Final FOD

resumen teórico

Temas

1) Conceptos básicos de BD
2) Archivos
3) Índices
4) Árboles
5) Hashing

1) Conceptos básicos sobre Bases de Datos

¿Qué es una BD?

- Una colección de datos interrelacionados.

Un dato es cualquier información dispuesta de manera adecuada para su tratamiento con computadora. La idea de conjunto expresa que son muchos. "Interrelación" indica que los datos no son aislados, que están relacionados entre sí y conforman un todo.
Sin embargo, esa definición no expresa una característica fundamental de las BD, que es para qué esos datos se almacenan y relacionan:

Una BD es una colección de datos interrelacionados con un propósito específico, vinculado a la resolución de un problema del mundo real.

Propiedades implícitas de una BD

Una Base de datos:
- Representa algunos aspectos del *mundo real*, a veces denominado Universo de Discurso.

- Es una *colección coherente* de datos con significados inherentes. Un conjunto aleatorio de datos no puede considerarse una BD. O sea los datos deben tener cierta lógica.

- Se diseña, construye y completa de datos para un *propósito específico.* Está destinada a un grupo de usuarios concretos y tiene algunas aplicaciones preconcebidas en las cuales están interesados los usuarios

- Está sustentada físicamente en archivos en *dispositivos de almacenamiento persistente* de datos (no volátil, como un disco rígido).

Gestores de BD

Un sistema de gestión de base de datos (SGBD) consiste en un conjunto de programas que permiten acceder y administrar una BD.

Actualmente cualquier sistema de Software requiere interactuar con una BD, es por eso que no se puede separar una BD de un SGBD.

Contiene dos lenguajes:
- LDD: Lenguaje de definición de datos. Se utiliza para definir el esquema que tendrá la BD, restricciones, semántica.
- LMD: Lenguaje de manipulación de datos, mediante el cual se puede recuperar información, agregar nueva o borrar datos ya existentes. Puede ser procedimental (especifica qué datos y como obtenerlos) o no procedimental (especifica qué datos pero no como obtenerlos. Más fácil).

Objetivos principales de un SGBD

- Evitar *redundancia e inconsistencia* de datos, es decir mantener la integridad de los datos.
- Facilitar el acceso a los datos y permitirlo en todo momento.
- Controlar concurrencia, es decir accesos al mismo tiempo a la misma información, para evitar anomalías.
- Proveer seguridad para restringir accesos no autorizados.
- Suministrar almacenamiento persistente, aún ante fallos.
- Backups (copias de seguridad).

2) Archivos

¿Qué son los archivos?
- Colección de registros guardados en almacenamiento secundario
- Colección de datos almacenados en dispositivos secundarios de memoria.
- Colección de registros que abarcan entidades con un aspecto común y originadas para algún propósito particular.

Los datos que necesitan ser preservados más allá de la ejecución de un algoritmo, deben ser almacenados en archivos. Para que esto suceda, un archivo no debe estar almacenado en memoria RAM, sino en algún dispositivo de memoria secundaria.

Las estructuras de datos, por lo general, son definidas y utilizadas sobre la memoria RAM y tienen duración mientras el programa se encuentra activo. Por esta razón, cuando es requerido tener persistencia en la información que maneja un algoritmo, es necesario almacenarla en archivos en el disco

Es responsabilidad del SO encargarse de cuestiones referidas al lugar de almacenamiento del archivo en disco, cómo se ubica la información en este y cómo será recuperada.

Entonces, un archivo es una estructura de datos homogénea. Se caracterizan por el crecimiento y las modificaciones que se efectúan sobre ellos. Crecimiento se refiere a agregar nuevos elementos, mientras que modificaciones hace referencia a alterar la información ya existente o eliminarla.

Memoria y almacenamiento

Existe la memoria volátil y aquella no volátil. La información almacenada en la memoria RAM es un patrón de corriente eléctrica que corre por circuitos. Esto implica que si llegara a interrumpirse la energía, dicha información se pierde. Por eso es volátil. Existen también, formas de almacenar permanentemente la información, grabando la información sobre un

disco rígido/cinta magnética/etc.

Memoria principal es la memoria de la computadora, la RAM, donde la información se accede mucho más rápido, tiene fácil acceso y con menos costo. Por otro lado, el almacenamiento secundario (disco rígido/cintas) requiere de mucho más tiempo para acceder a los datos, porque funciona mecánicamente (un cabezal debe posicionarse en el sector correcto, etc) por lo que se debe recuperar e intercambiar datos inteligentemente. Sin embargo, aunque es más rápida, existen ciertas desventajas de la memoria RAM respecto a la memoria secundaria: es muy costosa, volátil, y limitada.

Los archivos son manejados con algoritmos y éstos, no referencian al lugar físico donde se encuentra el archivo, sino que tienen una visión lógica de éste, lo utiliza como si estuviera almacenado en memoria RAM. Por eso se debe establecer una conexión con el SO.

- Archivo físico: es el archivo propiamente dicho, aquel que reside en un dispositivo de almacenamiento secundario, y es manejado por el SO (ubicación y operaciones disponibles).

- Archivo lógico: es el archivo utilizado por el algoritmo. Cuando un algoritmo necesita operar con un archivo hace una conexión con el SO ya que éste es el responsable de la administración de archivos. Dicha acción se denomina “Independencia física”.

Organización de archivos

Los archivos pueden analizarse desde diversas perspectivas. Pueden clasificarse según la información que contienen, por la estructura de dicha información, o por la organización física del archivo en memoria secundaria.

- *Secuencia de bytes.*
  No se puede determinar con facilidad el comienzo y fin de cada dato. Ej archivos de texto.
- *Estructurados (Registros y campos):*
  Pueden ser de longitud fija o variable.
  Campo: Un campo es la unidad más pequeña, lógicamente significativa de un archivo. Con longitud predecible (o long. Fija), puede generarse desperdicio de espacio. Con longitud variable, puede existir un indicador de longitud al principio de cada campo, o delimitador al final de cada campo (carácter especial no usado como dato). El problema de esta última opción, es que aquel carácter especial es un acuerdo que debe ser conocido por todos aquellos que interactúen con la BD.

Registro: Un registro es una colección de campos agrupados, que define un elemento del archivo. Pueden ser de longitud predecible/fija (en cant. de bytes o cant. de campos), cuyos campo a su vez puede ser también fijos o variables.

También, pueden ser de longitud variable, y para ello se utilizan distintas “técnicas” para determinar el tamaño de cada dato para poder ser leído correctamente:

- indicador de longitud: al comienzo, indica la cant. de bytes que contiene, que deberán ser leídos.
- segundo archivo: mantiene la información de la dirección del byte de inicio de cada registro

- delimitador: carácter especial no usado como dato (mismo problema que en delimitador de campos).

Un archivo de longitud variable puede ser leído carácter a carácter, por lo que a veces se utilizan los marcadores de fin de campo/registro. También es posible que un registro contenga 4 campos y luego de encontrar 4 marcas de fin de campo se asuma que terminó el registro, sin necesidad de contar con un separador de registro. O utilizar indicadores de longitud, por lo que no se requieren si o si todos los separadores.

Los registros de longitud fija tienen sus ventajas, el proceso de entrada y salida de información desde y hacia los buffers es responsabilidad del SO, los procesos de alta baja y modificación son menos complejos.
Sin embargo, a veces no se utiliza todo el espacio asignado al registro, lo que provoca desperdicio. Por eso es necesario contar con alguna organización de archivos que sólo utilice el espacio que requiere, los registros de longitud variable, donde el espacio utilizado por cada registro no está determinado a priori. La ventaja es que utiliza de mejor manera el espacio en disco. Sin embargo, los algoritmos de manejo de archivos de longitud variable son más complicados.

Acceso a los archivos

El acceso a los datos de un archivo puede ser:
- Secuencial *físico:* acceso a los registros del archivo uno tras otro, en el orden físico en el cual están almacenados.
- Secuencial *indizado* (lógico): el acceso a los elementos de un archivo se hace teniendo en cuenta algún tipo de organización previa, esto implica que no se toma en cuenta el orden físico (Un índice de un libro puede ser un ejemplo de la vida real).
- Directo: Se accede a un registro determinado del archivo en un solo acceso sin necesidad de acceder a los que lo preceden, en este tipo de acceso no importa que exista un orden físico o lógico predeterminado.

Tipos de archivos

Se clasifican según dos criterios: la forma de acceso y la cantidad de cambios

Según forma de acceso

- *Serie:* cada registro es accesible luego de procesar aquel que lo antecede, es decir, aquellos cuya forma de acceso es secuencial física.
- Secuencial: los registros son accedidos en el orden de alguna clave. Forma de acceso secuencial indizada (lógica).
- Directo: los registros pueden ser accedidos directamente. Forma de acceso directo.

Según cantidad (#) de cambios

- Estáticos: Pocos cambios. Puede actualizarse en procesamiento por lotes. No necesita de estructuras adicionales para agilizar los
cambios
- Volátiles: Sometido a operaciones frecuentes (agregar, borrar, actualizar). Su organización

debe facilitar cambios rápidos. Necesita estructuras adicionales para mejorar los tiempos de acceso.

Buffers

Los buffers son memoria intermedia (ubicados en RAM) entre un archivo y un programa, donde los datos residen provisoriamente hasta ser almacenados definitivamente en memoria secundaria, o donde residan una vez recuperados de la memoria secundaria.

Los buffers se utilizan por cuestiones de performance ya que una operación sobre el disco tiene una demora mucho mayor que una operación sobre RAM. Por eso, varias operaciones de lectura/escritura directamente sobre memoria secundaria reducirían notablemente la performance del algoritmo.

Es ineficiente transferir datos muy pequeños (de a 1 byte por ejemplo), debido al costo que implica el intercambio de información con el disco (es mecánico, tarda mucho). Por esta razón, se transfieren al buffer sectores enteros, entre los cuales se encuentran los datos deseados. Entonces, hay que organizar la información de tal manera que aquellos datos del sector que se transfirieron "demás" también sean útiles.

Algoritmica clásica sobre archivos

Se dividen en dos niveles:
- Físico: lo que respecta al almacenamiento secundario.
- Lógico: dentro del programa (crear/abrir, read/ write, EOF, seek)

Concepto de archivo maestro y archivo detalle:

➔ Maestro: archivo que resume información sobre un dominio de problema específico.

➔ Detalle: archivo que contiene movimientos realizados sobre la información almacenada en el maestro.

La algorítmica clásica sobre archivos se resumen en tres tipos: actualización, merge y corte de control.

- La actualización se dá entre archivo maestro con archivo detalle, tiene su primer variante en el caso donde exista un solo archivo detalle o el caso donde existan N archivos detalle donde N > 1.

- El merge consiste en la generación de un archivo maestro (que no existe) el cuál tiene el resumen de la información de uno o varios archivos existentes.
- El corte de control es el proceso mediante el cuál la información de un archivo se presenta de forma organizada con un formato especial.

Viaje del byte

- Hace referencia a el proceso que implica el intercambio de datos con algún dispositivo dealmacenamiento secundario.

Existen 2 archivos paralelamente:

-*Archivo físico:* es aquel que existe en el almacenamiento secundario. Es el archivo tal y como lo conoce el SO, y se encuentra en su directorio de archivos.

- *Archivo lógico:* Es el archivo visto por el programa. Permite a un programa describir las operaciones a efectuarse en un archivo. No se sabe cual archivo físico real se utiliza o donde esta ubicado.

El viaje del byte no es sencillo, abarca ciertos ciclos para escribir.

¿Quiénes están involucrados cuando se escribe un dato en un archivo?

- *Administrador de archivos :* conjunto de programas del S.O. (capas de procedimientos) que tratan aspectos relacionados con archivos y dispositivos de E/S. Las capas superiores se encargan de aspectos lógicos de datos (tabla), establecer si las características del archivo son compatibles con la operación deseada. En Capas Inferiores se tratan aspectos físicos (FAT), determinar donde se guarda el dato (cilíndro, superficie, sector). Si el sector está ubicado en RAM se utiliza, caso contrario debe traerse previamente.

- *Buffer de E/S:* agilizan la E/S de datos. Manejar buffers implica trabajar con grandes grupos de datos en RAM , para reducir el acceso a almacenamiento secundario.

- *Procesador de E/S:* dispositivo utilizado para la transmisión desde o hacia almacenamiento externo. Independiente de la CPU.

- *Controlador de disco (driver):* encargado de controlar la operación de disco: colocarse en la pista, colocarse en el sector, transferencia a disco.

Hay ciertas capas (pasos) de protocolo de transmisión de un byte que se llevan a cabo:

- El Programa pide al S.O. escribir el contenido de una variable en un archivo. Pasa el control al SO.
- El S.O. transfiere el trabajo al Administrador de archivos
- El Adm. busca el archivo en su tabla de archivos y verifica las características, deben ser compatibles.
- El Adm. obtiene de la FAT la ubicación física del sector, del archivo donde se guardará el byte.

Claves

Un registro es concebido como una cantidad de información que se lee o escribe. Por esta razón, es necesario extraer sólo un registro específico en vez del archivo completo. Para ello, es conveniente identificar cada registro con una llave o clave que se base en el contenido del mismo. Una clave, entonces, permite la identificación del registro, y debe permitir generar orden en el archivo por ese criterio
Una clave puede ser:

- Únivoca / Primaria: Identifica un elemento particular dentro de un archivo. El valor que tome dicha clave pertenece a un y sólo un elemento.
- Secundaria: reconocen un conjunto de elementos con igual valor, es decir, esa clave puede repetirse.

Forma canónica es una forma estándar para una clave, puede
derivarse a partir de reglas bien definidas.
- Representación única para la llave, ajustada a la regla. Ej: llave sólo con letras mayúsculas y sin espacios al final.
- Al introducir un registro nuevo. 1ro se forma una llave canónica para ese registro, 2do se la busca en el archivo. Si ya existe, y es univoca no se puede ingresar.

Osea: suponiendo que la clave es CIA (LON, RCA, WAR) + numero de ID. Una forma canónica sería CIA en mayúscula + n.º de ID.
Ver ejemplo de diapositiva, clase 5.

Performance de claves

El estudio de performance tiene en cuenta 3 puntos:
• Punto de partida para futuras evaluaciones
• Costo: acceso a disco, numero de comparaciones
• Caso promedio

- En el *caso secuencial:* el mejor caso es leer 1 registro, y el peor caso, leer n registros. El promedio es de n/2 comparaciones, la mitad de la cantidad de registros. Es de O(n), porque depende de la cantidad de registros.
Con la lectura de bloques de registros se mejora el acceso a disco, pero no varían las comparaciones.

- En el caso de acceso directo: Permite acceder a un registro preciso. Requiere una sola lectura para traer el dato, es de O(1). Debe necesariamente conocerse el lugar donde comienza el registro requerido.

Número relativo de registro *(NRR):* Indica la posición relativa con respecto al principio del archivo. Solo aplicable con registros de longitud fija. Ej. NRR 546 y longitud de cada registro 128 bytes, para leer ese determinado registro basta con calcular la distancia en bytes= 546 * 128= 69.888.

El acceso directo es preferible sólo cuando se necesitan pocos registros específicos, pero este método NO siempre es el más apropiado para la extracción de información, aunque a simple vista lo pareciera.
Por ejemplo, un caso en el que se quieran generar cheques de pago a partir de un archivo de registros de empleados. Como todos los registros se deben procesar, es más rápido y sencillo leer registro a registro desde el principio hasta el final, y no calcular la posición en cada caso para acceder directamente.

Operaciones

Altas, bajas, modificaciones, consultas.

Bajas (eliminación)

Cuando se quiere dar de baja un registro de un archivo, existen dos formas posibles: baja lógica y física.
La baja lógica consiste en borrar la información del archivo pero sin recuperar el espacio físico. No eliminar el registro de manera permanente, físicamente, sino considerarlo como eliminado. Existen estrategias para reconocerlos una vez eliminados, como la colocación de una marca de borrado. La ventaja es que con este criterio se puede anular la eliminación fácilmente, y cuestiones referidas a la performance. Sólo hay que realizar lecturas hasta encontrar el registro que se desea eliminar y, una vez encontrado, se realiza sólo 1 escritura para colocar la marca de no disponible. La desventaja es que no se recupera espacio en disco, y el archivo tiende a crecer continuamente, a menos que se utilicen técnicas para reutilizar/recuperar ese espacio.

Por otro lado, la baja física consiste en borrar aquellos registros de forma permanente. La ventaja es que el lugar de aquellos registros considerados borrados desaparece, el archivo se vuelve más pequeño y se cuenta con más espacio para utilizar. Siempre se administra un archivo de datos que ocupa el menor espacio posible. La desventaja tiene que ver con la performance de los algoritmos que implementan este tipo de solución. Este tipo de técnicas son muy costosas. La frecuencia con la que se realiza depende del dominio de aplicación y la necesidad de espacio.

Hay dos técnicas para realizar bajas físicas:

- La forma más simple es *copiar todo en un archivo nuevo* excepto los registros con marca de borrado. Una vez finalizado el proceso, se debe eliminar el archivo inicial del dispositivo de memoria secundaria, y renombrar al generado con el nombre original. Antes de eliminar el archivo original, se tienen en el disco 2 archivos, por lo que se debe contar con memoria suficiente para poder realizar esta técnica.
- Utilizar el mismo archivo de datos *realizando los corrimientos* necesarios. No se requiere mucha memoria, sin embargo en el peor de los casos se deben hacer n-1 escrituras, en el

caso que el elemento a eliminar sea el 1.

Tanto las bajas lógicas como físicas pueden ser realizadas en archivos de longitud fija como variable.

Aprovechamiento de espacio (altas)

El proceso de baja lógica marca cierta información del archivo como borrada. Sin embargo, sigue ocupando espacio en el disco y ¿qué hacer con esa información?

Existe una diferencia entre recuperar y reasignar:
Recuperar espacio: realizar periódicamente el proceso de baja física para compactar el archivo, quitando todos esos archivos que tienen marcas de borrado.
Reutilizar/reasignar espacio: utilizar los lugares marcados como borrados para hacer nuevas altas.

Cuando se quiere recuperar/reutilizar espacio borrado, aparece un problema conocido como ***FRAGMENTACIÓN***, que puede ser de dos tipos: interna o externa.

- Fragmentación interna: Aquella que se produce cuando a un elemento se le asigna más espacio del que requiere realmente. Cuando se desperdicia lugar en un registro, es decir, se le asigna lugar y no lo ocupa totalmente. En general sucede con registros de longitud fija, los registros de longitud variable tienden a evitar este problema. Una solución es que ese residuo, una vez ocupado el espacio libre, pase a ser otro espacio libre. El problema es cuando es demasiado pequeño.

- Fragmentación externa: Espacio libre entre dos registros, pero que es demasiado pequeño para almacenar otro. El espacio que sobra cuando se ocupa un lugar libre pasa a ser un nuevo espacio libre. El problema es cuando es tan pequeño que no se puede utilizar. Posibles soluciones serían unir espacios pequeños adyacentes o elegir los espacios que mejor se adecuen a cada caso.

F. interna → se asigna espacio que no se utiliza
F. externa → quedan espacios que son tan pequeños que no se pueden utilizar

Aprovechamiento en registros de longitud fija

Para poder recuperar espacio en registros de longitud fija, es necesario garantizar marcas especiales (de borrado) en los registros eliminados, y bajas lógicas (mecanismos para "ignorar" registros eliminados).
¿Cómo saber dónde existe espacio libre donde colocar el nuevo registro? Es decir, un espacio donde existe un registro eliminado.

- *Búsqueda* secuencial: usa las marcas de borrado. Para agregar un nuevo registro, recorre el archivo buscando una marca, es decir, el primer registro eliminado. Si no existe, se llega al final del archivo y se agrega allí. La desventaja es que es muy lento para operaciones frecuentes.

Por esta razón, es necesario contar con una forma de saber de inmediato si hay lugares vacíos en el archivo, sin necesidad de recorrerlo completamente. Una forma de saltar directamente a uno de esos lugares, en caso de existir. Recuperar espacio de forma eficiente.

- Lista *o pilas (header):* Se trata de una lista encadenada de registros disponibles. Al insertar un registro nuevo en un archivo de registros de longitud fija, cualquier registro disponible es bueno, debido a que todos los registros son de longitud fija y todos los espacios libres son iguales. Por esta razón, la lista NO necesita tener un orden particular.
Se trata de una lista invertida, cuando un registro se elimina, pasa a ser el header de la lista de disponibles. Por ejemplo:
1. En el encabezado de la lista estará NRR 4, el archivo tendrá:
   alfa beta delta * 6 gamma * -1 epsilon
2. Se borra beta, como inicial quedará 2, y el siguiente será 4.
   alfa * 4 delta * 6 gamma * -1 epsilon
3. Si se quiere agregar un elemento el programa solo debe chequear el header y desde ahí obtiene la dirección del primer registro disponible. Agrego omega (en el NRR 2) , como header queda 4 nuevamente.
   alfa omega delta * 6 gamma * -1 epsilon

Aprovechamiento en registros de longitud variable

En registros de longitud variable, el problema reside en que cada registro posee su propio tamaño y, por lo tanto, los lugares disponibles también. Por esta razón, los nuevos elementos no pueden colocarse en cualquier lugar disponible, deben "caber" necesariamente.
Para indicar lugares disponibles, se utiliza también una lista invertida donde a partir del registro cabecera se disponen las direcciones libres dentro del archivo. Además, ahora es necesario indicar también la cantidad de bytes disponibles en cada caso.

Para colocar un nuevo elemento al archivo, se busca un registro borrado de tamaño adecuado (lo suficientemente grande). Como se necesita buscar, no se puede organizar la lista de disponibles como una pila. Si se toma el primer registro borrado de tamaño adecuado, puede generar fragmentación, debido a que quizás es demasiado grande y se desperdicia espacio.
Para *minimizar la fragmentación*, existen estrategias de colocación, que permiten elegir el espacio más adecuado en cada caso. Son tres, y cada una tiene ventajas y desventajas.

- *Primer ajuste:* se selecciona la primer entrada de la lista de disponibles que sea capaz de almacenar al nuevo registro, y se le asigna al mismo. La ventaja es que minimiza la búsqueda, es más rápido y menos costoso. Sin embargo, no se preocupa por la exactitud del ajuste y tiende a generar fragmentación interna.

- *Mejor ajuste:* elige la entrada que más se aproxime al tamaño del nuevo registro y se le asigna completa. La ventaja es que reduce la fragmentación y aprovecha mejor el espacio. Sin embargo, exige búsqueda (procesar todos los disponibles). Puede generar fragmentación interna.

- *Peor ajuste:* selecciona la entrada más grande para el registro, y se le asigna solo el espacio necesario, el resto queda libre para otro registro. También exige búsqueda. Puede generar fragmentación externa.
En cuanto a las tres técnicas, la de primer ajuste es la más rápida, debido a que recorre la lista de libres hasta encontrar uno que quepa, lo asigna completo, y lo elimina de la lista de libres. Por otro lado, mejor y peor ajuste, son más ineficientes debido a que deben recorrer toda la lista buscando el mejor lugar según el caso. Una vez encontrado el lugar, mejor ajuste resuelve con mayor rapidez debido a que asigna todo el lugar completo y lo elimina de la lista de libres. Peor ajuste además de encontrarlo, debe asignar sólo lo que requiere y el espacio restante ponerlo en la lista de libres. Sin embargo, es el que más coincide con la filosofía de longitud variable, que cada registro ocupe sólo lo que necesita.

Modificación de registros

Cuando los registros de longitud fija se modifican, no existe ningún problema, porque la modificación consiste en sobreescribir el registro con el nuevo dato.
El problema surge cuando quiere modificarse un registro de longitud variable, debido a que puede significar que el nuevo registro necesite el mismo espacio, ocupe menos espacio, o requiera más espacio del que utilizaba antes. Si requieren el mismo espacio, no se genera ningún problema. Si el nuevo registro ahora ocupa menos espacio, tampoco se generaría algún problema además de generar fragmentación interna, debido a que el espacio sería capaz de almacenar el registro. El verdadero problema surge cuando el nuevo registro requiere un espacio mayor. Por esta razón, **el proceso de modificación en registros de longitud variable suele dividirse en dos etapas. Primero se da de baja el elemento viejo, y luego de modificarse, el registro nuevo es insertado según la política de ajuste que se maneje.**

Búsqueda de información

Cada búsqueda tiene un costo, que se basa en la cantidad de *accesos* a disco que se realizan en función de encontrar la información deseada y, en cada acceso, la cantidad (#) de *comparaciones*, es decir verificaciones de si el dato obtenido es el que se estaba buscando.

Los accesos son operaciones en memoria secundaria, y tienen un costo alto. En cambio, las comparaciones son operaciones en memoria, que pueden mejorarse con algoritmos más eficientes, y su costo es mucho menor.

Existen distintas formas de realizar una búsqueda. Puede ser secuencial, en la cual el registro deseado debe buscarse desde el comienzo del archivo; directa, utilizando el NRR, o binaria/dicotomica, incorporando el uso de llaves/claves.

Búsqueda secuencial se trata de acceder a un registro luego de todos sus predecesores, hasta llegar al dato deseado. En el peor caso se accede a todos los registros del archivo.
Es posible encontrar un registro en 1 sólo acceso a través del NRR, debido a que es posible calcular la distancia en bytes desde el principio del archivo hasta cualquier registro. Sin embargo, es muy poco probable que se conozca el NRR del registro que contiene el dato a

buscar.

Búsqueda binaria

Si se tiene un archivo de **longitud fija físicamente ordenado**, se puede mejorar la performance si se realiza la búsqueda bajo el mismo criterio que el orden del archivo. La búsqueda binaria consiste en partir el archivo a la mitad y comparar esa clave con aquella que se está buscando, para determinar en qué mitad está, y así sucesivamente hasta dar con el registro deseado.

Para poder realizar una búsqueda binaria, el archivo debe cumplir dos condiciones:

- Debe estar físicamente ordenado (para comparar sus claves).
- Ser de longitud fija (para poder acceder a la mitad).

Es de O(log2 N) considerando N como la cantidad de registros del archivo. Mejora la performance de la búsqueda secuencial.

La ventaja es que reduce el tiempo para encontrar información. La desventaja es que se debe garantizar que el archivo esté ordenado para poder llevarse a cabo, y es costoso.

Clasificación **(ordenación)**

¿Cómo ordenar un archivo? Puede ser:

- Si el archivo cabe en memoria RAM, se lleva allí para poder ordenarse. Esto implicaría leer los datos de memoria secundaria, pero sería secuencialmente, y como no requiere muchos desplazamientos en el disco (cabezal, etc) no requiere un costo excesivo. Una vez en memoria RAM se ordena con una performance alta, y vuelve a escribirse el archivo ordenado en memoria secundaria. Esta opción constituye la mayor performance para ordenar archivos físicamente, pero sólo sirve para archivos pequeños.

- Si el archivo no entra en memoria RAM, otra opción es transferir a ella sólo las claves por las que se desea ordenar el archivo, junto con los NRR, desde la memoria secundaria. De esta manera, al transferir sólo esos datos, es posible almacenar una mayor cantidad de registros en RAM y, además, ¿para qué llevar a ram todo el registro completo si lo que importa para ordenar el archivo es la clave? Sin embargo, luego debe leerse el archivo completo según el orden establecido para poder escribirse nuevamente en disco. Es un costo muy alto. Además, ¿Qué sucede cuando las *claves no entran en RAM?*

Una alternativa posible a este problema es partir el archivo, ordenar cada parte y juntar las partes ordenadas (merge).

Entonces…
La búsqueda binaria mejora la secuencial, pero requiere estar ordenada, con el costo que eso implica. Además, el ordenamiento en RAM es sólo para archivos pequeños.
Algunas formas de solucionar estos problemas, y mejorar el método de búsqueda son:

- No reordenar todo el archivo, sino sólo sus claves.
- Reorganizar con métodos más eficientes, como árboles o hashing.

3) Índices

Entonces, el propósito de ordenar un archivo radica en tratar de disminuir los accesos a memoria secundaria durante la búsqueda.

Hay distintos métodos de búsqueda:
- Secuencial (poco eficiente)
- Binaria (costosa)
- *Utilizando estructuras auxiliares.*

Una estructura auxiliar puede verse como un índice de un libro, que es una estructura ajena al contenido del libro, y de tamaño considerablemente menor, donde uno consulta directamente a qué página ir para encontrar la información que necesita.

Por ejemplo:
(tema, numero de hoja) → (clave, NRR/dirección del 1 byte del registro)

¿Qué son los índices?

- Herramienta para encontrar registros en un archivo. Consiste en un campo de llave (clave. búsqueda) y un campo de referencia que indica donde encontrar el registro dentro del archivo de datos.

- Tabla que opera con un procedimiento que acepta información acerca de ciertos valores de atributos como entrada (llave), y provee como salida, información que permite la rápida localización del registro con esos atributos.

- Estructura de datos con formato (clave, dirección) usada para decrementar el tiempo de acceso a unarchivo.

En definitiva:

Un índice es una estructura de datos adicional, que permite agilizar el acceso a la información almacenada en un archivo. Es **otro archivo, con registros de longitud fija**, independientemente de qué estructura tenga el archivo de datos. La característica principal de un índice es que permite imponer orden a un archivo, sin que este sea reacomodado.

La estructura de un índice puede variar, la más simple es un árbol.

En general, un índice puede mantenerse en memoria, y es más fácil de manejar que el archivo de datos, la búsqueda binaria es más rápida.

¿Cómo implementar índices?

Operaciones básicas en un archivo indizado:
- *Crear los archivos:* el índice y el archivo de datos se crean vacíos, solo con registro cabecera.

- *Cargar el índice en memoria:* se supone que cabe, ya que es lo suficientemente pequeño.

- *Reescritura del archivo de índice:* cambios reescribir.

- *Agregar nuevos registros:* Implica agregar nueva información al archivo de datos y al archivo de índices. Respecto al archivo de datos, el nuevo registro se copia al final del archivo, y es necesario saber el NRR (si es de longitud fija) o la distancia en bytes (si es variable) para poder utilizarlo en el índice. Para el índice, se ordena con cada nuevo elemento en forma canónica, en memoria, y se setea el flag anterior.

- *Eliminar un registro:* Para el archivo de datos, cualquier técnica de las vistas para reutilizar el espacio. Respecto al archivo de índices, no tiene sentido recuperar ese espacio borrado con otro registro nuevo, debido a que el índice tiene un orden y no debe alterarse. Se quita la entrada (o se podría marcar como borrado).

- *Actualización de registros:*
    - *Sin modificar la clave* ¿Qué pasa con el índice? Si es de longitud fija no hay que hacer más actividad. Si se trata de longitud variable se requiere hacer algunas modificaciones. Si el registro no cambia de longitud, se almacena en la misma posición física, el índice "no se toca". Si el registro cambia de longitud (se agranda) y se reubica en el archivo de datos, se debe guardar la nueva posición de inicio en el índice.

    - *Modificando la clave* ¿Qué sucede? Cuando se modifica el archivo de datos, también se debe actualizar y reorganizar el archivo de índices. Cómo simplificar → Modificar = Eliminar + Agregar (ya vistos)

Ventajas y desventajas

- *Ventajas:* Al ser de menor tamaño que el archivo de datos asociado, y ser de longitud fija, mejora la performance de búsqueda. Podrían caber en RAM. Siempre permite búsqueda binaria, porque al ser más pequeño el mantenimiento del orden es menos costoso y **siempre son de longitud fija**.

- *Desventajas:* ¿Qué sucede cuando no entra en RAM?

Persistencia de datos

Cuando los índices son demasiado grandes para entrar en memoria RAM, existen algunas soluciones posibles: árboles y dispersión (hashing)

Índices secundarios

No sería natural solicitar un dato por clave, por ejemplo pedir una canción por su ID ¿cómo podría saberse cuál es el ID de una canción? En su lugar, es más intuitivo solicitar la información a través un campo mas fácil de recordar (buscar una canción por su título o por su compositor). Este campo, puede repetirse por lo que no puede formar parte de la clave primaria. Se denomina clave secundaria.

El índice secundario relaciona una clave secundaria, con <u>una o más</u> claves primarias.

! Es decir, se cuenta con muchas claves primarias, que son unívocas, para identificar registros en particular. Se tiene, además, un archivo índice, que almacena esas claves primarias y dónde ir en el archivo original para encontrar el elemento que identifican. Por otro lado, se cuenta con una clave secundaria, que puede repetirse, pero es más fácil y natural recordarla. También, se cuenta con un índice secundario, que relaciona una clave secundaria con una o más claves primarias. Por ejemplo, del artista Beatles (clave secundaria) se tienen las canciones con ID 222, 333, 444 (claves primarias, que identifican a un registro en particular)

Acceso: primero por clave secundaria se accede al índice secundario, y se obtiene la clave primaria. Luego, con esa clave primaria se accede la índice primario y se obtiene la dirección del dato que se está buscando.

¿Por qué se accede de esta manera, en vez de que el índice secundario directamente tenga la dirección física? Primero, porque un archivo puede tener varios índices secundarios, y si el registro llegara a cambiar de ubicación en el archivo, habría que actualizar todos los índices secundarios. Por esta razón, sólo el índice primario tiene definida la dirección física del registro, y en caso de ser necesario, sólo debería cambiar el índice primario.
Segundo, por lo que sigue a continuación…

<u>Creación</u> de índice secundario

Al implementarse el archivo de datos, se deben crear todos los índices secundarios asociados, vacíos al principio. Similar a índices primarios.

<u>Altas</u> en índices secundarios

Cualquier alta en el archivo de datos implicaría una inserción. Esta operación es de bajo costo si el índice puede almacenarse en memoria principal, pero costosa si se encuentra en memoria secundaria.

<u>Bajas</u> en índices secundarios
Eliminar un registro del archivo de datos implica eliminar también su referencia en el índice primario y en todos los índices secundarios que tenga. También podría borrarse sólo del índice primario, como si fuera una barrera, pero en los índices secundarios se tendrían datos que ya no existen.

<u>Problema</u> de los índices secundarios

El principal problema de los índices secundarios es la *repetición de información.*

Por ejemplo, en el archivo de índices tendríamos

Beatles	222
Beatles	333
Beatles	444
Madona	555

De esta manera, el archivo de índices se debe reacomodar con cada adición, aunque se ingrese una clave secundaria ya existente (ejemplo Beatles), dado que existe un segundo orden por la clave primaria (el id de la canción).
La misma clave tiene varias ocurrencias, en distintos registros. Esto genera que se desperdicie espacio. Reduce la posibilidad de que el índice quepa en memoria.

Una primera solución a este problema implicaría almacenar en un registro todas las ocurrencias de una clave secundaria. Cada registro está formado por la clave secundaria + vector de claves primarias correspondientes.

Por ejemplo: Beatles 222 333 444

Al agregar un nuevo registro de una clave existente simplemente se debe insertar de forma ordenada la clave primaria en el vector de ocurrencias correspondiente. Al agregar un nuevo registro con una clave nueva, se genera un arreglo con la clave y un elemento en el vector de punteros

El problema que presenta esa solución es la elección del tamaño del vector, ya que conviene que sea de longitud fija. Al decidir previamente el tamaño del vector, puede haber casos en los que sea insuficiente, o puede haber casos en los que sobre espacio, provocando fragmentación interna

Mejora: clave secundaria + lista de claves primarias asociadas

Listas invertidas

Cuando en un archivo una clave secundaria lleva a un conjunto de una o más claves primarias, se puede utilizar una lista de referencias de claves primarias. No se desperdicia espacio, porque no debe realizarse una reserva de éste previamente, y pueden existir cualquier número de claves primarias para cada clave secundaria.
Si se agrega un elemento a la lista, no es necesaria una reorganización completa, debido a la naturaleza de las listas invertidas.
Primero se accede al archivo de claves secundarias y se obtiene la dirección de acceso a la lista de claves primarias. Dicha lista, se encuentra almacenada en otro archivo.

- Operaciones: Agregar un nuevo consiste en agregar concurrencias en la lista invertida. Lo mismo sucede con la operación borrar. Las modificaciones dependen del caso.

NRR	Archivo de índice secundario	
0	BEETHOVEN	3
1	COREA	2
2	DVORAK	7
3	PROKOFIEV	10
4	RIMSKY-KORSAKOV	6
5	SPRINGSTEEN	4
6	SWET HONEY IN...	9

NRR	Arch de listas de llaves primarias	
0	LON2312	-1
1	RCA2626	-1
2	WAR23699	-1
3	ANG3795	8
4	COL38358	-1
5	DG18807	1
6	MER76016	-1
7	COL31809	-1

NRR	Archivo de índice secundario	
0	BEETHOVEN	3
1	COREA	2
2	DVORAK	7
3	PROKOFIEV	10
4	RIMSKY-KORSAKOV	6
5	SPRINGSTEEN	4
6	SWET HONEY IN...	9

NRR	Arch de listas de llaves primarias	
0	LON2312	-1
1	RCA2626	-1
2	WAR23699	-1
3	ANG3795	8
4	COL38358	-1
5	DG18807	1
6	MER76016	-1
7	COL31809	-1
8	DG139201	5
9	FF245	-1
10	ANG36193	0

Ventajas y desventajas de índice secundario implementado con listas de llaves primarias

Ventajas: El único reacomodamiento en el archivo índice se produce cuando se agrega una nueva clave secundaria. Si se agrega o borran elementos de una clave secundaria ya existente, sólo se debe modificar el archivo que contiene la lista. Como se generan 2 archivos, uno de ellos podría residir en memoria secundaria, liberando la RAM.

Desventajas: el archivo de listas es conveniente que esté en memoria principal, debido a que podría haber muchos desplazamientos en disco. Costoso si hay muchos índices secundarios.

4) Árboles

El principal problema para administrar un índice en disco rígido, lo representa la cantidad de accesos necesaria para recuperar la información. El proceso de búsqueda de información es muy costoso, sumado al mantenimiento de la información ante operaciones de altas, bajar y modificaciones.
Un árbol es una estructura de datos que permite localizar en forma más rápida, información de un archivo. Tiene intrínsecamente búsqueda binaria (en el interior, propio, internamente).

Árboles binarios

¿Qué es un árbol binario?
Se denomina árbol binario a una estructura de datos dinámica no lineal, donde cada nodo tiene a lo sumo dos sucesores: a izquierda y a derecha.
Son la alternativa más simple para resolver el problema anteriormente planteado. En general tiene sentido cuando está ordenado.

Búsqueda

La búsqueda siempre parte desde la raíz y continúa hacia la izquierda o derecha según el elemento que se desee encontrar. De esta manera, en cada comparación se descarta la mitad del archivo restante, donde es seguro que el elemento no se encuentra. Tiene orden Log2(N) accesos a disco.

En general los árboles como estructura se implementan en memoria RAM. Sin embargo, para que pueda ser utilizado como una estructura índice de búsqueda se requiere persistencia de la información y, por lo tanto, deben implementarse sobre almacenamiento secundario. Un nodo, además de contener el elemento propiamente dicho, en los hijos izq y der no tiene punteros como siempre, tiene un número que hace referencia al NRR.

Performance

Anteriormente se vio que la búsqueda implementada con un archivo índices ordenado por clave, también es de orden logarítmico. Sin embargo, la ventaja de la organización en árboles está dada en la inserción de nuevos elementos.

Mientras que un archivo se ordena cada vez que ingresa un nuevo dato, con árboles esta operación es mucho más sencilla en términos de complejidad. Se debe agregar un nuevo elemento, buscar al padre, y actualizar al padre haciendo referencia al nuevo hijo: serían Log2(n) lecturas a disco para encontrar el padre del nuevo elemento y 2 escrituras (nuevo elemento y actualización del padre). Mientras que, de la otra manera, debería reordenarse todo el archivo.

Con respecto a las bajas, el análisis es el mismo. Para borrar un elemento, éste debe ser terminal y, de otra manera, se reemplaza por el menor de sus hijos mayores. Nuevamente serían Log2(N) lecturas y 2 escrituras.
En conclusión, la organización utilizando estructuras de árbol presenta *ventajas en cuanto a la performance de operaciones de inserción y bajas,* respecto a la estructura de un archivo ordenado por claves. En cuanto a la búsqueda, tienen un comportamiento similar.

*Si se requieren ordenar por dni, por legajo, por matricula y por cuit; se deben tener 4 árboles distintos ordenados por cada criterio. Sólo contienen ese dato, no todo el elemento.

Problema con árboles Binarios

Un árbol está balanceado cuando la altura de la trayectoria más corta hacia una hoja no difiere de la altura de la trayectoria más grande, es decir, todas las hojas se encuentran a la misma distancia del nodo raíz.

El problema de los árboles binarios es que se *desbalancean* fácilmente ¿Y cuál es el problema de que suceda esto? Que la performance de búsqueda ya no puede considerarse de orden logarítmico.

En el peor de los casos, él desbalanceo provoca que el árbol se transforma en una estructura de tipo lista,la performance de búsqueda decae transformándose en orden lineal.
En qué caso un árbol binario puede volverse una estructura de tipo lineal?
-> cuando se insertan elementos consecutivos, ejemplo: (5,7,8,10,11,12).

En conclusión, un árbol binario es una buena opción para implementar un índice, cuando está balanceado. Que el árbol permanezca balanceado depende de una correcta elección de la raíz. Sin embargo, cuando se genera un archivo que implementa un índice de búsqueda, es imposible saber con anterioridad cuál será la mejor raíz, debido a que los elementos van insertándose.

Árboles AVL

Un árbol AVL, es un árbol binario balanceado en altura (BA(1)).
Los árboles balanceados en altura tienen la característica que su construcción se determina según la siguiente regla: para cualquier nodo, la diferencia máxima entre las alturas de sus dos subarboles, derecho e izquierdo, no pude superar determinado delta (límite), donde ese delta es el nivel de balanceo en altura del árbol. Por ejemplo, si el árbol es BA(1) significa que la diferencia no puede superar 1, es el máximo desbalanceo posible.
Los árboles AVL son balanceados en altura, con delta 1 (BA(1))

Problema con AVL

El problema con los árboles AVL es que puede suceder que al insertar algún elemento, se infrinja la regla establecida. En ese caso, habría que rebalancear el árbol con algoritmos de rotación, y los costos de acceso a disco aumentan.

Entonces, se tiene una estructura que mantiene el balanceo acotado, pero con mayores costos en las operaciones de inserción y borrado. De esta manera, ni los áboles binarios ni los AVL representan soluciones viables para los índices de un archivo de datos.

Árboles binarios paginados

Como se vio antes en el concepto de buffering, cuando se transmite desde o hacia el disco rígido, la transferencia no se limita sólo a un registro sino que son enviados un conjunto de ellos, los que quepan en el buffer. Esto representa una mejora en la performance, ya que no requiere tantos accesos a disco.
Este concepto es importante cuando se genera el archivo que contiene al árbol binario. Dicho árbol se divide en páginas (se pagina), y cada una de ellas contiene un conjunto de nodos, que están ubicados en direcciones físicas cercanas.
De esta manera, cuando se accede a disco no se trasfieren una determinada cantidad de bytes, sino que se transmite una página completa.

Una organización de este tipo reduce notablemente la cantidad de accesos a disco. Como en cada página caben una cierta cantidad de nodos, en pocos accesos podría encontrarse el elemento buscado. Para transferir una página completa, si bien se accede a direcciones físicas diferentes, éstas son cercanas y los desplazamientos son mínimos y no implican un costo.

Performance

La performance de un árbol binario paginado depende de la cantidad de nodos que entren cada página. Si la página contiene 255 nodos, la performance de búsqueda sería de log256(N), es decir Logk+1(N), donde N es la cantidad total de elementos del árbol y k la cantidad de nodos que entran en una página.
Problema con binario paginado

El problema reside en cómo generar un árbol binario paginado. Para que una organización en páginas tenga sentido, todos los elementos de una página deberían llegar (insertarse en el árbol) consecutivamente para almacenarse en el mismo sector del disco. Esto es imposible, porque no se puede condicionar la llegada de elementos al archivo de datos. De esta manera, cada página debería quedar incompleta hasta que llegue un elemento que se relacione a ella, en vez de entrar en el primero disponible.
En este caso, habría que tener en cuenta las páginas, los elementos que caben en cada una de ellas, la forma en la que se construye un árbol binario y, además, cuestiones de balanceo. Un algoritmo de estas características sería muy costoso de implementar, y también en cuanto a la performance. Los árboles binario paginados traen un costo mucho mayor que el beneficio.

Árboles multicamino

Es una **generalización de árboles binarios, una estructura de datos en la que cada nodo puede tener k elementos y k+1 hijos**. Disminuye la profundidad del árbol.
Son n-arios, es decir, tienen varios hijos. Por esta razón, tienen poca altura (son más chatos) y requieren menos accesos para encontrar algo.
El concepto de **orden** en un árbol multicamino, hace referencia a la máxima cantidad de descendientes que puede tener un nodo.

Un árbol multicamino presenta otra forma de resolver el concepto de paginación. En este caso, el orden del árbol dependerá del tamaño de la página. Sin embargo, todavía existe el problema del balanceo.

Árboles Balanceados (B)

Estructuras NO lineales. Son árboles multicamino con una construcción especial en forma ascendente, que permite mantenerlo balanceado a bajo costo. Es decir, gracias a su construcción ascendente, siempre se mantienen balanceados.

Propiedades de un árbol B de orden M:

- Ningún nodo tiene más de M hijos.
- Los nodos internos tienen como mínimo [M/2] elementos (parte entera)
- Los nodos hoja tienen como mínimo [M/2] -1 elementos.
- La raíz tiene como mínimo 2 hijos, o sino ninguno.
- Todas las hojas están a igual nivel.
- Los nodos no terminales con K hijos contienen K-1 registros.

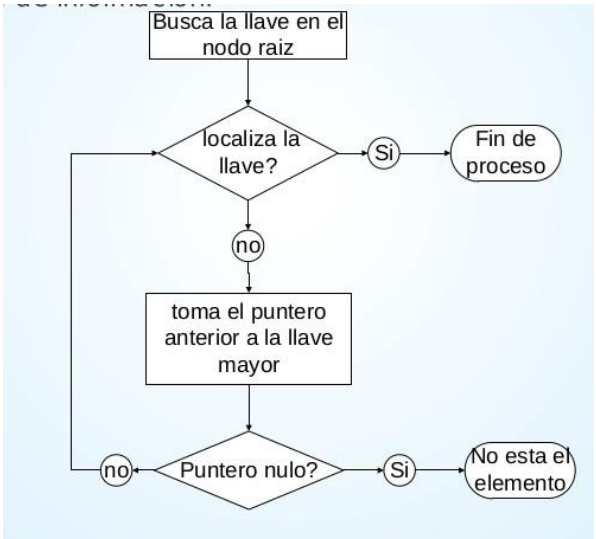
Creación

¿Cómo se construye un árbol B?
La característica principal de un árbol B es que todos sus nodos terminales están al mismo nivel, de manera tal que el árbol siempre se encuentra balanceado.

Por esta razón, no es posible construir un árbol B de la misma manera que los árboles tradicionales. Los árboles B se construyen de forma ascendente: cuando no se dispone de suficiente espacio, es la raíz quien debe alejarse de los nodos terminales y no al revés. Cuando llega un nuevo registro a un nodo lleno, y produce overflow, ese nodo se divide en 2 y un registro pasa arriba como nuevo padre. Crece el nivel del árbol. *División y promoción.*
* Cuando un nodo es de orden impar, ej 5, y al llegar otro produce overflow y queda un # de elementos pares, ej 6, se divide a la mitad y promociona el más chico del segundo nodo (los más grandes) En cambio, si es de orden par, ej 4, cuando llega uno y se divide el nodo, promociona justo el del medio y quedan ambos nodos con igual cantidad de elementos.

Búsqueda en árboles B

El proceso de búsqueda en un árbol B no difiere de un árbol binario común. Comienza la búsqueda por la raíz y se va bifurcando según el dato que se busque, descartando la mitad de los elementos, hasta encontrar el elemento deseado o un nodo sin hijos, que no contenga el elemento.



Performance de búsqueda:

En el caso de los árboles B, la eficiencia de búsqueda consiste en contar los accesos a disco que se requieren para encontrar un elemento o determinar que no se encuentra. La cantidad de accesos va a ser un número acotado entre el rango [1, H], siendo H la altura del árbol.
- *Mejor caso:* 1 lectura.
- *Peor caso:* h lecturas, donde h es la altura del árbol.

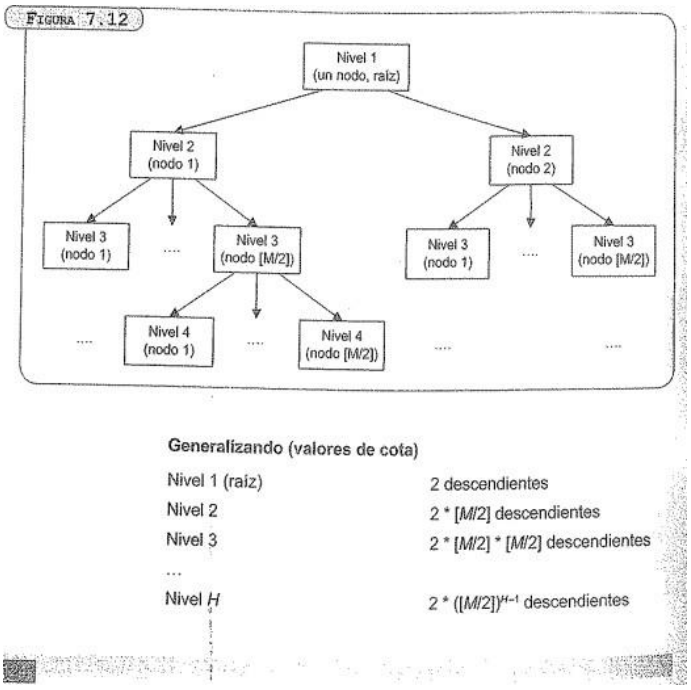
Hay que tener en cuenta que en los casos anteriores, siempre se calculó la eficiencia en función del a cantidad registros del archivo (N) . Para poder compararlos, es necesario

medir la eficiencia en variables similares, debe ser posible comparar H y N. Para ello se establece una cota para H, siempre teniendo en cuenta el peor caso, que es el que cumplirá siempre. Por eso, hay que tener en cuenta las propiedades de un árbol B: la raíz tiene al menos 2 hijos (si es que no tiene ninguno, los nodos internos [M/2] elementos

¿Cuál es el valor de h?
Axioma (principio): Si en un árbol balanceado de Orden M, si el número de elementos del árbol es N, hay N+1 punteros nulos en nodos terminales.

Cota para H

Nivel	# mínimo de descendientes
1	2
2	$2*[M/2]$
3	$2*[M/2]*[M/2]$
.....	.....
h	$2*[M/2]^{h-1}$
Relación entre h y # de nodos	
$N+1 >= 2*[M/2]^{h-1}$	
$h <= [1+\log_{[M/2]}((N+1)/2)]$	
Si M = 512 y N = 1000000 → h <= 3.37 (4 lecturas encuentra un registro)	



Inserción

Hay que recordar una propiedad de los árboles B, que es que los elementos siempre se insertan en los nodos terminales. Por lo tanto, en una inserción es necesario realizar H lecturas para encontrar el nodo al que corresponde el nuevo elemento.

Casos posibles:

- El registro tiene lugar en el nodo terminal, no se produce overflow: solo se hacen reacomodamientos internos en el nodo.

- El registro no tiene lugar en el nodo terminal, se produce
overflow: el nodo se divide y los elementos se reparten entre los nodos nuevos. Hay una promoción de una clave al nivel superior, y
esta promoción puede propagarse y generar una nueva raíz.

Performance de la inserción

- Mejor caso (sin overflow): H lecturas, porque hay que acceder a los nodos terminales, y 1 escritura.
- Peor caso (con overflow y se propaga hasta la raíz, aumenta en uno el nivel del árbol): H lecturas, 2h+1 escrituras (dos por nivel más la raíz)

Eliminación

Hay que tener en cuenta una característica que se hereda de los árboles binarios: para poder borrar un elemento, éste debe estar localizado en un nodo terminal. Si se desea borrar un elemento que no está en un nodo terminal, éste debe ser intercambiado por uno que si esté en una hoja, siempre con el menor de sus hijos mayores (subárbol derecho).

Posibilidades ante eliminación:

- *Mejor caso:* borra un elemento del nodo y no produce underflow,
solo reacomodos (numero de elementos >= [M/2]-1)

- *Peor caso:* se produce underflow, #elementos < [M/2] – 1

En este peor caso, donde se produce underflow, existen dos soluciones posibles:
redistribuir y concatenar
Dos nodos son hermanos si tienen el mismo padre.
Dos nodos son adyacentes hermanos si tienen el mismo padre y son
apuntados por punteros adyacentes en el padre.

- *Redistribución:* Cuando un nodo tiene underflow pueden trasladarse hacia él, claves de un nodo adyacente hermano, siempre y cuando este hermano pueda ceder alguna clave sin queda un underflow también. La clave raíz pasa al nodo en underflow y la clave del nodo adyacente pasa a ser raíz.

- *Concatenación:* En el caso de que un nodo esté un underflow, y a su nodo adyancete no le sobra ningún elemento, porque también quedaría en underflow, no se puede redistribuir. En este caso, se concatena con un nodo adyacente, disminuyendo el # de nodos (y en algunos casos la altura del árbol). Importante: el padre de esos 2 nodos que se concatenaron, también baja al nuevo único nodo.

Una ventaja de la redistribución es que no altera la cantidad de nodos del árbol, si se produce underflow sólo hay que reacomodar elementos entre los tres nodos involucrados, y no produce cambios en el resto del árbol. Primero hay que intentar redistribuir y, de no ser posible, luego concatenar.

Performance de la eliminación

- *Mejor caso* (borra de un nodo Terminal y sin underflow): H lecturas, 1 escritura

- *Peor caso* (se produce underflow y concatenación, lleva a decrementar el nivel del árbol en 1): 2h – 1 lecturas, H + 1 escrituras.

Modificación en árbol B

Se considera como una baja del elemento y una alta del elemento modificado.
Los árboles B representan una buena opción como estructura de datos para la organización índices asociados a un archivo de datos.

Árboles Balanceados B*

Los árboles B* tienen como característica que utilizan la redistribución no sólo en las bajas, sino también en las altas. Cuando tanto un nodo como su adyacente se encuentran completos, recién en ese momentos se produce la división. Esta forma de crear un árbol B* produce nodos más llenos (al menos ⅔) y no a la mitad como los árboles B. Por eso, todos los nodos quedan mejor distribuidos. De esta manera, se podría posponer la creación de páginas nuevas (nodos).
Se pueden generar árboles B más eficientes en términos de utilización de espacio.

Propiedades de un árbol B*

Suponiendo que es de orden M:

- Cada nodo del árbol tiene máximo M descendientes y M-1 elementos.
- Los nodos internos tienen como mínimo [(2M – 1) / 3] -1 elementos.
- Los nodos internos también contienen por lo menos [(2M – 1) / 3] –1 elementos.
- La raíz tiene al menos dos descendientes (o ninguno).
- Todas las hojas aparecen en igual nivel
- Una página que no sea hoja si tiene K descendientes contiene K-1 llaves.

! Existe una excepción a las propiedades de un árbol B* y es cuando sólo se tiene el nodo raíz. En este caso, al tener overflow, es imposible redistribuir porque no se cuenta con nodos hermanos. En ese caso, el nodo raíz se divide y sus dos hijos solamente completarán su espacio a la mitad.

Búsqueda en B*

Igual que un árbol B.

Bajas en B*

Igual que un árbol B.

Operaciones de Inserción en B*

La inserción en un árbol B* y su performance, dependerá de qué política se utilice para redistribuir.

Existen tres políticas posibles, que cada una determina qué hermano adyacente se tendrá en cuenta en caso de overflow:

- De un lado: El hermano adyacente tenido en cuenta para la redistribución será uno solo, izquierda o derecha (a menos que el nodo en overflow sea el extremo). De no ser posible porque ese hermano adyacente también se encuentra en overflow, se produce una división.

- De un lado u otro: si el nodo de la derecha está lleno y no se puede redistribuir, se busca el de la izquierda. **Y viceversa**. De no ser posible, se toma un hermano (ej izq si es política izq o derecha) y se divide, generando 3 nodos (2/3 llenos).

- De un lado y otro: se busca redistribuir por un lado y si no es posible, por el otro. De no ser posible tampoco, se divide entre los tres nodos, generando 4 con ¾ partes completas cada uno.

Archivos secuenciales indizados

Un archivo con acceso secuencial indizado es aquel que permite visualizar la información de dos maneras:
- Indizada: puede verse como un conjunto de registros ordenados por clave.
- Secuencial: puede accederse al archivo secuencialmente, con registros físicamente contiguos y ordenados nuevamente por una clave.

Problema

Se tiene un archivo con N registros que fueron organizados físicamente por orden de llegada y, además, un índice formado por un árbol B o B* que permite localización rápido a la información. Si se deseara acceder a la información de manera secuencial utilizando el orden de la llave, la única alternativa sería utilizando el índice. Recuperar todos los registros del archivo, significaría acceder a nivel hoja, volver al padre, y acceder al nodo terminal siguiente. Es mucho menos eficiente que tener la estructura ordenada físicamente.
Sin embargo, tener el archivo ordenado físicamente resuelve este problema pero resulta inaceptable al momento de realizar una búsqueda, una inserción o eliminación.

Se requiere una estructura en la que sean compatibles ambos casos.

Árboles B+

Existen soluciones de bajo costo para recuperar registros de un archivo en forma ordenada, sin necesidad de reacomodamientos físicos costosos.
Además, se requiere un mecanismo que permita localizar los datos contenidos en los nodos, a bajo costo, como podrían ser los árboles B y B*.

La estructura resultante de estos dos requerimientos es un árbol B+.
Consiste en un conjunto de grupos de registros ordenados por clave en forma secuencial, junto con un conjunto de índices, que proporciona acceso rápido a esos registros.

La característica principal de los árboles B+, las hojas (nodos terminales) están enlazadas y forman una lista. Todos los elementos se encuentran en las hojas. En cambio, los nodos no terminales funcionan como índices, hay punteros a los datos. Un árbol B+ utiliza la estructura de un B o de un B*, por lo tanto las inserciones y bajas se manejan de esa manera.

Esto permite acceso aleatorio rápido (como los B y B*) y también acceso secuencial rápido.

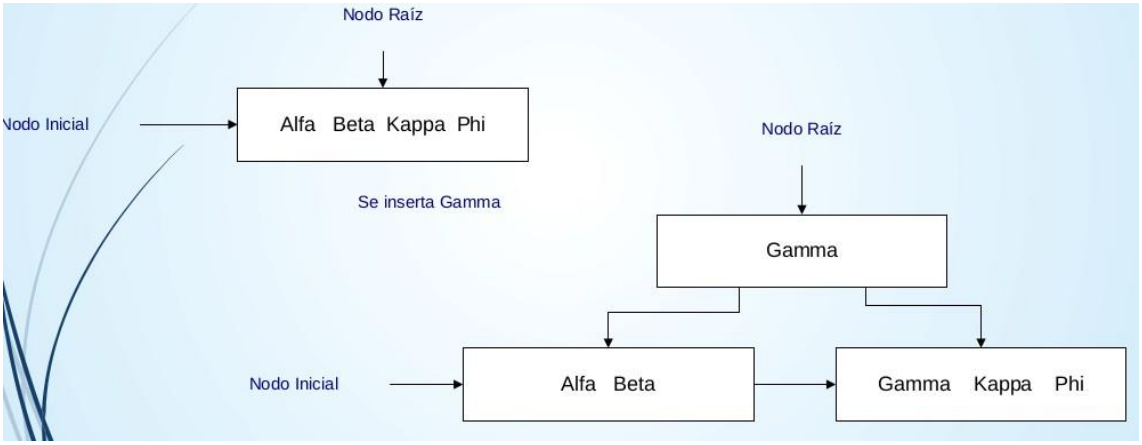
Conjunto índice: proporciona acceso indexado a los registros. En la raíz y nodos internos se encuentran las claves necesarias para establecer caminos de búsqueda.

Conjunto secuencia: Tienen todos los registros del archivo. Las hojas se vinculan entre sí (enlazan, como una lista, con punteros) para hacer el acceso secuencial más rápido.

Propiedades

En un árbol B+ de orden M:

- Cada nodo tiene máximo M descendientes y M-1 elementos.
- Los nodos internos tienen como mínimo [M/2 -1] elementos
- Las hojas también tienen como mínimo [M/2 -1] elementos
- La raíz tiene al menos dos descendientes, o ninguno.
- Todas las hojas aparecen en igual nivel
- Una página que no sea hoja si tiene K descendientes contiene K-1 llaves
- Los nodos terminales representan un conjunto de datos y son entrelazados entre ellos.
- Los nodos que no son hoja ni raíz tienen como mínimo M/2 descendientes



Separadores

- Derivados de las llaves de los registros que limitan un bloque en el conjunto de secuencia
- Los separadores más cortos, ocupan espacio mínimo

En el ejemplo, el separador sería Gamma.

Árbol B+ de prefijos simples

Se trata de un árbol B+ en el cual el conjunto índice está constituido por separadores más cortos, por ejemplo en el anterior podría ser G. El uso de prefijos simples busca aprovechar

mejor el uso de espacio físico.

Un árbol B+ de prefijos simples es un árbol B+ donde los separadores están representados por la mínima expresión posible de la clave, que sea capaz de terminar si la búsqueda se realiza a la izquierda o la derecha.
(Se utiliza lo mínimo que pueda servir como separador)

Búsqueda en B+

Es similar a los árboles B. Se sigue con la búsqueda hasta el último nivel del árbol (hojas), porque allí están todos los elementos.

Inserción (altas)

Siempre se agregan elementos en el último nivel del árbol. El problema se encuentra cuando se inserta un elemento en un nodo lleno, produciendo overflow.

¿Qué sucede en estos casos?
- El nodo en overflow se divide en 2, distribuyendo las claves equitativamente.
- Una copia de la clave del medio (o la más chica de los grandes), se promociona al nodo padre. Esto sucede solamente cuando el overflow sucede a nivel hoja, recordar que los nodos no terminales sólo son índices.

Bajas en B+

Las claves a eliminar siempre están en las hojas. Los underflow son tratados de igual manera que en los árboles B.
!! Importante: Las claves de la raíz o nodos internos NO se modifican. Son índices.

Comparación entre B y B+

	Árbol B	Árbol B+
Ubicación de datos	Nodos (cualquiera)	Nodo Terminal
Tiempo de búsqueda	=	=
Procesamiento secuencial	Lento (complejo)	Rápido (con punteros)
Inserción eliminación	Ya discutida	Puede requerir + tiempo

Resumen árboles

B → Mínimo en nodos no terminales (M/2) y en hojas (M/2)-1
B* → Mínimo [(2M – 1) / 3] -1
B+ → Mínimo (M/2) -1

El árbol B, es el único que difieren los mínimos entre las hojas y los nodos no terminales. En la raíz, para todos los B es mínimo 2 hijos o ninguno.

AVL → binario
Familia de árboles B → multicamino
Los árboles de familia B son muy flexibles y útiles para la administración de índices asociados a un archivo de datos. Sin embargo no es la única solución, y la mejor estructura para un archivo va a depender del archivo en sí y de su propósito. A veces utilizar un árbol no es necesario, por ejemplo si se tienen pocos registros.

5) Hashing

Si bien la performance de los árboles B es muy buena, se requieren entre 3 y 4 accesos a disco. Se podría presentar una situación en la práctica en la que esta cantidad de accesos resulte ineficiente.

- Secuencia: N/2 accesos promedio
- Ordenado: Log 2 N
- Árboles: 3 o 4 accesos

¿Qué es hashing o dispersión?

- Es una *técnica para generar una dirección base única para una clave dada.* La dispersión se usa cuando se requiere acceso rápido a una llave.

- Es una *técnica que convierte la clave del registro en un número aleatorio*, el que sirve después para determinar donde se almacena el registro.

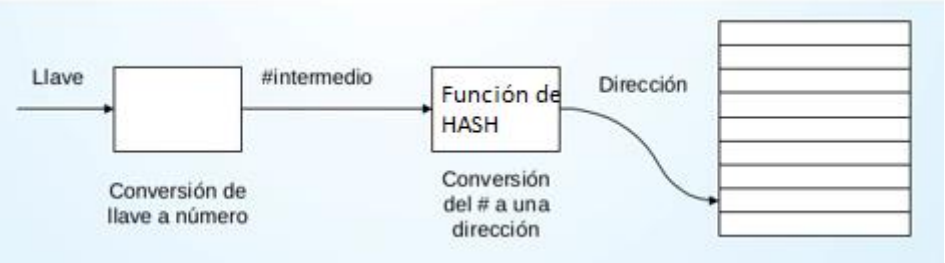
- Es una *técnica de almacenamiento y recuperación que usa una función de hash* para mapear registros en dirección de almacenamiento.

Es un método que mejora la eficiencia obtenida con árboles B, asegurando un promedio de 1 acceso para recuperar la información.

Atributos del hash

- *No requiere almacenamiento adicional:* no requiere índice, es el archivo de datos el que resulta disperso, dicha dispersión se genera a partir de la llave primaria.
- *Facilita inserción y eliminación rápida de registros:* se realiza de manera más eficiente en términos de acceso a disco (1 acceso a disco).

- Encuentra registros con muy pocos accesos al disco en promedio: el promedio es de un acceso a disco, si bien no es posible asegurar que cualquier registro sea hallado en un solo acceso, la mayoría de las búsquedas serán satisfechas con un acceso a disco.



Limitaciones: situaciones donde el método no es aplicable.

- *No podemos usar registros de longitud variable:* debido a que cada dirección física obtenida debe tener capacidad para almacenar un registro de tamaño conocido. Cuando se obtiene la dirección física, ésta tiene un espacio de almacenamiento asociado, conocido, limitado y NO puede ocurrir que el registro no quepa en él. Además, la función de hash genera, en teoría, un número único para cada clave, y ese espacio obtenido no le serviría al registro.

- *No es posible un orden lógico de datos:* los registros son esparcidos en el archivo, el orden está dado por la función de hash.

- *No permite claves duplicadas:* la función de hash no se puede aplicar a claves secundarias, porque es necesario que genere un número único.

Para determinar la dirección

Cuando se quiere almacenar un nuevo registro, la clave primaria de éste se convierte en un número **casi aleatorio**, a dicho número se le aplica la función de Hash que devuelve otro valor.
Dicho valor se mapea** y el valor mapeado corresponde a una dirección de memoria posible, donde se almacenará el registro. De esta manera, conociendo la clave primaria del registro y la función de hash utilizada para mapear, puede conocerse fácilmente dónde está almacenado el registro deseado.

**El mapeo de un valor resultado de aplicar la función de Hash implica en el hecho de que la función de Hash no contempla retornar un valor en el rango correcto de direcciones disponibles, por lo que se debe mapear dicho resultado dentro del rango para que se vuelva una dirección válida de memoria.
La función de hash NO es una función aleatoria.

Si al obtener la dirección de memoria donde almacenar el nuevo registro, sucede que ya está ocupada por otro registro, se produce colisión y, en algunos casos, overflow. El overflow conlleva un tratamiento especial.

Claves sinónimas: Dos claves son sinónimas cuando, al aplicarse al función de hash, obtienen el mismo resultado, que corresponde a la misma dirección donde almacenarse.

Dispersión: el método de hashing presenta 2 alternativas para su implementación:

- Hashing con espacio de direccionamiento estático: aquella política en la cual el espacio disponible para dispersar los registros en un archivo está fijado previamente. La función de hash aplicada a una clave da como resultado una dirección física posible. Tiene espacio determinado y limitado. Por ejemplo, en un aula hay 100 bancos y no puede modificarse, independientemente si llegan 20 100 o 500 alumnos.

- Hashing con espacio de direccionamiento dinámico: aquella política en la cual el espacio disponible para dispersar los registros en un archivo aumenta o se disminuye según las necesidades de espacio del archivo en cada momento.

Parámetros/propiedades de la dispersión

En el método de hashing, cuando se utiliza con espacio de direccionamiento estático, existen 4 propiedades que determinan cuán eficiente será la dispersión: *función de hash, tamaño de los nodos, densidad de empaquetamiento* y el *método de tratamiento de desbordes.*

1) Función de Hash:

La función de hash es como una caja negra que, a partir de una clave, se obtiene la dirección donde debe almacenarse el registro.
Dicha función transforma una clave primaria de un registro en otro valor que estará dentro de un determinado rango, dicho valor se utiliza como dirección física de acceso para insertar un registro en un archivo de datos.

Por ejemplo, suponiendo que la función de hash consiste en sumar los dígitos de una clave primaria, si llega un legajo con nro 12, y llega otro con nro 21, ambos darían 3 y deberían almacenarse en la misma dirección de memoria. Esta función no es eficiente.

Existen algunas diferencias con los índices:
- En la dispersión *no hay relación aparente entre llave y dirección.* Las claves son independientes, no influyen una sobre la otra. En cambio, en los índices por ejemplo cuando eran de longitud fija de 128 se podría utilizar el NRR, por ejemplo 4, para determinar en qué dirección está el registro.

- Dos llaves distintas pueden transformarse en iguales direcciones (colisiones).

Colisión

Situación en la que un registro es asignado a una dirección que está utilizada por otro registro.

Overflow

Situación en la que se produce una colisión y, además, no queda espacio para el registro nuevo.Soluciones posibles:

- Algoritmos de dispersión sin colisiones o que estas colisiones nunca produzcan overflow. Son algoritmos perfectos → imposibles de conseguir.
- Minimizar el número de colisiones a una cantidad aceptable. Existen distintos métodos para reducir las colisiones:
    - Esparcir registros: buscar una función de dispersión que distribuya los registros de la forma más aleatoria posible.
    - Usar más espacio en disco: Si se tienen 10 registros y 10 direcciones para almacenarlos, es probable que haya muchas colisiones. Si se tienen 100 direcciones para almacenarlos, disminuye el número de colisiones que se producen. Distribuir pocos registros en muchas direcciones, baja la densidad de empaquetamiento. La ventaja es que disminuye colisiones y, por lo tanto, overflow, pero desperdicia espacio y la respuesta a cuánto espacio desperdiciado es tolerable tiene muchas posibles respuestas.
    - Colocar más de un registro por dirección: direcciones con N claves, mejoras notables. Se relaciona con el tamaño de cubeta

2) Tamaño de las cubetas (Nodos)

El tamaño de cada nodo estará sujeto a la posibilidad de transferencia de información en cada operación de E/S entre la RAM y el disco.

- Cada dirección de memoria puede almacenar más de un registro. Por ejemplo sillas en un banco.
- A mayor tamaño de la cubeta:
    - Menor overflow
    - Mayor fragmentación (interna)
    - Búsqueda más lenta dentro de la cubeta (No es tan importante porque se trata de una búsqueda local)

3) Densidad de empaquetamiento

La relación entre el espacio disponible para el archivo de datos, y la cantidad de registros que integran dicho archivo.

Se trata de la proporción de espacio del archivo asignado, que en realidad almacena registros. La proporción del archivo que está utilizada. Proporción de ocupación.
DE = (número de registros almacenados el archivo % capacidad total del archivo)
capacidad total del archivo= cantidad de nodos x tamaño de cada nodo

Cuanto mayor sea la DE, mayor será la posibilidad de colisiones, dado que se dispone de menos espacio para almacenar registros. Si embargo, con una DE muy baja hay más

desperdicio de espacio, fragmentación.
La DE no es constante. Al principio cuando se crea el archivo y, supongamos que tiene 1 registro, es muy pequeña. A medida que se van agregando registros, el archivo crece y la DE también.

Estimación del overflow
Sabiendo que:
- N # de cubetas,
- C capacidad de nodo,
- R # reg. Del archivo
- DE = R / (C x N)

Probabilidad que una cubeta reciba I registros *(distribución de Poisson).* Sirve para estimar el overflow que habrá.

Con el ejemplo de los alumnos y las sillas, se trata de la probabilidad de que cierta cantidad de alumnos deban ir al mismo banco.

Análisis numéricos de Hashing

En general si hay n direcciones, entonces el # esperado de direcciones con I registros asignados es N*P(I).

Las colisiones aumentan con al archivo más "lleno"

4) Tratamiento de Colisiones con Overflow

Hemos visto que el % de overflow se reduce, pero el problema se mantiene dado que no llegamos a 0%. Cuando ocurre overflow, deben realizarse 2 acciones: encontrar lugar para el registro en otra dirección, y asegurarse que posteriormente el registro pueda ser encontrado allí.

Algunos métodos para el tratamiendo de overflow son:

- *Saturación progresiva:* Cuando se completa el nodo, se busca el próximo hasta encontrar uno libre, y se coloca el nuevo registro allí. Como los registros van almacenándose en el próximo nodo libre/vacio, cuando se realiza una búsqueda, esta termina al encontrar un nodo libre ¿Qué pasa entonces si se quiere eliminar un registro que estaba, por ejemlo, en el medio del archivo? En necesario colocar una marca de borrado, que indique que ese nodo está libre, es decir puede ser nuevamente utilizado, pero estuvo ocupado. Sirve para no detener la búsqueda.

- *Saturación progresiva encadenada:* similar a saturación progresiva, pero los registros de saturación se encadenan y "no ocupan" necesariamente posiciones contiguas. Se forma una lista, por lo que el algoritmo es más complejo. Cada cubeta contiene el registro y puntero a otra cubeta que debería estar allí.

! Importante: Si una clave A se ubica en un nodo 1, y una clave B debe ubicarse en la misma (sinónimas), pero se produce overflow y se ubica en el proximo nodo libre 2. Suponiendo que llega una clave C que debe ubicarse en el nodo 2 (base), la que debe reacomodarse es la clave B, porque la idea es que la mayoría de las claves estén en el lugar que les corresponde originalmente (base)

- *Doble dispersión:* el problema que tiene la saturación progresiva es que, a medida que se producen saturaciones, los registros tienden a esparcirse en nodos cercanos. Esto produce exceso de saturación sectorizada. Una solución es almacenar los registros de overflow en zonas no relacionadas.
La doble dispersión es una técnica a través de la cual se resuelven overflows aplicando una segunda función de hash a la clave. Esta segunda función de hash produce un número, el cual no se trata de una dirección sino de un desplazamiento. Cuando se produce overflow, se suma a la dirección original el resultado de la segunda función de hash, tantas veces como sea necesario hasta encontrar una dirección con espacio.

- *Área de desborde separado:* No utiliza nodos direccionables por la función de Hash para los registros en overflow, estos van a nodos especiales.
Los registros están en su dirección base y sus sinónimos van al área de desborde. Es mejor porque no hay que reacomodar/reubicar y en la base, todos se ubican en su lugar correspondiente.
Ubicación del desborde:
    - A intervalos regulares entre direcciones asignadas.
    - Cilindros de desborde.

El hashing estático resulta ser un método que si bien satisface las búsquedas en un acceso en el 99,9% de los casos (siempre y cuando la DE sea menor al 75%).
Sin embargo existen situaciones en las cuáles más de un acceso sean requeridas, y estas situaciones se dan cuando se generan situaciones de saturación.

Hashing dinámico

Con hashing estático surgen algunos problemas. Requiere un número de direcciones fijas, virtualmente imposible.
Cuando un archivo se completa, es necesario obtener mayor cantidad de direcciones para nodos y también modificar la función de hash para que pueda direccionar a esos nuevos nodos. El archivo debe ser redispersado y eso tiene un costo muy alto. Mientras se realiza esta operación, no es posible que el usuario final acceda al archivo.

Una solución es utilizar hashing con espacio de direccionamiento dinámico. El tamaño varía en cada momento. Por ejemplo, puede ir agregándose bancos a medida que llegan alumnos. En teoría, es ilimitado (depende del almacenamiento).
La eficiencia de búsqueda en Hash Dinámico SIEMPRE es uno. Ya que no encuentra cubetas en saturación que impliquen más accesos.

Varias posibilidades:
- Hash virtual
- Hash dinámico
- Hash Extensible

Hash extensible

El principio del método consiste en comenzar a trabajar con un único nodo para almacenar registros, e ir aumentando la cantidad de direcciones disponibles a medida que se van completando dichos nodos.
Este método, como todos los que trabajan con espacio de direccionamiento dinámico, no utiliza el concepto de DE. Esto se debe a que el espacio que utiliza va aumentando o disminuyendo según las necesidades.

¿Cómo trabaja?

La función de hash retorna un string de bits, cuya cantidad determina la cantidad máxima de direcciones a las que puede acceder el método.
Hash extensible trabaja bajo las siguientes pautas:

- Se utilizan solo los bits necesarios de acuerdo a cada instancia del archivo.
- Los bits tomados forman la dirección del nodo que se utilizará.
- Si se intenta insertar a una cubeta llena, deben reubicarse todos los registros allí - contenidos entre el nodo viejo y el nuevo, para ello se toma un bit más, para que pueda ser posible referenciar a más direcciones que ahora se necesitan.
- La tabla tendrá tantas entradas (direcciones de nodos)como 2^i , siendo i el número de bits actuales para el sistema.

El método de Hash Dinámico requiere de una estructura auxiliar (requiere espacio adicional), una tabla que se almacena en memoria principal (RAM), que contiene la dirección física de cada nodo.

Se debe tener en cuenta que la función de hash no retorna una dirección, sino una secuencia de bits. Esta secuencia de bits permite obtener de la tabla en memoria principal, la dirección física del nodo para almacenar la clave.

Elección de organización

Los archivos se utilizan para acomodar datos para satisfacer requerimientos.

Organización	Acc.un reg. CP	Todos reg. CP
Ninguna	Lento	Lento
Secuencial	Lento	Rápido
Index sec.	Buena	Rápida
Hash	Rápido	lento

Para poder realizar una correcta elección sobre la organización de un archivo, hay que tener en cuenta algunos factores.

¿Qué examinar?

- Características del archivo: número de registros, tamaño de registros.

Requerimientos de usuario: tipos de operaciones, número de accesos a archivos, características del hardware (tamaño de sectores, bloques, pistas, cilíndros, etc.),parámetros, tiempo (necesario para desarrollar y mantener el soft, para procesar archivos),uso promedio (número de registros usados/ número de registros).

Organización	Acceso a un registro por clave primaria	Acceso a todos los registros por clave primaria
Ninguna	Muy ineficiente	Muy ineficiente
Secuencial	Poco eficiente	Eficiente
Secuencial indizada	Muy eficiente	El más eficiente
Hash	El más eficiente	Muy ineficiente