Fundamentos de organización de datos, resumen materia

Introducción y definiciones

- SGBD (Sistema de gestión de base de datos): conjunto de programas necesarios para acceder y administrar una BD. Dentro de la SGBD algunos programas son para especificar el esquema de una BD y otros para manipular datos
- Buffer: espacio de memoria en el que se almacenan datos de manera temporal. Normalmente son para un único uso: evitar que el programa o recurso que lo requiere se quede sin datos durante una transferencia de datos irregular o por la velocidad del proceso

---

- Datos: representación de un hecho conocido que puede registrarse y tiene un resultado implícito. Aquellos que necesitan ser preservados más allá de la ejecución de un algoritmo deben residir en archivos
- Base de datos: colección de *datos* relacionados

Las bases de datos tienen propiedades implíticas

- Representa aspectos del mundo real
- Colección coherente de datos con significados inherentes
- Se diseña, construye y completa de datos para un propósito específico
- Sustentada físicamente en archivos en dispositivos de almacenamiento persistentes

---

Almacenamiento secundario: diseñado para almacenar datos a los que no es necesario acceder con frecuencia, como los discos duros

Almacenamiento primario: área de almacenamiento de datos los cuales se acceden con frecuencia, como las memorias RAM

Hay dos diferencias entre el almacenamiento primario y secundario es

- Capacidad: RAM tiene capacidad limitada
- Tiempo de acceso: RAM tiene un acceso en nanosegundos vs. disco rígido que tiene orden de milisegundos

---

Los registros pueden ser de longitud fija cuyo tamaño está determinado por la suma del tamaño de campos que contiene o pueden ser de longitud variable el cual se adapta a las necesidades de cada dato

Archivos

> Un archivo es una colección de registros semejantes guardados en almacenamiento secundario el cual contiene elementos del mismo tipo por lo tanto es una estructura homogénea de datos

A la hora de trabajar con archivos es necesario tener 2 puntos de vistas: la *visión física* y la *visión lógica*. La física se refiere a que un archivo está almacenado en memoria secundaria y el SO se ocupa de su administración. El punto de vista lógico se refiere a una referencia del archivo el cual se encuentra en la memoria RAM y por lo tanto los algoritmos no interactúan directamente con el archivo de memoria secundaria

Entonces a partir de esto se trabajan con *archivos físicos* y *archivos lógicos.* El archivo físico reside en memoria segundaria y lo administra el SO y el archivo lógico es usado desde el algoritmo, lo cual se conoce como *independencia física*

Forma de acceso

- Secuencial: el acceso a cada elemento se realiza luego de haber accedido a su anterior
- Secuencial indizado: acceso a los registros de acuerdo al orden establecido por otra estructura

- Directo: accedo a un registro determinado sin necesidad de haber accedido a los predecesores

Organización de un archivo

1. Secuencia de bytes: como archivos de texto que se leen/ recuperan caracteres el cual no tienen un formato previo
2. Conjunto de registros: tienen una estructura y pueden estar conformados por campos
    - Registro: conjunto de campos agrupados que definen un elemento del archivo
    - Campo: unidad más pequeña lógicamente significativa de un archivo

- Archivos físicos: aparecen en el disco rígido y están a cargo del sistema operativo
- Archivos lógicos: se manipulan desde el lenguaje de programación

Operaciones sobre archivos

A la hora de manipular los archivos, las operaciones principales que podré hacer serán: definir, abrir para operar, leer, escribir, controlar el fin, ir a un lugar específico, saber la cantidad de registros, saber la posición de trabajo actual, etc.

Definición/ declaración (usando Pascal)

```
type emple= record
      nombre: string[50];
      direccion: string[40];
      edad: integer;
    end;
    numero= file of integer;
    empleado= file of emple;
var arch_num: numero;
var arch_emp: empleado;
```

Relacionar archivo físico con el lógico

```
begin
  assign(arch_num,'pepe.dat');
  assign(arch_emp, 'pipo.dat'); {donde 'pipo.dat' se refiere al archivo físico}
```

Funciones necesarias

```
rewrite(nombre_logico); {crea el archivo}
reset(nombre_logico); {abre el archivo}
close(nombre_logico); {cierra el archivo e indica que no se va a trabajar más con
                       el archivo. Significa poner una marca de EOF al final del mismo}
EOF(nobre_logico); {boolean que indica si estoy al final del archivo}
FilseSize(nombre_logico); {función que devuelve el tamaño del archivo}
FilePos(nombre_logico); {indica en qué posición estoy del archivo}
Seek(nombre_logico, posicion); {va a una posición del archivo, el índice empieza en 0}
```

Buffers de memoria

Las lecturas y escrituras desde o hacia un archivo se realizan sobre buffers. Se denomina buffers a una memoria intermedia entre un archivo y un programa, donde los datos residen provisoriamente hasta ser almacenados. La justificación anterior se debe a cuestiones de performance. Una operación sobre RAM tiene una demora del orden de nanosegundos y una operación sobre disco tiene una demora del orden de milisegundos. Por lo tanto, varias operaciones de lectura y/o escritura directamente sobre la memoria secundaria reducirían notablemente la performance de un algortimo. Por este motivo, el sistema operativ, al controlar dichas operaciones, intenta minimizar el tiempo requerido para estas.

La operación *read* lee desde un buffer y, en caso de no contar con información, el sistema operativo realiza automáticamente una operación *input,* trayendo más información al buffer. La diferencia radica en que cada operación *input* transfiere desde el disco una serie de registros. De esta forma, cada determinada cantidad de instrucciones *read,* se realiza una operación *input*. Cada *read* se mide en nano segundos, en tanto que los *input* se miden en milisegundos, pero se realizan con menor frecuencia

Algoritmia clásica sobre archivos

- Archivo maestro: archivo que resume información sobre un dominio de problema específico
- Archivo detalle: contiene novedades o movimientos realizados sobre la información almacenada en el maestro

Un ejemplo sería tener un archivo maestro el cual tiene información de todos los productos de un supermercado y tengo un archivo detalle que detalla todas las ventas realizadas un determinado día

Actualización de un archivo maestro con 1 archivo detalle

Será necesario tener en cuenta las siguientes precondiciones

1. Hay un archivo maestro
2. Existe un único archivo detalle que modifica al maestro
3. Cada registro del detalle modifica a un registro del maestro
4. No todos los registros del maestro son necesariamente modificados
5. Cada elemento del maestro que se modifica es alterado por uno y solo un elemento del archivo detalle
6. Ambos archivos están ordenados por el mismo criterio

```
program maestro_1detalle;
type
  producto= record
    cod:str4;
    descripcion:string[20];
    pu:real;
    stock:integer;
  end;
  venta_prod=record
    cod:str4;
    cant_vendida:integer;
  end;
  detalle= file of venta_prod;
  maestro= file of producto;
var
  regm:producto;
  regd:venta_prod;
  mae:maestro;
  det:detalle;
begin
  assign(mae,'maestro');
  assign(det,'detalle');
  reset(mae);
  reset(det);
  while (not eof(det)) do begin
    read(mae,regm);
    read(det,regd);
    {busca en el maestro el producto del detalle}
    while (regm.cod <> redg.cod) do
      read(mae,regm);
    {modifica el stock del producto con la cantidad vendida de ese producto}
    regm.stock:=regm.stock-regd.cant_vendida;
    {se reubica el puntero en el maestro}
    seek(mae,filepos(mae)-1);
    {se actualiza el maestro}
    write(mae,regm);
  end;
  close(det);
  close(mae);
end.
```

En caso de tener en el repeticiones de un producto en el archivo detalle, podré hacer un “merge” y luego actualizar, sería

```
program maestro_1detalle;
type
  producto= record
    cod:str4;
    descripcion:string[20];
    pu:real;
    stock:integer;
  end;
```

```
  venta_prod=record
    cod:str4;
    cant_vendida:integer;
  end;
  detalle= file of venta_prod;
  maestro= file of producto;
var
  regm:producto;
  regd:venta_prod;
  mae:maestro;
  det:detalle;
  cod_actual:str4;
  tot_vendido:str4;
begin
  assign(mae,'maestro');
  assign(det,'detalle');
  reset(mae);
  reset(det);
  while (not eof(det)) do begin
    read(mae,regm);
    read(det,regd);
    {busca en el maestro el producto del detalle}
    while (regm.cod <> redg.cod) do
      read(mae,regm);
    {se totaliza la cantidad vendida del detalle}
    cod_actual:=regd.cod;
    tot_vendido:=0;
    while (regd.cod=cod_actual) do begin
      tot_vendido:=tot_vendido+regd.cant_vendida;
      read(det,regd);
    end;

    {modifica el stock del producto con la cantidad vendida de ese producto}
    regm.stock:=regm.stock-regd.cant_vendida;
    {se reubica el puntero en el maestro}
    seek(mae,filepos(mae)-1);
    {se actualiza el maestro}
    write(mae,regm);
  end;
  close(det);
  close(mae);
end.
```

Actualización de un archivo maestro con N archivos detalle

Se plantea un proceso de actualización donde la cantidad de archivos detalle es N, donde N > 1 y el resto de las precondiciones son las mismas

```
program mergeRepetidos;
  const valoralto = '9999';
  type vendedor = record
         cod: string[4];
         producto: string[10];
         montoVenta: real;
       end;
       ventas = record
         cod: string[4];
         total: real;
       end;
     maestro = file of ventas;
     arc_detalle=array[1..100] of file of vendedor;
     reg_detalle=array[1..100] of vendedor;
  var min: vendedor;
      deta: arc_detalle;
      reg_det: reg_detalle;
      mae1: maestro;
      regm: ventas;
      i,n: integer;


procedure leer (var archivo:detalle; var dato:vendedor);
    begin
      if (not eof( archivo ))
        then read (archivo, dato)
        else dato.cod := valoralto;
    end;

procedure minimo (var reg_det: reg_detalle; var min:vendedor; var deta:arc_detalle);
    var i,minimo,indiceMin: integer;
    begin
```

```
      { busco el mínimo elemento del
        vector reg_det en el campo cod,
        supongamos que es el índice i }
      minimo:= valoralto;
      indiceMin:=-1;
      for i:=1 to n do begin
        if (reg_det[i].cod <minimo) then begin
          minimo:=reg_det[i].cod;
          indiceMin:=i;
        end;
      end;
      min = reg_det[i]; // guardo el minimo del embudo
      leer( deta[i], reg_det[i]; //leo y así actualizo registro de reg_deta
    end;


begin
  Read(n)
  for i:= 1 to n do begin
    assign (deta[i], 'det'+i);
    { ojo lo anterior es incompatible en tipos}
    reset( deta[i] );
    leer( deta[i], reg_det[i] );
  end;
  assign (mae1, 'maestro'); rewrite (mae1);
  minimo (reg_det, min, deta);
  while (min.cod <> valoralto) do begin
       regm.cod := min.cod;
       regm.total := 0;
       while (regm.cod = min.cod ) do begin
         regm.total := regm.total+
                       min.montoVenta;
         minimo (regd1, regd2, regd3, min);
       end;
       write(mae1, regm);
     end;
 end.
```

Proceso de baja

Proceso que permite quitar información de un archivo. Puede llevarse a cabo de 2 maneras

Baja física

Consiste en borrar efectivamente la información del archivo, recuperando el espacio físico

Un elemento es quitado del archivo y es reemplazado por otro elemento del mismo archivo decrementando en uno su cantidad de elementos. La ventaja de este método consiste en administrar un archivo que ocupe el lugar mínimo necesario pero su desventaja tiene que ver con la performance final de los algoritmos que implementen esta solución. Hay 2 técnicas

- Generar un nuevo archivo con los elementos válidos

```
program nuevoArchivo;
const valoralto='ZZZ';
type
  empleado=record
  nombre:string[50];
  direccion:string[50];
  edad:integer;
  observaciones:string[200];
  end;
archivo_empleado=file of empleado;

var
  reg:empleado;
archivo,archivoNuevo:archivo_empleado;
procedure leer(var archivo:archivo_empleado; var dato:empleado);
begin
  if (not eof(archivo)) then
    read(archivo,dato);
  else
    dato.nombre:valoralto;
end;

{programa principal}
```

```
  begin
    assign(archivo,'archivo.txt');
    assign(archivoNuevo,'nuevo.txt');
    reset(archivo);
    rewrite(archivoNuevo);
    leer(archivo,reg);

  {se copian los registros previos a Carlos Garcia}
  while (reg.nombre <>"Carlos García") begin
    write(archivoNuevo,reg);
    leer(archivo,reg);
  end;
  {se descarta a Carlos Garcia}
  leer(archivo,reg)

  {se copian los registros restantes}
  while (reg.nombre<>valorAlto) do begin
    write(archivoNuevo,reg);
    leer(archivo,reg);
  end;
  close(archivoNuevo);
  close(archivo);
  end.
```

En resumen, el algoritmo consiste en recorrer el archivo original de datos copiando al nuevo archivo toda la información menos la correspondiente al elemento que se desea sacar. Finalizado el proceso, se debe eliminar el archivo inicial del dispositivo de memoria secundaria

Según su performance, se determina que este método necesita leer tantos datos como tenga el archivo original y escribir todos los datos. Suponiendo que el archivo tiene N registros, habrá N lecturas y N-1 escrituras. Debe tenerse en cuenta que finalizado el proceso de generación, coexistirán dos archivos en el disco por lo tanto la memoria secundaria debe tener capacidad para ambos

- Usar el mismo archivo de datos, generando los reacomodamientos necesarios: el algoritmo requiere localizar el dato a eliminar, copiar sobre este registro el elemento siguiente y así sucesivamente. Según performance, la cantidad de lecturas a realizar es N, en tato que la cantidad de escrituras dependerá del lugar donde se encuentre el elemento a borrar. A diferencia del método anterior, no es necesario contar con mayor capacidad en el disco rígico

Baja lógica

Consiste en que el elemento que se desea quitar es marcado como borrado pero sigue ocupando espacio dentro del archivo. La ventaja tiene que ver con la performance, basta con localizar el registro a eliminar y colocar sobre él una marca de borrado. Su desventaja es que al no recuperarse el espacio borrado, el tamaño del archivo tiene a crecer continuamente.

```
program borradoLogico;
const valorAlto='ZZZ';
type
  empleado=record
    nombre:string[50];
    dirección:string[50];
  end;
  archivoemple=file of Empleado;
var
  reg:empleado;
  archivo:archivoemple;
procedure leer(var archivo:archivoemple; var dato:empleado);
begin
  if (not eof(archivo)) then
    read(archivo,dato);
  else
    dato.nombre:=valorAlto;
end;

{programa principal}
begin
  assign(archivo,'archivoemple');
  reset(archivo);
  leer(archivo,reg);
  {se avanza hasta Carlos García}
  while (reg.nombre <>"Carlos García") do begin
    leer(archiv,reg);
  end;
```

```
  {se genera una marca de borrado}
  reg.nombre:="*";
  {se borra lógicamente a Carlos García}
  seek(archivo,filepos(archivo)-1);
  write(archivo,reg);
  close(archivo);
end.
```

Recuperación de espacio

El proceso de baja marca la información de un archivo como borrada pero esa información sigue ocupando espacio en el disco rígido. Hay dos posibles opciones para resolver ese problema

- Recuperación de espacio: periódicamente usar el proceso de baja física para realizar un proceso de compactación del archivo
- Reasignación de espacio: otra alternativa posible consiste en recuperar el espacio usando los lugares indicados como borrados para el alta de nuevos elementos del archivo

  Esta técnica consiste en reutilizar el espacio indicado como borrado para que nuevos registros se inserten en dichos lugares

  Este proceso puede realizarse buscando los lugares libres desde el comienzo del archivo, pero se debe considerar que de esa manera sería muy lento. La alternativa consiste en recuperar el espacio de forma eficiente. A medida que los datos se borran del archivo, se genera una lista encadenada invertida con las posiciones borradas. A medida que quitan elementos, este proceso se reitera en la lista de elementos borrados que se va construyendo.

  Para recuperar el espacio, el proceso de alta debe, primeramente verificar la lista por espacios libres. En caso de encontrar algún espacio libre, el registro se deberá insertar en esa posición, actualizando el registro cabecera.

  Si el archivo no dispone de lugares para reutilizar entonces el registro cabecera no tiene ninguna dirección válida. En ese caso, para ingresar un nuevo elemento se debe acceder a la última posición del archivo.

Alta en lista enlazada

```
var
  flor,aux:registro;
begin
  flor.cod:=cod;
  reset(arch);
  read(arch,aux);
  if aux.cod < 0 then begin //significa que hay algun registro que borré y puedo reutilizarlo
    seek(arch, Abs(aux.cod)); //función para el absoluto, si no hago (aux.cod * -1)
    read(arch,aux); //leo lo que tiene esa posicion para despues llevarlo a head
    seek(arch,filepos(arch)-1);
    write(arch,flor); //inserto el nuevo elemento;
    seek(arch,0); //vuelvo a head
    write(arch,aux); //escribo dato que había en la posición donde inserté
  end
  else //en caso de que aux.cod > 0 significa que no hay un espacio libre por lo tanto tengo que insertar al final del archivo
  begin
    seek(arch,filesize(arch);
    write(arch,flor);
  end;
end;
```

Campos y registros con longitud variable

La información de un archivo siempre es homogénea. De esta forma, cada uno de los datos es del mismo tamaño, generando lo que se denomina archivos con registros de longitud fija. La longitud de cada registro está detemirnada por la información que guarda.

Administrar archivos con **registros de longitud fija** tiene algunas ventajas como, el proceso de entrada y salida de información desde y hacia los buffers es responsabilidad del sistema operativo, los procesos de alta, baja y modificación se corresponden con todo lo visto hasta el momento. Pero hay determinados problemas donde no es posible o no es deseable usar registros de longitud fija.

Hay que tener en cuenta que con los registros de longitud fija hay desperdicio de espacio y mayor tiempo de procesamiento. Es de interés contar con alguna organización de archivos que solo use el espacio necesario para almacenar la información. Es necesario entonces definir el archivo de otra forma, para ello existen diferentes soluciones.

En un archivo con **registros de longidud variable** necesitaremos saber dónde empieza y termina cada registro o campo del archivo. Entonces, es necesario colocar marcas que delimiten cada elemento. Estas marcas se denominan *marcas de fin de registro* y pueden ser cualquier carácter que luego no deberá poder ser parte de un elemento de dato del archivo

Otra forma de manipular los registros de longitud variable será usar indicadores de longitud de campo y/o de registro. Antes de almacenar un registro, se indica su longitud; luego, los siguientes bytes corresponden a elementos de datos de dicho registro.

Eliminación con registros de longitud variable

El proceso de baja lógica no tiene diferencias sustanciales con respecto a lo discutido anteriormente. Cuando se desea recuperar el espacio borrado lógicamente con nuevos elementos, deben tenerse en cuenta nuevas consideraciones. Tienen que ver con el espacio disponible. Para insertar un elemento no basta con disponer de lugar, es necesario además que el lugar sea del tamaño suficiente. Para ello se genera una lista invertida donde a partir de un registro cabecera se dispone de las direcciones libres dentro del archivo. Es necesario ahora indicar, además, la cantidad de bytes disponibles en cada caso para su reutilización.

Debe localizarse el lugar más adecuado dentro del archivo para el nuevo elemento. Hay 3 formas genéricas para la selección del espacio

1. Primer ajuste: selecciono el primer espacio disponible donde quepa el registro a insertar
2. Segundo ajuste: selecciona el espacio más adecuado para el registro. Se considera el más adecuado como aquel de menos tamaño donde quepa el registro
3. Tercer ajuste: consiste en seleccionar el espacio de mayor tamaño asignando para el registro solo los bytes necesarios

Fragmentación

- FRAGMENTACIÓN INTERNA: se produce cuando a un elemento de dato se le asigna mayor espacio del necesario. Se genera por reservar lugar que no se usa

Los registros de longitud fija tienden a generar fragmentación interna, se asigna tanto espacio como lo necesario de acuerdo con la definición del tipo de dato. Este espacio no siempre se condice con lo que realmente usa el registro

- FRAGMENTACIÓN EXTERNA: espacio disponible entre dos registros pero que es demasiado pequeño para poder ser reutilizado. Se genera por dejar espacios tan pequeños que no pueden ser usados.

La fragmentación tiene que ver con la utilización de espacio.

Las técnicas de primer y mejor ajuste suelen imprmentar una variante que genera fragmentación interna y la técnica de peor ajuste solo asigna el espacio necesario

Las 3 técnicas definidas tienen sus ventajas y desventajas. El primer ajuste tiende a ser el más rápido porque solo debe recorrerse la lista de registros borrados hasta encontrar el primer lugar donde quepa el nuevo elemento de datos. Dicho lugar se quita de la lista, asignando todo el espacio disponible al nuevo elemento. Las alternativas de mejor y peor ajuste, en comparación con primer ajuste son mas ineficientes. En ambos, es necesario recorrer toda la lista para encontrar el lugar más adecuado según su caso. Una vez localizado dicho lugar, mejor ajuste se resuelve con mayor rapidez.

Modificación de datos con registros de longitud variable

Cuando se trabaja con archivos con registros de longitud fija, una modificación consiste en sobreescribir un registro con el nuevo dato. Cuando los archivos soportan registros de longitud variable, surge un nuevo problema. Modificar un registro existente puede significar que el nuevo registro requiera el mismo espacio en disco, que ocupe menos espacio o requiera uno de mayor tamaño.

El problema surge cuando el nuevo registro ocupa mayor espacio que el anterior. En este caso, no es posible usar el mismo espacio físico y el registro necesita ser reubicado.

Para evitar todo este análisis y para facilitar el algoritmo de modificación sobre archivos con registros de longitud variable, se estila dividir el proceso de modificación en dos etapas: en la primera se da de baja al elemento de dato viejo, mientras que en la segunda etapa el registro es insertado de acuerdo con la política de recuperación de espacio determinada

Manejo de índices

Proceso de búsqueda

Cuando se realiza la búsqueda de un dato, se debe considerar la cantidad de accesos a disco en pos de encontrar esa información, y en cada acceso, la verificación de si el dato obtenido es el buscado. Surgen dos parámetros a analizar: cantidad de accesos y cantidad de comparaciones. El primero es una operación sobre memoria secundaria (implica un costo alto) y el segundo es sobre memoria principal, por lo que el costo es relativamente bajo).

El mejor caso es ubicar el dato deseado en el primer registro (1 lectura), y el peor caso, en el útlimo registro (N lecturas, siendo N la cantidad de registros). El caso promedio consiste entonces en realizar N/2 lecturas. Es decir, la performance depende de la cantidad de registros del archivo, siendo N el orden. En conclusión, mejor caso= 1 acceso, peor caso N accesos y caso promedio N/2 accesos

Ordenamiento de archivos

Si el archivo a ordenar puede almacenarse en forma completa en memoria RAM, una alternativa es

- Trasladar el archivo completo desde memoria secundaria hasta principal y luego ordenarlo allí. Luego, la ordenación efectuada en memoria principal será realizada con alta performance Esta opción solo puede ser usada en archivos pequeños

En caso de que no quepa en memoria RAM puedo

- Transferir a memoria principal solo la clave por la cual se ordena cada registro del archivo, junto con los NRRs que se encuentran en memoria secundaria. De esa forma, al transferir solo estos datos, es posible almacenar mayor cantidad de registros en RAM y así podré ordenar un archivo de mayor tamaño.

  El algoritmo solo ordena en memoria principal las claves. Posteriormente, se debe leer nuevamente cada registro del archivo y escribirlo sobre el archivo ordenado. Implica muchos desplazamientos en memoria secundaria entonces es alto el costo a pagar para poder usar la memoria principal como medio de ordenación

Esta tercera opción surge cuando el archivo es realmente grande, de modo tal que las claves no se pueden ordenar en memoria principal porque no entran. La estrategia será así

1. Dividir el archivo en particiones de igual tamaño, de modo tal que cada partición quepa en memoria principal
2. Transferir las particiones, de a una, a la memoria principal
3. Ordenar cada partición en memoria principal y reescribirlas ordenadas en memoria secundaria
4. Realizar un merge de las particiones, generando un nuevo archivo ordenado

Indización

El propósito de ordenar un archivo radica en tratar de minimizar los accesos a memoria secundaria durante la búsqueda de información. Con el uso de una fuente de información organizada adicional, se puede lograr mejorar la performance de acceso a la información deseada.

Un **índice** es una estructura de datos adicional que permite agilizar el acceso a la información almacenada en un archivo. En dicha estructura se almacenan las claves de los registros del archivo, junto a la referencia de acceso a cada registro asociadoa la clave.

Esta estructura de datos es otro archivo con registros de longitud fija. La característica fundamental de un índice es que posibilita imponer orden en un archivo sin que realmente este se reacomode

El índice es un nuevo archivo ordenado, con registros de longitud fija, con la diferencia de que contiene solo un par de datos del archivo original

Una vez disponible el índice, la búsqueda de un dato se realiza primero en el índice, de allí se obtiene la dirección efectiva del archivo y luego se accede directamente a la información buscada.

Se tratará el caso del índice creado a partir de la clave primaria, denominado índice primario.

Al crearse el archivo, con clave primaria formada por los atributos, se crea el índice primario asociado. Dado que el índice es un archivo de registros con longitud fija, es posible realizar sobre este una búsqueda binaria. Hallada la referencia correspondiente a la dirección del primer byte del registro buscado, se accede directamente al archivo de datos según lo indicado en dicha referencia.

Resumiento, se realiza una búsqueda con potencial bajo costo en el índice y luego un acceso directo al archivo de datos, por lo que la cantidad de accesos a memoria secundaria está conficionada por la búsqueda del índice

Creación de índice primario

Al crearse el archivo, se crea también el índice asociado, ambos vacíos, solo con el registro encabezado

Altas en índice primario

La operación de alta de un nuevo registro al archivo de datos consiste simplemente en agregar dicho registro al final del archivo. Con el NRR o la dirección del primer byte más la clave primaria, se genera un nuevo registro de datos a insertar en forma ordenada en el índice.

Modificaciones en índice primario

Si el archivo está organizado con registros de longitud fija, el indice no se altera.

Si el archivo está organizado con registros de longitud variable y el registro modificado no cambia de longitud, nuevamente el índice no se altera. Si el registro modificado cambia de longitud, particularmente agrandando su tamaño, este debe cambiar de posición

Bajas en índice primario

Eliminar un registro del archivo de datos implica borrar la información asociada en el índice primario. Se debe borrar física o lógicamente el registro correspondiente en el índice

Ventajas del índice primario

Al ser de menor tamaño que el archivo asociado y tener registros de longitud fija, posibilita mejor la performance de búsqueda. Además se pueden realizar búsquedas binarias.

Índices para claves candidatas

Las **claves candidatas** son claves que no admiten repeticiones de valores para sus atributos, similares a una clave primaria, pero que por cuestiones operativas no fueron seleccionadas como clave primaria

Índices secundarios

Se denomina **clave secundaria** a aquellas que soportan valores repetidos. Es necesario crear otro tipo de índice mediante el cual se pueda acceder a la información de un archivo, pero con datos fáciles de recordar.

Un **índice secundario** es una estructura adicional que permite relacionar una clave secundaria con una o más claves primarias, dato que, como se mencionó, varios registros pueden contener la misma clave secundaria

Creación de un índice secundario

Al impantarse el archivo de datos, se deben crear todos los índices secundarios asociados

Altas en índice primario

Cualquier alta en el archivo de datos genera una inserción en el índice secundario, que implica reacomodar el archivo en el cual se almacena.

Modificación en índice secundario

Tendré dos alternativas

1. Ocurre cuando se produce un cambio en la clave secundaria, en este caso se debe reacomodar el índice secundario
2. Ocurre cuando cambia el resto del registro y no generará cambios en el índice secundario

Bajas en el índice secundario

Cuando se elimina un registro del archivo de datos, esta operación implica eliminar la referencia a ese registro del índice primario más todas las referencias en índices secundarios. La eliminación a realizarse en el índice secundario, almacenado en un archivo con registros de longitud fija, puede ser física o lógica

Alternativas de organización de índices secundarios

La organización de índices presentada hasta el momento consiste en almacenar la misma clave secundaria en distintos registros, tantas como ocurrencias haya. Esto implica mayor espacio, generando una menor posibilidad de que el índice quepa en memoria

---

Árboles

El principal problema para poder administrar un índice en disco rígido lo representa la cantidad de accesos necesarios para recuperar la información. Como fue visto, el proceso de búsqueda es costoso y el mantenimiento, altas, bajas y modificaciones también debe ser considerado como parte de actualización del archivo de datos.

Las estructuras tipo árbol presentan algunas mejoras tanto para la búsqueda como para el mantenimiento del orden de la información

Los **árboles binarios** representan la alternativa más simple para la solución de este problema

Árbol binario

Estructura de datos dinámica no lineal, en la cual cada nodo puede tener a lo sumo dos hijos y tiene sentido cuando está ordenado

La búsqueda se realiza a partir del nodo raíz y se recorre. Se chequea un nodo, si es el deseado, la búsqueda finaliza o se decide si la búsqueda continúa a izquierda o derecha descartando la mitad de los elementos restantes. La búsqueda de información tiene orden logarítmico

Una ventaja de la organización mediante árboles binarios está dada en la inserción de nuevos elementos. Mientras que un archivo se desordena cuando se agrega un nuevo dato, la operatoria resulta más sencilla en términos de complejidad computacional. La secuencia de pasos para insertar un nuevo elemento es la siguiente

1. Agrega el nuevo elemento de datos al final del archivo
2. Buscar al padre de dicho elemento
3. Actualizar el padre, haciendo referencia a la dirección del nuevo hijo

Se hacen log2(N) lecturas para localizar el padre del nuevo elemento y dos operaciones de escritura (el nuevo elemento y la actualización del padre)

Para quitar un elemento de un árbol, este debe ser necesariamente un elemento terminal. Si no lo fuera, debe intercambiarse el elemento en cuestión con el menor de sus hijos mayores.

En conclusión los árboles binarios representan una buena elección en términos de inserción y borrado de elementos

Se entiende por **árbol balanceado** a aquel árbol donde la trayectoria de la raíz a a cada una de las hojas está representada por igual cantidad de nodos. Es decir, todos los nodos hoja se encunetran a igual distancia del nodo raíz

Árboles AVL

Los árboles balanceados en altura son árboles binarios cuya construcción se determina respetando un precepto muy simple: la diferencia entre el camino más corto y el más largo entre un nodo terminal y la raíz no puede diferir en más de un determinado delta y dicho delta es el nivel de balanceo en altura del árbol.

Un **árbol AVL** es un árbol balanceado en altura donde el delta determinado es uno, es decir, el máximo desbalanceo es uno. Sin embargo, cunado se intenta insertar un elemento, se viola el precepto establecido. Se

debe llevar a cabo un rebalanceo de este árbol. Los costos computacionales de acceso a disco aumentan considerablemente, por lo que su implantación deja de ser viable

Paginación de árboles binarios

Cuando se transfiere desde o hacia el disco rígido, dicha transferencia no se limita a un registro, sino que son eviados un conjunto de registros, es decir los que quepan en un *buffer* de memoria. Este concepto es de utilidad cuando se genera el archivo que contiene el árbol binario. Dicho árbol se divide en paginas

Cuando se transfieren datos, no se accede al disco para transferir unos pocos bytes, sino que se transmite una página completa.

Este tipo de organización reduce el número de accesos a disco necesarios para poder recuperar la información. Esto es considerado que para transferir una página completa, si bien se accede a direcciones físicas, diferentes, tales direcciones son cercanas y los desplazamientos realizados apra su acceso son despreciables.

Al dividir un árbol binario en páginas, es posible realizar búsquedas más rápidas de datos almacenados en memoria secundaria

Árboles multicamino

Los árboles binarios son un tipo de estructura de datos que resulta muy simple. Se podría definir a un árbol ternario como aquel en el cual cada nodo puede tener cero, uno, dos o tres hijos. Un árbol ternario es similar a uno binario, solo que cada nodo tiene la posibilidad de poseer hasta dos elemetnos y hasta tres hijos

Un **árbol multicamino** es una estructura de datos en la cual cada nodo puede contener k elementos y k+1 hijos. El *orden de un árbol multicamino* es la máxima cantidad de descendientes posibles de un nodo.

Así, si el orden de un árbol binario es de dos, el orden de un árbol ternario será tres y, en general, un árbol multicamino de orden M contendrá un máximo de M descendientes por nodo. Un árbol multicamino representa otra forma de resolver el concepto de página vertido anteriormente

Árboles B

Los **árboles B** son árboles multicamino con una construcción especial que permite mantenerlos balanceados a bajo costo y tienen las siguientes propiedades básicas

1. Cada nodo del árbol puede contener, como máximo, M descendientes y M-1 elementos. Entonces, establece que si el orden de un árbol B es por ejemplo 256, indica que la máxima cantidad de descendientes de cada nodo es de 256
2. La raíz no posee descendientes directos o tiene al menos 2. Esto posibilita que la raíz no disponga de descendientes pero cuando sea necesario deberá contar con dos, como mínimo
3. Un nodo con x descendientes directos contiene x-1 elementos. Determina que cualquier nodo que posea una cantidad determinada de hijos, por ejemplo 256, necesariamente debe contener 255 elementos
4. Los nodos terminales, tienen como mínimo M/2-1 elementos y como máximo M-1 elementos. Establece la mínima cantidad de elementos contenidos en un nodo terminal. Al ser M el orden del árbol, la máxima cantidad admisible de elementos será M-1 pero, además la mínima cantidad de elementos es uno menos que la parte entera de la mitad del orden. Así si 256 fuera el orden establecido, un nodo terminal deberá poseer 127 elementos como mínimo y 255 como máximo
5. Los nodos que no son terminales ni raíz tienen como mínimo M/2 elementos. Implica una restricción parecida a la anterior: cualquier nodo que no sea terminal o raíz debe tener una cantidad mínima de descendientes. Siguiendo con un árbol de orden 256, cada nodo deberá tener un mínimo de 128 descendientes
6. Todos los nodos terminales se encuentran al mismo nivel. Esta propiedad establece el balanceo del árbol. la distancia desde la raíz hasta cada nodo terminal debe ser la misma

   CREACIÓN DE ÁRBOLES B

   Un árbol balanceado tiene como principal característica que todos los nodos terminales se encunetran a la misma distancia del nodo raíz. No es posible concebir un árbol B de la misma forma que un árbol tradicional. Cuando no se dispone de suficiente espacio, es la raíz la que debe alejarse de los nodos terminales.

Si el nodo raíz tiene lugar, se procederá a insertarlos en esto nodo teniendo en cuenta que los elementos de datos deben quedar ordenados dentro del nodo. La llegada de un nuevo elemento provocará un *overflow.* Este overflow significa que en el nodo no hay capacidad disponible para almacenar un nuevo elemento de datos. Ante la ocurrencia de un overflow el proceso es el siguiente

1. Se crea un nodo nuevo
2. La primera mitad de las claves se mantienen en el nodo viejo
3. La segunda mitad de las claves se trasladan al nodo nuevo
4. La menor de las claves de la segunda mitad se promociona/ asciende al nodo padre

BÚSQUEDA EN UN ÁRBOL B

El proceso de búsqueda coincide con la primera parte del proceso mostrado. El primer paso en la inserción consiste en localizar el nodo que debería contener al nuevo elemento. Comenzando desde el nodo raíz, se procede a buscar el elemento en cuestión. En caso de encontrarlo en dicho nodo, se retorna una condición de éxito. Si no se encuentra, se procede a buscar en el nodo inmediato siguiente que debería contener al elemento

La *eficiencia de búsqueda en un árbol B* consiste en contar los accesos al archivo de datos, que se requieren para localizar un elemento o para determinar que el elemetno no se encuentra. El resultado es un valor acodado en el rango entero [1, H] siendo H la altura del árbol tal como fuera definida previamente. Si el elemento se encuentra ubicado en el nodo raíz, la cantidad de accesos requeridos es 1. En caso de localizar al elemento en un nodo terminal, serán requeridos H accesos

En resumen, la performance de la inserción en un árbol B está compuesta por H lecturas, 1 escritura (en el mejor caso) o H lectuturas y (2*H)+1 escrituras en el peor caso

ELIMINACIÓN EN ÁRBOLES B

Para poder borrar un elemento debe estar lcoalizado en un nodo terminal. Si se desea eliminar un elemento el cual no está en una hoja, deberá intercambiarse con el elemento más extremo de la hoja correspondiente.

La segunda situación posible es aquella que genera una situación de *underflow.* En este caso, el nodo al que se quita un elemento deja de cumplir la condición de contener al menos [M-2]-1 elementos. A fin de poder ilustrar esta situación, es necesario rpesentar algunas definiciones extras.

Se denomina **nodos hermanos** a aquellos nodos que tienen el mismo nodo padre

Se denomina **nodos adyacentes hermanos** o **nodos hermanos adyacentes** a aquellos nodos que, siendo hermanosm son además dependientes de punteros consecutivos del padre

CONCLUSIONES ÁRBOLES B

Los árboles balanceados representan una buena solución como estructura de datos, cualquier operación se realiza en términos aceptables de performance y se debe tener siempre en cuenta que las estructuras de árboles serán usadas para administrar los índices asociados a claves primarias, candidatas o secundarias

Árboles B*

Los árboles B* representan una variable de los árboles B. Define una alternativa para los casos de overflow. Si la cantidad de niveles del árbol crece más lentamente, la performance final de la estructura es mejor.

Si se aplica el concepto de redistribuir, cunado un nodo se completa, resubica sus elementos usando un nodo adyacente hermano. Cuadno no sea posible esta redistribución, se estará ante una situación donde tanto el nodo que genera overflow como su adyacente hermano están complejos. Abre la posibilidad de dividir partiendo de dos nodos completos y generando 3 nodos complejos en dos terceras partes

Un **árbol B*** es un árbol balanceado con las siguientes propiedades

1. Cada nodo del árbol puede contener como máximo M descencientes y M-1 elementos
2. La raíz no posee descendientes o tiene al menos dos
3. Un nodo con x descendientes contiene x-1 elementos
4. Los nodos terminales tienen como mínimo (2M-1)/3]-1 elementos y como máximo M-1 elementos
5. Los nodos que no son terminales ni raíz tienen como mínimo (2M-1)/3] descendientes

6. Todos los nodos terminales se encuentran al mismo nivel

Se puede observar que cuando el árbol B* tiene un solo nivel y este nivel se completa, no está disponible la posibilidad de redistribuir, dado que la raíz no tiene hermanos. Entonces, la propiedad que indica la mínima cantidad de elementos tiene una excepción de tratamiento, al dividir el ndoo raíz y generar dos hijos.

La operación de búsqueda sobre un árbol B* es similar a la representada anteriormente para árboles B. La naturaleza de ambos árboles para localizar un elemento no presenta diferencias de tratamiento.

La operación de baja resulta nuevamente similar, no se genera *underflow* como para quellos donde si se genera insuficiencia

OPERACIÓN DE INSERCIÓN EN ÁRBOLES B*

Las diferentes variantes de inserción tiene que ver con alternativas propuestas en los casos de redistribución. En un árbol B* puede ser regulado de acuerdo con 3 políticas básicas: política de un lado, política de un lado u otro, política de un lado y otro lado.

Cada política detrermina, en caso de overflow, el nodo adyacente hermano a tener en cuenta. La política de un lado determina que el nodo adyacente hermano considerado será solo uno, definiendo la política de izquierda, o en su efecto, la política de derecha. En caso de completar un nodo, intenta redistribuir con el hermano indicado.

En caso de no ser posible porque el hermano también está completo, tanto el nodo que genera overflow como dihco hermano son divididos en dos nodos llenos a tres nodos dos tercios llenos

La performance resultante de la inserción sobre árboles B* dependerá de cada política. Ante la ocurrencia de overflow, como mínimo cada una de las políticas requiere dos lecturas y tres escrituras.

En caso de necesitar realizar una divisicón, la política de un lado necesita a su vez realizar cuatro escrituras. La política de un lado y el otro genera cuatro escrituras, dado que además de involucrar a los tres nodos terminales, se deberán tener en cuenta el nodo padre y el nuevo generado

Acceso secuencial indizado

Se denomina **archivo con acceso secuencial indizado** a aquel que permite dos formas para visualizar la información

1. Indizada: el archivo puede verse como un conjunto de registros ordenados por una clave o llave
2. Secuencial: se puede acceder secuencialmente al archivo, con registros físicamente contiguos y ordenados nuevamente por una nueva clave

Suponga que se dispone de un archivo con N registros que fueron organizados físicamente por oden de llegada, pero que se creó una estructura de índice usando un árbol B o B*. La única forma de extraerlos en orden es usando el índice. Para recuperar los N registros en orden, es necesario recorrer todos los nodos del árbol. Esto significa acceder a un nodo terminal, volver a su padre y acceder al nodo terminal siguiente.

Si el archivo requiere muhco acceso secuencial utilizando el índice, la solución resulta muy ineficiente

Árboles B+

Un **árbol B+** es un árbol multicamino con las siguientes propiedades

1. Cada nodo del árbol puede contener, como máximo M descencientes y M-1 elementos
2. La raíz no posee descendientes o posee al menos 2
3. Un nodo con x descendientes contiene x-1 elementos
4. Los nodos terminales tienen como mínimo [(M/2) -1] elementos, y como máximo M-1 elementos
5. Los nodos que no son terminales ni raíz tienen como mínimo [M/2] descendientes
6. Todos los nodos terminales se encuentran al mismo nivel
7. Los nodos terminales representan un conjunto de datos y son enlazados entre ellos

Así los árboles B+ diferencian los elemtnos que constituyen datos de aquellos que son separadores.

En el momento en que el nodo se satura, se produce una divisicón. Se genera un nuevo nodo terminal donde se redistribuyen los elementos. Se debe notar que la clave utilizada como separador es la misma que está

contenida en el nodo terminal, es decir, una copia del elemento y no el elemento en sí

El proceso de creación del árbol B+ sigue los lineamientos discutidos anteriormente para árboles B

Siempre se insertan en nodos terminales. Si se produce una saturación, el nodo se divide y se promociona una copia. En caso de dividir un nodo no terminal, se debe promocionar hacia el padre el elemento en sí y no una copia del mismo, es decir, solo ante la división de un nodo terminal se debe promocionar una copia.

Para borrar un elemento de un árbol B+, siempre se borra de un nodo terminal, y si hubiese una copia de ese elemento en un nodo no terminal, esta copia se mantendría

Árboles B+ de prefijos simples

Un **árbol B+ de prefijos simples** es un árbol B+ donde los separadores están representados por la mínima expresión posible de la clave, que permita decidir si la búsqueda se realiza hacia la izquierda o hacia la derecha

Conclusiones de los árboles balanceados

Los árboles balanceados son estructuras muy poderosas y flexibles para la administración de índices asociados a archivos de datos. La mejor estructura para un archivo va a depender del archivo en sí y del propósito de uso de dicho archivo. Es decir, si el archivo de datos se limita a unos pocos registros que pueden estar contenidos en un nodo, usar un árbol no sería necesario

Las características compartidas entre los árboles de la familia B son

1. Manejo de bloques o nodos de trabajo
2. Todos los nodos terminales están a la misma distancia del nodo raíz
3. Crecen de abajo hacia arriba, a diferencia de los árboles binarios comunes
4. Son bajos y anchos, a diferencia de los binarios, que son altos y delgados
5. Es posible implementar algoritmos que manipulen registros de longitud variable para, de esta forma, mejorar aún más la performance y el uso del espacio

---

Hashing(dispersión)

Un archivo directo es un archivo en el cual cualquier registro puede ser accedido sin acceder antes a otros registros. En un archivo serie, en cambio, un registro está disponible solo cuando el registro predecesor fue procesado.

Hashing es un método que mejora la eficiencia obtenida con árboles balanceados, asegurando en promeido un acceso para recuperar la información. Este método tiene las siguientes definiciones

- Técnica para generar una dirección base única para una clave dada. La dispersión se usa cunado se requiere acceso rápido mediante una clave. Esto plantea la dispersión como una técnica que permite generar la dirección donde se almacena un registro a partir de la llave o clave de dicho registro
- Técnica que convierte la clave asociada a un registro de datos en un número aleatorio. Agrega la conversión de la clave en un número aleatorio que será la base para determinar dónde el registro se almacenará
- Técnica de almacenamiento y recuperación que usa una función para mapear registros en direcciones de almacenamiento en memoria secundaria. Esta definición menciona que la técnica usa una función, que será responsable de obtener la dirección física de almacenamiento del registro

El hashing presenta una serie de atributos. Pueden resumirse en

- No se requiere almacenamiento adicional. Significa que cuando se elige la opción de dispersión como método de organización de archivos, es el archivo de datos el que resulta disperso
- Facilita la inserción y eliminación rápida de registros en el archivo
- Localiza registros dentro del registro con un solo acceso a disco

Sin embargo, el método de dispersión presenta limintaciones. Estas indican situaciones donde el método no es aplicable, o donde, a partir de su aplicación, no es posible lograr otras caracterísicas que podrían ser deseadas para el archivo de datos. Estas limintaciones son las siguientes

- No es posible aplicar la técnica de dispersión en archivos con registros de longitud variable
- No es posible obtener un orden lógico de los datos
- No es posible tratar con claves duplicadas

Tipos de dispersión

El hashing presenta 2 alternativas para su implementación: tratamiento de espacio de forma estática o tratamiento de espacio en forma dinámica

- **Hashing con espacio de direccionamiento estático:** política donde el espacio disponible para dispersar los registros de un archivo de datos está fijado previamente. Así la función de hash aplicada a una clave da como resultado una dirección física posible dentro del espacio disponible para el archivo
- **Hashing con espacio de direccionamiento dinámico:** aquella política donde el espacio disponible para dispersar los registros de un archivo de datos aumenta o disminuye en función de las necesidades de espacio que en cada momento tiene un archivo

Parámetros de la dispersión

1. Función de hash: esta función puede verse como una caja negra que recibe como entrada una clave y produce una dirección de memoria. Una **función de hash** es una función que transforma un valor, que representa una llave primaria de un registro, en otro valor dentro de un determinado rango

   La función de hash debe retornar como resultado una dirección que se encuentre en un determinado rango. Sin embargo, una vez aplicada dicha función sobre una clave, el valor resultante puede ser cualquiera.

   No es posible almacenar dos registros en el mismo espacio físico. Así, es necesario encontrar una solución a este problema, las alternativas para este caso son 2

   - Elegir un algoritmo de dispersión perfecto que no genere colisiones
   - Minimizar el núemro de colisiones a una cantidad aceptable y de esta manera tratar dichas colisiones como una condición excepcional. Existen diferentes modos de reducir el número de colisiones, las alternativas disponibles son

     1. Distribuír los registros de la forma más aleatoria posible
     2. Utilizar más espacio de disco
     3. Ubicar o almacenar más de un registro por cada direción física en el archivo

   Uno de los objetivos fundamentales cuando se usa el método de dispersión es la selección de la función de hash. Esta función debe esparcir los registros de la manera más uniforme posible
2. Tamaño de cada nodo de almacenamiento: dentro de los parámetros que afectan la eficiencia del método de dispersión está presente el tamaño o capacidad de cada nodo. La capacidad estará definida por la posibilidad de transferencia de información en cada operación de entrada/salida desde RAM hacia disco y viceversa
3. Densidad de empaquetamiento: se define la **densidad de empaquetamiento (DE)** como la relación entre el espacio disponible para el archivo de datos y la cantidad de registros que integran dicho archivo. La cantidad de registros que cada nodo puede almacenar se conoce como **regisros por nodo (RPN)**. La densidad de empaquetamiento se calcula como DE= r / RPN * n. Si se debieran esparcir 30 registros entre 10 direcciones con capacidad de 5 registros por cada dirección, la DE sería de 0.6 o 60%, o sea (30/5*10).

   Cuanto mayor sea la DE, mayor será la posibilidad de colisiones, si la DE se mantiene baja, se dispone de mayor espacio para esparcir registros. Cuando la DE se mantiene baja, se desperdicia espacio en el disco, dado que se utiliza menor espacio que el reservado, generando fragmentación.

   La DE no es constante
4. Métodos de tratamiento de desbordes (overflow): un desborde ocurre cuando un registro es direccionado a un nodo que no dispone de capacidad para almacenarlo. Deben realizarse dos acciones: encontrar lugar para el registro en otra dirección y asegurarse de que el registro posteriormente sea encontrado en esa nueva dirección

Resolución de colisiones con overflow

Aunque la función de hash sea eficiente y aun con DE relativamente baja, es probable que las colisiones produzcan overflow. Se debe contar con algún método para reubicar a aquellos registros que no pueden ser almacenados en la dirección base obtenida a partir de la función de hash

Se presentan 4 métodos para la solución de las colisiones

1. Saturación progresiva: el método consiste en almacenar el registro en la dirección siguiente más proxima al nodo donde se produce la saturación
2. Saturación progresiva encadenada: el método funciona igual que la saturación progresiva, un elemento que se intenta ubicar en una dirección completa es direccionado a la inmediata siguiente con espacio disponible. La diferencia radica en que, una vez localizada la nueva dirección, esta se encadena o enlaza con la dirección base inicial, generando una cadena de búsqueda de elementos
3. Doble dispersión: el método consiste en disponer de dos funciones de hash. La primera obtiene a partir de la llave la dirección de base, en la cual el registro será ubicado. De producirse overflow, se utilizará la segunda función de hash. Esta segunda función no retorna una dirección, sino que su resultado es un desplazamiento. Este desplazamiento se suma a la dirección base obtenida con la primera función, generando así la nueva dirección donde se intentará ubicar al registro.

   La doble dispersión tiende a esparcir los registros en saturación a lo largo del archivo de datos
4. Área de desbordes por separado: ante la ocurrencia de overflow, los registros son dispersados en nodos que no se corresponden con su dirección base original. Así, a medida que se completa un archivo por dispersión, pueden existir muchos registros ocupando direcciones que originalmente no les correspondían, disminuyendo la performance final del método de hashing utilizado

Hashing extensible

El método hash extensible es una alternativa de implementación para hash con espacio de direccionamiento dinámico. 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 los nodos se completan.

No usa el concepto de DE. Con el hash extensible las direcciones de nodos no están prefijadas a priori entonces la función de hash no puede retornar una dirección fija

Para el método extensible, la función de hash retorna un string de bits. La cantidad de bits que retorna determina la cantidad máxima de direcciones a las que puede acceder un método.

El método necesita, para su implementación, de una estructura auxiliar.

Resumiento, el método de hash extensible trabaja según las siguientes pautas

- Se utilizan solo los bits necesarios de acuerdo con cada instancia del archivo
- Los bits tomados forman la dirección del nodo que se utilizará
- Si se intenta insertar en un nodo lleno, 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 como 2i siendo i el número de bits actuales del sistema

Conclusiones

Los árboles balanceados representaron una solución aceptable en términos de eficiencia, reduciendo el número de accesos a disco. No obstante para aquellos donde la eficiencia de búsqueda debe ser extrema, se debe realizar un solo acceso para recuperar la información