Conceptos de Bases de Datos

Agustin Murray

5 de mayo de 2024

Índice

1. Conceptos generales de bases de datos

1.1. Conceptos Básicos

- Se considera **Base de Datos**, a una colección o conjunto de datos interrelacionados con un propósito específico vinculado a la resolución de un problema del mundo real.
- Cualquier información dispuesta de manera adecuada para su tratamiento por una computadora.
- Una colección de archivos diseñados para servir a múltiples aplicaciones.

1.2. Orígenes de las BD

En los orígenes de las **BD**, las unidades de almacenamiento de gran volumen eran muy lentas, entonces se buscaba reducir el acceso a los mismos. Para esto, era necesario repetir datos en distintos puestos de trabajo y al final del día actualizar los mismos en todas las unidades para evitar problemas de pérdida/modificación de información.
Con el tiempo, la tecnología fue avanzando y los sistemas de información evolucionaron. Se logra integrar aplicaciones, interrelacionar archivos y eliminar la redundancia de datos.

1.3. Gestores de Bases de Datos

Un **Sistema de Gestión de Bases de Datos (SGBD)** consiste en un conjunto de programas necesarios para acceder y administrar una BD.
Actualmente, cualquier sistema de software necesita interactuar con información almacenada en una BD y para ello requiere del soporte de un SGBD.
Un SGBD posee dos tipos diferentes de lenguajes: uno para especificar el esquema de una BD, y el otro para la manipulación de los datos.
La definición del esquema de una BD implica:

- Diseño de la estructura que tendrá efectivamente la BD.
- Describir los datos, la semántica asociada y las restricciones de consistencia.

Para ello se utiliza un lenguaje especial, llamado **Lenguaje de Definición de Datos (LDD)**. El resultado de compilar lo escrito con el LDD es un archivo llamado Diccionario de Datos. Un Diccionario de Datos es un archivo con metadatos, es decir, datos acerca de los datos.

Los objetivos más relevantes de un SGBD son:

- **Controlar la concurrencia:** varios usuarios pueden acceder a la misma información en un mismo periodo de tiempo. Si el acceso es para consulta, no hay inconvenientes, pero si más de un usuario quiere actualizar el mismo dato a la vez, se puede llegar a un estado de inconsistencia que, con la supervisión del SGBD, se puede evitar.
- **Tener control centralizado:** tanto de los datos como de los programas que acceden a los datos.

- **Facilitar el acceso a los datos:** dado que provee un lenguaje de consulta para recuperación rápida de información.
- **Proveer seguridad para imponer restricciones de acceso:** se debe definir explícitamente quiénes son los usuarios autorizados a acceder a la BD.
- **Mantener la integridad de los datos:** esto implica que los datos incluidos en la BD respeten las condiciones establecidas al definir la estructura de la BD y que, ante una falla del sistema, se posea la capacidad de restauración a la situación previa.

2. Archivos, estructuras y operaciones básicas

2.1. Definición

Un **archivo** es una colección de registros semejantes, guardados en dispositivos de almacenamiento secundario de una computadora.
Los archivos se caracterizan por el crecimiento y las modificaciones que se efectúan sobre estos. El crecimiento indica la incorporación de nuevos elementos, y las modificaciones involucran alterar datos contenidos en el archivo, o quitarlos.

2.2. Aspectos físicos

Almacenamiento primario (RAM):

- Capacidad de almacenamiento limitada.
- Volátil.
- Alto costo.
- Acceso rápido (orden de los nanosegundos).

Almacenamiento secundario (cintas y discos):

- Alta capacidad de almacenamiento.
- No volátil.
- Menor costo que el primario.
- Acceso lento en comparación con el primario (orden de los milisegundos). Se busca optimizar este aspecto:
  1. Búsqueda de un único dato: Obtención en un intento o en pocos.
  2. Búsqueda de varios datos: obtención de todos de una sola vez.
- **Cintas:** acceso secuencial, económico, estable en diferentes condiciones ambientales, fácil de transportar.
- **Discos:** acceso directo, se almacenan datos en sectores.

2.3. Niveles de visión

- **Archivo físico:** es el archivo residente en la memoria secundaria y es administrado (ubicación, tipo de operaciones disponibles) por el sistema operativo.
- **Archivo lógico:** es el archivo utilizado desde el algoritmo. Cuando el algoritmo necesita operar con un archivo, genera una conexión con el sistema operativo, el cual será el responsable de la administración. Esta acción se denomina independencia física.

2.4. Organización interna de los datos

- **Secuencia de bytes:** Se determina como unidad más pequeña de L/E al byte. No se puede determinar fácilmente el comienzo y el final de cada dato por lo que generalmente son archivos de texto.
- **Campos:** Se determina como unidad más pequeña de L/E al campo. El campo es un ítem de datos elemental y se caracteriza por su tipo de dato y su tamaño.
- **Registros:** Se determina como unidad más pequeña de L/E al registro. El registro es un conjunto de campos agrupados que definen un elemento del archivo. Los campos internos a un registro deben estar lógicamente relacionados, como para ser tratados como una unidad.

2.5. Acceso a información contenida en los archivos

Básicamente, se pueden definir tres formas de acceder a los datos de un archivo:

- **Secuencial:** el acceso a cada elemento de datos se realiza luego de haber accedido a su inmediato anterior. El recorrido es, entonces, desde el primero hasta el último de los elementos, siguiendo el orden físico de estos.
- **Secuencial indexado:** el acceso a los elementos de un archivo se realiza teniendo presente algún tipo de organización previa, sin tener en cuenta el orden físico.
- **Directo:** es posible recuperar un elemento de dato de un archivo con un solo acceso, conociendo sus características, más allá de que exista un orden físico o lógico predeterminado.

2.6. Operaciones básicas sobre archivos

Para poder operar con archivos, son necesarias una serie de operaciones elementales disponibles en todos los lenguajes de programación que utilicen archivos de datos. Estas operaciones incluyen:

- La definición del archivo lógico.
- La definición de la forma de trabajo del archivo (creación inicial, utilización).
- La administración de datos (L/E info).

2.6.1. Definición de archivos

Como cualquier otro tipo de datos, los archivos necesitan ser definidos. Se reserva la palabra clave **file** para indicar la definición del archivo.

```
Var archivo_logico: file of tipo_de_dato;
```

Otra opción para definir archivos se presenta a continuación:

```
Type archivo= file of tipo_de_dato;
Var archivo_logico: archivo;
```

2.6.2. Correspondencia archivo lógico - archivo físico

Se debe indicar que el archivo lógico utilizado por el algoritmo se corresponde con el archivo físico administrado por el sistema operativo. La sentencia encargada de hacer esta correspondencia es:

```
Assign (nombre_lógico, nombre_físico);
```

2.6.3. Apertura y creación de archivos

Hasta el momento, se ha detallado cómo definir un archivo y se ha establecido la relación con el nombre físico. Para operar con un archivo desde un algoritmo, se debe realizar la apertura.

- La operación **rewrite** indica que el archivo va a ser creado y, por lo tanto, la única operación válida sobre el mismo es escribir información.
- La operación **reset** indica que el archivo ya existe y, por lo tanto, las operaciones válidas sobre el mismo son L/E de información.

```
rewrite(nombre_lógico);
reset(nombre_lógico);
```

2.6.4. Cierre de archivos

Para mantener válida la marca de end of file (EOF) se cierra el archivo de la siguiente forma:

```
close(nombre_lógico);
```

2.6.5. Lectura y escritura de archivos

Para leer o escribir información en un archivo, las instrucciones son:

```
read(nombre_lógico, var_dato);
write(nombre_lógico, var_dato);
```

Buffers de memoria: Las lecturas y escrituras desde o hacia un archivo se realizan sobre buffers. Se denomina buffer a una memoria intermedia (ubicada en RAM) entre un archivo y un programa, donde los datos residen provisionalmente hasta ser almacenados definitivamente en la memoria secundaria, o donde los datos residen una vez recuperados de dicha memoria secundaria. La razón de esto es la mejora de performance al trabajar con mayor frecuencia en memoria principal.
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.
De forma similar procede la operación write; en este caso, se escribe en el buffer, y si no se cuenta con espacio suficiente, se descarga el buffer a disco por medio de una operación output, dejándolo nuevamente vacío.

2.6.6. Operaciones adicionales con archivos

- Control fin de datos:

```
eof(nombre_lógico);
```

  La función retornará verdadero si el puntero del archivo referencia EOF, y falso en caso contrario.

- Control de tamaño del archivo:

```
filesize(nombre_lógico);
```

- Control de posición de trabajo dentro del archivo:

```
Filepos(nombre_lógico);
```

- Ubicación física en alguna posición del archivo:

```
seek(nombre_lógico, posición);
```

- Truncamiento de un archivo a partir de la posición actual:

```
truncate(nombre_lógico);
```

3. Algorítmica clásica sobre archivos

3.1. Objetivo

Desarrollar algoritmos considerados clásicos en la operativa de archivos secuenciales. Estos algoritmos se resumen en tres tipos: de actualización, merge y corte de control.

El primer caso, con todas sus variantes, permite introducir problemas donde se actualiza el contenido de un archivo resumen o "maestro", a partir de un conjunto de archivos con datos vinculados a este archivo maestro.

En el segundo caso, se dispone de la información distribuida en varios archivos que se reúne para generar un nuevo archivo, producto de la unión de los anteriores.

Por último, el corte de control, muy presente en la operativa de BD, determina situaciones donde, a partir de información contenida en archivos, es necesario generar reportes que resuman el contenido, con un formato especial.

Se denomina archivo maestro al archivo que resume información sobre un dominio de problema específico. Ejemplo: el archivo de productos de una empresa que contiene el stock actual de cada producto. Por otra parte, se denomina archivo detalle al archivo que contiene novedades o movimientos realizados sobre la información almacenada en el maestro. Ejemplo: el archivo con todas las ventas de los productos de la empresa realizadas en un día particular.

3.2. Actualización de un archivo maestro con un archivo detalle (I)

Presenta la variante más simple del proceso de actualización. Las precondiciones del problema son las siguientes:

- Existe un archivo maestro.
- Existe un único archivo detalle que modifica al maestro.
- Cada registro del detalle modifica a un registro del maestro. Esto significa que solamente aparecerán datos en el detalle que se correspondan con datos del maestro. Se descarta la posibilidad de generar altas en ese archivo.
- No todos los registros del maestro son necesariamente modificados.
- Cada elemento del maestro que se modifica es alterado por uno y solo un elemento del archivo detalle.
- Ambos archivos están ordenados por igual criterio. Esta precondición, considerada esencial, se debe a que hasta el momento se trabaja con archivos de datos de acuerdo con su orden físico.

```
begin
  assign(mae1, 'maestro');
  assign(det1, 'detalle');
  reset(mae1);
  reset(det1);
  while not eof(det1) do begin
    read(mae1, regm);
```

```
    read(det1, regd);
    while(regm.cod<>regd.cod) do read(mae1,regm);
    regm.stock := regm.stockk - regd.cant_vendida;
    seek(mae1, filepos(mae1)-1);
    write(mae1, regm);
  end;
  close(det1);
  close(mae1);
end.
```

3.3. Actualización de un archivo maestro con un archivo detalle (II)

Este es otro caso, levemente diferente del anterior; solo se modifica una precondición del problema y hace, de esta forma, variar el algoritmo resolutorio.

- Cada elemento del archivo maestro puede no ser modificado, o ser modificado por uno o más elementos del detalle (hay registros repetidos).

```
program ejemplo_3_3;
  const valoralto='9999';
  type str4 = string[4]

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

  begin
    assign(mae1, 'maestro');
    assign(det1, 'detalle');
    reset(mae1);

\include{1.tex}
    reset(det1);
    read(mae1, regm);
    read(det1, regd);

    while(regd.cod<>valoralto) do
    begin
      aux:=regd.cod;
      total:=0;

      while(aux = regd.cod) do
      begin
        total := total + regd.cant_vendida;
        leer(det1,regd);
      end;
```

```
        while (regm.cod<>aux) do read(mae1,regm);

        regm.cant := regm.cant - total;

        seek(mae1, filepos(mae1)-1);

        write(mae1, regm);

        if not eof(mae1) then read(mae1,regm);
      end;
      close(det1);
      close(mae1);
  end.
```

El procedimiento de lectura, denominado leer, es el responsable de realizar el read correspondiente sobre el archivo, en caso de que este tuviera mas datos. En caso de alcanzar el fin de archivo, se asigna a la variable dato.cod, por la cual el archivo está ordenado, un valor imposible de alcanzar en condiciones normales de trabajo. Este valor indicará que el puntero del archivo ha llegado a la marca de fin.

3.4. Actualización de un archivo maestro con N archivos detalle

Bajo las mismas consignas del ejemplo anteriosr, se plantea un proceso de actualización donde, ahora, la cantidad de archivos detalle se lleva a N (siendo $N > 1$) y el resto de las precondiciones son las mismas.

El ejemplo 3.4 presenta la resolución de un algoritmo de actualización a partir de tres archivos detalle. Para ello, se agrega un nuevo procedimiento, denominado mínimo, que actúa como filtro. El objetivo de este proceso a partir de la información recibida es retornar el elemento más pequeño de acuerdo con el criterio de ordenamiento del problema. El objetivo del procedimiento mínimo es determinar el menor de los tres elementos recibidos de cada archivo y leer otro registro del archivo desde donde provenía ese elemento.

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

procedure minimo(var r1,r2,r2,min:venta_prod);
begin
  if(r1.cod<=rd2.cod) and (r1.cod<=r3.cod) then
  begin
    mon:=r1;
    leer(det1,r1);
  end
  else if (r2.cod<=r3.cod) then
  begin
    min:=r2;
    leer(det,r2);
```

```
   end
   esle
   begin

\include{1.tex}
     min := r3;
     leer(det3,r3);
   end;
end;

begin
   {asignacion y apertura de archivos correspondientes}

   while(min.cod<>valoralto) do
   begin
     aux:=min.cod;
     total_vendido:=0;
     while(aux=min.cod) do
     begin
       total_vendido:=total_vendido+min.cantvendida;
       minimo(regd1,regd2,regd3,min);
     end;
     while(regm.cod<>min.cod) do read(mae1,regm);

     regm.cant:=regm.cant-total;
     seek(mae1, filepos(mae1)-1);
     write(mae1,regm);
     if not eof(mae1) then read(mae1,regm);
   end;
   close(det1);
   close(det2);
   close(det3);
   close(mae1);
end.
```

3.5. Proceso de generación de un nuevo archivo a partir de otros existentes. Merge

3.5.1. Primer ejemplo

El primer ejemplo plantea un problema muy simple. Las precondiciones son las siguiente:

- Se tiene información en tres archivos detalle.
- Esta información se encuentra ordenada por el mismo criterio en cada caso.
- La información es disjunta; esto significa que un elemento puede aparecer una sola oportunidad en todo el problema.

```
begin
  leer(det1, regd1);
  leer(det2, regd2);
  leer(det3, regd3);

  minimo(regd1, regd2, regd3, min);
  while(min.codigo<>valoralto) do
  begin
    write(mae1, min);
    minimo(regd1, regd2, regd3, min);
  end;
  close(det1);
  close(det2);
  close(det3);
  close(mae1);
end.
```

3.5.2. Segundo ejemplo

Como segundo ejemplo se presenta un problema similar, pero ahora los elementos se pueden repetir dentro de los archivos detalle, modificando de esta forma la tercera precondición del ejemplo anterior. El resto de las precondiciones permanecen inalteradas.

```
begin
  leer(det1, regd1);
  leer(det2, regd2);
  leer(det3, regd3);

  minimo(regd1, regd2, regd3, min);

  while(min.codigo<>valoralto) do
  begin
    codprod:=min.codigo;
    cantotal:=0;
    while(codprod=min.codigo)
    begin
      cantotal:=cantotal+min.cant;
      minimo(regd1,regd2,regd3,min);
    end;
    write(mae1,min);
  end;
  close(det1);
  close(det2);
  close(det3);
  close(mae1);
end.
```

3.6. Corte de control

Se denomina corte de control al proceso mediante el cual la información de un archivo es presentada en forma organizada de acuerdo con la estructura que tiene el archivo.
Suponga que se almacena en un archivo la información de ventas de una cadena de electrodomésticos. Dichas ventas han sido efectuadas por los vendedores de cada sucursal de cada ciudad de cada provincia del país. Luego, es necesario informar al gerente de ventas de la empresa el total de ventas producidas.

Deben tenerse en cuenta las siguientes precondiciones:

- El archivo se encuentra ordenado por provincia, ciudad, sucursal y vendedor.
- Se debe informar el total vendido en cada sucursal, ciudad y provincia, así como el total final.
- En diferentes provincias pueden existir ciudades con el mismo nombre, o en diferentes ciudades pueden existir sucursales con igual denominación.

```
begin
  {asignaciones y aperturas correspondientes.}
  leer(archivo, reg);
  total:=0;
  while(reg.Provincia<>valoralto) do
  begin
    writeln('Provincia:', reg.Provincia);
    prov:=reg.Provincia;
    totprov:=0;

    while(prov=reg.Provincia) do
    begin
      writeln('Ciudad:', reg.Ciudad);
      ciudad:=reg.Ciudad;
      totciudad:=0;

      while(prov=reg.Provincia) and (Ciudad=reg.Ciudad) do
      begin
        writeln('Sucursa:',reg.sucursal);
        sucursal:=regSucursal;
        totsuc:=0;
        while((prov=reg.Provincia) and
        (Ciudad=reg.Ciudad) and Sucursal=reg.Sucursal) do
        begin
          write('Vendedor:', reg.Vendedor);
          writeln(reg.MontoVenta);
          totsuc:=totsuc+reg.MontoVenta;
          leer(archivo, reg);
        end;
        writeln('Total sucursal', totsuc);
        totciudad:=totciudad+totsuc;
      end;
      writeln('Total ciudad', totciudad);
```

```
        totprov:=totprov+totciudad;
      end;
      writeln('Total provincia', totprov);
      total:=total+totprov;
    end;
    writeln('Total empresa', total);
    close(archivo);
end.
```

4. Eliminación de datos. Archivos con registros de longitud variable

4.1. Proceso de bajas

Se denomina **proceso de baja** a aquel proceso que permite quitar información de un archivo.
El proceso de baja puede ser analizado desde dos perspectivas diferentes: aquella ligada con la algorítmica y performance necesarias para borrar la información, y aquella que tiene que ver con la necesidad real de quitar información de un archivo en el contexto informático actual.
En la actualidad, las organizaciones que disponen de BD consideran la información su bien más preciado. De esta forma, el concepto de borrar información queda condicionado. En general, el conocimiento adquirido no se quita, sino que se preserva en archivos o repositorios históricos.
El proceso de baja puede llevarse a cabo de dos modos diferentes:

- **Baja física:** consiste en borrar efectivamente la información del archivo, recuperando el espacio físico.
- **Baja lógica:** consiste en borrar la información del archivo, pero sin recuperar el espacio físico respectivo.

4.1.1. Baja física generando un nuevo archivo de datos

```
program ejemplo_4_1;
...
begin
...
while(reg.Nombre<>'Carlos Garcia')
begin
  write(archivonuevo, reg);
  leer(archivo, reg);
end;
leer(archivo, reg);
while(reg.Nombre<>valoralto)do
begin
  write(archivonuevo, reg);
  leer(archivo,reg);
end;
...
end.
```

Un análisis de performance básico determinó que este método necesita leer tantos datos como tenga el archivo original y escribir todos los datos, salvo el que se elimina. Esto significa n lecturas y $n-1$ escrituras; en ambos casos, las lecturas y escrituras se realizan en forma secuencial.

4.1.2. Baja física utilizando el mismo archivo de datos

```
program ejemplo_4_2;
...
begin
...
while(reg.Nombre<>'Carlos Garcia')
begin
  leer(archivo,reg);
end;
leer(archivo, reg);
while(reg.nombre<>valoralto)do
begin
  seek(archivo, filepos(archivo)-2);
  write(archivo, reg);
  seek(archivo, filepos(archivo)+1);
  leer(archivo, reg);
end;
end.
```

El análisis de performance, para este ejemplo, determina que la cantidad de lecturas a realizar es n, en tanto que la cantidad de escrituras dependerá del lugar donde se encuentre el elemento a borrar; en el peor de los casos, deberán realizarse nuevamente $n-1$ escrituras. No se necesita tanta memoria secundaria como en el ejemplo anterior.

4.1.3. Baja lógica

Se realiza una baja lógica sobre un archivo cuando el elemento que se desea quitar es marcado como borrado, pero sigue ocupando el espacio dentro del archivo. La ventaja del borrado lógico tiene que ver con la performance, basta con localizar el registro a eliminar y colocar sobre él una marca que indique que se encuentra no disponible. Entonces, la performance necesaria para llevar a cabo esta operación es de tantas lecturas como sean requeridas hasta encontrar el elemento a borrar, más una sola escritura que deja la marca de borrado lógico sobre el registro.
La desventaja de este método está relacionada con el espacio en disco. Al no recuperarse el espacio borrado, el tamaño del archivo tiende a crecer continuamente.

```
program ejemplo_4_3;
...
begin
...
while(reg.Nombre<>'Carlos Garcia') do
begin
  leer(archivo, reg);
end;
reg.nombre:='***' {marca de borrado}
Seek(archivo, filepos(archivo)-1);
write(archivo, reg);
close(archivo);
end.
```

4.2. Recuperación de espacio

El proceso de baja lógica marca la información de un archivo como borrada. Ahora bien, esa información sigue ocupando espacio en el disco duro. La pregunta a responder sería: ¿qué hacer con dicha información? Hay dos respuestas posibles:

- **Recuperación de espacio:** periódicamente utilizar el proceso de baja física para realizar un proceso de compactación del archivo. El mismo consiste en quitar todos aquellos registros marcados como borrados, utilizando para ello cualquiera de los algoritmos vistos anteriormente para borrado físico.
- **Reasignación del espacio:** otra alternativa posible consiste en recuperar el espacio, utilizando los lugares indicados como borrados para el ingreso (altas) de nuevos elementos al archivo.

4.2.1. Reasignación de espacio

Esta técnica consiste en reutilizar el espacio indicado como borrado para que nuevos registros se inserten en dichos lugares. Así, el proceso de alta discutido en capítulos anteriores se vería modificado; en lugar de avanzar sobre la última posición del archivo (donde se encuentra la marca de **EOF**), se debe localizar alguna posición marcada como borrada para insertar el nuevo elemento en dicho lugar.
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. Para ello, a medida que los datos se borran del archivo, se genera una lista encadenada invertida con las posiciones borradas.

4.3. Campos y registros con longitud variable

La información en un archivo siempre es homogénea. Esto es, todos los elementos almacenados en él son del mismo tipo. De esta forma, cada uno de los datos es del mismo tamaño, generando lo que se denomina archivos con registros de longitud fija.
Administrar archivos con registros de longitud fija tiene algunas importantes ventajas: 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 de datos se corresponden con todo lo visto hasta el momento.
No obstante, hay determinados problemas donde no es posible, no es deseable trabajar con registros de longitud fija.
Para evitar situaciones donde sobra memoria en cada elemento del archivo, es de interés contar con alguna organización que solo utilice el espacio necesario para almacenar la información. Este tipo de soluciones se representan con archivos donde los registros utilicen longitud variable. En estos casos, como el nombre lo indica, la cantidad de espacio utilizada por cada elemento del archivo no está determinada a priori.
Cada elemento del dato debe descomponerse en cada uno de sus elementos constitutivos y así, elemento a elemento, guardarse en el archivo. En el caso de necesitar transferir un string, debe hacerse caracter a caracter; en caso de tratarse de un dato numérico, cifra a cifra.

```
program ejemplo_4_5;
```

```
...
begin
...
while apellido<>'zzz' do
begin
  BlockWrite(empleados, apellido, length(apellido)+1);
  BlockWrite(empleados, '#', 1);
  ...
  BlockWrite(empleados, documento, length(documento)+1);
  BlockWrite(empleados, '@', 1);
end;
close(empleados);
end.
```

La primera conclusión que se puede obtener a partir del uso de registros de longitud variable es que la utilización de espacio en disco es optimizada, respecto del uso, con registros de longitud fija. Sin embargo, esto conlleva un algoritmo donde el programador debe resolver en forma mucho más minuciosa las operaciones de agregar y quitar elementos.

4.3.1. Alternativas para registros de longitud variable

Cuando se plantea la utilización de espacio con longitud variable sobre técnicas de espacio fijo, existen algunas variantes que se pueden analizar. Estas tienen que ver con utilizar indicadores de longitud de campo y/o registro. De esta manera, antes de almacenar un registro, se indica su longitud; luego, los siguientes bytes corresponden a elementos de datos de dicho registro.

4.4. Acceso a archivos de longitud variable

4.4.1. Clave o llave

Se le asigna a cada registro una clave de identificación. La misma puede adoptar una **forma canónica**, por ejemplo: usar letras mayúsculas sin espacios.
Al agregar un registro, se debe comprobar que no hay otro registro con la misma clave. De ser así, entonces se inserta el registro, caso contrario, se deben comprobar/modificar los datos del nuevo registro.

4.4.2. Acceso directo

Modo de acceso que permite saltar directamente hasta el lugar de un registro en particular.

- El acceso directo es preferible solo cuando se necesitan pocos registros específicos.
- Es necesario conocer el lugar de comienzo del registro requerido.
- Con registros de longitud fija, puede utilizarse un registro encabezado, el cual suele estar primero que todos y contiene información general del archivo, como la cantidad de registros, el tamaño del registro, tipos de delimitadores, etc.

NRR (número relativo de registro): Indica la posición de un registro con respecto al inicio del archivo.

- Archivos con registros de longitud fija: Utilizando el NRR se calcula la distancia en bytes para acceder al registro buscado.
- Archivos con registros de longitud variable: No se conoce el tamaño de cada registro, no es posible aplicar el NRR. Se usa lógica secuencial exclusivamente.

4.5. Eliminación con registros de longitud variable

El proceso de baja sobre un archivo con registros de longitud variable es, a priori, similar a lo discutido anteriormente. Un elemento puede ser eliminado de manera lógica o física. En este último caso, el modo de recuperar espacio es similar a lo planteado en el ejemplo 4.3.
El proceso de baja lógica no tiene diferencias sustanciales con respecto a lo discutido anteriormente. Sin embargo, cuando se desea recuperar el espacio borrado lógicamente con nuevos elementos, deben tenerse en cuenta nuevas consideraciones. Estas tienen que ver con el espacio disponible. Mientras que con registros de longitud fija los elementos a eliminar e insertar son del mismo tamaño, utilizando registros de longitud variable esta precondición no está presente.
El proceso de inserción debe localizar el lugar dentro del archivo más adecuado al nuevo elemento. Existen tres formas genéricas para la selección de este espacio:

- **Primer ajuste:** consiste en seleccionar el primer espacio disponible donde quepa el registro a insertar.
- **Mejor ajuste:** consiste en seleccionar el espacio más adecuado para el registro. Se considera el espacio más adecuado como aquel de menor tamaño donde quepa el registro.
- **Peor ajuste:** consiste en seleccionar el espacio de mayor tamaño, asignando para el registro solo los bytes necesarios.

4.6. Fragmentación

- Se denomina **fragmentación interna** a aquella que se produce cuando a un elemento de dato se le asigna mayor espacio del necesario.
- Se denomina **fragmentación externa** al espacio disponible entre dos registros, pero que es demasiado pequeño para poder ser reutilizado.

4.6.1. Fragmentación y recuperación de espacio

Como se mencionó anteriormente, el procedimiento de recuperación de espacio generado por bajas, utilizando registro de longitud variable, presenta tres alternativas.
Cada una de estas alternativas selecciona el espacio considerado más conveniente. Las técnicas de primer y mejor ajuste suelen implementar una variante que genera fragmentación interna. Así, una vez seleccionado el lugar libre, el espacio asignado corresponde a la totalidad de lo disponible.
Por el contrario, la técnica de peor ajuste solo asigna el espacio necesario. De esta forma, es posible que genere fragmentación externa dentro del archivo. Es deseable, en esos casos, disponer de un algoritmo que se ejecute periódicamente para la recuperación de estos espacios no asignados *(garbage collector)*.

4.7. Modificación de datos con registro de longitud variable

Modificar un registro existente puede significar que el nuevo registro requiere el mismo espacio en disco, que ocupe menos espacio o que requiera uno de mayor tamaño. Como es natural, el problema no se genera cuando ambos registros requieren el mismo espacio. Se puede suponer que si el nuevo elemento ocupa menos espacio que el anterior, no se genera una situación problemática dado que el espacio disponible es suficiente, aunque en ese caso se generaría fragmentación interna.
El problema surge cuando el nuevo registro ocupa mayor espacio que el anterior. En este caso, no es posible utilizar el mismo espacio físico y el registro necesita ser reubicado.
En general, 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 baja al elemento de dato viejo, mientras que en la segunda etapa el nuevo registro es insertado de acuerdo con la política de recuperación de espacio determinada.

5. Búsqueda de información. Manejo de índices

5.1. 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 (comparación). Es así que surgen dos parámetros a analizar: cantidad de accesos y cantidad de comparaciones. El primero de ellos es una operación sobre memoria secundaria, lo que implica un costo relativamente alto; mientras que el segundo es sobre memoria principal, por lo que el costo es relativamente bajo. Se tomará en cuenta solo el costo de acceso a memoria secundaria para los análisis de performance tanto de este capítulo como de los subsiguientes.

El proceso de búsqueda implica un análisis de situaciones en función del tipo de archivo sobre el que se quiere buscar información.

El caso más simple consiste en disponer de un archivo serie, es decir, sin ningún orden preestablecido más que el físico, donde, para acceder a un registro determinado, se deben visitar todos los registros previos en el orden que estos fueron almacenados.

Mejor caso $\rightarrow$ 1 acceso Peor caso $\rightarrow N$ accesos
Promedio $\rightarrow N/2$ accesos

Si el archivo está físicamente ordenado y el argumento de búsqueda coincide con el criterio de ordenación, la variación en relación con el caso anterior se produce para el caso de que el dato buscado no se encuentra en dicho archivo. En ese caso, el proceso de búsqueda se detiene en el registro cuyo dato es "mayor.ª buscado.

Búsqueda binaria Si se dispone de un archivo con registros de longitud fija y además físicamente ordenado, es posible mejorar esta performance de acceso si la búsqueda se realiza con el mismo argumento que el utilizado para ordenar este archivo.

```
function busqueda_binaria(archivo, clave, long_archivo);
var
  menor,mayor,clave_encontrada,registro:integer;
begin
  registro:=-1;
  menor:=1;
  mayor:=long_archivo;

  mientras(menor<=mayor) do begin
    medio:=(mayor+menor)/2;
    buscar reg con NRR = medio
    clave_encontrada = clave can nica
    correspondiente al reg. le do
    if clave<clave_encontrada then mayor:=medio+1
    else if clave > clave_encontrada then menor:=medio+1;
    else registro:=NRR;
  end;
  busqueda_binaria:=registro;
end;
```

Se concluye que la búsqueda binaria es de orden $\log_2(N)$. No obstante, se debe considerar el costo adicional de mantener el archivo ordenado para posibilitar este criterio de búsqueda, y el número de accesos disminuye pero aún dista bastante de recuperar la información en un acceso a disco.

5.2. Ordenamiento de archivos

Si el archivo a ordenar puede almacenarse en forma completa en memoria RAM, una alternativa muy atractiva es trasladar el archivo completo desde memoria secundaria hasta memoria principal, y luego ordenarlo allí. Si bien esto implica leer los datos de memoria secundaria, su acceso es secuencial sin requerir mayores desplazamientos, por lo que su costo no es excesivo. Luego, la ordenación efectuada en memoria principal será realizada con alta performance. La operatoria finaliza escribiendo nuevamente el archivo ordenado en memoria secundaria, otra vez en forma consecutiva, con pocos desplazamientos de la cabeza lectora-grabadora del disco.

Si el archivo no cabe completo en memoria RAM, una segunda alternativa constituye transferir a memoria principal de cada registro del archivo solo la clave por la que se desea ordenar, junto con los NRRs, a los registros correspondientes en memoria secundaria.

El algoritmo ordena en memoria principal solo las claves. Posteriormente, se debe leer nuevamente cada registro del archivo (de acuerdo con el orden establecido por las claves) y escribirlo sobre el archivo ordenado. De este modo, el proceso requiere leer el archivo en forma completa dos veces y reescribirse ordenado una vez. Debido a la cantidad de desplazamientos implicados si el archivo es realmente grande, esta alternativa se ve imposibilitada.

Para archivos realmente grandes surge una nueva alternativa, que consiste en los siguientes pasos:

1. Dividir el archivo en particiones de igual tamaño, de modo tal que cada partición quepa en memoria principal.

2. Transferir las particiones (de una a una) a memoria principal. Esto implica realizar lecturas secuenciales sobre memoria secundaria, pero sin ocasionar mayores desplazamientos.

3. Ordenar cada partición en memoria principal y reescribirlas ordenadas en memoria secundaria. También en este caso la escritura es secuencial (estos tres pasos anteriores se denominan sort interno).

4. Realizar el merge o fusión de las particiones, generando un nuevo archivo ordenado. Esto implica la lectura secuencial de cada partición nuevamente y la reescritura secuencial del archivo completo.

Se puede concluir que la ordenación es una operación de $O(N^2)$ evaluada en términos de desplazamiento.

5.2.1. Selección por reemplazo

Este método se basa en el siguiente concepto: seleccionar siempre de memoria principal la clave menor, enviarla a memoria secundaria y reemplazarla por una nueva clave que está esperando ingresar a memoria principal. Sintetizando, los pasos a cumplir serían:

1. Leer desde memoria secundaria tantos registros como quepan en memoria principal.

2. Iniciar una nueva partición.

3. Seleccionar, de los registros disponibles en memoria principal, el registro cuya clave es menor.

4. Transferir el registro elegido a una partición en memoria secundaria.

5. Reemplazar el registro elegido por otro leído desde memoria secundaria. Si la clave de este registro es menor que la clave del registro recientemente transferido a memoria secundaria, este nuevo registro se lo guarda como no disponible.

6. Repetir desde el Paso 3 hasta que todos los registros en memoria principal estén no disponibles.

7. Iniciar una nueva partición activando todos los registros no disponibles en memoria principal y repetir desde el paso 3 hasta agotar todos los elementos del archivo a ordenar.

Generalizando, se puede establecer que, en promedio, este método aumenta el tamaño de las particiones al doble de la cantidad de registros que se podrían almacenar en memoria principal.

5.2.2. Selección natural

El método de selección natural en relación con selección por reemplazo es una variante que reserva y utiliza memoria secundaria en la cual se insertan los registros no disponibles. De este modo, la generación de una partición finaliza cuando este nuevo espacio reservado está en *overflow* (completo).

Comparando los tres métodos analizados:

- **Ventajas:**
  - **Sort interno:** produce particiones de igual tamaño. Algorítmica simple. No es un detalle menor que las particiones tengan igual tamaño; cualquier variante del método de merge es más eficiente si todos los archivos que unifica son de igual tamaño.
  - **Selección natural y selección por reemplazo:** producen particiones con tamaño promedio igual o mayor al doble de la cantidad de registros que caben en memoria principal.
- **Desventajas:**
  - **Sort interno:** es el más costoso en términos de performance.
  - **Selección natural y selección por reemplazo:** tienen a generar muchos registros no disponibles. Las particiones no quedan, necesariamente, de igual tamaño.

Merge en más de un paso: En pos de buscar mejoras aun mayores que las alternativas ya vistas, se plantea la posibilidad de realizar el merge o fusión en más de un paso. Esto, como su nombre lo indica, consiste en generar archivos intermedios que son conjuntos equitativos de particiones. Estos archivos intermedios deberán sufrir otro merge para obtener el archivo original ordenado.
Al analizar esta alternativa, vamos a ver que la cantidad de desplazamientos requeridos es considerablemente menor que la del merge tradicional.

5.2.3. Indización

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 asociado a la clave. Es necesario que las claves permanezcan ordenadas.
Esta estructura de datos es otro archivo con registros de longitud fija, independientemente de la estructura del archivo original. La característica fundamental de un índice es que posibilita imponer orden en un archivo sin que realmente este se reacomode.
En el peor caso, se plantea de nuevo la situación de ordenar este nuevo archivo en memoria secundaria, con alguno de los criterios planteados previamente en este capítulo.
Veremos dos formas de aprovechar la indización: con índice primario o secundario.

Indización con índice primario:

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. A partir de esta operación, con el NRR o la dirección del primer byte, según corresponda, más la clave primaria, se genera un nuevo registro de datos a insertar en forma ordenada en el índice.

Modificaciones en índice primario
Se considera la posibilidad de cambiar cualquier parte del registro excepto la clave primaria. Si el archivo está organizado con registros de longitud fija, el índice 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, es decir, debe reubicarse. En este caso, esta nueva posición del registro es la que debe quedar asociada en la clave primaria respectiva del índice.

Bajas en índice primario
Se debe borrar física o lógicamente el registro correspondiente en el índice. No se debe recuperar el espacio físico en el índice con otro registro que se inserte, debe mantenerse el orden.

Ventajas del índice primario
La principal ventaja que posee el uso de un índice primario radica en que, al ser de menor tamaño que el archivo asociado y tener registros de longitud fija, posibilita mejorar la performance de búsqueda. 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. El tratamiento de un índice que soporte una clave candidata es similar al definido anteriormente para un índice primario.

Indización con índice secundario:
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. De esta manera surge el uso de índices secundarios.
Un índice secundario es una estructura adicional que permite relacionar una clave secundaria con una o más claves primarias, dado que varios registros pueden contener la misma clave secundaria. Solo el índice primario tiene la dirección física y es el único que debe alterarse; todos los índices secundarios permanecen sin cambios. El índice secundario está almacenado en un archivo con registros de longitud fija.

Creación de índice secundario
Al implantarse el archivo de datos, se deben crear todos los índices secundarios asociados, naturalmente vacíos, solo con el registro encabezado.

Altas en índice secundario
Cualquier alta en el archivo de datos genera una inserción en el índice secundario, que implica reacomodar el archivo en el cual se almacena. Esta operación es de bajo costo en términos de performance si el índice puede almacenarse en memoria principal.

Modificaciones en el índice secundario
Aquí se deben analizar dos alternativas. La primera de ellas ocurre cuando se produce un cambio en la clave secundaria. En este caso, se debe reacomodar el índice secundario, con los costos que ello implica. El segundo caso ocurre cuando cambia el resto del registro de datos (excepto la clave primaria), no generando así ningún cambio en el índice secundario.

Bajas en 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. Esta eliminación puede ser lógica o física.
Puede borrarse solo la referencia en el índice primario. En este caso, el índice primario actúa como una especie de barrera de protección, que aísla a los índices secundarios de este tipo de cambios en el archivo de datos.

Alternativas de organización de índices secundarios
La primera alternativa de mejora sobre el espacio ocupado por el índice secundario implica almacenar en un mismo registro todas las ocurrencias de la misma clave secundaria. Cada registro estará formado por la clave secundaria, más un arreglo

de claves primarias correspondientes a dicha clave. Esto simplifica agregar un nuevo registro, pero al ser de longitud fija, se debe definir un tamaño fijo de arreglo. Esto puede generar insuficiencia de espacio y fragmentación.
Otra alternativa es pensar en una lista de claves primarias asociada a cada clave secundaria. De esta manera, no se debe realizar reserva de espacio y puede existir cualquier número de claves primarias por cada clave secundaria. Esta lista se denomina lista invertida y se encuentra almacenada en otro archivo, el cual se recorre de acuerdo con el camino establecido por el atributo enlace, que indica cuál es el próximo registro asociado, que en caso de no existir, contendrá un puntero nulo.
Ambos archivos están organizados con registros de longitud fija.
Esta opción tiene las siguientes ventajas:

- El único reacomodamiento en el índice se produce cuando se agrega una nueva clave primaria.
- Si se agregan o borran datos de una clave secundaria ya existente, solo se debe modificar el archivo que contiene la lista, y en cada caso solo se debe modificar la lista.
- Dado que se generan dos archivos, uno de ellos podría residir en memoria secundaria, liberando así memoria principal.

Índices selectivos
Consiste en disponer de índices que incluyan solo claves asociadas a una parte de la información existente, es decir, aquella información que tenga mayor interés de acceso.
De esta forma, el índice incluye solo un subconjunto de datos asociados a los registros del archivo que interesa consultar, generando una estructura que actúa como filtro de acceso a la información de interés en el archivo.

6. Árboles. Introducción

6.1. Árboles binarios

Un **árbol binario** es una estructura de datos dinámica no lineal, en la cual cada nodo puede tener a lo sumo dos hijos.
Mientras que sobre memoria RAM estos hijos se representaban mediante punteros, ahora el valor de cada uno es representado mediante un valor entero. Ese valor indica el NRR del hijo dentro del archivo de datos.

```
type registroarbolbinario = record
     elemento_de_dato: tipo_de_dato;
     hijo_izquierdo, hijo_derecho: integer;
end;

indiceBinario = file of registroarbolbinario;
```

6.2. *Performance* de los árboles binarios

La búsqueda de un elemento de datos comienza siempre desde la raíz, recorriendo a izquierda o a derecha según el elemento que se desea ubicar. De esta forma, en cada revisión se descarta la mitad del archivo restante, donde el dato buscado seguro no se encuentra. Es así que la *performance* para buscar un elemento es del orden $\mathbf{Log_2(N)}$ accesos a disco, siendo N la cantidad de nodos que contiene el árbol.
Una ventaja de la organización mediante árboles binarios está dada en la inserción de nuevos elementos. Mientras que en un archivo se desordena cuando se agrega un nuevo dato, si la organización se realiza con la política de árbol binario, la operatoria resulta más sencilla en términos de complejidad computacional. La secuencia de pasos para insertar un nuevo elemento es la siguiente:

1. Agregar el nuevo elemento de datos al final del archivo.

2. Buscar el padre de dicho elemento. Para ello se recorre el archivo desde la raíz hasta llegar a un nodo terminal.

3. Actualizar el padre, haciendo referencia a la dirección del nuevo hijo.

Se puede observar la *performance* desde el punto de vista de accesos a disco: se debe realizar $\mathbf{Log_2(N)}$ lecturas, necesarias para localizar el padre del nuevo elemento, y dos operaciones de escritura (nuevo elemento y actualización del padre). De este modo, si se compara este método con reordenar todo el archivo, como se presentó en el Capítulo 5, esta opción resulta mucho menos costosa en términos de *performance* final.
La operación de borrado presenta un análisis similar. 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. Deberán realizarse nuevamente $\mathbf{Log_2(N)}$ lecturas y dos escrituras.
Como conclusión, los árboles binarios representan una buena elección en términos de inserción y borrado de elementos, que supera ampliamente los resultados obtenidos en el Capítulo 5. En lo que se refiere a operaciones de búsqueda, los árboles binarios tienen un comportamiento similar al comportamiento obtenido al disponer de un archivo ordenado.

6.2.1. Archivos de datos vs. índice de datos

Cada estructura ordenada solo necesita el atributo por el cual ordena. Así, un árbol binario que ordena por código, por ejemplo, solo debe tener esa información. De este modo, se debe separar el archivo de datos de los índices de dicho archivo. El archivo de datos se trata como un archivo serie, donde cada elemento es insertado al final del archivo y no hay orden físico de datos. Se genera, además, un archivo por cada árbol binario que necesite implantar un índice diferente. Cada archivo de índice tendrá la estructura presentada a continuación:

```
type clientes = record
  codigo:integer;
  nombre,dni:string;
  fecha_nacimiento:fecha;
end;

archivocliente = file of clientes;
```

Se incluirá la dirección (NRR) del registro completo en el archivo de datos.

6.2.2. Problemas con los árboles binarios

Si el árbol NO está **balanceado**, la performance de búsqueda ya no puede considerarse de orden logarítmico. El caso degenerado de un árbol binario transforma al mismo en una estructura tipo lista y, en ese caso, la performance de búsqueda decae, transformándose en orden lineal.
La correcta elección de la raíz del árbol determinará si el mismo permanecerá balanceado o no. No obstante, cuando se genera un archivo que implanta un índice de búsqueda, es imposible a priori determinar cuál es la mejor raíz, dado que dependerá de los elementos de datos que se inserten.

6.2.3. 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 que un determinado delta, y dicho delta es el nivel de balanceo en altura del árbol.
Así, un **árbol AVL** es un árbol balanceado en altura donde el delta determinado es uno, es decir, el máximo desbalanceo posible es uno.
Por cuestiones de costos computacionales de acceso a disco considerables, los árboles binarios y los AVL no representan una solución viable para los índices del archivo de datos.

6.3. Paginación de árboles binarios

Las operaciones de lectura y escritura de datos en un archivo utilizando *buffers* presentan una mejora de performance. Este concepto es de utilidad cuando se genera el archivo que contiene el árbol binario. Dicho árbol se divide en páginas, es decir, se pagina, y cada página contiene un conjunto de nodos, los cuales están ubicados en direcciones físicas cercanas.
La cuestión a analizar, ahora, es cómo generar un árbol binario paginado. Para lograr esta construcción, hay que pensar en páginas, los elementos que caben en cada una de ellas, la forma en que se construye un árbol binario y, además, los

temas relacionados con el balanceo. Dividir el árbol en páginas implica un costo extra necesario para su reacomodamiento y para mantener su balanceo interno. Un algoritmo que soporte esta construcción será muy costoso de implementar y luego también en cuanto a *performance*.
Aquí se plantea una disyuntiva, el paginado de árboles binarios con su beneficio y el costo que ello trae aparejado. La solución para este problema consiste en adoptar la idea de manipular más de un registro y tratar de disponer de algoritmos a bajo costo para construir un árbol balanceado.

6.4. Árboles multicamino

Un **árbol multicamino** es una estructura de datos en la cual cada nodo puede contener k elementos y $k + 1$ hijos.
Se define el concepto de orden de **orden de un árbol multicamino** como la máxima cantidad de descendientes posibles en un nodo.
Un árbol multicamino representa otra forma de resolver el concepto de página vertido anteriormente. En este caso, el orden del árbol dependerá del tamaño de la página y de los elementos que se coloquen en ella.
Sin embargo, queda latente aún el problema del balanceo. El Capítulo 7 aborda dicho problema.

7. Familia de árboles balanceados

7.1. Árboles B (balanceados)

Los **árboles B** son árboles multicamino con una construcción especial que permite mantenerlos balanceados a bajo costo.
Un árbol B de orden M posee las siguientes propiedades básicas:

1. Cada nodo del árbol puede contener, como máximo, M descendientes y $M-1$ elementos.

2. La raíz no posee descendientes directos o tiene al menos dos.

3. Un nodo con x descendientes directos contiene $x-1$ elementos.

4. Los nodos terminales (hojas) tienen, como mínimo, $\left(\frac{M}{2}\right)-1$ elementos, y como máximo, $M-1$ elementos.

5. Los nodos que no son terminales ni raíz tienen, como mínimo, $\frac{M}{2}$ elementos.

6. Todos los nodos terminales se encuentran al mismo nivel.

Hasta el momento, se dispone de una definición de árbol B y de seis propiedades. Se deberá ahora determinar la estructura de datos que brinde soporte a estos árboles. En el Capítulo 6, cuando se presentó una estructura tipo árbol como implementación posible para los índices de un archivo de datos, se discutió que **dichas estructuras contendrían solamente la información para permitir el acceso eficiente a estos archivos**. Así, por un lado, se manipula el archivo de datos como un archivo serie y, por otro, cada uno de los archivos que implementaban a los índices respectivos, con una referencia al registro completo en el archivo de datos. En este capítulo, se continúa en ese sentido.

7.1.1. Creación de árboles B

El proceso de creación comienza con una estructura vacía. En el momento inicial, el único nodo existente en el árbol será un nodo raíz, el cual no contendrá ningún elemento. Asimismo, el archivo del índice se encontrará vacío.
Cuando llega el primer elemento, se inserta en la primera posición libre del nodo raíz. El proceso detecta que, en el archivo de índice, el nodo raíz es nulo; por lo tanto, se inserta el primer registro del archivo en el primer lugar de los datos. El número de lugares ocupados se establece en 1 y, al no tener este nodo raíz descendientes, los punteros a los hijos se establecen en nulo (utilizando el valor -1 por ejemplo). El nodo siguiente es otro registro del archivo; por ese motivo se selecciona -1 como indicador de nulo. El primer registro del archivo está ubicado en la posición cero.
Los elementos siguientes comenzarán siempre el proceso de inserción a partir del nodo raíz. Si el nodo raíz tiene lugar, se procederá a insertarlos en este nodo teniendo en cuenta que los elementos de datos deben quedar ordenados dentro del nodo. Este proceso no resulta costoso, dado que el registro que contiene al nodo raíz está almacenado en memoria; por lo tanto, su ordenación solo debe contabilizar accesos a memoria RAM.
La llegada de un nuevo elemento provocará un overflow en el nodo. Este overflow significa que en el nodo no hay capacidad disponible para almacenar un nuevo elemento de datos. En este caso, 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 al nodo padre.

El proceso continúa con la inserción de elementos en el árbol de la misma forma; cuando el árbol crece en altura, es la raíz la que se aleja de los nodos terminales, de modo tal que siempre el árbol crezca de manera balanceada. En todo momento, durante el proceso de creación del árbol, las propiedades descritas anteriormente fueron respetadas.

7.1.2. Búsqueda en un árbol B

El proceso de búsqueda comienza desde el nodo raíz. En caso de encontrarlo en dicho nodo, se retorna una condición de éxito (esto implica retornar la dirección física en memoria secundaria, asociada al registro que contiene la clave encontrada). Si no se encuentra, se procede a buscar en el nodo inmediato siguiente que debería contener el elemento, procediendo de esa forma hasta encontrar dicho elemento, o hasta encontrar un nodo sin hijos que no incluya el elemento.

7.1.3. Eficiencia de búsqueda en un árbol B

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 elemento no se encuentra.
El resultado es un valor acotado en el rango entero [1, H], siendo H la altura del árbol tal como fuera definida previamente en el Capítulo 6. 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 (o que el elemento no se encuentre), serán requeridos H accesos.
Será de gran utilidad el siguiente axioma que no vamos a demostrar: Ün árbol B asociado a un archivo de datos que contiene N registros contendrá un total de N+1 punteros nulos en sus nodos terminales..ºperando, se obtiene la siguiente cota que representa el peor caso:

$$H < 1 + \log_{\frac{M}{2}} \cdot \frac{N+1}{2}$$

Puede observarse una notable mejora en el proceso de búsqueda de información sobre un árbol B.

7.1.4. Eficiencia de inserción en un árbol B

H lecturas y 1 escritura (mejor caso)
H lecturas y $(2 \cdot H) + 1$ escrituras (peor caso)

7.1.5. Eliminación en árboles B

El proceso de eliminación en un árbol tipo B sigue un conjunto de pasos que garantizan que después de eliminar un elemento, el árbol mantenga sus propiedades de orden y balance. Aquí tienes un resumen sintetizado del proceso:

1. **Identificación del elemento a eliminar:** Se busca el elemento que se desea eliminar en el árbol.

2. **Intercambio:** Si el elemento a eliminar no está en un nodo terminal, se intercambia con el mayor de sus claves menores (MN) o la menor de sus claves mayores (UW). Esto se hace para mantener la estructura del árbol.

3. **Eliminación del elemento:** Una vez intercambiado (si es necesario), se elimina el elemento del nodo terminal donde se encuentra.

4. **Verificación de underflow:** Después de eliminar el elemento, se verifica si el nodo terminal donde se encontraba tiene menos elementos de los requeridos mínimamente ($\lfloor \frac{M}{2} \rfloor - 1$). Si tiene suficientes elementos, el proceso de eliminación finaliza. Si no, se pasa al siguiente paso.

5. **Corrección de underflow:** Se buscan nodos adyacentes hermanos que puedan ceder elementos al nodo afectado. Se consideran dos acciones principales: concatenación y redistribución.

   - **Concatenación:** Si la concatenación no produce un nodo con más elementos de los permitidos, se pueden fusionar los nodos afectados en uno solo.
   - **Redistribución:** Si hay un nodo adyacente con suficientes elementos, se redistribuyen entre los nodos afectados. La mitad de los elementos permanece en cada nodo y se toma un elemento como separador, que se mueve al nodo padre.

6. **Propagación de cambios:** Si la operación de concatenación se realiza en un nodo que no es la raíz y causa un underflow en el nodo padre, se aplican los mismos pasos de corrección de underflow en el nodo padre. Este proceso puede propagarse hacia arriba en el árbol, incluso hasta la raíz, pudiendo modificar la estructura de niveles del árbol.

En resumen, el proceso de eliminación en un árbol tipo B implica intercambiar, eliminar y corregir underflows mediante concatenación o redistribución, asegurando que el árbol mantenga sus propiedades de orden y balance.

7.1.6. Eficiencia de eliminación en un árbol B

El mejor caso ocurre cuando se elimina un elemento de un nodo terminal sin generar insuficiencia. Esto requiere H lecturas y una escritura.

El peor caso se da cuando se necesita concatenar nodos, lo que implica leer un nodo adyacente hermano por cada nivel. La cantidad de lecturas en este caso es $2 \times H - 1$, y la cantidad de escrituras es $H - 1$.

Resumiendo (siendo H la cantidad de niveles):

	Mejor caso	Peor caso
Cantidad de lecturas	H	$2 \times H - 1$
Cantidad de escrituras	1	$H - 1$

El análisis de eficiencia de la eliminación plantea una situación muy beneficiosa, considerando la cantidad de operaciones de baja registradas sobre archivos.

7.1.7. Modificaciones en árboles B

La opción más simple para tratar un caso de modificación es proceder tomando un cambio como una baja del elemento anterior y un alta del nuevo elemento. Esta consideración no requiere mayor análisis de operatividad y tiene como resultado, en el análisis de performance, el obtenido de sumar una operación de baja seguida de una de alta. En el peor caso, dicha situación se mantiene acotada en el tiempo de respuesta.

7.1.8. Algunas conclusiones sobre árboles B

- **Eficiencia de los árboles B:** Según lo analizado hasta ahora, los árboles balanceados son una solución sólida para manejar índices en archivos de datos, ya que todas las operaciones (consultas, inserciones, eliminaciones y modificaciones) se realizan con un rendimiento aceptable.
- **Uso de árboles B para administrar índices:** Estos árboles se utilizan principalmente para administrar índices asociados a claves primarias, candidatas o secundarias en archivos de datos. Los archivos de índices contienen la estructura de clave, hijos y referencias al registro original en el archivo de datos.
- **Consideraciones sobre índices secundarios:** Los índices secundarios, al igual que los que almacenan claves candidatas, referencian la clave primaria en lugar de la posición física en el archivo de datos, por razones de rendimiento.
- **Modificaciones en índices:** Si se modifica la posición física de un registro, solo se modifica el índice correspondiente a la clave primaria. En caso de modificar la clave primaria, se deben actualizar todos los índices asociados. Sin embargo, se sugiere diseñar claves primarias bajo la premisa de que nunca serán modificadas, como se explicará en detalle en futuros capítulos.

7.2. Árboles B*

Los árboles B* representan una variante sobre los árboles B. Se define una alternativa para los casos de overflow. Así, antes de dividir y generar nuevos nodos se dispone de una variante, redistribuir también ante una saturación. Esta acción deporará la generación de nuevos nodos y, por ende, tendrá el efecto de aumentar en forma más lenta la cantidad de niveles del árbol. Esto resulta en una mejora de la performance final de la estructura.

Propiedades de un árbol B*

- Cada nodo del árbol puede contener, como máximo, M y $M - 1$ elementos.
- La raíz no posee descendientes o tiene al menos dos.
- Un nodo con x descendientes contiene $x - 1$ elementos.
- Los nodos terminales tienen, como mínimo, $\left\lceil \frac{2M-1}{3} \right\rceil$ elementos, y como máximo, $M - 1$ elementos.

- Los nodos que no son terminales ni raíz tienen, como mínimo, $\left\lceil \frac{2M-1}{3} \right\rceil$ descendientes.
- Todos los nodos terminales se encuentran al mismo nivel.

Las operaciones de búsqueda y eliminación con similares a las de árboles B. El libro no tiene como objetivo profundizar su tratamiento.

7.2.1. Operaciones de inserción sobre árboles B*

El proceso de inserción en un árbol B* puede ser regulado de acuerdo con tres políticas básicas, política de un lado, política de un lado u otro lado, política de un lado y otro lado.
Así, cada política determina, 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 uno solo. 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 dicho hermano son divididos de dos nodos llenos a tres nodos dos tercios llenos.
La política de un hermano adyacente o el otro hermano adyacente representa una alternativa al caso anterior. Aquí, en caso de producirse una saturación en un nodo, se intenta primero redistribuir con un adyacente hermano. Si no es posible, se intenta con el otro. Si nuevamente no es posible, la alternativa es dividir de dos nodos llenos a tres nodos dos tercios llenos.
La última política alternativa plantea ambos adyacentes hermanos. Aquí, la forma de trabajo es similar al caso anterior. Primero, redistribuir hacia un lado y, si no es posible, con el otro hermano. La diferencia aparece cuando los tres nodos están completados. Aquí se toman los tres nodos y se generan cuatro nodos con tres cuartas partes completas cada uno. Esta práctica genera árboles de menor altura pero necesita de un mayor número de operaciones de E/S sobre disco para poder ser implementada.

7.2.2. Análisis de performance de inserción en árboles B*

La performance resultante de la inserción sobre árboles B* dependerá de cada política. Así, ante la ocurrencia de overflow, como mínimo cada una de las políticas requiere dos lecturas y tres escrituras.
En caso de necesitar realizar una división, la política de un lado necesita a su vez realizar cuatro escrituras. Es la misma situación generada por la política de un lado o el otro lado, dado que en este caso se divide nuevamente de dos nodos completo a tres nodos.
Por último, la política de un lado y el otro lado 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.

7.3. Manejo de buffers. Árboles B virtuales

Cuando se decide la capacidad de cada nodo, esta misma está determinada básicamente por la cantidad de información que podrá manejar el SO, a través del uso de buffers. Así, el concepto de buffer y el de nodo están íntimamente relacionados. El SO administra un número importante de buffers. Es posible que aquellos buffers más utilizados se mantengan en memoria principal a fin de minimizar el número de

accesos a disco (LRU).
Un árbol B que administre un índice de acceso muy frecuente significará para el SO que el buffer que contenga el nodo raíz sea muy utilizado. El SO tenderá a mantener dicho buffer en memoria, ahorrándose una E/S. Si se hace esto con otros principales, se ahorran varios accesos a memoria secundaria.

Archivos secuenciales indizados

- Permiten dos formas de acceso:
  - Indizado: acceso aleatorio por clave
  - Secuencial: acceso secuencial ordenado por clave
- Alternativas de implementación con los métodos vistos hasta ahora:
  - Orden físico: ineficiente al sufrir modificaciones
  - Árbol B: rápida recuperación para acceso aleatorio pero ineficiente para acceso secuencial
- Es necesaria una solución que permita ambos accesos de forma eficiente $\rightarrow$ $\acute{A}rbolesB+$

7.4. Árboles B+

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

- Cada nodo del árbol puede contener, como máximo, M descendientes y $M-1$ elementos.
- La raíz no posee descendientes o tiene al menos dos.
- Un nodo con x descendientes contiene $x-1$ elementos.
- Los nodos terminales tienen, como mínimo $([M/2]-1)$ elementos, y como máximo, $M-1$ elementos.
- Los nodos que no son terminales ni raíz tienen, como mínimo, $\left[\frac{M}{2}\right]$ descendientes.
- Todos los nodos terminales se encuentran al mismo nivel.
- Los nodos terminales representan un conjunto de datos y son enlazados entre ellos (punteros).

El proceso de creación del árbol B+ sigue los lineamientos de los árboles B. Los elementos siempre se insertan en nodos terminales. Si se produce una saturación, el nodo se divide y se promociona una copia del menor de los elementos mayores, hacia el nodo padre. Si el padre no tiene espacio para contenerlo, se dividirá nuevamente (sin copiar).
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 mantendrá porque sigue actuando como separador. Si al borrar se produce underflow, se realiza una distribución siguiendo una política asignada. Si no es posible redistribuir, se fusiona según la política, en este caso se pierde el señalador padre.

7.4.1. 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 por izquierda o por derecha.

8. Dispersión (*hashing*)

Hashing o dispersión es un método que mejora la eficiencia obtenida con árboles balanceados, asegurando en promedio un acceso para recuperar la información.
Se presentan a continuación definiciones para el método:

- Técnica para generar una dirección base única para una clave dada. La **dispersión** se usa cuando se requiere acceso rápido mediante una clave.
- Técnica que convierte la clave asociada a un registro de datos en un número aleatorio, el cual posteriormente es utilizado para determinar dónde se almacena dicho registro.
- Técnica de almacenamiento y recuperación que usa una función para mapear registros en direcciones de almacenamiento en memoria secundaria.

Beneficios

- No requiere almacenamiento adicional.
- Facilita la inserción y eliminación rápida de registros.
- Encuentra registros con muy pocos accesos al disco (menos de 2 en promedio).

Costos

- No es posible el uso de registros de longitud variable.
- No existe orden físico de los datos.
- No permite claves duplicadas.

8.1. Tipos de dispersión

Se denomina **hashing con espacio de direccionamiento estático** a aquella 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.
Se denomina **hashing con espacio de direccionamiento dinámico** a 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 el archivo. Así, la función de hash aplicada a una clave da como resultado un valor intermedio, que será utilizado para obtener una dirección física posible para el archivo. Estas direcciones físicas no están establecidas a priori y son generadas de manera dinámica.

8.1.1. Parámetros de la dispersión

Los cuatro parámetros a estudiar son los siguientes:

- Función de hash.
- Tamaño de cada nodo de almacenamiento.
- Densidad de empaquetamiento.
- Métodos de tratamientos de *overflow*.

8.2. Función de hash

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, que se utiliza como dirección física de acceso para insertar un registro en un archivo de datos.

Colisión: Ocurre cuando, luego de aplicar la función de dispersión a dos claves diferentes, se obtiene como resultado la misma clave.
Las colisiones ocasionan problemas. No es posible almacenar dos registros en el mismo espacio físico. Así, es necesario encontrar una solución a este problema. Las alternativas en este caso son dos:

- Elegir un algoritmo de dispersión perfecto, que no genere colisiones (no aplicable).
- Minimizar el número de colisiones a una cantidad aceptable, y de esta manera tratar dichas colisiones como una condición excepcional.

Para disminuir el número de colisiones, pueden tenerse en cuenta las siguientes alternativas:

- Direcciones con N claves $\rightarrow$ mejoras notables
- **Esparcir registros**: buscar métodos que distribuyan los registros de la forma más aleatoria posible entre las direcciones disponibles.
- **Usar memoria adicional**: distribuir pocos registros en muchas direcciones.
- **Colocar más de un registro por dirección**:
  - Direcciones con N claves $\rightarrow$ mejoras notables
  - Las direcciones que pueden almacenar 1 o más registros se denominan cubetas o compartimentos.
  - Ejemplo: archivo con direcciones de 512 bytes y cada registro a almacenar tiene un tamaño de 80 bytes.
    - Se puede almacenar hasta 6 registros por cada dirección asignada al archivo (cada dirección tolera hasta 5 sinónimos).

Algoritmos de dispersión:

- **Uniforme:** reparte los registros en forma uniforme en el espacio de direcciones disponibles (difícil de lograr).
- **Aleatoria:** las claves son independientes, no influyen una sobre la otra. Cualquier dirección tiene la misma probabilidad de ser elegida para una clave.

8.3. Tamaño de cada nodo de almacenamiento

La capacidad del nodo queda determinada por la posibilidad de transferencia de información en cada operación de entrada/salida desde RAM hacia disco, y viceversa.
A **mayor tamaño** del compartimiento:

- Menor probabilidad de overflow.
- Mayor fragmentación (espacios vacíos).
- Búsqueda más lenta dentro del nodo.

8.4. 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.

$$DE = \frac{r}{RPN \cdot n}$$

- **r**: Cantidad de registros que componen un archivo.
- **n**: Cantidad de nodos direccionables.
- **RPN**: Cantidad de registros que cada nodo puede almacenar.

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

8.5. Métodos de tratamiento de overflow

Un desborde u overflow ocurre cuando un registro es direccionado a un nodo que no dispone de capacidad para almacenarlo. Cuando esto ocurre, deben realizarse dos acciones: encontrar un lugar para el registro en otra dirección y asegurarse de que el registro posteriormente sea encontrado en esa nueva dirección.

8.6. Estudio de la ocurrencia de overflow

- **N:** representa el número de direcciones de nodos disponibles en memoria secundaria.
- **K:** determina la cantidad de registros a dispersar.
- **i:** determina la cantidad de registros que contendrá un nodo en un momento específico.
- **C:** determina la capacidad de cada nodo.

Luego, es necesario poder determinar la probabilidad de que un nodo reciba **i** registros.

$$P(i) = \frac{K! \cdot (\frac{1}{N})^i \cdot (1 - \frac{1}{N})^{K-i}}{i! \cdot (K-i)!}$$

8.7. 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 o saturación.
Se presentan cuatro métodos para reubicar aquellos registros que no pueden ser almacenados en la dirección base obtenida a partir de la función de hash.

8.7.1. Saturación progresiva

Inserción: Cuando se completa una dirección en memoria se busca en las siguientes direcciones en secuencia, hasta encontrar una vacía para almacenarlo.

Búsqueda: Comienza en la dirección base y continúa buscando en localidades sucesivas:

- Al llegar a la clave buscada: se finaliza con éxito.
- Al llegar a una dirección vacía: la clave buscada no está en el archivo.
- Al llegar al final del archivo, se continúa por el inicio: circularidad.
- Al llegar al lugar de comienzo: la clave buscada no está en el archivo ($DE = 1$).

Eliminación: Debe ser posible utilizar el espacio liberado para posteriores inserciones. Pero el espacio liberado por una eliminación no debe obstaculizar las búsquedas posteriores. Las búsquedas finalizan al encontrar una dirección vacía, entonces no se deben dejar direcciones vacías. Para esto, se marca el espacio liberado (por ejemplo con). Esto evita romper las secuencias de búsqueda y dejarlo disponible para posteriores adiciones. No es necesario marcarlo si el siguiente espacio está vacío.

- Ventaja: simplicidad.
- Desventaja: tiende a agrupar zonas contiguas. Las búsquedas son largas con DE que tienden a 1.

8.7.2. Saturación progresiva encadenada

Técnica para evitar los problemas causados por la acumulación de registros. Es similar a la de saturación progresiva excepto que las claves sinónimo se enlazan con apuntadores, entonces, ya no es necesario buscar de forma secuencial. Cada dirección base contiene un número que indica el lugar del siguiente sinónimo.

- Ventaja: Solo se necesita acceder a las direcciones que contienen registros con claves sinónimo. (Mejora el número de accesos promedio y ya no son necesarias las marcas de eliminación).
- Desventaja: Debe agregarse un campo de enlace a cada registro (requiere mayor espacio de almacenamiento).

8.7.3. Área de desbordes por separado

Aquí se distinguen dos tipos de nodos: aquellos direccionables por la función de hash (área principal de datos) y aquellos de reserva, que solo podrán ser utilizados en caso de saturación pero que no son alcanzables por la función de hash (área de saturación).
Cuando se agrega un registro nuevo: Si hay lugar en la dirección base, se coloca en el área principal, si no, se coloca en el área de saturación.

- Ventaja: Se mejora el tratamiento de inserciones y eliminaciones.
- Desventaja: Si el área de saturación separada está en un cilindro diferente del de la dirección base, toda búsqueda de registro en saturación implicará un desplazamiento. Esta incrementa el costo.

8.7.4. 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 un 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 el registro. En caso de generarse nuevamente overflow, se deberá sumar de manera reiterada el desplazamiento obtenido, y así sucesivamente hasta encontrar una dirección con espacio suficiente para albergar el registro.
La doble dispersión tiende a esparcir los registros en saturación a lo largo del archivo de datos, pero con un efecto lateral importante. Los registros en overflow tienden a ubicarse lejos de sus direcciones de base, lo cual produce un mayor desplazamiento de la cabeza lectora/grabadora del disco duro, aumentando la latencia entre pistas y, por consiguiente, el tiempo de respuesta.

8.8. Hash asistido por tabla

El método de hash se destaca por su eficiencia en la recuperación de información, logrando un acceso exitoso en más del 99.9 % de los casos cuando la densidad de elementos es inferior al 75 %. Además, las operaciones de inserción y eliminación mantienen niveles de eficiencia similares para estos porcentajes.

Sin embargo, existen situaciones donde puede ser necesario más de un acceso para recuperar o almacenar un registro, especialmente cuando hay saturación.

Una alternativa es el hash asistido por tabla, que utiliza tres funciones de hash para direccionar registros:

- La primera función de hash retorna la dirección física del nodo donde se debe almacenar el registro (F1H).
- La segunda función retorna un desplazamiento, similar al método de doble dispersión (F2H).
- La tercera función retorna una secuencia de bits que no pueden ser todos unos (F3H).

Este método requiere una estructura adicional y, aunque sigue utilizando espacio de direccionamiento estático, permite acceder a un registro en un solo acceso.

8.8.1. El proceso de eliminación

Es necesario hacer mención del proceso de eliminación que utiliza la variante de hash asistido por tabla.
A fines prácticos, el proceso para dar de baja un registro del archivo se puede establecer de la siguiente manera:

1. Se localiza el registro de acuerdo con el proceso de búsqueda definido anteriormente.

2. Si se encuentra la llave buscada, se reescribe el nodo correspondiente sin el elemento a borrar.

Es importante destacar la simplicidad del proceso de eliminación. Sin embargo, hay un caso especial a considerar:
Supongamos que se borra una clave de un nodo completo. Cuando se intente insertar un nuevo elemento en el nodo, habrá lugar. Si el nodo ya había producido overflow, la tabla en memoria contiene un valor correspondiente a la F3h de la llave que produjo saturación. En ese caso, el nuevo elemento a insertar debe cumplir la siguiente propiedad:

$$\text{F3H(nuevo elemento)} < \text{Tabla(dirección del nodo)}$$

Es decir, la tercera función de hash debe otorgar un valor menor al dato que se encuentra en la tabla en memoria para el nodo en cuestión.

8.9. Hash con espacio de direccionamiento dinámico

Otra alternativa posible es trabajar con archivos que administren el espacio de direccionamiento de manera dinámica. Esto implica no establecer a priori la cantidad de nodos disponibles, sino permitir que esta crezca a medida que se insertan nuevos registros.
Surge así la necesidad de utilizar hash con espacio de direccionamiento dinámico. Este tipo de hash dispersa las claves en función de las direcciones disponibles en cada momento, y la cantidad de direcciones puede crecer sin límites, según las necesidades de cada archivo particular.

Existen diferentes alternativas de implementación para hash con espacio dinámico:

- Hash virtual
- Hash dinámico
- Hash extensible

En este libro, se considerará solamente la última alternativa, el hash extensible.

8.9.1. Hash extensible

El método de hash extensible es una alternativa de implementación para hash con espacio de direccionamiento dinámico. Su principio consiste en comenzar con un único nodo para almacenar registros e ir aumentando la cantidad de direcciones disponibles a medida que los nodos se completan.

Al igual que otros métodos que trabajan con espacio dinámico, el hash extensible no utiliza el concepto de densidad de elementos (DE) debido a que el espacio en disco utilizado aumenta o disminuye según la cantidad de registros del archivo en cada momento.

Para el método extensible, la función de hash retorna un string de bits que determina la cantidad máxima de direcciones a las que puede acceder el método. Este método necesita una estructura auxiliar, una tabla que se administra en memoria principal, que contiene la dirección física de cada nodo.

El tratamiento de dispersión con la política de hash extensible comienza con un solo nodo en disco y una tabla que contiene la dirección del único nodo disponible. Resumiendo, 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 entre el nodo viejo y el nuevo; para ello se toma 1 bit más.
- La tabla tendrá tantas entradas (direcciones de nodos) como 2^i, siendo i el número de bits actuales para el sistema.

El proceso de búsqueda asegura encontrar cada registro en un solo acceso. Se calcula la secuencia de bits para la llave, se toman tantos bits de esa llave como indique el valor asociado a la tabla, y la dirección del nodo contenida en la celda respectiva debería contener el registro buscado. En caso de no encontrar el registro en dicho nodo, el elemento no forma parte del archivo de datos.

La siguiente tabla resume los diferentes métodos de gestión de archivos:

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