

En los nodos **hoja**, que están enlazados, están los valores de clave ordenados junto con los RIDs (*rowid*) que apuntan a las tuplas que contienen ese valor de clave. Los nodos hoja, que forman el conjunto secuencia, se encuentran enlazados para poder recuperar por búsquedas secuenciales, a veces se encuentran doblemente enlazados, para facilitar búsquedas ascendentes y descendentes por el valor de la clave.



También existen las **Tablas Organizadas por Índice** (*IOT*). En este caso, las hojas contienen las tuplas en lugar del RID. Así, una IOT solo puede estar organizada de esta forma mediante una clave, aunque se pueden definir índices adicionales basados en otras claves.



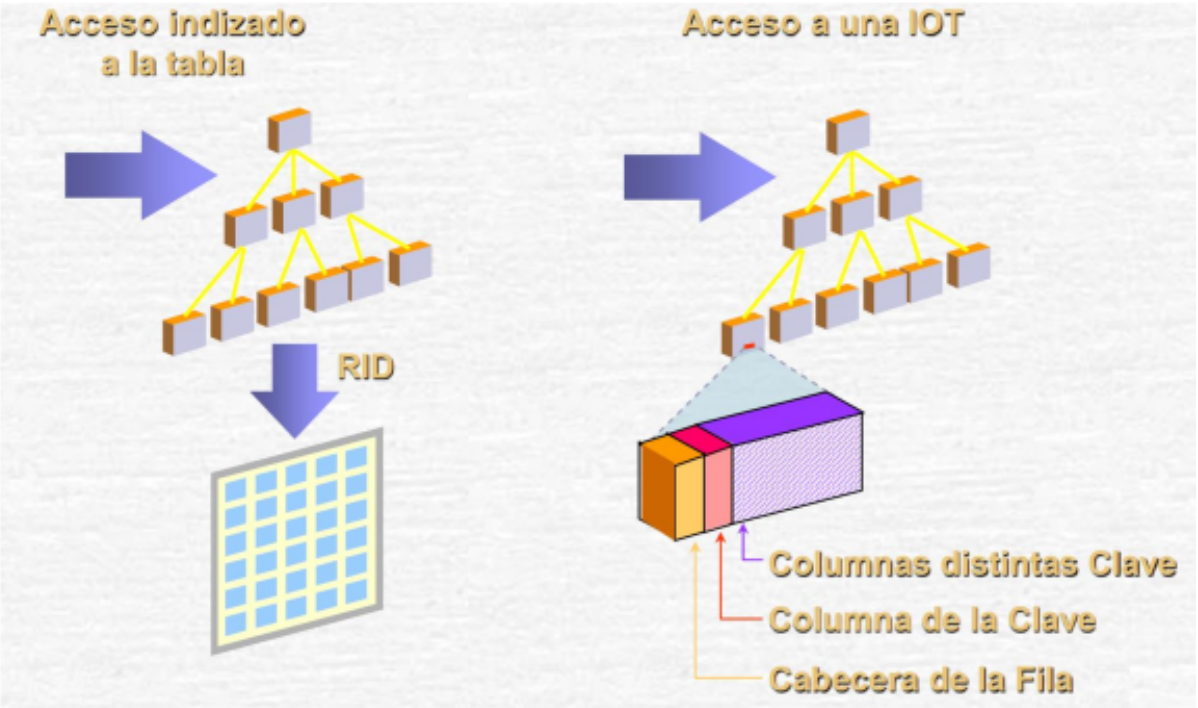


Tabla normal	IOT
Identificador único ROWID (RID) ROWID implícito Soporta varios índices	Identificado por clave primaria No ROWID Sólo un índice IOT, varios B-tree

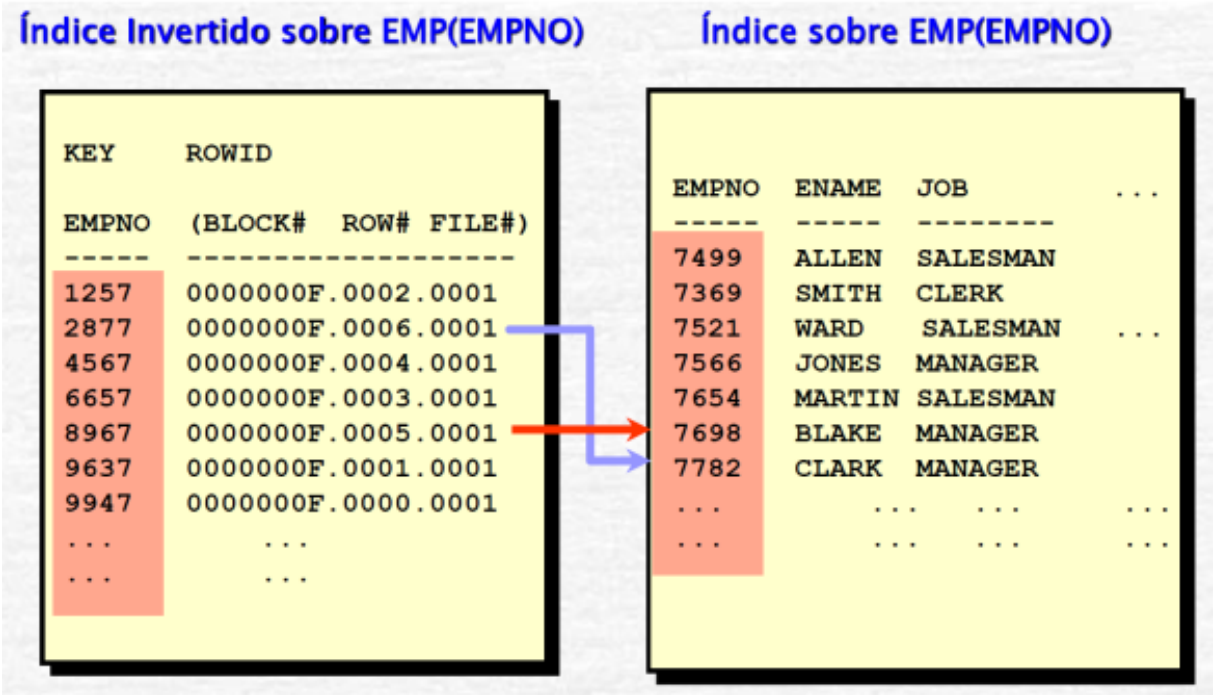
Tabla normal	IOT
Una recuperación completa devuelve las tuplas desordenadas	Una recuperación completa devuelve las tuplas ordenadas según CP

1.14 Índices por clave invertida

Los **índices por clave invertida** (*reverse index*) invierten los datos del valor de la clave. Es adecuado para búsquedas basadas en predicados = y mejora el rendimiento de la inserción de tuplas si se insertan de forma ascendente para valores de la clave. También mejora accesos concurrentes.

Para el empleado 7698 almacena 8967.

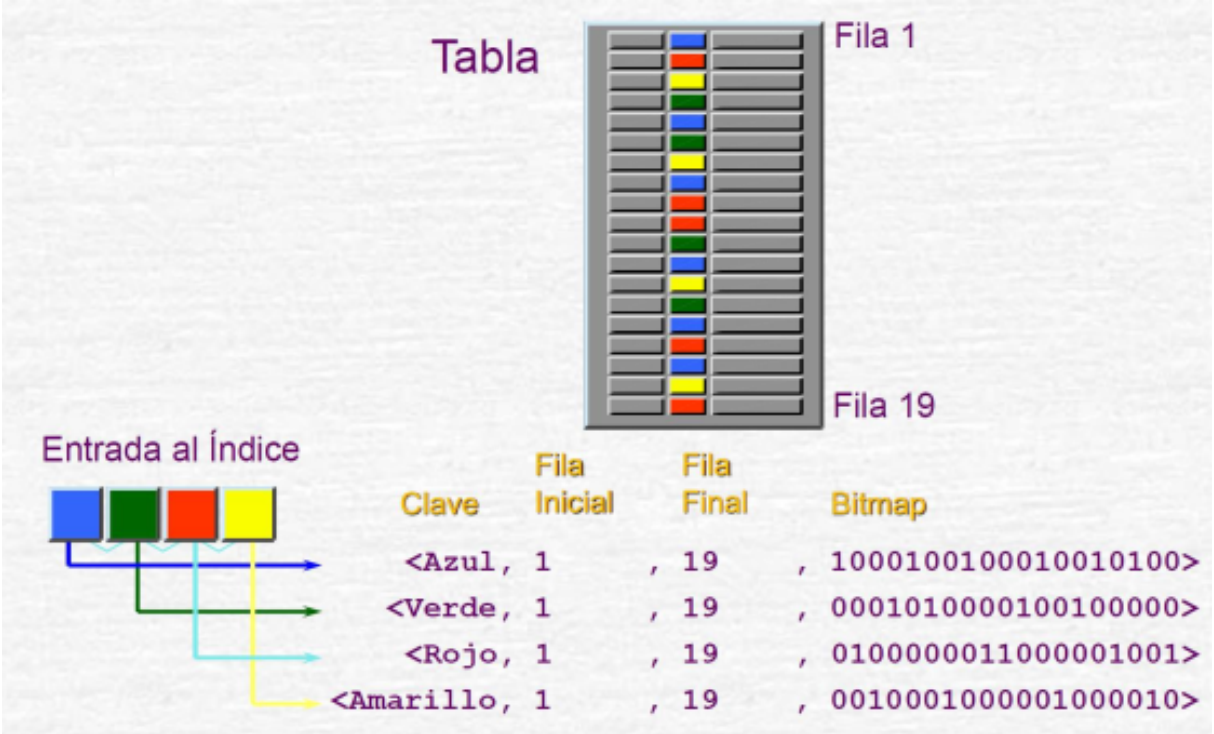
Con este índice se reducen los embotellamientos (retención de bloques de BD) en el índice cuando se están introduciendo los datos de forma ascendente para los valores de la clave, puesto que todos irían a la misma entrada de índice.



1.15 Índices BITMAP

En los **índices BITMAP** (*índices de mapa de bit*), para cada valor que toma la clave almacena una secuencia de bits (tantos como tuplas contenga la tabla), en los que hay un 1 si el valor está presente en la tupla y un 0 si no lo está.

B-tree	BITMAP
Adecuado para columnas que tomen muchos valores	Adecuado para columnas que tomen pocos valores
Actualización de claves poco costosa	Actualización de claves muy costosa
Ineficiente para consultas usando predicados OR	Eficiente para consultas usando predicados OR



En la anterior imagen, el índice tiene 4 entradas, una por cada color. Nos dice qué tuplas tiene cada color. Si queremos ver las tuplas del color verde o azul, haremos el OR entre los mapa de bits correspondientes.

Una **ventaja** es acelerar consultas tipo OR, las del tipo AND no porque no hay.

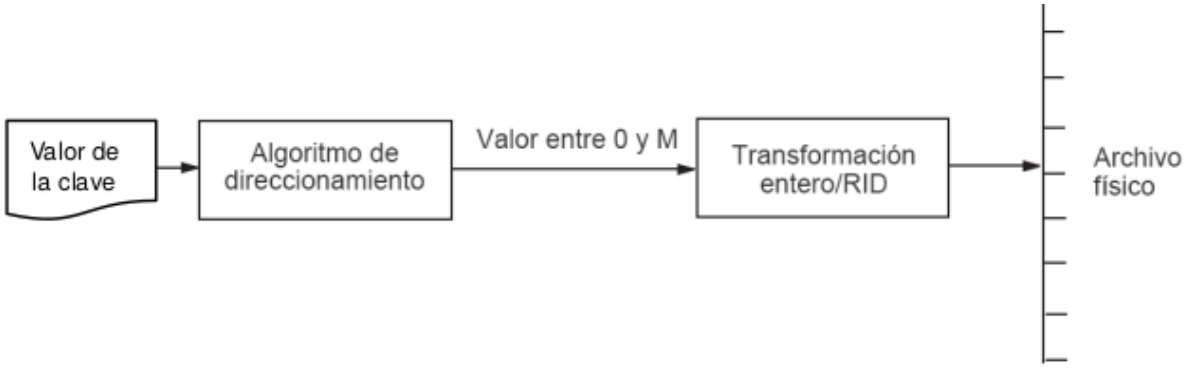
Un **inconveniente** es cuando hay muchos valores porque hay que montar un mapa de bits.

2 Tema 5. El nivel interno. Métodos de organización y acceso a los datos

2.1 Acceso directo

El **acceso directo** es una forma de acceder a un registro almacenado. En este caso, no tenemos estructura adicional sino que usamos un algoritmo o función sobre un campo determinado del mismo para identificar la posición del registro deseado. Para ello, debemos tener un campo que identifique unívocamente al registro.

Con respecto al **funcionamiento**, lo usual es que no se pueda establecer una **clave física** totalmente correlativa y única para cada registro, por lo que nuestro algoritmo tiene que transformar los valores de un cierto campo en una dirección. Además, deberá tener una entrada de un **campo clave** y proporcionará una salida que será un valor entero positivo transformable en **RID**.



Estos algoritmos de direccionamiento no suelen mantener el orden de la clave. Los registros no están almacenados según el orden de su clave física. Es por ello por lo que tendremos problemas a la hora de recuperar intervalos de datos.

Los **algoritmos** usados son variados:

- **Dependen del tipo de clave**: si la clave es alfanumérica, se transforma a un valor numérico.

- Suelen estar basados en un mecanismo de **generación de números pseudoaleatorios**:

    - **Cuadrados centrales**: se eleva la clave al cuadrado y se eligen tantos digitos centrales como sea enecesario.
    - **Congruencias**: se divide la clave por M y se toma el resto (M suele ser primo).
    - **Desplazamiento**: se superponen adecuadamente los digitos binarios de la clave y luego se suman
    - **Conversión de base**: se cambia la base de numeración y se suprimen algunos dígitos resultantes.

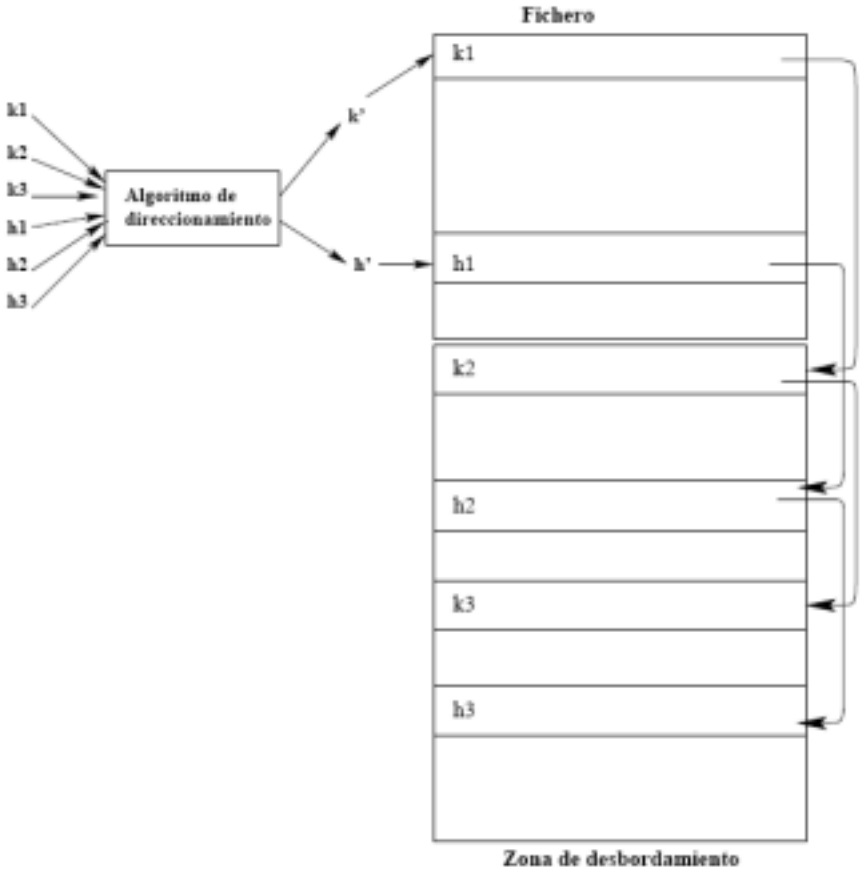
Estos algoritmos tienen varios **problemas**:

- Salvo que el campo clave se diseñe para ello, es prácticamente imposible encontrar una transformación que dé un valor entero positivo en un rango de valores limitado tal que dos claves diferentes den siempre valores distintos. Se producen **colisiones**.

- Producen **huecos**, zonas vacías del rango de salida, no asignadas por el algoritmo, que generan huecos en el fichero de datos.

Para **gestionar colisiones y huecos** tenemos que combinar acceso directo con una gestión mediante **listas de colisión**, que mantienen los registros con claves que producen colisión en dichas lista. Si estas listas crecen el acceso directo no resulta adecuado pues hay que mantener las listas y la zona de desbordamiento es casi como el fichero original. Puede haber colisión:

- El **registro problemático** se almacena en la zona de desbordamiento.
- Los **sinónimos** (registros con claves que producen colisión) se conectan mediante una lista.



Si crecen las listas de sinónimos, el acceso directo puro no resulta adecuado. Hay que mantener listas y la zona de desbordamiento es casi como el fichero original.

Han aparecido técnicas más sofisticadas: **hashing**.

2.2 Hashing básico

Aparece para solucionar el problema del acceso directo. Ya que los valores de las claves no estaban uniformemente distribuidos en el intervalo $[0, M]$, sino que se acumulan en una parte de él. La solución es asignar más espacio a esa parte del intervalo.

La técnica consiste en dividir el fichero en **buckets** (cubos) y el algoritmo de direccionamiento asignará cubos, no direcciones concretas. En cada *bucket* habrá más de un registro y ciertos rangos de valores tendrán asignados más buckets que otros. Complementamos esto con el uso de **cubos de desbordamiento**.

Tendremos como **parámetros**:

- **Número** de cubos.
- **Tamaño** de los cubos (relación con bloques físicos): *slots*.
- La **transformada** clave/dirección, que tiene en cuenta la distribución de la clave según rangos para que los cubos queden equitativamente rellenos.

Para **insertar** un registro, transformamos la clave, localizamos su cubo, se inserta si hay sitio y si no se sitúa en el cubo de desbordamiento conectándolo con el cubo a donde realmente le corresponde el registro mediante punteros.

Para **buscar** un registro, transformamos la clave, localizamos su cubo y dentro del cubo buscamos secuencialmente: si hemos encontrado el registro, el proceso termina; en caso contrario, se impone un barrido por punteros a través de los cubos de desbordamiento.

2.3 Hashing dinámico

El hashing básico tiene el problema de que hay que conocer la distribución previa de las claves para asignar adecuadamente los buckets, de lo contrario sigue habiendo huecos y colisiones. Además, al aumentar el número de registros, aumentan los registros en las páginas de desbordamiento y a veces hay que reorganizar los datos.

La solución es trabajar de forma dinámica. El **Hashing dinámico** trabaja partiendo de una configuración uniforme y de pocos cubos y los restantes, se van generando cuando los necesite, asignando a los rangos conforme la afluencia de registros lo pide. Es un hashing dinámico o extensible.

La **técnica** es la siguiente:

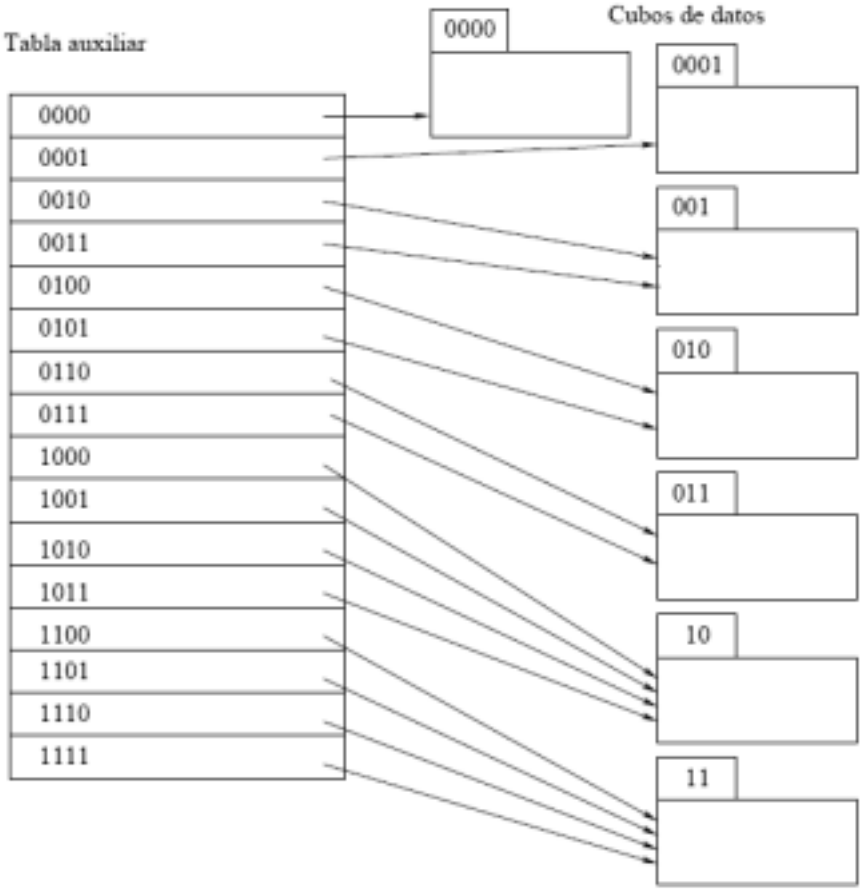
El valor transformado del campo clave da una entrada de una tabla índice que está en memoria. Allí está la dirección del cubo donde están los registros que tienen asociado este valor transformado (puede que varias entradas de la tabla lleven al mismo cubo). Inicialmente, todas las entradas apuntan al mismo cubo. Cuando se insertan más registros, se generan nuevos cubos y cambian las salidas de la tabla índice.

En el **algoritmo de Hashing Dinámico**, partimos de:

- k una clave física para direccionar.
- $k' = h(k)$ un entero entre 0 y M

- n un número de bits que tiene k' en binario.
- $d \leq n$, los primeros d dígitos de k', que seleccionan el cubo donde está el registro(pseudoclave).
- $b < d <= n$, inicialmente el archivo tiene 2^b cubos distintos y como máximo tendrá 2^d. Si son necesarios más, hay que aumentar d.

Entonces, si tenemos una tabla índice con 2^d filas, en la primera columna de esta tabla (valores de campo clave) se sitúan las posibles soluciones de d dígitos binarios (d es la profundidad global de la tabla). En principio, las entradas cuyos b primeros dígitos son iguales apuntan al mismo cubo. Allí se almacenan los registros cuyo valor de k' tiene esos b primeros dígitos. Todos los cubos suelen tener profundidad local igual a b. Por último, al llenar un cubo se divide en 2, poniendo en uno los registros con el dígito $b+1$ de k' a 0 y en otro los que lo tienen igual a 1. Aumenta entonces la profundidad local de estos cubos en uno.



De este modo, solventamos los problemas del acceso directo, aunque tenemos inconvenientes pues tenemos que usar una tabla índice adicional (y por tanto acceder más a disco si no cabe en memoria), y el tamaño de la tabla depende de d.