*Resumen Teoría*

***Clase 1***

***Base de datos***

¿Qué es una base de datos? 1) Es una colección de datos relacionados, donde un dato representa hechos conocidos que pueden registrarse y que tienen un resultado implícito.

2) Colección de archivos diseñados para servir a múltiples aplicaciones.

Propiedades implícitas Una BD 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, ósea los datos deben tener cierta lógica). Una BD 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 Por último, está sustentada físicamente en archivos en dispositivos de almacenamiento persistente de datos

***Archivos***

Definición

1. Colección de registros guardados en almacenamiento secundario.
2. Colección de datos almacenados en dispositivos secundarios de memoria
3. Colección de registros que abarcan entidades con un aspecto común y originadas para algún propósito particular.
4. Colección de bytes que representa información.

***Archivos – Organización***

Secuencia de bytes

• No se puede determinar fácilmente comienzo y fin de cada dato.

• Archivos de texto.

Registros y Campos

• Campo: Unidad más pequeña, lógicamente significativa de un archivo

• Registro: Conjunto de campos agrupados que definen un elemento del archivo

***Archivos – Acceso***

Secuencial Físico: acceso a los registros uno tras otro y en el orden físico en el que están guardados

Secuencial indizado (lógico): acceso a los registros de acuerdo al orden establecido por otra estructura

Directo: se accede a un registro determinado sin necesidad de haber accedido a los predecesores

***Archivos – Tipos***

***Forma de acceso***

•Serie: cada registro es accesible solo luego de procesar su antecesor, simples de acceder (acceso secuencial físico)

•Secuencial: los registros son accesibles en orden de alguna clave (acceso secuencial indizado/ secuencial lógico)

• Directo: se accede al registro deseado (acceso directo)

***# 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***

Memoria intermedia entre un archivo y un programa, donde los datos residen provisoriamente hasta ser almacenados definitivamente en memoria secundaria o donde los datos residen una vez recuperados de dicha memoria secundaria. Los buffers ocupan lugar en RAM y es el SO quien se encarga de manipular los buffers.

***Archivos – Operaciones básicas***

Contamos con 2 niveles de archivos, uno Físico (almacenamiento secundario) y otro Lógico (dentro del programa).

Crear •Abrir •Read/Write •Eof •Seek(localización)

En cuanto a las instrucciones Read y Write, estas operaciones leen y/o escriben sobre los buffers relaciona dos a los archivos, no se realizan directamente sobre la memoria secundaria. En ambos casos la variable debe ser del mismo tipo que los elementos que se declararon como parte del archivo.

***Clase 2 Archivos - Algorítmica Clásica***

***Operaciones usuales:*** •Agregar nuevos elementos •Actualizar un archivo maestro con uno o varios archivos detalles • Corte de control •Merge de varios archivos.

*Actualización Maestro Detalle*

• Se denomina maestro al archivo que resume un determinado conjunto de datos.

• Se denomina detalle al que agrupa información que se utilizará para modificar el contenido del archivo maestro.

*Archivos – Merge*

• Involucra archivos con contenido similar, el cual debe resumirse en un único archivo.

Ver que mas se puede agregar

***Clase 3***

*Memoria principal (RAM)*

Es rápida y de simple acceso, pero su uso tiene algunas desventajas respecto al almacenamiento secundario: • Capacidad limitada •Mayor costo •Es volátil

*Almacenamiento secundario*

Necesita más tiempo para tener acceso a los datos que en RAM.

• Su acceso es tan “lento” que es imprescindible enviar y recuperar datos con inteligencia

• Al buscar un dato, se espera encontrarlo en el primer intento (o en pocos)

• Si se buscan varios datos, se espera obtenerlos todos de una sola vez

*Viaje de un byte*

*Archivo físico:*

• Archivo que existe en almacenamiento secundario

• Es el archivo tal como lo conoce el S.O. y que aparece 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 cuál archivo físico real se utiliza o donde está ubicado

***Quienes están involucrados***

**Administrador de archivos:** conjunto de programas del S.O. (capas de procedimientos) que tratan aspectos relacionados con archivos y dispositivos de E/S.

• En Capas Superiores: aspectos lógicos de datos (tabla). Establecer si las características del archivo son compatibles con la operación deseada

• En Capas Inferiores: 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:** encargado de controlar la operación de disco.

• Colocarse en la pista

• Colocarse en el sector

• Transferencia a disco

**Capas del protocolo de transmisión de un byte**

•El Programa pide al S.O. escribir el contenido de una variable en un archivo

•El S.O. transfiere el trabajo al Administrador de archivos

•El Adm busca el archivo en su tabla de archivos y verifica las características

•El Adm obtiene de la FAT(tabla de asignación de archivos) la ubicación física del sector del archivo donde se guardará el byte.

•El Adm se asegura que el sector del archivo está en un buffer y graba el dato donde va dentro del sector en el buffer

•El Adm de archivos da instrucciones al procesador de E/S (donde está el byte en RAM y en que parte del disco deberá almacenarse)

• El procesador de E/S encuentra el momento para transmitir el dato a disco, la CPU se libera • El procesador de E/S envía el dato al controlador de disco (con la dirección de escritura)

• El controlador prepara la escritura y transfiere el dato bit por bit en la superficie del disco.

***Archivos – Tipos***

*Archivos como secuencias de datos*

• No se puede determinar fácilmente comienzo y fin de cada dato.

• Ejemplo: archivos de texto

*Archivos estructurados*

**Campos**

•Unidad lógicamente significativa más pequeña de un archivo. Permite separar la información •Identidad de campos: variantes, pro y contras.

•**Longitud predecible** (long. Fija), desperdicio de espacio, si el tamaño es pequeño al agrandarlo se podría desperdiciar más espacio)

•**Indicador de longitud** (al ppio de cada campo)

• **Delimitador al final de cada campo** (carácter especial no usado como dato)

**Registros**

• Organización de registros

• **Longitud predecible** (en cant. de bytes o cant. de campos)

• Campos fijos o variables

• **Longitud variable**

• Indicador de longitud (al comienzo, indica la cant. de bytes que contiene)

• Segundo archivo (mantiene la info de la dirección del byte de inicio de cada registro)

• Delimitador (carácter especial no usado como dato)

• Estudio de casos: ventajas y desventajas.

------------------------------------------------------------------------------------------------------------------------------

• Se concibe al Registro como la cantidad de información que se lee o escribe

• Objetivo: acceder sólo un registro específico

• Es conveniente identificar un registro con una llave o clave que se base en el contenido del mismo

**Clave**

• Permite la identificación del registro

• Deben permitir generar orden en el archivo por ese criterio

Puede ser Univoca/Primaria o secundaria.

• U/P: Identifican un elemento particular dentro de un archivo

•S: Generalmente no identifican un único registro

**Forma canónica:** forma estándar para una llave, puede derivarse a partir de reglas bien definidas.

• Representación única para la llave, ajustada a la regla •

Ej: Clave 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

------------------------------------------------------------------------------------------------------------------------------

***Estudio de perfomance***

Caso secuencial

•Mejor caso: leer 1 reg. , peor caso leer n registros

•Supongamos que tenemos 1000 registros, buscar uno en particular mejor caso 1, peor caso 1000, promedio 500, en realidad el mejor caso es 0, el buffer puede estar en memoria. •Promedio: n/2 comparaciones

•Es de O(n), porque depende de la cantidad de registros

•Lectura de Bloques de registros

•mejora el acceso a disco,

•no varían las comparaciones.

Acceso directo

• Permite acceder a un registro preciso

• Requiere una sola lectura para traer el dato [O(1) ].

• Debe necesariamente conocerse el lugar donde comienza el registro requerido

Numero 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 ⎝ 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.**

***Eliminación de archivos (bajas)***

•Registro de longitud fija: agregar o modificar, sin inconvenientes

•Registros de longitud variable: problemas

•Ej: Intentar modificar un registro, tal que el modificado quede de mayor tamaño ×Soluciones posibles:

• Agregar los datos adicionales al final del archivo (con un vínculo al registro original) ◊ complica el procesamiento del registro.

• Reescribir el registro completo al final del archivo ◊ queda un espacio vacío (desperdiciado) en el lugar origen

•La operación agregar no genera inconvenientes.

***Baja Lógica***

• Cualquier estrategia de eliminación de registros debe proveer alguna forma para reconocerlos una vez eliminados (ejemplo: colocar una marca especial en el reg. eliminado).

• Con este criterio se puede anular la eliminación fácilmente.

• Cómo reutilizar el espacio de registros eliminados?

• Los programas que usan archivos deben incluir cierta lógica para ignorar los registros eliminados

***Baja Física***

• Recuperar el espacio

• La forma más simple es copiar todo en un nuevo archivo a excepción de los registros eliminados ⎝ Baja Física

• Frecuencia

• Tiempo (depende del dominio de aplicación)

• Ante la necesidad de espacio

***Aprovechamiento del espacio***

•Reg longitud fija: es necesario garantizar:

•Marca especiales en los reg. borrados en Baja Lógica

•Reg longitud variable: los nuevos elementos deben “caber” en el lugar

***Recuperación del espacio para su reutilización cuando se agreguen registros***

• Búsqueda secuencial -> usa las marcas de borrado.

• Para agregar, se busca el 1º reg. eliminado. Si no existe se llega al final del archivo y se agrega allí.

• Es muy lento para operaciones frecuentes.

• Es necesario

• Una forma de saber de inmediato si hay lugares vacíos en el archivo

• Una forma de saltar directamente a unos de esos lugares, en caso de existir

***Aprovechamiento de espacio (reg. long. fija)***

• Recuperación de espacio con Lista o pilas (cabecera)

• Lista encadenada de regs disponibles.

• Al insertar un reg nuevo en un archivo de reg. con long fija, cualquier registro disponible es bueno. • La lista NO necesita tener un orden particular, ya que todos los reg. son de long. fija y todos los espacios libres son iguales

***Aprovechamiento de espacio (reg. long. variable)***

•Marca de borrado al igual que en reg. de long. fija (ej:\*)

•El problema de los registros de longitud variable está en que no se puede colocar en cualquier lugar, para poder ponerlo debe caber, necesariamente.

•Lista: No se puede usar NRR como enlace. Se utiliza un campo binario que explícitamente indica en enlace (conviene que indique el tamaño).

•Cada registro indica en su inicio la cant. de bytes

•Reutilización: buscar el registro borrado de tamaño adecuado (lo suficientemente grande). •Como se necesita buscar, no se puede organizar la lista de disponibles como una pila.

•El tamaño “adecuado” del primer registro borrado a reutilizar ->origina Fragmentación

**Aprovechamiento de espacio (Fragmentación)**

•**Interna**: ocurre cuando se desperdicia espacio en un registro, se le asigna el lugar pero no lo ocupa totalmente.

•Generalmente ocurre con Reg. longitud fija

•Reg. longitud variable evitan el problema

•Espacio asignado => No ocupado

•**Externa:** ocurre cuando se desperdicia espacio entre registros

•Generalmente ocurre con Reg. longitud variable

•Espacio no asignado => No ocupado

**Estrategias de colocación en registros de longitud variable:**

•Primer ajuste: se selecciona la primer entrada de la lista de disponibles, que pueda almacenar al registro, y se le asigna al mismo. Asigna el espacio completo (por definición)

• Minimiza la búsqueda

• No se preocupa por la exactitud del ajuste

•Mejor ajuste: elige la entrada que más se aproxime al tamaño del registro y se le asigna completa.

•Exige búsqueda

•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

***En general:***

•Las estrategias de colocación tienen sentido con reg. de longitud variable

•Primer ajuste: más rápido. Puede generar fragmentación interna (porque se le asigna todo el espacio, por definición)

•Mejor ajuste: Puede generar fragmentación interna (porque se le asigna todo el espacio, por definición)

•Peor ajuste: Puede generar fragmentación externa. No genera Fragmentación interna por definición

***Clase 4***

***Búsqueda de información***

* ***Costo(tener en cuenta)***
  + cantidad de comparaciones (operaciones en memoria)
  + Se pueden mejorar con algoritmos más eficientes.
  + cantidad de accesos (operaciones en disco) (critico, cuanto menos mejor, ya que 1 acceso a memoria = 1 millon de comparaciones en memoria).

Info extra: archivos serie son ineficientes, en el otro extremo están los directos que son los mas eficientes, y en el medio tendríamos los secuenciales con acceso secuencial indexado.

Un archivo serie tiene un orden lineal y la cantidad de accesos a discos es n^2, siendo n la cantidad de registros del archivo(en promedio se leen n^2 registros).

* ***Buscar un registro***
  + + rápido si conocemos el NRR (directo)(si lo conozco, con hacer un seek a esa posición ya consigo la información, como es el caso de los accesos directos).
  + Secuencia debe buscarse desde el principio
  + Trataremos de incorporar el uso de claves o llaves.
* ***Búsqueda binaria 🡪 precondiciones***
  + Archivo ordenado por clave(es similar a trabajar con un vector ordenado).
  + Registros de longitud fija
* ***Búsqueda 🡪 partir el archivo a la mitad y comparar la clave,***
  + puedo acceder al medio por tener long. Fija
  + Si N es el # de registros, la performance será del orden de Log2 N
  + Se mejora la performance de la búsqueda secuencial.
  + Resumen teorico: Puedo ir dividiendo el archivo con filesize con una búsqueda dicotómica, que tiene una eficiencia de log en base n, que es mayor que la eficiencia que se obtiene en una búsqueda lineal(n/2).
* ***Búsqueda binaria***
  + acota el espacio para encontrar información
  + costo 🡪 mantener ordenado el archivo (si el costo no se mantiene, dejamos de tener la posibilidad de hacer una búsqueda binaria).
  + Hay que tener en cuenta si el tipo de archivo es volátil o no, puesto a que dependiendo de esto será necesario realiza, en mayor o menos medida, las operaciones necesarias para organizar dicho archivo.
  + Requiere que el archivo este ordenado, lo cual es difícil si tenemos en cuenta que la mayoría de archivos son dinámicamente inestables, por lo que tenerlos ordenados es bastante ineficiente.

***Como clasificar (ordenar) un archivo***

* + En RAM (Levanto toda la información(n lecturas), supongamos en un vector, la llevo a RAM, hago n lecturas, lo ordeno y luego lo llevo a disco(n escrituras). En todo el proceso tendría eficiencia de 2N, que es superior a leer a leer y escribir de a un dato, que tiene orden factorial de n). Tiene una limitación, que ocurre cuando el archivo no entra en memoria RAM. Como alternativa está el siguiente criterio.
  + Claves en RAM (cuando ordeno un archivo, lo hago por un criterio o campo en particular del archivo. En este caso levanto del registro el campo por el que voy a ordenar el registro, me guardo el campo particular y el número de registro donde ese campo está en el archivo). En resumen, solo levanto una parte del archivo, que puedo ordenar y guardar en RAM porque es más pequeño que el archivo completo.

Tengo N lecturas del registro original, más una lectura y una escritura para el archivo nuevo (el ordenado), 3N que es más eficiente que ordenar sobre disco que tiene factorial de N.

* + Archivos grandes (caso donde no aplica ninguno de los 2 anteriores).
    - En estos casos lo que sea es: partir el archivo, ordenar cada parte y juntar las partes con un merge. Dicho merge agrega a la eficiencia otro nivel de complejidad e ineficiencia, lo que hace esta alternativa peor que las anteriores, pero aun así sigue siendo mejor a la ordenación en disco.
* ***Conclusiones***

La búsqueda binaria mejora a la secuencial, pero trae ciertos problemas, el número de accesos baja pero no llega a uno (única manera es con acceso directo y eso es otra política). Tenemos la opción de mejorar la eficiencia si tenemos un archivo ordenado que mejora la eficiencia, aunque mantener ese orden es demasiado costoso. Además debemos tener en cuenta que la clasificación en RAM aplica para archivos pequeños.

Se debería mejorar el método de ordenación, no reordenando todo el archivo, sino mediante la utilización de árboles.

***Clase 5***

***Búsqueda de datos – Índices***

* Búsqueda de información:
  + debemos minimizar el número de accesos
  + Secuencial (poco eficiente)(n/2) (eficiencia lineal)
  + Binaria (muy costosa)(eficiente pero problema de mantener orden)(logn)
  + Estructuras auxiliares(permiten ver la información como si estuviera ordenada => los índices)(Ej de libro, donde ultimas paginas tienen índices ordenados por temas con referencias a las páginas)
* El uso de índices proporciona varios caminos de acceso a un archivo, lo que significa que el archivo de datos va a ser único y además, que un archivo de datos tendrá tantos índices o criterios de búsqueda como el problema lo requiera.

***Índices 🡪 definición***

* **Herramienta** para encontrar registros en un archivo. Consiste de un *campo de llave* (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** (*clave, dirección*) usada para decrementar el tiempo de acceso a un archivo, ya que sobre la tabla se hace búsqueda dicotómica y una vez que se encontró el elemento se accede a donde está el registro.

1ero: como se ve en un libro – 2do: caso índice (distancia en bytes por caso donde se usan registros de longitud variable, se pueden usar tanto fija como variable). 3ero: se obtiene eficiencia logarítmica por uso de árboles con búsqueda binaria.

* **Característica fundamental de un árbol:** permite imponer orden sin que el archivo se desordene. Tenemos datos desordenados e índices que actúan como filtro, que permiten operar como si dichos datos estuvieran ordenados.

***Llave primaria*** ***: cía grabadora(se repite) + Nº de identificación de la cía(no se repite)***

* + Forma canónica: cía en mayúsculas + Nº identificación
  + No se puede hacer búsqueda binaria sobre el archivo ya que tiene reg. de longitud variable (no se puede usar en NRR como medio de acceso)
  + Es con lo que ordenamos el archivo, se utilizan 2 atributos donde ambos combinados forman la clave primaria
  + Esto esta aplicado a un ejemplo de la teoría, donde se utilizan registros de longitud variable.

***Dos Archivos: índice(o estructura de índice) y datos***

* Se construye un índice: llave de 12 caracteres (alineada a izq. y completada con blancos) más un campo de referencia (dir. del primer byte del registro correspondiente)
* Estructura del índice: archivo ordenado de reg. de long fija(por el campo de referencia) (puede hacerse búsqueda binaria por tener long fija). (nos permite ver los archivos desordenados como si estuvieran ordenados).
* En memoria
* Más fácil de manejar que el arch. de datos

***Operaciones básicas de un archivo indizado***

* Crear los archivos (el índice(o tantos índices como sean necesarios como índices tuviera el archivo de datos) 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. Se almacena en un arreglo ya que la búsqueda es muy eficiente) Subir índice a memoria.
* Reescritura del archivo de índice (cambios 🡪 reescribir) Esto una vez terminada la acción.
* ***Agregar nuevos registros*** 
  + Implica agregar al archivo de datos y al archivo de índices
  + Archivo de datos: copiar al final (se debe saber el NRR (fija) o distancia en bytes (variable) para el índice)
  + Índice ordenarse con cada nuevo elemento en forma canónica (en mem.), setear el flag anterior
* ***Eliminar un registro***
  + Arch. datos 🡺 Cualquier técnica de las vistas para reutilizar el espacio
  + Arch. índices 🡺 se quita la entrada (ó se podría marcar como borrado).
* ***Actualización de registros***
  + Sin modificar la clave (que pasa con el índice?)
  + Si el registro no cambia de longitud, se almacena en la misma posición física, el índice “no se toca”.
  + Si el reg. cambia de longitud (se agranda) y se reubica en el arch. de datos 🡺 se debe guardar la nueva posición inicial en el índice
  + Si se trata de long. Fija, no hay que hacer mas actividad
  + Modificando la clave (que sucede?)
  + Se modifica el archivo de datos + se debe modificar el indice
  + Se debe actualizar y reorganizar el archivo de índices
  + Cómo simplificar 🡺 Modificar = Eliminar + Agregar (ya vistos)

***Ventajas***

* Se almacena en memoria principal(la búsqueda logarítmica se hace en RAM por lo que es muy eficiente)
* Permite búsqueda binaria(genera eficiencia de búsqueda logarítmica)
* El mantenimiento es menos costoso(mantener el índice actualizado mientras lo tengo como un vector en RAM es muy eficiente)

***Desventajas***

* Si no caben en memoria RAM? (Caso de tener muchos índices, muchos vectores)(Al momento de terminar cada tarea tengo que actualizar todos los índices que fueron modificados, esto genera más trabajo de operación de entrada/salida.
* Reescritura del archivo de índices?
* Persistencia de datos (En caso de un imprevisto, como podría ser una corte en la energía necesaria para el ordenador, corte de luz, todos los índices que estaban en memoria RAM se pierden, por lo que el primer trabajo será reconstruir estos índices cuando sea posible).

---------------------------------------------------------------------------------------------------

***Apuntes de audios***

Solución al problema de persistencia de datos: el indicé en todo momento debe estar sobre **disco** (persistido). Aun así esto genera pérdidas en lo que a performance se refiere.

Posible solución => implementación de árboles (sobre disco y no en RAM), estructuras que permiten hacer búsquedas binarias (eficientes). Con esto logramos persistencia y despreocuparnos del tamaño del índice, ya que lo tenemos en disco. La eficiencia sigue siendo logaritmo en base 2, pero no en RAM, sino en disco por lo que perdemos algo de eficiencia.

Otra posible solución que no implementa índices seria implementando organización por dispersión (hashing), que permite encontrar registros con un acceso a disco siempre.

---------------------------------------------------------------------------------------------------

***Índices secundarios***

* ***Definición***
  + No sería natural solicitar un dato por clave, se utiliza normalmente un campo más fácil de recordar, dicho campo es uno que pertenece a una llave secundaria porque puede repetirse (las claves secundarias se pueden repetir).
    - No es lo mismo la construcción de un índice sin repeticiones a la construcción de uno que si las tiene.
    - Las claves univocas no repiten información, un elemento en una clave univoca aparece una sola vez en el archivo, mientas que las claves secundarias permiten que un elemento de la clave aparezca más de una vez en el archivo.
  + El índice secundario relaciona la llave secundaria con la llave primaria
    - Una clave primaria es una clave univoca que fue elegida como primaria. En caso de tener 2 claves univocas, elijo una.
    - Voy a tener mi archivo de datos y mi archivo con la clave primaria. Además pueden haber otras claves univocas o también claves secundarias por eso el primer punto del índice secundario.
    - La clave secundaria no va a tener dirección física del archivo de datos, no tiene asociada la posición donde el elemento está en el archivo de datos, tiene asociada la clave primaria (accedo por la clave secundaria, busco por la clave secundaria, encuentro la clave primaria, que tengo que buscarla en la clave primaria y ahí acceder al elemento de datos). De esa manera solo la clave primaria tiene direcciones fijas del archivo de datos. En caso de que se modifique el archivo de datos, esas direcciones físicas solo alteran el índice primario, el resto de los índices no se altera, lo que termina siendo más eficiente.
    - Cualquier índice que no sea el primario accede al archivo de datos a través del índice primario
  + Acceso 🡺 1º por llave secundaria (se obtiene la clave primaria) y luego llave primaria (en índice primario)
* ***Problemas:*** *la repetición de información*
  + El arch. de índices se debe reacomodar con cada adición:
    - Puedo agregar una clave secundaria nueva
    - Puedo agregar una clave secundaria que ya existe
  + Misma clave varias ocurrencias, en distintos registros
  + Tengo que organizar la información de dada una clave secundaria, tengo todos los lugares donde esa clave esta. Si tengo misma ocurrencia de la clave en distintos registros, tengo que compactar todos esos registros en uno solo.
    - Esto tiene un inconveniente tener una sola clave con las n referencias(vector de punteros) por las n ocurrencias.(preguntar)
* ***Soluciones***
  + Arreglo: clave + vector de punteros con ocurrencias
  + Al agregar un nuevo reg. de una clave existente no se debe reacomodar nada-> solo reacomodar el vector de ocurrencias
  + Al agregar un nuevo reg. con una clave nueva, se genera un arreglo con la clave y un elemento en el vector de punteros
  + ***Problema:*** *elección del tamaño del vector.*
    - Tamaño fijo(necesito que la cantidad de claves primarias que tiene asociadas la clave secundaria, no este limitada en espacio)
      * Otra alternativa es usar tamaño variable, pero esto anula completamente el uso de la búsqueda binaria.
    - Puede haber casos en que sea insuficiente
    - Puede haber casos que sobre espacio, provocando fragmentación interna
    - Mejora: clave + lista de punteros con ocurrencias
* ***Listas Invertidas***
  + ***Definición:***
    - Archivos en los que una llave secundaria lleva a un conjunto de una o más claves primarias 🡺 lista de referencias de claves primarias
    - No se pierde espacio (no hay reserva)
    - Si se agrega un elem. a la lista 🡺 no se necesaria una reorganización completa

***Anotaciones de audios***

En caso de uso de claves primarias, la dirección es la física del archivo, el caso de las otras claves, la dirección es de la clave primaria porque esa dirección nos permite ser más ordenados en cuanto a la manera de trabajar, además de más eficiente en el momento donde se trabaja sobre disco.

Además los índices secundarios necesitan agregar un archivo intermedio porque es la única manera de manejar las repeticiones preservando los conceptos de índices primarios. Con esto se logra que funcionen similarmente a los índices primarios, aunque con el uso de un archivo auxiliar, lo que los vuelve mas ineficientes.

* ***Ventajas***
  + El único reacomodamiento en el arch. índice -> al agregar o cambiar un nombre
  + Borrar o añadir grabaciones para un compositor->sólo cambiar el archivo de listas
  + Como el reacomodamiento es a bajo costo se podría almacenar el arch. índice en mem. secundaria , liberando RAM
* ***Desventaja***
  + el arch. de listas es conveniente que esté en memoria ppal. porque podría haber muchos desplazamientos en disco 🡺 costoso si hay muchos índices secundarios

---------------------------------------------------------------------------------------------------

***Clase 6***

***Arboles 🡪 introducción***

* ***Problemas con los índices?***
  + La búsqueda binaria aun es costosa(Log base 2, ineficiente comparado a lo que viene más adelante)
  + Mantener los índices ordenados es costoso(alternativa de subir a memoria RAM como vector, sigue siendo costoso)
  + Solución 🡪 RAM
  + Objetivo 🡪 persistencia de datos (archivos y no vectores en RAM, por lo que la eficiencia baja de RAM nanosegundos, a archivo disco milisegundos). El archivo tiene que tener una estructura más fácil de trabajar = árbol (búsqueda binaria, eficiente, la inserción y la modificación también son relativamente eficientes).
* ***Árboles***
  + Estructuras de datos que permiten localizar en forma más rápida información de un archivo, tienen intrínsecamente búsqueda binaria
* ***Que es un árbol binario?***
  + Estructuras de datos donde cada nodo tiene dos sucesores, a izquierda y a derecha
* ***Un árbol binario, puede implantarse en disco?***
  + Si es posible, vamos a implantar un árbol binario en disco y vamos a convertir una estructura dinámica en memoria RAM, en una estructura dinámica que la manejamos como un archivo para lograr la persistencia.
  + Como lograr la persistencia? (convertir punteros a los hijos, que son direcciones a memoria RAM, en direcciones que podamos manipular dentro de un archivo, y también en disco como memoria persistida.
* ***Como crear un árbol***

***Arboles binarios***

* ***Árbol balanceado***: 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.
* Si mis nodos terminales están todos a la misma distancia del nodo raíz, el árbol esta balanceado. Caso en que un nodo terminal este a una distancia menor o mayor que otro nodo terminal, entonces el árbol deja de estar balanceado.
* ***Tiene eficiencia en log base 2 de n (si esta balanceado)***: son difíciles de mantener en este estado.
  + Si mis nodos terminales están todos a la misma distancia del nodo raíz, el árbol esta balanceado. Si algún nodo terminal está a alguna altura menor o mayor al resto, el árbol esta balanceado.
* Inconveniente de los binarios: se desbalancean fácilmente.
  + Si el árbol no está balanceado, no todos los registros se van a recuperar con la misma eficiencia.
  + El peor caso, es que el árbol se configure como una lista, donde la eficiencia pasaría a ser secuencial.
  + La eficiencia también depende del orden de llegada de las claves, ya una secuencia de llegada inadecuada puede desbalancear el árbol.

***Árboles AVL***

* Árbol binario balanceado en altura (BA(1)) en el que las inserciones y eliminaciones se efectúan con un mínimo de accesos.
* Árbol balanceado en altura:
  + Para cada nodo existe un límite en la diferencia que se permite entre las alturas de cualquiera de los subárboles del nodo (BA(k)), donde k es el nivel de balance)
  + Si yo miro el trayecto más largo hacia un nodo terminal y lo resto al trayecto más corto a otro nodo terminal, la diferencia no puede ser mayor que 1. Si todas las diferencias se mantienen igual a 1, entonces el nodo esta balanceado en altura.
  + Ventaja: al lograr que el árbol se mantenga nivelado, entonces la eficiencia se convierte en algo parecido a eficiencia log en base 2 de n, y un árbol AVL pasa a ser una buena solución porque no se desbalancea mucho.
* **Características/Conclusiones**
  + Se necesitan algoritmos complejos, que se encargan de reacomodar punteros para mantener el orden (acceder a muchos registros de mi archivo de datos, modificando la dirección de los hijos para mantener el balance).
  + El reacomodamiento que genera el algoritmo de un árbol AVL es tal que la cantidad de accesos que tengo que hacer en disco es muy grande.
  + Estructura que debe ser respetada
  + Mantener árbol, rotaciones restringidas a un área local del árbol
    - Binario: 🡪 Búsqueda: Log2(N+1)
    - AVL: 🡪 Búsqueda: 1.44 log2(N+2)
    - Ambas performance por el peor caso posible

***Árboles binarios paginados***

* Problemas de almacenamiento secundario, buffering, páginas de memoria, varios registros individuales, minimiza el número de accesos
* Problema: construcción descendente, como se elige la raíz?, cómo va construyendo balanceado?
* Cuando leo del archivo me traigo mucho registros del árbol, con lo que con pocas operaciones de acceso voy a traer gran parte de los registros que necesito (que caben en un buffer de memoria). La cantidad de accesos a disco depende del tamaño de los registros y del tamaño del buffer. En resumen, con una operación de entrada a salida al disco (1 acceso), me traigo una gran cantidad de registros (x niveles de un árbol).
* Es necesario que todos los elementos de una sección del árbol estén en la misma sección del disco, para que a la hora de hacer un acceso a disco, obtenga la mayoría de ellos.
* Paginados: metidos en recuadros (menos accesos a disco). Es necesario lograr manejarlo de una buena forma, y lograr que sea balanceado.

***Árboles multicamino***

* Generalización de árboles binarios, c/nodo tiene k punteros y k-1 claves (o registros), disminuye la profundidad del árbol,
  + Orden del árbol.
  + Donde de cada padre pueden salir más de 2 hijos

***Arboles balanceados (B)***

Los arboles binarios no son balanceados, los balanceados tampoco van a ser binarios.

* Son árboles multicamino (como mínimo cada nodo va a tener 2 elementos y 3 hijos) con una construcción especial en forma ascendente que permite mantenerlo balanceado a bajo costo.
* En caso de árboles binarios, la construcción es descendente, cada vez que llega un elemento nuevo recorro la raíz hasta un hijo y lo inserto abajo. Esta forma de construcción hace que el árbol se pueda desbalancear fácilmente. Caso contrario, forma de construcción ascendente, cuando quiero insertar un nodo y no hay lugar, hago subir el nivel del árbol, la raíz la hago alejar de los nodos terminales, y se debe mover la raíz, esta se mueve parejo respecto a cada nodo terminal (sube y se aleja en 1 de los nodos terminales).
* Propiedades de un árbol B de orden M (cantidad máxima de hijos por nodo):
  + Ningún nodo tiene más de M hijos
  + C/nodo (menos raíz y los terminales) tienen como mínimo [M/2] hijos
  + La raíz tiene como mínimo 2 hijos (o sino ninguno)
  + Todos los nodos terminales a igual nivel
  + Nodos no terminales con K hijos contienen K-1 registros. Los nodos terminales tienen:
  + Mínimo [M/2] –1 registros
  + Máximo M – 1 registros

***Performance de búsqueda***

* Mejor caso: 1 lectura
* Pero caso: h lecturas (con h altura del árbol)(todos los nodos)
* Cuál es el valor de h?
  + Axioma: árbol balanceado de Orden M, si el número de elementos del árbol es N 🡪 hay N+1 punteros nulos en nodos terminales.
  + **Como se hace la búsqueda:** 
    - Comienzo en el nodo raíz
      * De no estar en la raíz, toma el puntero anterior a la clave mayor. Si bifurca a un puntero nulo es que no existe, de no ser nulo chequea ese nuevo nodo, y si no está ahí recorre el árbol en profundidad.

***Performance de la inserción***

* Mejor caso (sin overflow)
  + H lecturas
  + 1 escritura
* Peor caso (overflow hasta la raíz, aumenta en uno el nivel del árbol)
  + H lecturas
  + 2h+1 escrituras (dos por nivel más la raíz)
* Estudios realizados
  + M = 10 25% divisiones
  + M = 100 2% divisiones

***Eliminación***

* Siempre eliminar de nodos terminales (trabajamos con árboles)(única garantía de que el árbol no se rompe es que cuando borre un elemento esta en un nodo terminal)
* Si se va a eliminar un elemento que no está en nodo terminal 🡪 llevarlo primero a nodo terminal
* Posibilidades ante eliminación
  + Mejor caso: borra un elemento del nodo y no produce underflow, solo reacomodos ( # elementos >= [M/2]-1
  + Peor caso: se produce underflow, #elementos < [M/2] – 1
* Dos soluciones
  + Redistribuir
  + Concatenar
* Definición: nodo adyacente hermano
  + Dos nodos son adyacentes hermanos si tienen el mismo padre y son apuntados por punteros adyacentes en el padre.

***Performance de la eliminación***

* Mejor caso (borra de un nodo Terminal)
  + H lecturas
  + 1 escritura
* Peor caso (concatenación lleva a decrementar el nivel del árbol en 1)
  + 2h – 1 lecturas
  + H + 1 escrituras

***Redistribuir***

* Cuando un nodo tiene underflow puede trasladarse llaves de un nodo adyacente hermano (en caso que este tenga suficientes elementos)
* Cuando un nodo tiene underflow puede trasladarse llaves de un nodo adyacente hermano (en caso que este tenga suficientes elementos)

***Concatenación***

* Si un nodo adyacente hermano está al mínimo (no le sobra ningún elemento) no se puede redistribuir, se concatena con un nodo adyacente disminuyendo el # de nodos (y en algunos casos la altura del árbol)

***Árboles Balanceados 🡪 B\****

* La redistribución podría posponer la creación de páginas nuevas
* Se pueden generar árboles B más eficientes en términos de utilización de espacio
* Definición: Árbol B especial en que cada nodo está lleno por lo menos en 2/3 partes
* Propiedades (orden M)
  + Cada página tiene máximo M descendientes
  + Cada página, menos la raíz y las hojas, tienen al menos [(2M – 1) / 3] descendientes
  + 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
  + Una página hoja contiene por lo menos [(2M – 1) / 3] –1 llaves, y no más de M-1.
* Operaciones de **Búsqueda**
  + Igual que el árbol B común
* Operaciones de **Inserción**
  + Tres casos posible
    - **Derecha**: redistribuir con nodo adyacente hermano de la derecha (o izq. si es el último)
    - **Izquierda o derecha**: si el nodo de la derecha está lleno y no se puede redistribuir, se busca el de la izquierda.
    - **Izquierda y derecha**: busca llenar los tres nodos, estos tendrán un ¾ parte llena.

***Archivos secuenciales indizados***

* Permiten una mejor recorrida por algún tipo de orden
  + Indizado (ordenado por una llave)
  + Secuencial (acceder por orden físico, devolviendo el registro en orden de llave)
* Hasta ahora métodos disjuntos, se opta:
  + rápida recuperación (Árbol)
  + Recuperación ordenada (secuencial)
* Debemos encontrar una solución que agrupe ambos casos
* **Conjunto de secuencias**
  + Conjunto de registros que mantienen un orden físico por llave mientras que se agregan o quitan datos, si podemos mantenerlo podemos indizarlos
* **Posible solución**
  + Mantener bloques de datos
  + Cada bloque con registros y puntero al siguiente
* **Costo**
  + Aumenta el tamaño del archivo (fragmentación interna)
  + No hay orden físico salvo dentro del un bloque.
  + Tamaño del bloque
  + Debe permitir almacenar varios bloques en RAM (redistribucióon)
  + Las E/S deben ser rápidas y sin necesidad de desplazamientos
  + Como logramos ahora una rápida búsqueda?

-------------------------------------------------------------------------------------------------------------------

***Árboles Balanceados 🡪 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 los registros.
* ***Propiedades***
  + Cada página tiene máximo M descendientes
  + Cada página, menos la raíz y las hojas, tienen entre [M/2] y M hijos
  + 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 linkeados juntos.

\*Los nodos no terminales no tienen datos sino punteros a los datos.

* ***Separadores***
  + Derivados de las llaves de los registros que limitan un bloque en el conjunto de secuencia
  + Separadores más cortos, ocupan espacio mínimo
* ***Árbol B+ de prefijos simples***
  + Árbol B+ en el cual el conjunto índice está constituido por separadores más cortos

-------------------------------------------------------------------------------------------------------------------

|  |  |  |
| --- | --- | --- |
|  | ***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*** |

***Conclusiones***

* + Operaciones clásicas
  + Comparaciones

|  |  |  |
| --- | --- | --- |
|  | 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 |

***Clase 7***

***Hashing (Dispersión) 🡪 Introducción***

* Necesitamos un mecanismo de acceso a registros con una lectura solamente
  + Secuencia: N/2 accesos promedio
  + (archivo serie)(acceso dicotómico)Ordenado: Log2 N
  + Árboles: 3 o 4 accesos
* Clave Primarias 🡪 características
  + No se repiten
  + El resto de las claves actúan a través de ella
  + Cuando se aprenda a modelar, tendrán más características que las hacen especiales

***Definición***

* Técnica para generar una dirección base única para una llave dada. La dispersión se usa cuando se requiere acceso rápido a una llave. Esta técnica convierte la llave del registro en un número aleatorio, el que sirve después para determinar donde se almacena el registro.
* También podemos decir que es una técnica de almacenamiento y recuperación, que usa una función de hash para mapear registros en dirección de almacenamiento.

-------------------------------------------------------------------------------------------------------------------

* Archivos secuenciales indizados
  + Archivo de datos
  + Archivo con indice primario
  + Archivos con indices univocos o secundarios
* Archivos directos
  + UN ACCESO
  + No puede haber estructuras adicionales
  + Se organiza EL archivo de datos
  + Solo puede organizarse por un UNICO criterio
  + Ese criterio es la clave primaria

***Atributos del hash***

* No requiere almacenamiento adicional (índice)
* Facilita inserción y eliminación rápida de registros
* Encuentra registros con muy pocos accesos al disco en promedio
* **Como opera:** a través de una clave se genera un número, al cual se le aplica la función de hash para generar una dirección de disco.

***Costo***

* No podemos usar registros de longitud variable
* No puede haber orden físico de datos (de ninguna manera vamos a encontrar una versión de hash que me permita sacar un listado ordenado de los elementos por el criterio de la clave primaria a bajo costo).
* No permite llaves duplicadas (no se podría porque tienen elementos duplicados, por lo que la dirección se podría repetir, además la llaves primarias no admiten resultados)

***Para determinar la dirección***

* La clave se convierte en un número casi aleatorio
* # se convierte en una dirección de memoria
* El registro se guarda en esa dirección
* Si la dirección está ocupada colisión/overflow(tratamiento especial)

-------------------------------------------------------------------------------------------------------------------

***1) Función de hash***

* Caja negra que a partir de una clave se obtiene la dirección donde debe estar el registro (modulo que tiene como entrada la dirección de la clave primaria y como salida una dirección de disco donde se va a guardar el elemento).
* Diferencias con índices
  + Dispersión no hay relación aparente entre llave y dirección(dos claves consecutivas pueden ir a lugares diferentes, porque no sabemos cómo opera la función de hash(caja negra)).
  + 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 un registro es asignado a una dirección que esta utilizada por otro registro y no queda espacio para este nuevo

**Soluciones**

* Algoritmos de dispersión sin colisiones o que estas colisiones nunca produzcan overflow (perfectos) (imposibles de conseguir). En caso de conseguir una, tendríamos un problema adicional, lo que conlleva a que cada uno de mis registros fue un sector de disco diferente, esto genera una gran cantidad de fragmentación.
* Cierto número de colisiones es aceptable para achicar la fragmentación del archivo. Lo ideal sería conseguir una función de hash que no produzca overflow.
* Almacenar los registros de alguna otra forma, esparcir

**Soluciones para las colisiones**

* Esparcir registros: buscar métodos que distribuyan los registros de la forma más aleatoria posible
* Usar memoria adicional: distribuir pocos registros en muchas direcciones, baja la densidad de empaquetamiento:
  + Disminuye el colisiones y por ende overflow
  + Desperdicia espacio
* Colocar más de un registro por dirección: direcciones con N claves, mejoras notables

**Algoritmos simples de dispersión**

* Condiciones
  + Repartir registros en forma uniforme
  + Aleatoria (las claves son independientes, no influyen una sobre la otra)

**Tres pasos**

* Representar la llave en forma numérica (en caso que no lo sea)
* Aplicar la función
* Relacionar el número resultante con el espacio disponible



***2) Tamaño de las cubetas (nodo)***

* Puede tener más de un registro
* A mayor tamaño
  + Menor overflow
  + Mayor fragmentación
  + Búsqueda más lenta dentro de la cubeta (este concepto realmente afecta al problema?)

***3) Densidad de empaquetamiento***

* Proporción de espacio del archivo asignado que en realidad almacena registros
* Cuanto mas alta sea (tiende a 1), tenemos en el archivo tantos registros como el archivo puede contener
* DE = número de registros del archivo (espacio ocupado)  
   capacidad total del archivo (espacio disponible)
  + (valor entre 0 y 1)
* Densidad de empaquetamiento menor
  + Menos overflow
  + Más desperdicio de espacio
* Es necesario analizar el comportamiento de un archivo directo
* Cuando encontrar un registro requiere un solo acceso y cuando requiere más cantidad de accesos
* Estimar el Overflow
  + Analizar probabilísticamente si la inserción de un registro genera o no colisión
  + Analizar si la colisión genera o no overflow
* Es necesario
  + Conocer elementos básicos de probabilidades
  + Vamos a utilizar la distribucion de Poisson

**Estimación del overflow 🡪 sabiendo que**

* N # de cubetas (direcciones)
* C capacidad de nodo (de cada cubeta)
* R # reg. Del archivo
* DE = R / (C x N)
* Probabilidad que una cubeta reciba I registros (distribución de Poisson)



* C +1 = overflow

**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” (por lo que también aumentan los overflows)
  + Ej: N = 10000 R = 10000 DE = 1 100%
  + P(0)= 0.3679 3679 (probabilidad de no recibir registros)  
    P(1)= 0.3679 \* 10000 3679 qué significa? (recibir 1 registro)  
    P(2)= 0.1839 1839 (2 registros)  
    P(3)= 0.0613 613
  + overflow = 1839 + 2 \* 613 = 3065 (alto)

**Ahora supongamos que el problema es**

* R = 500 N= 1000 DE = 50%  
   P(0) = 0.607 607  
   P(1) = 0.303 \* 1000 303  
  saturación = N \* [ 1 \* P(2) + 2 \* P(3) + 3 \* P(4) + 4 \* P(5) ] = 107
* **Saturación menor** densidad overflow  
   10% 4.8%  
   50% 21.4%  
   100% 36.8%
* **los números bajos de overflow (baja densidad)** 🡪 **muchas cubetas libres**

**Resumen:**

* Con valores razonables de C (nodos de capacidad con valor razonable), con densidades de equipamiento razónales, el overflow es muy bajo, la probabilidad vaya a parar a donde debe ir es casi 1. Por ende, con una buena función de hash, una buena densidad de empaquetamiento y un buen amaño de nodo, el overflow se casi anula o achica.

**Tratamiento de Colisiones con Overflow**

* Hemos visto que el % de overflow se reduce, pero el problema se mantiene dado que no llegamos a 0%
* Métodos:
  + Saturación progresiva
  + Saturación progresiva encadenada
  + Doble dispersión
  + Área de desborde separado
* ***Saturación progresiva:***
  + Cuando se completa el nodo, se busca el próximo hasta encontrar uno libre.
  + Búsqueda? => Llega la clave y se le aplica la función de hash, voy a esa dirección, si no está ahí y está lleno el nodo, paso a la siguiente hasta encontrar un nodo que NUNCA se llenó, y no uno no lleno, puesto que puede estar más adelante.
  + Eliminación, no debe obstaculizar las búsquedas
  + Ventajas:
    - Simpleza
    - En el disco, probablemente sectores consecutivos de disco van a tener sectores consecutivos.
  + Desventajas:
    - Puede generar zonas de overflows consecutivas.
    - Búsqueda hay que hacerla aun encontrado un nodo no lleno.
* ***Saturación progresiva encadenada:***
  + similar a saturación progresiva, pero los reg. de saturación se encadenan y “no ocupan” necesariamente posiciones contiguas
  + Utilización de punteros hacia el siguiente nodo que tiene su overflow. Me evita pasar por el nodo saturado, la búsqueda ahora es mas eficiente
* ***Dispersión doble:*** 
  + ***2 funciones de hash***
  + Busca no generar zonas de overflow consecutivas
  + Saturación tiende a agrupar en zonas contiguas, búsquedas largas cuando la densidad tiende a uno
  + Solución almacenar los registros de overflow en zonas no relacionadas (se obtiene usando la segunda función de hash, como desplazamiento desde la anterior).
  + Esquema con el cual se resuelven overflows aplicando una segunda función a la llave para producir un Nº C, el cual se suma a la dirección original tantas veces como sea necesario hasta encontrar una dirección con espacio.
* ***Encadenamiento en áreas separadas:***
  + ***En los casos anteriores:***
    - En caso de un overflow, el nodo se va a ubicar en otro lugar del archivo y va a utilizar otro espacio que potencialmente era de otro registro.
    - Esto último genera ineficiencia, por lo que esta nueva técnica busca resolver esto mediante un área de desborde SEPARADA.
  + No utiliza nodos de direcciones para los overflow, estos van a nodos especiales
  + Cada nodo que va generando un overflow se linkea con un nodo en el área de overflow, donde se guarda el elemento.
  + Ejemplo:
  + Se mejora el tratamiento de inserciones o eliminaciones. Empeora el TAP.
  + Ubicación del desborde
  + A intervalos regulares entre direcciones asignadas
  + Contra: Cilindros de desborde – Donde van a estar los nodos en overflow, por lo que acceder a esos nodos conllevara mayor tiempo promedio de acceso a disco.

---------------------------------------------------------------------------------------------------

***Hash con espacio de direccionamiento estático***

* Necesita un número de direcciones fijas, virtualmente imposible. Cuanto tiene que la cantidad de nodos? En un archivo la cantidad de registros es variable y va a ir creciendo a lo largo del tiempo, la capacidad de cada nodo está determinada por el SO y el tamaño del registro, la cantidad se calcula, pero en algún momento el archivo va a crecer junto a la densidad de empaquetamiento, por lo que hay que generar un nuevo archivo aumentando la capacidad del nodo (política de hash estático).
* Cuando el archivo se llena
  + Saturación excesiva
  + Redispersar, nueva función, muchos cambios
* Solución 🡪 espacio de direccionamiento **dinámico**
  + Reorganizar tablas sin mover muchos registros
  + Técnicas que asumen bloques físicos, pueden utilizarse o liberarse.
  + Cambiar la cantidad de nodos, sin cambiar la función de hash (cambiar la función de hash significaría re dispersar todo el archivo)
* Varias posibilidades
  + Hash virtual
  + Hash dinámico
  + Hash Extensible

En los 3 casos lo que se busca son soluciones donde la cantidad de nodos se maneja dinámicamente y donde la función de hash es la misma.

***Hash Extensible***

* Adapta el resultado de la función de hash de acuerdo al número de registros que tenga el archivo, y de las cubetas necesitadas para su almacenamiento.
* Función: Genera secuencia de bits (normalmente 32)
* Muestra que es posible manejar una cantidad variable de direcciones manteniendo una misma función de hash.
* Este principio viola una de los principios de hashing: en este caso es necesaria una estructura auxiliar (tabla o matriz) que vamos a usar como intermediario para poder usar igual función de hash con cantidades variables de espacio.

**Como trabaja**

* 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.
* La tabla tendrá tantas entradas (direcciones de nodos) como 2i, siendo i el número de bits actuales para el sistema.

-------------------------------------------------------------------------------------------------------------------

**Elección de organización**

Archivos

* Acomodar datos para satisfacer rápidamente requerimientos
* Accesos: resumen

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

No hay método que con hash recupere ordenados de forma eficiente.

**Elección de organización**

* Captar los requerimientos de usuario
* Que examinar
  + Características del archivo
    - Número de registros, tamaño de registros
  + Requerimientos de usuario
    - Tipos de operaciones, número de accesos a archivos
    - Características del hard (como está organizado el disco y como lo maneja el SO)
      * Tamaño de sectores, bloques, pistas, cilíndros, etc.
    - Parámetros
    - Tiempo (necesario para desarrollar y mantener el soft, para procesar archivos)
    - Uso promedio (# reg. Usados/ #registros