ESTRUCTURAS DE ALMACENAMIENTO Y MÉTODOS DE ACCESO

Los datos que constituyen la BD están almacenados físicamente en un medio de almacenamiento en el ordenador; más concretamente en almacenamiento secundario de disco magnético, que es el soporte más difundido para almacenar ficheros de bases de datos en línea, por varias razones:

- las BD suelen consistir en grandes cantidades de información permanente, cuyo tamaño no cabe en memoria principal
- En el almacenamiento secundario es más difícil que ocurra un fallo que suponga la pérdida de datos
- Su coste es inferior

Estudiaremos formas de organizar ficheros de datos en disco para conseguir un acceso a la BD con rendimiento aceptable, es decir, diferentes técnicas para almacenar en disco grandes cantidades de datos estructurados.
Los administradores (DBA) y los diseñadores de bases de datos deben conocer las ventajas e inconvenientes de cada técnica que ofrece un SGBD, para diseñar e implementar la BD y para operar con los datos; el SGBD suele ofrecer varias opciones para organizar los datos y en el Diseño Físico de la BD es necesario elegir la técnica de organización adecuada para la BD que se diseña. Además, los implementadores de los SGBD deben conocer las posibles técnicas de organización, para implementarlas de manera eficiente y ponerlas a disposición del DBA y los diseñadores.

Una aplicación, en un momento dado, necesita acceder a parte de la BD. Deberá localizarla en disco, copiarla en memoria principal, procesarla y, en el caso de que la haya modificado, reescribirla en disco.

Conceptos generales

Los datos en disco se organizan en FICHEROS de REGISTROS.
Un **registro** es una colección de valores o elementos de información relacionados entre sí (que tienen que ver con un mismo concepto de la realidad). Los registros describen 'entidades' y sus 'atributos'. Cada atributo corresponde a un **campo** del registro, que toma valores de cierto tipo de datos.

Ejemplo:

```
NOMBRE DEL TIPO DE REGISTRO     NOMBRE DEL CAMPO     TIPO DE DATOS DEL CAMPO (valores que puede aceptar)

type EMPLEADO = record          NOMBRE               : packed array [1..30] of character;
                                NSS                  : packed array [1..9] of character;
                                SALARIO              : integer;
                                CÓDIGO_PUESTO        : integer;
                                DEPARTAMENTO         : packed array [1..20] of character;
                end;
```

Los registros se almacenan de forma tal que sea posible recuperarlos (leerlos) de forma eficiente siempre que se necesiten.

Hemos de recordar que un disco magnético está dividido en pistas, y éstas en sectores.
La división de una pista en **bloques** de igual tamaño (páginas) la realiza el Sistema Operativo cuando da formato al disco.

Consideraremos la **transferencia de datos entre** la **memoria principal y** el **disco en unidades de bloques**.

Cuando se necesita la información contenida en cierto bloque, se le pasa al dispositivo de E/S del disco la dirección hardware (en disco) del bloque y también la dirección de un buffer de memoria[1]. Una operación de lectura copiaría el bloque del disco al buffer. Una operación de escritura copiaría el bloque del buffer al disco.

El tiempo total de localización y transferencia de un bloque desde el disco a memoria principal (t_T) suele ser la suma del tiempo de búsqueda, más el retardo rotacional, más el tiempo de transferencia de bloque. Este tiempo t_T es mucho mayor que el tiempo que la CPU tarda en procesar la información contenida en dicho bloque: el "cuello de botella" está en la localización de los datos en disco. La solución es conseguir estructurar los ficheros de forma que se **minimice el número de transferencias de bloques** necesarias para localizar y transferir los datos solicitados desde el disco a memoria principal.

Un **fichero** es una secuencia de registros. Lo más común es que todos los registros sean del mismo tipo, por ejemplo en un fichero ASIGNATURAS todos los registros serán del mismo tipo (con campos para el código y el nombre de la asignatura, su número de créditos, etc.). Si no es así, el fichero se denomina mixto.
Un **fichero mixto** es aquel que contiene registros de diferentes tipos que están íntimamente relacionados entre sí; se almacenan juntos en los bloques de disco, forman parte del mismo fichero físico (aunque corresponderían a diferentes relaciones o tablas). Esto resulta ventajoso para aumentar la eficiencia de consultas u operaciones en las que intervienen ambas tablas (JOIN). Por ejemplo, en un mismo fichero, junto al registro de cada ESTUDIANTE podrían colocarse los registros BOLETÍN_NOTAS de dicho estudiante.

Si todos los registros del fichero tienen exactamente el mismo tamaño (en bytes), se dice que el fichero se compone de **registros** de **longitud fija**. Véase la figura 1, al final del tema.
Si no es así, y los registros del fichero pueden tener tamaños distintos, se dice que el fichero esta compuesto por registros de **longitud variable**.

Varias son las causas por las que los registros de un fichero pueden ser de longitud variable:

- ✓ Si el fichero contiene registros del mismo tipo y ...
  - · Algún campo tiene longitud variable[2] (por ejemplo el nombre de las asignaturas), o
  - · Algún campo puede tener múltiples valores[3] (atributo multivaluado para los teléfonos de los estudiantes), o
  - · Algún campo es opcional (algunos registros no contienen dichos campos)[4]
- ✓ Siempre que el fichero contiene registros de distinto tipo (fichero mixto)[5]

Los **registros** de un fichero **se asignan a bloques** de disco, pues el bloque es la unidad de transferencia de datos entre la memoria principal y la secundaria (el disco). Está claro que si el tamaño de bloque es mayor que el de registro, en cada bloque de disco se almacenará más de un registro de datos.

[1] El buffer es un área reservada contigua, en memoria principal, que (normalmente) tiene el tamaño de un bloque. Puede denominarse también "área intermedia".
[2] En este caso, es necesario utilizar, dentro de los registros, una marca de "fin de campo". Véase figura 1(b).
[3] Será necesario, además de la marca de "fin de campo", una marca de separación entre los distintos valores.
[4] En este caso, se almacena en cada campo un par <nombre-campo=valor-campo>, además de la marca "fin de campo" y otra de "fin de registro". Véase figura 1(c).
[5] Cada registro irá precedido de un código indicativo de su tipo de registro.

Consideramos un bloque de tamaño B bytes.

- En un fichero con registros de longitud fija de R bytes, si $B \geq R$, el número de registros por bloque (el **factor de bloques**[6], fbl) será

$$fbl = \lfloor B / R \rfloor$$

  Normalmente la longitud del registro no será un divisor exacto del tamaño de bloque (es decir, no suele suceder que en un bloque quepa un número entero de registros), así que en cada bloque se tendrá un espacio desocupado:

$$B - (fbl * R)$$

  Para aprovechar este espacio no utilizado, podría almacenarse ahí una parte de un registro y el resto del registro en otro bloque (que sería el bloque consecutivo o, si no es así, el primer bloque contendría un puntero hacia el bloque que contiene el resto del registro).
  Este tipo de organización se llama **extendida**, pues los registros "pueden extenderse más allá del final de un bloque" (nota: siempre que el tamaño del registro sea superior al del bloque será necesario utilizar una organización extendida).
  Si no se permite que un mismo registro esté almacenado parte en un bloque y parte en otro bloque, la organización del fichero será **no extendida**. Véase la figura 2.

- En el caso de un fichero con registros de longitud variable, puesto que cada bloque puede contener un número distinto de registros, se define el factor de bloques (fbl) como el número medio de registros por bloque.
  De esta forma, es posible estimar cuántos bloques serán necesarios para almacenar un total de r registros:

$$b = \lceil r / fbl \rceil$$

  Con este tipo de ficheros también pueden utilizarse ambos tipos de organización: extendida o no extendida. Será necesaria una organización extendida si el tamaño medio de los registros es grande (pues así se reducirá el espacio perdido por bloque).

Descriptor de fichero (cabecera)

Contiene información necesaria para permitir, a los programas, acceder a los registros del fichero. Incluye, por ejemplo, información para determinar la dirección de los bloques en disco, así como de descripción del formato de los registros:

- Fichero de organización no extendida y registros de longitud fija:
  - longitud y orden de los campos
- Fichero de longitud variable:
  - códigos de tipo de campo (para registros con campos opcionales),
  - caracteres separadores entre campos,
  - códigos de tipos de registro (para ficheros mixtos)

Operaciones con ficheros

Suelen dividirse en operaciones de obtención y de actualización de datos.

- Operaciones de **obtención de datos**: no alteran la información, pues sólo localizan ciertos registros para examinar y procesar los valores de sus campos.
- Operaciones de **actualización**: modifican el fichero, pues insertan, eliminan registros o cambian los valores de los campos de algunos registros ya existentes.

En ambos casos, normalmente, será necesaria una operación previa de selección de los registros que se desea obtener o actualizar, con base en una condición de selección o de

[6] En algunos libros aparece como "factor de bloqueo", pero es más correcto llamarlo "factor de bloques".

búsqueda. Ésta es una expresión booleana que indica aquello que deben satisfacer los valores de los registros para ser seleccionados. Si varios registros satisfacen la condición, sólo se localiza el primero de ellos (según el orden físico dentro del fichero).
El Registro Actual es el registro localizado más recientemente.

Las operaciones "reales" para localizar, leer y actualizar los registros de un fichero dependen del sistema; la siguientes son algunas representativas:

- Buscar o Localizar[7]: busca el primer registro que satisface una condición de búsqueda. Transfiere el bloque que lo contiene a un buffer de la memoria principal (si no está ya ahí). El registro se localiza en el buffer y pasa a ser el registro actual.
- Leer u Obtener: copia el registro actual del buffer a una variable de programa o a un área de trabajo del programa de usuario.
- BuscarSiguiente: busca en el fichero el siguiente registro que satisface la condición de búsqueda. Transfiere el bloque que lo contiene a un buffer de la memoria principal (si no está ya ahí). El registro se localiza en el buffer y pasa a ser el registro actual.

- Eliminar: borra del buffer el registro actual y (antes o después) actualiza el fichero en disco para que refleje la eliminación.
- Modificar: cambia algunos valores de campos del registro actual y (antes o después) actualiza el fichero de disco para que refleje la modificación.
- Insertar, añade un nuevo registro en el fichero, para ello localiza el bloque donde se debe insertar, transfiere dicho bloque al buffer (si no está ya ahí), introduce el nuevo registro en el buffer y (antes o después) escribe el contenido del buffer en el disco para que el fichero refleje la inserción.

- Otras operaciones serían, por ejemplo, la lectura ordenada de (todos los) registros, la de buscarTodos (para obtener todos los registros que cumplen cierta condición) o la reorganización de los registros del fichero (se verá), y por supuesto, las de abrir (que incluye leer el descriptor del fichero y preparar el buffer) y cerrar el fichero.

ORGANIZACIÓN DE FICHEROS

La **organización de ficheros** es la estructuración de los datos de un fichero en **registros**, **bloques** y **estructuras de acceso**, lo cual incluye cómo se colocan los registros y los bloques en el disco y cómo se interconectan entre sí, puesto que los datos deben almacenarse de forma tal que sea posible localizarlos eficientemente cuando se necesiten.

En cambio, un **método de acceso** consiste en un grupo de programas que permite realizar las operaciones antes citadas (leer, insertar, etc.) sobre los datos de un fichero.
En general es posible aplicar varios métodos de acceso distintos (acceder utilizando diferentes técnicas de acceso) a una misma organización de fichero, aunque algunos de los métodos de acceso sólo podrán aplicarse a ficheros organizados de determinada manera (por ejemplo, no puede aplicarse un método de acceso indexado a un fichero sin índice).

Lo adecuado es conseguir una combinación "organización" + "método de acceso" tal que se maximice la eficiencia de las operaciones que se realicen sobre el fichero más frecuentemente.

[7] Se utilizan dos verbos diferentes para indicar si el registro sólo se va a obtener (buscar) o también se va a actualizar (localizar)

Tipos de Organización de Ficheros

1. Organización Primaria
   - 1.1. Ficheros no ordenados (organización en montículo o montón)
   - 1.2. Ficheros ordenados (organización secuencial)
   - 1.3. Ficheros dispersos (organización directa)

2. Organización Secundaria: ficheros indexados o indizados (organización indexada o indizada)
   - 2.1. Índices ordenados de un nivel
   - 2.2. Índices multinivel

1. ORGANIZACIÓN PRIMARIA

1.1. FICHEROS DE REGISTROS NO ORDENADOS

Los registros se colocan en orden de inserción y al final del fichero.

La operación de inserción de un nuevo registro resulta muy eficiente: el descriptor de fichero almacena la dirección del último bloque con datos; se copia en un buffer dicho bloque; se le añade el nuevo registro y se copia el bloque del buffer al disco (reescritura).
La búsqueda de un registro que satisfaga cierta condición necesita una búsqueda **lineal**, bloque a bloque, lo cual resulta costoso puesto que examina (y por tanto transfiere) muchos bloques: si el fichero consta de b bloques, examinará una media de $b/2$ bloques para encontrar el primer registro que cumple la condición, y si ningún registro la cumple o la satisfacen varios, los examinará todos (b).

La eliminación de un registro existente supone a) encontrar el registro que se desea borrar (búsqueda lineal en el fichero), b) copiar el bloque que lo contiene en un buffer de memoria, c) eliminar el registro del buffer y d) copiar el bloque del buffer en el disco (reescritura).

Otra opción sería la eliminación por marca (en lugar del "borrado físico" del registro dentro del bloque): a) encontrar el registro que se desea borrar (búsqueda lineal), b) copiar el bloque que lo contiene en un buffer de memoria, c) marcarlo como no válido (o eliminado) y d) copiar el bloque del buffer en el disco. Obviamente, es necesario que cada registro incluya un campo adicional (el marcador de eliminación) que indique si es válido (contiene datos "reales") o no. Además, el algoritmo de búsqueda deberá examinar sólo los registros válidos dentro de cada bloque.

En ambas técnicas de borrado se desperdicia el espacio que ocupaban los registros eliminados. Para recuperar el espacio desocupado puede realizarse periódicamente una reorganización del fichero, es decir acceder de forma lineal al fichero, empacando los registros para eliminar huecos o registros no válidos (los bloques quedan llenos de registros).

En lugar de reorganizar cada cierto tiempo el fichero, podrían aprovecharse los huecos (o los registros marcados como no válidos) para almacenar nuevos registros; esto complicaría el algoritmo de inserción (que hasta ahora era muy simple) puesto que sería necesario seguir la pista de las posiciones que van quedando vacías.

La modificación o actualización de un valor dentro de un registro, supone a) localizar el registro (búsqueda lineal), b) copiar el bloque que lo contiene en un buffer de memoria, c) modificar el valor del registro en el buffer y d) reescribir en el disco el bloque del buffer.

Este tipo de ficheros (de registros no ordenados) puede utilizar una organización extendida o no extendida, así como registros de longitud fija o variable.
Si los registros son de longitud variable, puede suceder que, una vez modificado el registro, no quepa en el espacio que antes ocupaba en el bloque. En este caso se debe eliminar el registro antiguo e insertar uno nuevo (modificado).

La lectura ordenada según el valor de cierto campo de los registros del fichero supone la creación de una copia (ordenada) del fichero. Como esto resulta muy costoso, sobre todo si el fichero ocupa mucho espacio, para crear la copia se suele utilizar una técnica especial de ordenación externa, por ejemplo alguna variación de la ordenación por fusión (de bloques) o mergesort.

1.2. FICHEROS DE REGISTROS ORDENADOS (SECUENCIALES)

Estos ficheros constan de registros almacenados de forma **ordenada físicamente según** el **valor** de cierto **campo**, denominado campo de ordenación. Un ejemplo sería un fichero LIBROS cuyos registros estuvieran almacenados en orden alfabético de sus títulos.

Si dicho campo es una clave candidata del fichero (es decir, su valor identifica unívocamente cada uno de los registros), se le llama clave de ordenación. Sería el caso de tener el fichero de los libros con los registros ordenados según su código ISBN.

Este tipo de organización tiene **ventajas** con respecto a la organización no ordenada:

1) La lectura ordenada resulta muy eficiente si el orden es el de los valores del campo de ordenación (listado de títulos de libros en orden alfabético), puesto que es una lectura secuencial del fichero, que no necesita de ordenaciones adicionales.
2) Encontrar el siguiente registro en orden según el campo de ordenación (dado un título de un libro, encontrar el siguiente en orden alfabético), no suele necesitar acceso a otro bloque, puesto que está en el mismo bloque que el registro actual (excepto cuando el registro actual es el último del bloque, claro!).
3) Si la condición de búsqueda está basada en el valor del campo de ordenación, el acceso resulta mucho más rápido, puesto que puede utilizarse el algoritmo de búsqueda binaria.

Nótese que si es necesario leer el fichero siguiendo un orden de los valores de un **campo no de ordenación** (listado de libros ordenados por su fecha de publicación), o si la condición de la búsqueda está basada en el valor de un campo no de ordenación (listado de libros publicados en Madrid, antes del 16-Nov-1999), usar este tipo de organización **no** supone ninguna **ventaja** con respecto a la de los ficheros no ordenados, pues la búsqueda sería lineal y la lectura ordenada supondría una copia ordenada del fichero.

La inserción y eliminación de registros son operaciones costosas, puesto que deben conservar el orden de los registros.

La inserción supone a) encontrar la posición correcta dentro de un bloque, según el valor de su campo de ordenación, en el que debe ser almacenado (se podría encontrar mediante búsqueda binaria), y b) abrir espacio en el fichero, lo cual implica el desplazamiento (lectura y escritura) de los registros[8], operación que puede llevar mucho tiempo si el fichero es grande.

[8] Si consideramos que el fichero contiene r registros en total, el número promedio de desplazamientos es de r/2 registros en cada operación de inserción.

La eliminación de registros tiene el mismo problema, aunque es menos grave si se utiliza el borrado por marca y se reorganiza el fichero periódicamente.

Para hacer más eficiente la inserción, podría dejarse espacio libre en cada bloque para nuevos registros. Pero una vez agotado este espacio, surge de nuevo el problema inicial.

Otra opción sería utilizar un fichero no ordenado auxiliar. Sería un fichero temporal de desborde. Al fichero ordenado se le llama ahora fichero maestro o principal. Los nuevos registros son almacenados al final del fichero de desborde (montículo) y periódicamente se realiza una fusión (reorganización) de los ficheros principal y auxiliar. De esta forma la inserción es muy eficiente, pero la búsqueda resulta más complicada, pues el registro debe buscarse primero en el fichero principal, mediante una búsqueda binaria y si no es encontrado ahí, debe buscarse en el fichero de desborde mediante búsqueda lineal.

En la modificación del valor de un campo de un registro determinado, podemos distinguir varios casos:
- Si la condición de búsqueda del registro se basa en el campo clave de ordenación, se utiliza la búsqueda binaria para localizarlo. Si no, se debe realizar una búsqueda lineal.
- Además, si el campo que se desea modificar no es el de ordenación, el registro es actualizado y reescrito (sin problema si los registros son de longitud fija).
- Mientras que si el campo que debe ser modificado es el de ordenación, el nuevo valor puede provocar el cambio de lugar (para mantener el orden) del registro en el fichero, lo cual supondría la eliminación del registro antiguo y la inserción del modificado en el lugar adecuado.

La lectura ordenada según el campo de ordenación es muy eficiente (consiste en una sencilla lectura secuencial de los registros), siempre que no se tenga en cuenta el fichero de desborde; pero si éste se considera, es necesario a) reorganizar los ficheros (es decir, ordenar los registros del fichero auxiliar y fusionarlos con los del fichero principal, desechando los registros marcados como eliminados) para después b) leer de forma secuencial el fichero (obteniendo los registros en orden).

1.2B. FICHEROS SECUENCIALES ENCADENADOS

Los registros están dispuestos secuencialmente de forma lógica (y no física, como hasta ahora hemos visto), utilizando punteros para establecer el orden de los registros según el valor del campo de ordenación.

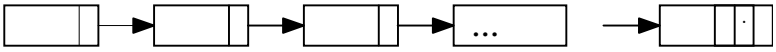
Así, cada registro contiene un campo con la dirección del siguiente registro en orden. Dicho registro siguiente puede estar almacenado en "cualquier lugar" dentro del fichero.
De este modo se tiene el fichero ordenado, sin necesidad de que los registros lo estén físicamente.

La inserción, modificación y eliminación son ahora operaciones mucho más flexibles. Por ejemplo, al insertar un nuevo registro, es almacenado al final del fichero, se localiza el lugar que le corresponde según el orden y se reajustan los punteros convenientemente.

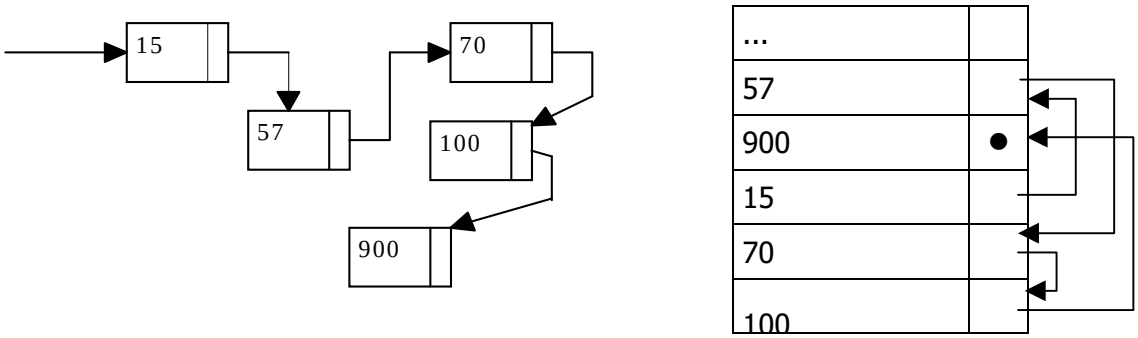
Suele ser necesario mantener un par de registros especiales que almacenan las direcciones del primer y último registros (según el orden).

Este tipo de organización puede implementarse con diversas estructuras de datos, como las listas lineales, las listas múltiples, los anillos o listas circulares, los árboles binarios, etc.

Listas lineales.



A cada registro le precede y antecede otro, en orden de clave. Cada registro apunta al siguiente.



No es posible acceder de forma directa a un registro determinado, puesto que es necesario leer cada registro para saber dónde está el siguiente. Así, la búsqueda de cualquier registro es siempre lineal (a partir del registro inicial).

La inserción se lleva a cabo colocando el nuevo registro en una zona libre del fichero y reajustando los apuntadores.

La modificación de un registro supone a) la búsqueda lineal del registro, siguiendo los punteros, b) actualización del registro y c) reescritura del registro en disco.
Esta reescritura se realizará en la misma zona del fichero si los registros son de longitud fija.
Si la longitud de los registros es variable y no cabe en el espacio que ya ocupaba, es necesario eliminar el registro antiguo, almacenar el registro en otra zona del disco y reajustar los punteros adecuadamente.
Si la modificación ha implicado la variación de la posición del registro dentro del fichero (cambio del valor del campo de ordenamiento), es preciso reajustar los punteros para mantener el orden. Esto no será necesario si la modificación no afecta al orden de los registros.

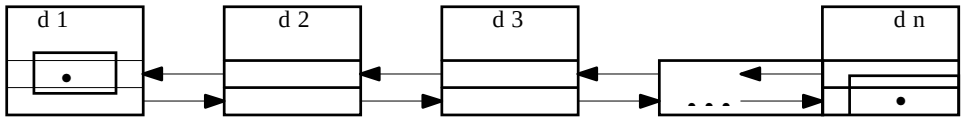
La eliminación tan solo supone el reajuste de los apuntadores. Los registros borrados son zonas de espacio que se desperdician.
Es posible aprovechar tales espacios desocupados, pero si éstos **no** se recuperan, los ficheros muy volátiles[9] quedan con muchos huecos (espacio desperdiciado), si además los ficheros tienen una alta tasa de inserciones (al final del fichero), su tamaño crece mucho, de forma que la lectura ordenada resulta un proceso muy lento, al tener que realizar muchos movimientos (accesos a bloques distintos), además serán necesarias frecuentes reorganizaciones del fichero.

[9] Ficheros sobre los que se realiza muchas inserciones, eliminaciones y/o modificaciones

Listas múltiples.

Los registros están encadenados por más de una lista. Este tipo de estructuras permite, por ejemplo...

1) **Recorrido hacia atrás**.
   Acceso a registros anteriores sin necesidad de recorrer de nuevo el fichero desde el principio de la lista.



   Las operaciones de borrado, inserción y modificación son análogas a las de las listas simples, pero requieren el reajuste de un número mayor de apuntadores.

2) **Reducción del tiempo de acceso**.
   Por ejemplo, si disponemos de un fichero que contiene información histórica desde 1900 hasta el día de hoy, para realizar la extracción de datos referentes a 1996 podríamos utilizar una serie de registros especiales que apuntaran al (contuvieran la dirección del) primer registro de cada década, para acceder rápidamente a la zona adecuada y buscar a partir de ahí.

3) **Ficheros clasificados por más de una clave**.
   Para ello se utilizan las llamadas listas cruzadas.
   Los registros se almacenan ordenados físicamente según uno de los campos y el resto de ordenaciones se implementa mediante apuntadores.
   Cada registro tiene tantos campos-puntero como listas diferentes haya.
   En este caso, por ejemplo la operación de inserción implicaría a) inserción física del nuevo registro según el valor del campo de ordenación física, b) recorrido secuencial de una lista hasta encontrar un valor del campo correspondiente a dicha lista, superior al del nuevo registro y reorganización de punteros, y c) repetir el paso b para cada una del resto de listas.
   También pueden implementarse todas las ordenaciones de forma lógica, mediante listas (en este caso, los registros estarán almacenados desordenadamente).

   Este tipo de estructura reduce el tiempo de acceso a los registros del fichero según el valor de cada uno de los campos sobre el que haya construida una lista.

Existen más estructuras, como las de **anillo** o **listas circulares** (sin registros de inicio ni de fin, pero con un indicador de fin) y los **árboles binarios**.

Un árbol binario...

- Es una estructura no lineal.
- Los registros están almacenados sin seguir un orden físico ni lógico.
- Son la base para implementar cualquier otro tipo de árbol
- Los árboles binarios de búsqueda posibilitan la implementación de ficheros permitiendo un procesamiento de su contenido muy eficiente

1.3. FICHEROS DE ACCESO DIRECTO (TÉCNICAS DE DISPERSIÓN. HASHING)

Este tipo de organización de ficheros permite un acceso muy rápido a determinados registros según ciertas condiciones de búsqueda. La condición de búsqueda debe ser una igualdad sobre un solo campo, el campo de dispersión. Si éste es un campo clave (suele serlo) se le llama clave de dispersión.

Este tipo de organización se basa en el establecimiento de una función h de dispersión, que se aplica al valor k del campo de dispersión de los registros de un fichero, y produce la dirección del bloque de disco en el que está almacenado el registro que contiene dicho valor. La localización del registro dentro del bloque puede realizarse en el buffer de memoria principal.

Algunos ejemplos de funciones h:

a. Módulo. h(k) = K mod M
b. Plegado (folding). Si k= k1k2k3k4, entonces h(k) = k1k2 ⊕ k3k4, donde la operación ⊕ puede ser la suma aritmética o la operación lógica XOR
c. Truncamiento. Si k= k1k2k3k4, entonces, por ejemplo h(k) = k1k3

Una de las desventajas de la dispersión es que no garantiza que valores distintos de k produzcan direcciones diferentes. Y esto es debido a que el espacio del campo de dispersión (el número de posibles valores de k) es mucho mayor que el espacio de direcciones (número de direcciones disponibles para los registros).

Tipos de dispersión:

1. Interna: en ficheros internos (ficheros temporales pequeños, en memoria principal)
2. Externa: en ficheros de disco (es la que más nos interesa)

En la **dispersión interna**, el espacio de direcciones destino es un conjunto de M registros (es decir, el fichero se implementa en memoria principal mediante un array de M posiciones). La función de dispersión, por tanto, transformará el valor del campo de dispersión en un entero entre 0 y M-1 (correspondiente a los índices del array). Véase figura 4(a).

Una **colisión** tiene lugar cuando al aplicar h al valor del campo de dispersión de un registro que se desea insertar en el fichero, se obtiene una dirección (posición) en la que ya está almacenado otro registro. Es necesario introducir el nuevo registro (que se denomina sinónimo del que ya ocupa su posición en el fichero) en otro lugar, mediante una técnica de resolución de colisiones.

Técnicas de Resolución de Colisiones

1) **Direccionamiento abierto**. Se examinan secuencialmente las siguientes posiciones hasta encontrar una vacía, donde se almacena el nuevo registro. Es un procedimiento muy lento.
2) **Encadenamiento**. Se mantiene un área de desbordamiento (se extiende el array con varias posiciones) donde almacenar cada sinónimo. Cada posición contiene un campo más (puntero). Se coloca el registro nuevo en una posición de desborde desocupada, y es apuntada desde la dirección que le correspondía por su dirección de dispersión (o bien, desde un sinónimo anterior ya en la zona de desbordamiento). Así, se tendrá una lista de sinónimos (registros de desborde) para cada dirección de dispersión. Véase figura 4(b).

   Los algoritmos de inserción, eliminación, consulta y modificación son más sencillos que para el direccionamiento abierto.
3) **Dispersión múltiple**. Se aplica una segunda función de dispersión. Si se produce de nuevo una colisión, se aplica direccionamiento abierto, o bien una tercera función de dispersión y luego direccionamiento abierto.

La función h debe ser sencilla y rápida de calcular. Además su principal objetivo es conseguir una distribución uniforme de los registros en el espacio de direcciones, es decir, sin acumulamientos en zonas determinadas del fichero. De esta manera, se minimizará el número de colisiones y se ocupará el máximo número de posiciones del fichero (se reducirá el desaprovechamiento del espacio en disco).

En la **dispersión externa**, el espacio de direcciones destino es un conjunto de **cubetas**.
Una cubeta es un conjunto de registros, de forma que puede contener un bloque de disco o bien varios bloques contiguos de disco (física o lógicamente).
La función h ahora se aplica al valor del campo de dispersión de un registro y devuelve el número de cubeta donde se almacenará dicho registro.

En el descriptor del fichero se mantiene una tabla que establece la correspondencia entre cada número de cubeta y la dirección de disco en la que ésta comienza. Véase figura 5.

El uso de cubetas decrementa el número de colisiones, pues los registros sinónimos que quepan en la cubeta se dispersan sin problemas dentro de ésta (se almacenan secuencialmente).

El problema surge cuando la cubeta está completa, y un nuevo registro debe almacenarse en esa cubeta. La solución sería que cada cubeta contenga un puntero al primer registro de una lista de registros de desborde, almacenados en una cubeta de desborde (común a todas las demás). Si ésta se completa puede encadenarse con otra cubeta de desborde. Véase figura 6.

La dispersión posibilita el acceso más rápido posible a cierto registro dado su valor de campo de dispersión (es un acceso directo, sin tener en cuenta el resto de registros). La mayoría de las veces basta un acceso a bloque para localizar un registro.
Sin embargo, esta técnica no resulta muy útil si se desea leer en orden los registros, pues la mayor parte de las funciones de dispersión no mantienen los registros ordenados según el valor del campo de dispersión (aunque algunas sí lo hacen, como la función identidad, p. ej.).
Por otra parte, también tiene el inconveniente de asignar una cantidad fija de espacio a cada fichero. Veamos, si se dispone de M cubetas, cada una con capacidad para m registros como máximo. A lo más, caben m*M registros en el espacio asignado. Si el número (real) de registros del fichero es mucho menor que m*M, se desaprovechará mucho espacio. Si el número de registros es muy superior a m*M, el número de colisiones será muy elevado, de forma que la obtención de registros será muy lenta, debido a las largas listas de registros de desborde.

La búsqueda de un registro con base en el valor de un campo que no sea el de dispersión es tan costosa como en un fichero no ordenado, pues debe realizarse una búsqueda secuencial.

La eliminación de un registro implica sacarlo de la cubeta en la que se encuentra. Podría pasarse uno de los registros sinónimos desde la lista de desborde a la cubeta, para aprovechar ese espacio. Si el registro que se desea eliminara ya estuviera en una lista desborde, bastaría con sacarlo de dicha lista.

Vemos que es necesario seguir la pista de las posiciones vacías del área de desborde mediante una lista enlazada de posiciones de desborde desocupadas.

En la modificación de un valor dentro de un registro, si la condición de búsqueda es una comparación de igualdad sobre el campo de dispersión, la localización del registro es muy eficiente aplicando la función h; en caso contrario es necesario realizar una búsqueda lineal del registro. Además, si debe modificarse un campo no de dispersión, se actualiza el registro y se reescribe en la misma cubeta donde estaba; en cambio, si el campo cuyo valor debe cambiar sí que es el campo de dispersión, la modificación puede implicar que el registro "salte" a otra cubeta, lo cual provoca la eliminación del registro antiguo y la inserción del nuevo.

2. ORGANIZACIÓN SECUNDARIA. FICHEROS DE ÍNDICES

Los índices son estructuras de acceso adicionales (de ahí la denominación de organización secundaria), puesto que un cada índice es utilizado junto con un fichero de datos (fichero principal) cuyos registros están organizados siguiendo algún tipo de organización primaria.

El índice agiliza la obtención de registros (del fichero principal) en respuesta a ciertas condiciones de búsqueda o de selección.

Tipos de índices:

2.1. Índices ordenados de un nivel
   a. Primarios
   b. Secundarios
   c. De Agrupamiento

2.2. Índices multinivel
   a. Árboles B
   b. Árboles B^+

2.1. ÍNDICE ORDENADO DE UN SOLO NIVEL

Es una estructura de datos definida con base en los valores de un campo del fichero principal, llamado genéricamente campo de indexación o indización.

El fichero de índice contiene un conjunto de registros, llamados entradas. Cada entrada contiene un campo para contener los valores del campo de indexación, junto con una lista de punteros, cada uno de los cuales apunta a uno de los bloques que contienen algún registro con dicho valor en tal campo, dentro del fichero principal (de forma análoga a como cada línea (entrada) del índice de un libro (fichero) contiene cada palabra importante o tópico (valor de indexación), junto con los números (punteros) de todas las páginas (bloques) en las que aparece tal tópico).

Las entradas del índice están ordenadas (según el valor del campo de indexación), así que es posible realizar búsquedas binarias en el índice de forma eficiente, puesto que el índice suele ser de poco tamaño. Si el índice contiene b_i bloques, la búsqueda binaria necesita acceder a una media de $\log_2(b_i)$ bloques.

Un índice es **denso** si contiene una entrada por cada registro del fichero de datos. En otro caso, el índice es **no denso**, por ejemplo cuando contiene una entrada por cada bloque del fichero principal.

2.1.A. ÍNDICES PRIMARIOS.

Son índices no densos y se construyen sobre ficheros de datos con registros ordenados según los valores de un campo que es clave (suele ser la clave primaria).

El campo de indexación es el campo clave de ordenamiento del fichero de datos

El índice es un fichero con registros (entradas) de dos campos y longitud fija.
Los campos son:

- Uno (K) de igual tipo de datos que la clave de ordenamiento del fichero de datos.
- El otro (P) un puntero a bloque de disco.

Una entrada contiene los valores <K(i), P(i)>, donde K(i) es el valor de la clave del primer registro[10] del bloque i (llamado registro ancla del bloque), y P(i) es la dirección de comienzo del bloque i.

El número total de entradas en el índice es el número de bloques del fichero de datos ordenado en disco.
El fichero de índice está ordenado según los valores K(i).

Véase figura 7.

El número de bloques del fichero índice es mucho menor que el número de bloques del fichero de datos, puesto que:

- El número de entradas del índice es muy inferior al de registros del fichero de datos (tiene una entrada por cada bloque)
- El tamaño de cada entrada es mucho menor que el tamaño de un registro de datos, pues sólo tiene dos campos.

Así que en un bloque de disco caben más entradas de índice que registros de datos.
Por tanto, la búsqueda binaria en el índice necesita muchos menos accesos a bloques que si se realizara directamente la búsqueda en el fichero de datos.

La búsqueda de un registro cuya clave tiene el valor k implica a) realizar una búsqueda binaria en el índice hasta encontrar una entrada i tal que $K(i) \leq k < K(i+1)$; entonces el registro buscado está en el bloque con dirección P(i)[11] y b) una vez que el bloque con dirección P(i) ya se ha copiado en el buffer, leerlo de forma secuencial hasta encontrar el registro.

La inserción en la posición correcta de un nuevo registro en el fichero de datos implica, además de lo que ya vimos para la inserción en un fichero con registros ordenados (ver sección 1.2), la modificación de las entradas del índice, en el caso de que la inserción haya alterado los registros ancla de algunos bloques.

El borrado de registros también puede suponer la modificación del fichero de índice, si se ha eliminado algún registro ancla de bloque.

[10] Si K(i) contiene el valor de la clave del **último** registro del bloque i se mejora el tiempo de acceso.
[11] Esto es posible gracias al ordenamiento (físico o lógico) del fichero de datos según el valor de la clave.

2.1.B. ÍNDICES DE AGRUPAMIENTO.

También son índices no densos y se construyen sobre ficheros de datos con registros ordenados según los valores de un campo que NO es clave.

El campo de indexación es el campo de ordenamiento del fichero, y es posible que existan varios registros de datos con el mismo valor para tal campo, que se denomina ahora campo de agrupamiento.

Este tipo de índice acelera la obtención de todos los registros que tienen el mismo valor en el campo de indexación o agrupamiento.

El índice es un fichero con entradas de dos campos y longitud fija:
- Uno, K, de igual tipo de datos que el campo de agrupamiento del fichero de datos.
- El otro, P, un puntero a bloque de disco.

Una entrada contiene los valores <K(i), P(i)>, donde K(i) es un valor (distinto) del campo de agrupamiento y P(i) es un puntero al primer bloque del fichero de datos que contiene un registro con ese valor K(i) en el campo de agrupamiento; los registros de datos con el mismo valor K(i) están tras él en el fichero principal, pues éste está ordenado por ese campo (no clave).

El índice contiene una entrada i por cada valor distinto del campo de agrupamiento del fichero de datos.
El fichero de índice está ordenado según los valores K(i).

Véase figura 8.

La inserción y eliminación de un registro son operaciones costosas debido a la ordenación física de los registros de datos.

Esto puede aliviarse reservando un bloque completo para cada valor distinto del campo de agrupamiento, de forma que los registros con dicho valor se colocan en ese bloque y, si se necesita más espacio, se enlaza un nuevo bloque (vacío) con el que se ha completado.

Véase figura 9.

2.1.c. ÍNDICES SECUNDARIOS.

En este caso, el campo de indexación es un campo (de los registros de un fichero de datos) que no es de ordenamiento.

Cada entrada del índice contiene dos campos:
- Uno, K, de igual tipo de datos que el campo de indexación.
- El otro, P, un puntero a bloque o bien a registro del fichero en disco

Pueden existir muchos índices secundarios (muchos campos de indexación) para un mismo fichero de datos.

ÍNDICE SECUNDARIO SOBRE UN CAMPO CLAVE DEL FICHERO PRINCIPAL

Recordemos que el campo no es de ordenamiento y, puesto que es clave, ahora se le llama clave secundaria.

Es un índice denso, pues contiene una entrada i por cada registro del fichero de datos.

Cada entrada es un registro de longitud fija y dos campos <K(i), P(i)>, donde K(i) es el valor de la clave secundaria de un registro y P(i) es un apuntador, bien a ese registro, o bien al bloque donde está dicho registro (en cuyo caso, una vez copiado el bloque en el buffer de memoria, para acceder al registro será necesario buscarlo secuencialmente dentro del bloque hasta dar con él).

Las entradas están ordenadas según el valor de K(i), de forma que es posible realizar búsquedas binarias en el índice.
Véase figura 10.

Los registros en el fichero de datos no están ordenados según los valores de la clave secundaria, así que no es posible utilizar anclas de bloque; por esto se necesita una entrada de índice por cada registro y no por cada bloque (el índice es denso).

Cuando se construye un índice secundario sobre un fichero de datos, según los valores de uno de sus campos, se mejora mucho la búsqueda de un registro arbitrario (dado su valor para dicho campo), puesto que si el índice no existiera, debería realizarse una búsqueda lineal.

Por otra parte, un índice secundario necesita más espacio que un índice primario y su tiempo de búsqueda es superior, pues contiene más entradas.

ÍNDICE SECUNDARIO SOBRE UN CAMPO NO CLAVE DEL FICHERO PRINCIPAL

Recordemos que el campo no es de ordenamiento y, puesto que no es clave, es posible que varios registros de datos contengan el mismo valor en el campo de indexación.

A la hora de construir índices de este tipo, existen varias opciones; estas son algunas:

a) Incluir varias entradas de índice con el mismo valor de K(i). Es decir, cada entrada será un registro de longitud fija que contendrá los valores <K(i), P(i)>, donde P(i) es un puntero a uno de los registros con dicho valor K(i) en el campo de indexación.
En este caso el índice es denso.

b) Incluir una única entrada de índice por cada valor distinto del campo de indexación, y permitir que las entradas del índice tengan longitud variable[12], de forma que cada entrada de índice pueda contener varios punteros <K(i), [$P(i_1)$, $P(i_2)$..$P(i_n)$]>, donde cada $P(i_j)$ apunta al bloque donde esté uno de los n registros con valor K(i) en el campo de indexación.

En los casos a) y b) será necesario modificar el algoritmo de búsqueda binaria en el índice.

c) Incluir una única entrada por cada valor distinto del campo de indexación, pero las entradas son de longitud fija. Ahora se utiliza un nivel adicional de indirección para manejar "punteros múltiples".
Es decir, cada entrada contendrá los valores <K(i), P(i)>, donde P(i) es un puntero a un bloque de apuntadores a registros (cada uno de los cuales apunta a un registro del fichero de datos con valor K(i) en el campo de indexación). Si hay muchos registros con valor K(i) y se agota el espacio en el bloque de apuntadores, éste se enlaza con un nuevo bloque de apuntadores.
Ahora, la obtención de datos necesita un acceso a bloque adicional, pues hay un nivel extra, pero los algoritmos de inserción, búsqueda y eliminación son sencillos.
Véase figura 11.

Una alternativa al uso del nivel adicional, sería que P(i) apuntara a un registro de datos con valor K(i), el cual estuviera ligado al resto de registros con el mismo valor para la clave secundaria, mediante una lista enlazada en el fichero de datos.

Los índices secundarios en general, representan un ordenamiento lógico de los registros, según los valores del campo de indexación.

[12] Debido a que el campo apuntador es multivaluado.

2.2. ÍNDICE DE MÚLTIPLES NIVELES

Para construir un índice de este tipo, se parte de la existencia de un índice ya creado sobre el fichero de datos en disco (principal).

Considera este índice (que ahora llamaremos índice de primer nivel, o de nivel base) como un fichero ordenado, con un valor distinto para cada K(i), de modo que es posible crear otro índice primario sobre dicho fichero; y éste será el índice de segundo nivel.

Para crearlo podemos utilizar anclas de bloque, por lo que el índice de segundo nivel tendrá una entrada por cada bloque del índice de primer nivel (será no denso).

De la misma manera, el tercer nivel es un índice primario sobre el de segundo nivel y tendrá una entrada por cada bloque del índice de nivel 2. Véase figura 12.

Sólo es necesario crear el índice de un nivel más, t+1, si el índice anterior, de nivel t, necesita más de un bloque de almacenamiento en disco.

Así, puede repetirse este proceso hasta que todas las entradas de un nivel t del índice quepan en un único bloque. En ese momento, se habrá completado el índice; y el índice de nivel t será el de nivel superior.

Este esquema de múltiples niveles es útil para cualquier tipo de índice (primario, de agrupamiento o secundario) siempre que el índice de primer nivel tenga valores distintos para K(i) y entradas de longitud fija.

Este tipo de organización reduce el número de bloques leídos al buscar un registro dado su valor del campo de indexación. El procedimiento de búsqueda sobre un índice de varios niveles desecha en cada paso, más registros que la búsqueda binaria en un índice de un solo nivel.

Observar que en el caso de que el índice de múltiples niveles fuera denso, es decir, el índice de nivel base es denso), es posible determinar si un registro está o no en el fichero principal sin necesidad de acceder a dicho fichero (¿cómo?), mientras que si el índice de múltiples niveles no es denso, sí será necesario acceder al fichero principal para poder determinar si contiene el registro.

Puesto que todos los niveles del índice son ficheros ordenados físicamente, surgen los consabidos problemas de ineficiencia al realizar operaciones de inserción, eliminación o modificación de registros de datos que impliquen inserción o eliminación de entradas del índice.

Estos problemas se reducen dejando espacio en todos los bloques de los ficheros de índice para nuevas entradas. Así, surgen los índices dinámicos de múltiples niveles que suelen ser implementados con estructuras de datos llamadas árboles B y árboles B^+, que permiten al índice expandirse y contraerse dinámicamente.

ÍNDICES DINÁMICOS DE MÚLTIPLES NIVELES CON BASE EN ÁRBOLES B Y B+

Antes de nada, recordemos algunos conceptos:

Un **árbol** está compuesto por un conjunto de nodos.

Cada **nodo**, salvo el nodo **raíz**, tiene un nodo padre y ninguno o varios nodos **hijos**. Los nodos con hijos se llaman nodos **internos**, mientras que los que no tienen hijos son nodos **hoja**.

El **nivel** de profundidad del nodo raíz es 0 y el de un nodo cualquiera es el nivel del padre +1.

El **subárbol** de un nodo es ese nodo y el conjunto de sus descendientes (sus hijos, los hijos de sus hijos, y así sucesivamente hasta llegar a nodos hoja).

Un **árbol** está **equilibrado** si todos sus nodos hoja están en el mismo nivel de profundidad.
El que un árbol esté equilibrado es una situación deseable, pues garantiza que no existen nodos en niveles muy altos que, en una búsqueda, necesitarían muchos accesos a bloques.

ÁRBOL DE BÚSQUEDA

Es un árbol guía en la búsqueda de un registro, dado el valor de uno de sus campos.

Un árbol de búsqueda de orden p es un árbol cuyos nodos cumplen:
- Contienen un máximo de $p-1$ valores
- Contienen como mucho p punteros a nodos hijos

Y cada nodo es de la forma: $<P_1, K_1, P_2, K_2,...,P_{q-1}, K_{q-1}, P_q>$, donde $q \leq p$ y
P_i es un puntero a un nodo hijo o nulo
K_i es un valor de búsqueda <u>único</u> dentro del árbol de búsqueda[13]

Y además cumplen estas dos restricciones:
a) En cada nodo se cumple que $K_1<K_2< ...<K_{q-1}$
b) Para todo valor X dentro del subárbol al que apunta Pi, se cumple:
$K_{i-1}<X<K_i$ para $1<i<q$
$X<K_i$ para $i=1$
$K_{i-1}<X$ para $i=q$

Estas dos reglas indican el procedimiento de una búsqueda de un registro con valor X en su campo de indexación (seguir siempre el puntero P_i que las cumpla).

Para usar los árboles de búsqueda con el objetivo de localizar registros almacenados en ficheros en disco...
- Los valores en los nodos serán los de uno de los campos del fichero principal, el llamado campo de búsqueda (igual al campo de indexación, si es un índice de múltiples niveles el que guía la búsqueda). Cada valor estará asociado a un puntero a registro o a bloque.
- Cada puntero será un apuntador al registro (del fichero en disco) que contiene el valor correspondiente, o bien al bloque en el que está dicho registro, o bien a un bloque de punteros (cada uno de los cuales apuntará a uno de los registros con igual valor en el campo de búsqueda) .
- El árbol de búsqueda se almacenará también en disco, y se asigna cada nodo del árbol a un bloque del disco.

[13] Aunque no tiene por qué ser único dentro del fichero de datos

La inserción de un registro de datos implica actualizar el árbol de búsqueda, añadiendo el valor del campo del nuevo registro y un puntero a éste. Son necesarios algoritmos de inserción y eliminación de valores en el árbol que no violen las dos restricciones anteriores. Estos algoritmos, en general, no garantizan que el árbol esté equilibrado.

La eliminación de un registro de datos puede dejar algunos nodos del árbol (bloques de disco) casi vacíos, con lo que se desperdiciará espacio y se aumentará el número de niveles en el árbol.

ÁRBOL B (BALANCEADO)

Es un árbol de búsqueda al que se le añaden más restricciones, de modo que resuelven los dos inconvenientes que acabamos de citar. Estas restricciones aseguran que este tipo de árboles en todo momento esté equilibrado y que el espacio desperdiciado (por eliminaciones) nunca es demasiado, pero al mismo tiempo, esto implica que los algoritmos de inserción y eliminación de valores en el árbol sean más complejos.

I. Árbol B de orden p, utilizado como estructura de acceso según un campo clave para buscar registros de un fichero de datos

Definición:

1. Cada nodo interno es de la forma
   $<P_1,<K_1,Pr_1>,P_2,<K_2,Pr_2>,...,P_{q-1},<K_{q-1},Pr_{q-1}>,P_q>$, donde $q \leq p$ y
   - P_i es un puntero de árbol a otro nodo (bloque de disco en fichero de índice) o nulo
   - K_i es un valor de búsqueda
   - Pr_i es un puntero de datos (al bloque donde está el registro de datos o al propio registro (bloque + desplazamiento) cuyo valor del campo clave de búsqueda es K_i)

2. En cada nodo se cumple que $K_1<K_2< ...<K_{q-1}$

3. Para todo valor X del campo clave de búsqueda en el subárbol al que apunta P_i, se cumple:
   - $K_{i-1}<X<K_i$ para $1<i<q$
   - $X<K_i$ para $i=1$
   - $K_{i-1}<X$ para $i=q$
4. Cada nodo contiene como mucho p apuntadores de árbol

5. Cada nodo, salvo la raíz y las hojas, tiene un mínimo de $\lceil p/2 \rceil$ punteros de árbol.
   La raíz tiene un mínimo de dos, salvo si es el único nodo del árbol.

6. Un nodo con q punteros de árbol ($q \leq p$), tiene $q-1$ valores del campo clave de búsqueda, y $q-1$ punteros de datos.

7. Todos los nodos hoja (sus punteros de árbol son todos nulos) están en el mismo nivel de profundidad.

II. Árbol B de orden p, utilizado como estructura de acceso según un campo NO clave para buscar registros de un fichero de datos

En este caso, los punteros Pr_i apuntan a un bloque (o lista enlazada de bloques) de punteros a los registros con el mismo valor en su campo de búsqueda.

Inserción en un árbol B.

- Todo árbol B comienza con un único nodo raíz (y hoja) en el nivel 0.

- Una vez completado (con p-1 valores), insertar una nueva entrada implica la partición del nodo raíz en dos nodos de nivel 1, de forma que:
  - El nodo raíz queda con el valor del medio m y dos punteros a sus hijos
  - Los valores menores que m pasan al nodo de la izquierda
  - Los valores superiores a m quedan en el nodo hijo de la derecha

- Si se completa un nodo no raíz, y se debe insertar una nueva entrada en dicho nodo, éste se divide en dos nodos de su mismo nivel.
  - El valor del medio m pasa al nodo padre, así como los punteros a sus hijos
  - Los valores menores que m pasan al nodo de la izquierda
  - Los valores superiores a m quedan en el nodo hijo de la derecha

  Si el padre también está completo, se dividirá a su vez en dos, propagándose esto hasta la raíz. Al dividirse el nodo raíz, se crea un nuevo nivel en el árbol

Eliminación en un árbol B.

- Si la eliminación de un valor hace que un nodo quede ocupado hasta menos de la mitad[14], se combina con sus nodos vecinos (el padre ocupa la posición del hijo y el siguiente nodo vecino pasa a ser el nodo padre).

(Explicación más o menos informal) Cuando se elimina una entrada de un nodo que sólo contiene esa entrada, debe eliminarse ese nodo junto con el puntero que lo apunta desde el padre; esto conlleva que el valor que lo apuntaba ha de "saltar" de tal nodo padre: para saber "dónde" hay que colocarlo hay que pensar que ya no está ahí y que hay que insertarlo (seguir algoritmo de inserción de una nueva entrada). Cuando se elimina una entrada de un nodo con más entradas, (a) si al eliminar la entrada, no existe "insuficiencia", pues 'no pasa nada', pero (b) si el nodo queda con menos entradas de la cuenta (insuficiencia), primero se echa mano de algún nodo hermano con el que fusionar el nodo actual; si no tiene hermanos, se acude al padre (el valor padre pasa a ocupar la posición del valor eliminado; si no tiene padre, se acude a los nodos hijos: el último valor del primer hijo pasa a ocupar la posición del eliminado. Cualquiera de estos cambios puede implicar reestructurar los subárboles, etc.

[14] Cada nodo interno debe tener un mínimo de $\lceil p/2 \rceil$ punteros de árbol

ÁRBOL B^+

Variación de árbol B; también empleado para implementar índices dinámicos de múltiples niveles

La principal diferencia es que en los árboles B todos los valores del campo de búsqueda aparecen en algún nivel del árbol, junto con sus punteros de datos correspondientes, mientras que en los árboles B^+, los punteros de datos aparecen sólo en los nodos hoja.

Nodo Hoja:

- Una entrada por cada valor del campo de búsqueda, junto con un puntero al registro (o a su bloque en disco) si el campo de búsqueda es clave o, si no lo es, a un bloque de apuntadores (nivel de indirección adicional).
- Todos los nodos hoja están enlazados entre sí, para permitir el acceso ordenado a los registros según el campo de búsqueda.
- Estructura de un nodo hoja:
  1. Cada nodo hoja es de la forma
     $<<K_1,Pr_1,>,<K_2,Pr_{2>},...,<K_{q-1},Pr_{q-1}>, P_{sig}>$
     donde $q \leq p$, Pr_i es un puntero de datos y P_{sig} es un puntero al siguiente nodo hoja.
  2. En cada nodo hoja se cumple que $K_1<K_2< ...<K_{q-1}$
  3. Cada Pr_i, es un puntero a datos del fichero en disco:
     · al registro cuyo valor del campo de búsqueda es K_i o
     · al bloque que contiene el registro, o
     · al bloque de punteros a los registros con valor K_i, si el campo de búsqueda no es clave
  4. Cada nodo tiene un mínimo de $\lfloor p/2 \rfloor$ valores.
  5. Todos los nodos hoja están en el mismo nivel.

Nodo Interno:

- Algunos valores del campo de búsqueda de los nodos hoja se repiten en los nodos internos, para guiar las búsquedas
- Estructura de un nodo interno:
  1. Cada nodo interno es de la forma
     $<P_1,K_1,P_2,K_2,...,P_{q-1},K_{q-1},P_q>,,$ $q \leq p$, P_i un puntero de árbol y K_i un valor de búsqueda
  2. En cada nodo interno se cumple que $K_1<K_2< ...<K_{q-1}$
  3. Para todo valor de búsqueda X en el subárbol al que apunta P_i, se cumple:
     $K_{i-1} < X \leq K_i$ para $1<i<q$
     $X \leq K_i$ para $i=1$
     $K_{i-1} < X$ para $i=q$
  4. Cada nodo contiene como mucho p apuntadores de árbol
  5. Cada nodo tiene un mínimo de $\lceil p/2 \rceil$ punteros de árbol. La raíz tiene un mínimo de dos, salvo si es el único nodo del árbol.
  6. Un nodo con q punteros de árbol ($q \leq p$), tiene $q-1$ valores del campo clave de búsqueda.
- Las entradas de los nodos internos contienen valores de búsqueda y apuntadores de árbol, pero no apuntadores de datos, por lo que caben más entradas en un nodo interno de un árbol B+ que en un nodo similar en un árbol B. Así que si el tamaño de bloque (nodo) es el mismo, el orden p de un árbol B^+ será mayor que para el árbol B correspondiente, es decir, el árbol B^+ tendrá menos niveles, con lo que se reducirá el tiempo de búsqueda.

I. Árbol B+ de orden p, utilizado como estructura de acceso según un campo clave para buscar registros de un fichero de datos

Los punteros P_i en los nodos internos son apuntadores de árbol a bloques nodos de árbol.
Los punteros Pr_i en los nodos hoja son apuntadores de datos a registros o bloques del fichero de datos. Salvo P_{sig} , que es un puntero de árbol al siguiente nodo hoja.

Gracias a este encadenamiento de los nodos hoja, es posible el acceso ordenado a los registros de datos, recorriendo los nodos de izquierda a derecha mediante los punteros P_{sig}. De esta forma, el índice puede utilizarse tanto para procesamiento secuencial de registros[15], como para el acceso directo a un registro individual (dado el valor del campo de búsqueda).

II. Árbol B+ de orden p, utilizado como estructura de acceso según un campo NO clave

En este caso, los punteros en los nodos hojas, Pr_i , son apuntadores a bloques de punteros (a los registros del fichero de datos).

Por supuesto, cada nodo de un árbol B^+ contiene la información necesaria para la implementación de los algoritmos de inserción y eliminación de valores en el árbol, es decir:

- Tipo del nodo (interno u hoja)
- Número de entradas actuales (q)
- Puntero al nodo padre y a los hermanos

---

[15] También sería posible incluir en cada nodo hoja un puntero al nodo anterior (P_{ant})

ALGUNAS CONSIDERACIONES

- En la mayor parte de los sistemas comerciales de BD, el índice no es parte del fichero sino que es una estructura de acceso independiente, que se crea y destruye dinámicamente.

- Si se va a acceder con frecuencia a un fichero de datos según condiciones de selección o búsqueda con base en los valores de determinado campo, es conveniente crear un índice sobre el mismo. Normalmente, el sistema creará un índice secundario, pues...
  - Evita la ordenación física de los registros del fichero de datos.
  - Puede crearse un índice secundario con casi cualquier organización de registros, de forma que supone un complemento de otros métodos primarios (hashing, ordenamiento...). Incluso puede usarse con archivos mixtos.

- Para crear un índice secundario implementado con un árbol B^+ es necesario...
  1. Examinar todos los registros del fichero
  2. Crear las entradas a nivel de hoja
  3. Ordenar dichas entradas y llenar los nodos hoja, creando el resto de niveles del árbol

- Es posible utilizar los índices para imponer una restricción de clave (valores únicos) sobre el campo de indexación del archivo. Cada vez que se inserta un nuevo registro en el fichero de datos, primero se busca en el índice el lugar donde debe ser introducido[16]; y al mismo tiempo puede comprobarse si existe ya un registro en el fichero con ese mismo valor del campo de indexación, en cuyo caso, se rechaza la inserción.

- Un fichero totalmente invertido es aquel que tiene un índice secundario[17] construido sobre cada uno de sus campos. Como todos los índices son secundarios, los registros nuevos pueden ser insertados al final del fichero de datos (no es necesario conservar ningún orden físico), por lo que éste puede ser un fichero no ordenado.

- Un fichero tiene una organización secuencial indexada si el fichero está ordenado por un campo clave y tiene un índice primario de múltiples niveles basado en dicha clave de ordenamiento. Esto permite el acceso a los registros del fichero tanto de forma secuencial como directa.

[16] Ver "especificación de índices en SQL", en el anexo A
[17] Implementado mediante un árbol B^+

ANEXO A: Especificación de índices en SQL estándar.

Debido a que los índices son caminos físicos de acceso a los datos y no conceptos lógicos, no se incluyeron en SQL2 las instrucciones para crear y eliminar índices.

Sin embargo, versiones anteriores de SQL (y muchos SGBD actuales) sí cuentan con dichas instrucciones de creación y eliminación de índices.

En las primeras versiones de SQL, la parte DDL (*data definition language*) no incluye cláusulas para definir restricciones de clave ni de integridad referencial en la instrucción CREATE TABLE. En particular no disponía de las cláusulas PRIMARY KEY o UNIQUE, por lo que) para especificar una restricción de clave era necesario crear un índice (llamado índice único debido a que precisamente se utiliza para asegurar la no repetición de valores de su campo de indexación).

En la versión SQL2 este tipo de restricción de integridad se impone incluyendo la cláusula PRIMARY KEY o UNIQUE en la definición del atributo[18]. Esto implica la definición automática (por parte del sistema) de un índice único sobre dicho atributo.

En la mayor parte de los SBD relacionales, un fichero corresponde a una relación (tabla) base; un índice se construye sobre una relación base, según los valores de uno de los atributos de la relación (atributo de indexación).

En los SGBD relacionales, los índices pueden crearse y eliminarse de forma dinámica, sin afectar a la relación sobre la que han sido construidos.

La existencia de un índice sobre un atributo incrementa la velocidad de las consultas en cuya condición (de selección, de reunión) se especifica un valor para tal atributo.

Por otro lado, cada índice construido para un atributo de una relación hace que las inserciones, borrados y actualizaciones de la relación consuman más tiempo y sean más complejas.

Para crear un índice sobre el atributo genero de la relación PELICULA, se utilizaría la sentencia

```
CREATE INDEX INDICE_GENERO_PELICULA
ON PELICULA(genero);
```

Es posible indicar al sistema si se desea que las entradas del índice estén en orden ascendente (ASC, opción por defecto) o descendente (DESC) según los valores del atributo de indexación.

También puede crearse un índice sobre la combinación de varios atributos,

```
CREATE INDEX INDICE_NOMBRES_DIRECTOR
ON DIRECTOR(apellido1 ASC, apellido2 ASC, nombre DESC);
```

SQL ofrece dos opciones adicionales al crear índices.

1. Para imponer una restricción de clave sobre el atributo de indexación mediante un índice, es necesario crearlo indicando que el atributo de indexación ha de contener valores únicos, utilizando la cláusula UNIQUE.

   ```
   CREATE UNIQUE INDEX INDICE_NIF_DIRECTOR
   ON DIRECTOR(nif);
   ```

   Si se intenta crear este índice sobre una tabla base cuyas tuplas no cumplan que el valor de nif sea único, el sistema no creará el índice.

   De esta forma es más eficiente imponer la unicidad de los valores de un atributo que si no existiera el índice sobre el mismo (¿Por qué? ¿Qué hará el sistema si no existe índice, para comprobar la unicidad de los valores de un atributo?).

[18] La unicidad de un atributo es un concepto lógico, ligado a la definición de la tabla donde está el atributo, y conviene separarlo de cuestiones de almacenamiento físico.

2. SQL permite crear índices de agrupamiento con la cláusula CLUSTER (grupo).

   De este modo, el sistema almacenará (en el fichero de datos) los registros ordenados según el valor del campo de indexación y construirá un índice sobre dicho atributo. Las sentencias DML que incluyen dicho atributo, en la condición de reunión o de selección, son más eficientes.

   Para conseguir que las tuplas (registros) de películas estén indexados y agrupados (físicamente en el fichero de datos) por director, crearemos este índice

```
CREATE INDEX INDICE_DIRECTOR_PELICULA
ON PELICULA(director)
CLUSTER;
```

   Una relación base puede tener

   - ✓ Uno o ningún índice de agrupamiento
   - ✓ Cualquier número de índices no de agrupamiento

Para evitar confusiones, la siguiente tabla establece la correspondencia existente entre los conceptos de SQL estándar y los estudiados en la teoría de este tema:

SQL	Teoría
índice de agrupamiento único	índice primario
índice de agrupamiento no único	índice de agrupamiento
índice no de agrupamiento	índice secundario

Para destruir un índice se utiliza la sentencia DROP; conviene eliminar un índice cuando ya no se van a realizar consultas en las que interviene un atributo indexado, y así evitar el coste del mantenimiento del índice y aprovechar el espacio que ocupa en disco.

```
DROP INDEX INDICE_DIRECTOR_PELICULA;
```

Cada SGBD implementa los índices utilizando su propia técnica; por ejemplo puede utilizar árboles B o B^+ (como Oracle), o bien utilizar estructuras de almacenamiento basadas en dispersión, etc.

Algunas notas sobre los índices en Oracle

En Oracle se recomienda crear un índice ...

- ✓ cuando la tabla contiene gran cantidad de datos (ocupa mucho espacio)

- ✓ sobre columnas que suelen aparecer...
  - en cláusulas WHERE con una comparación de igualdad o de menor o mayor que
  - en joins de varias tablas (es decir, columnas que suelen ser atributos de reunión)

  en estos casos, el SGBD NO utiliza el índice cuando la consulta...
  - no incluye WHERE,
  - contiene GROUP BY y/o DISTINCT,
  - modifica una columna indexada mediante una función (SUBSTR, || ...)

  y el SGBD SÍ puede utilizar el índice si la consulta...
  - contiene ORDER BY
  - aplica las funciones MIN y/o MAX sobre una única columna

- ✓ sobre columnas con gran variedad en sus valores, pues el índice supondrá una mayor discriminación en las búsquedas.
  Por ejemplo no convendría crear un índice sobre una columna es_valido cuyos valores sólo pueden ser (SI, NO). Sí convendría hacerlo sobre una columna cod_postal.
  En el caso de que la clave abarque más de una columna, es conveniente definir el índice poniendo las columnas en orden de mayor a menor variedad en sus valores. Y si tienen variedad parecida, poniendo primero las columnas accedidas más a menudo.

Es interesante hacer notar que en NO existe entrada en el índice Oracle para una fila que tenga un valor NULL en la columna o atributo de indexación. Si el índice está construido sobre varias columnas, no existe entrada para una tupla con alguna de estas columnas con valor NULL.

Para más información acerca de los índices en Oracle, puede acudirse a los manuales de referencia de ORACLE

ANEXO B. Clusters en Oracle

Por último, y para evitar confusiones, daremos una pequeña introducción a un método opcional para almacenar datos del que habla Oracle: la estructura de datos llamada cluster.

Un cluster es un grupo de una o más tablas almacenadas juntas físicamente, debido a que contienen información común. Los ficheros de este tipo son los que hasta ahora hemos denominado ficheros mixtos. Un ejemplo sería un fichero en el que se almacenaran registros de empleados y de los departamentos a los que éstos pertenecen.

Normalmente se construye un cluster para tablas que a) están relacionadas entre sí mediante una o más columnas comunes a dichas tablas; por ejemplo EMP y DEPT comparten el campo deptno –número de departamento, y b) con mucha frecuencia las tablas son referenciadas conjuntamente, es decir en consultas que incluyen la reunión de las mismas.

La columna o conjunto de columnas que tienen en común las tablas en un cluster, forman la clave del cluster (*cluster key*)[19]. En nuestro ejemplo deptno es la clave del cluster.

El que ambas tablas estén almacenadas en el mismo fichero incrementa la eficiencia de las consultas que incluyen la reunión de las tablas que forman el cluster, puesto que las tuplas relacionadas están físicamente adyacentes.
Otra ventaja es que, debido a que los datos en una clave del cluster (los valores del deptno) sólo se almacenan una vez para las tablas que lo forman, los clusters permiten almacenar un conjunto de tablas aprovechando más el espacio que si las tablas estuvieran almacenadas de forma individual (sin cluster)[20].

Oracle indica que se debe crear un índice sobre el cluster de forma manual, una vez creado el cluster de tablas. Lo llama índice de cluster (*cluster index*) pues está creado sobre la clave del cluster (y contiene una entrada por cada valor de dicha clave de cluster).

Para localizar cierta fila en un cluster, se utiliza el índice de cluster para encontrar el valor de la clave de cluster que apunta al bloque de datos asociado a tal valor. Por tanto, Oracle accede a una fila del cluster con un mínimo de dos operaciones de E/S (serán más de dos).

El índice cluster debe ser creado (con CREATE INDEX sobre la clave de cluster) antes de poder ejecutar cualquier sentencia DML (incluyendo INSERT y SELECT) sobre las tablas en el cluster. Por tanto, no es posible introducir datos en una tabla de cluster hasta que no se haya creado el índice cluster.

Diferencias entre un índice de tabla y un índice de cluster:

- Las claves que son NULL sí tienen una entrada en el índice cluster.
- Las entradas del índice apuntan al primer bloque en la cadena para un valor de clave de cluster dado.
- Un índice cluster contiene una entrada por valor de clave de cluster, en vez de una entrada por fila de cluster.
- La ausencia de un índice de tabla no afecta a los usuarios, pero los datos en un cluster no pueden ser accedidos a menos que exista un índice de cluster. Si se destruye un índice de cluster, los datos en el cluster permanecen, pero serán inaccesibles hasta que sea creado un nuevo índice de cluster.

Para más información acerca de los clusters en Oracle, puede acudirse a los manuales de referencia de ORACLE

[19] Para cada columna especificada (al crear el cluster) como parte de la *clave de cluster*, cada tabla (incluida en el cluster) debe tener una columna que se corresponda en tamaño y tipo con dicha columna en la clave de cluster.
[20] Véase la figura de la página siguiente.

Como ejemplo, se muestran las tablas EMP y DEPT que comparten la columna deptno (es, por tanto, la clave del cluster). En la figura siguiente aparecen las dos posibilidades: bien crear un cluster para ambas tablas, o bien almacenar cada una por separado.

En caso de crear un cluster, Oracle almacena en los mismos bloques de datos la fila de cada departamento y las de los empleados que pertenecen a éste. Obsérvese que los valores de la clave de cluster (deptno) sólo se almacenan una vez para las dos tablas.

Nota: aunque en el dibujo aparezca así, las tuplas de departamento dentro del cluster no tienen por qué estar en orden según los valores del atributo deptno; las tuplas de empleados correspondientes a cada departamento tampoco tienen por qué estar almacenadas en orden según los valores del atributo empno.