[75.06] Organización de Datos

María Inés Parnisari

7 de diciembre de 2013

Índice

Objetivo: cómo representar y almacenar datos en archivos para su resguardo, recuperación y transmisión de forma eficiente y segura.

Dato: representación de un objeto (cosa concreta o abstracta, o evento pasado o futuro) en una unidad lógica llamada **registro**, formado por **campos** que describen al objeto (los atributos).

Archivo de datos: unidad lógica de almacenamiento permanente de **registros**, administrada por un **sistema operativo**. Dentro de un archivo los registros pueden agruparse en **bloques** o **páginas**.

Diseño de datos: tiene dos etapas que son independientes del lenguaje de programación utilizado.

1. **Diseño conceptual**[1]: implica definir los atributos.

   Atributos: representan a un dato, y tienen un **valor** (que puede ser **nulo**). Cada atributo tiene una lista de valores permitidos (el **dominio**). Pueden ser:

Simples. *Edad*	Obligatorios. *Nombre y apellido*	Monovalentes. *Sexo*	ID único. *DNI*
Compuestos. *Dirección*	Opcionales *E-mail*	Polivalentes. *Teléfonos*	ID externos. *Código postal*

   Para cada atributo debe definirse:

   - Identidad: el nombre.
   - Estructura, en caso de que sean compuestos.
   - Cardinalidad ("?" = opcional, "*" = ninguno o varios, "+" = uno o varios)
   - Extensión

2. **Diseño lógico**: implica definir tipos o dominios para atributos, y la forma de almacenamiento y recuperación.

   a) Tipos de valores: `int, float, char, char[], string, boolean, binary`

   b) Forma de almacenamiento: registros de longitud fija o registros de longitud variable.

Parte I

Sistemas de archivos[2]

Sistema operativo: responsable de manejar **recursos** físicos (hardware) y lógicos (carpetas y archivos). Provee una interfaz controlada entre ambos.

Conceptos de sistemas operativos:

- **Proceso**: programa en ejecución. Tiene asignado un **espacio de direcciones**. El espacio de direcciones contiene el programa ejecutable, los datos del programa y su pila. Toda la información de cada proceso se almacena en una **tabla de procesos**.

  Varios procesos pueden ser instancias de un mismo programa. Un sistema que puede ejecutar varios procesos "simultáneamente" se denomina **multitarea**. En estos sistemas, la CPU le asigna un quantum de tiempo a cada proceso, y cuando este tiempo se acaba, se ejecuta otro proceso.

[1] *Bases de Datos: conceptos*. Silberschatz. Capítulo 2.
[2] *Sistemas Operativos Modernos*. Tanenbaum. Capítulo 4.

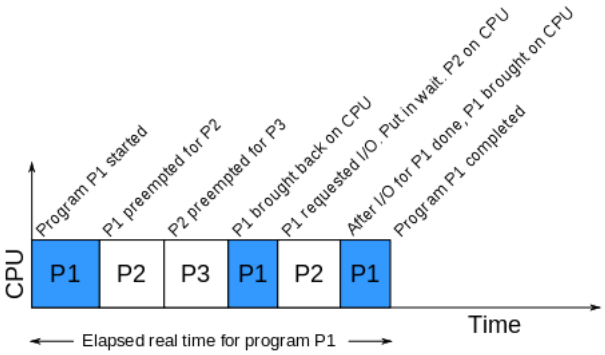


Figura 1: Multitasking.

- **Espacio de direcciones**: porción de la memoria principal que se le asigna a un proceso. Es un mecanismo de protección manejado por el sistema operativo y por el hardware.
- **Archivos y directorios**: los directorios agrupan archivos u otros directorios. La jerarquía de archivos tiene forma de árbol. Existe un **directorio raíz**, y el **directorio actual**.

Funciones del sistema operativo:

1. Identificar y localizar **archivos** mediante jerarquía de **directorios**.
2. Permitir a usuarios o grupos de usuarios establecer **permisos** de acceso a archivos.
3. Asignar **espacio** en discos, administrando la ocupación y liberación de archivos.
4. Coordinar la relación entre uno o varios procesos en ejecución, y su deseo de acceder al mismo dispositivo de almacenamiento simultáneamente (**concurrencia**).
5. Coordinar la comunicación entre la CPU (que maneja tiempos electrónicos rápidos) y los dispositivos de almacenamiento (que maneja tiempos mecánicos lentos), mediante el uso de **buffers**.

Sistema de archivo: parte del sistema operativo que administra los archivos. Los sistemas de archivos se almacenan en disco.

1 Archivos

Requerimientos para el almacenamiento de información a largo plazo:

- Debe ser posible almacenar mucha información.
- La información debe sobrevivir a la terminación del proceso que la utilice.
- Varios procesos deben ser capaces de acceder a la misma información concurrentemente.

Archivo: unidad lógica de información creada por los procesos.
Estructura de archivos:

- Secuencia de bytes: provee la mayor flexibilidad.
- Secuencia de registros de longitud fija.
- **Árbol**: formado por registros que tienen un campo **llave.**

Tipos de archivos según la función que cumplen:

- *Archivos regulares*: contienen información del usuario. Pueden ser **ASCII** o binarios.

- *Archivos especiales de caracteres*: modelan dispositivos de E/S.
- *Archivos especiales de bloques*: modelan discos.
- *Archivos especiales de red*: para comunicar procesos.
- *Directorios*: mantienen la estructura del sistema de archivos.

Tipos de archivos según los datos que contienen:

1. **De datos maestros**. Representan cosas reales. *Ejemplo: un archivo de productos.*
2. **Transaccionales**. Representan eventos. *Ejemplo: un archivo con registros de ventas de productos.*
3. **De reporte**: para presentarle a un usuario. *Ejemplo: ".pdf"*
4. **De trabajo**: resultados parciales de procesamiento. *Ejemplo: ".tmp"*
5. **De control de datos:** para almacenar metadatos o administrar el acceso a otros archivos.
6. **De intercambio de datos**: representan datos de forma estándar para trabajar más fácil entre varias aplicaciones. *Ejemplo: XML.*
7. **De recursos de programa**. *Ejemplo: video, audio, imágenes...*
8. **De productos de programas**. *Ejemplo: ".doc", ".xls"...*
9. **De empaquetado de archivos**. *Ejemplo: ".tar", ".rar", ".zip"...*

Posibles atributos de archivos:

- Atributos de **protección**: usuarios permitidos, contraseña, creador, propietario.
- Atributos **bandera** (binarios): "sólo lectura", "oculto", "del sistema", "de archivo", "ascii/binario", "de acceso aleatorio", "temporal", "de bloqueo".
- Atributos **extras**: longitud de registro, posición de la llave en el registro, longitud de la llave, hora de creación, hora de último acceso, hora de última modificación, tamaño actual, tamaño máximo.

Operaciones posibles en un archivo:

`Create`	`Open`	`Read`	`Seek`	`Get attributes`	`Rename`
`Delete`	`Close`	`Write`	`Append`	`Set attributes`	

- `Create` crea un archivo sin datos y establece algunos atributos.
- `Open` lleva los atributos y la lista de direcciones de disco a memoria principal. Se debe especificar un modo de apertura.
- `Close` obliga a escribir el último bloque del archivo (*flush*).
- `Read` requiere tres parámetros: el *file descriptor* (entero que referencia al archivo), cuántos datos se necesitan leer, y dónde colocarlos (un buffer).
- `Rename` evita tener que copiar el archivo en otro nuevo con el nuevo nombre.

2 Directorios

Tipos de directorio:

- *Directorios jerárquicos*, son árboles con uno o más niveles.
- *Directorio de un solo nivel*, que puede simular un directorio jerárquico si a los archivos los nombramos como si fueran un directorio. Ejemplo: "`usr/ast/file`".

Nombres de rutas:

- **Absoluto**: ruta desde el directorio raíz al archivo.
- **Relativa**: se utiliza en conjunto con el directorio de trabajo.

Operaciones posibles en un directorio:

`Create dir`	`Open dir`	`Read dir`	`Link`	`Rename`
`Delete dir`	`Close dir`		`Unlink`	

- ***Link***: llamada al sistema que permite a un archivo aparecer en más de un directorio.
  - ***Hard link***: relación bidireccional entre una ruta y un archivo. Dos rutas apuntan a la misma estructura que contiene metadatos sobre un archivo.
    En UNIX BSD: Al hacer un *hard link*, se incrementa el contador de *hard links* en el nodo-i del archivo. Todos los nodos-i tienen, al menos, 1 en su contador. Cuando se borra un *hard link*, el contador decrece. Cuando el contador llega a cero, el archivo se borra definitivamente.
  - ***Soft link / symbolic link***: relación unidireccional entre una ruta y un archivo.
    En UNIX BSD: La carpeta original (1) apunta al nodo-i del archivo X. El siguiente directorio (2) que quiera tener una referencia al archivo, crea un archivo que contiene la ruta del archivo original X. Si es borra el archivo X, la ruta (2) tiene un puntero inválido.

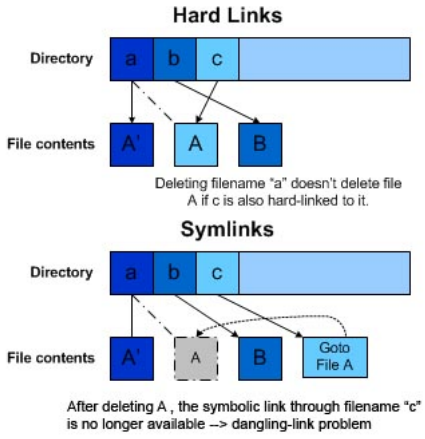


Figura 2: Hard links y soft links.

Los *hard links* no requieren espacio extra en el disco, y los soft links sí.

Los *soft links* pueden apuntar a archivos en otras máquinas, en la misma red o por Internet. Los hard links solo pueden apuntar a archivos dentro de la misma partición.

3 Distribución del sistema de archivos

Ejemplo: UNIX BSD.

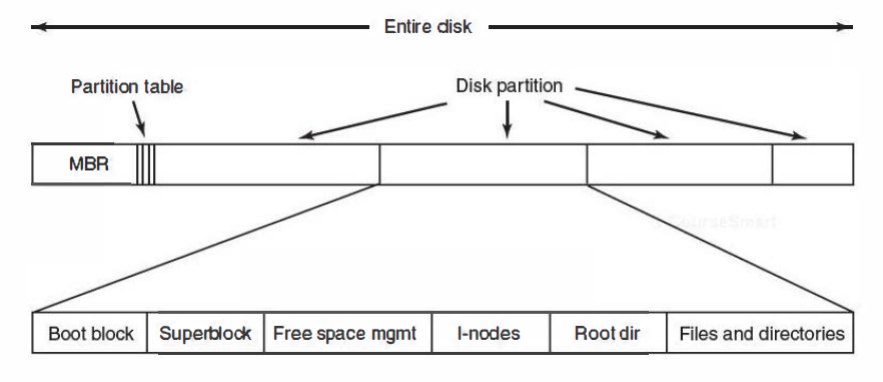


Figura 3: Distribución posible de un sistema de archivos

1. **MBR** (*Master Boot Record*): se utiliza para arrancar la computadora.
2. **Tabla de particiones**: proporciona las direcciones de inicio y fin de cada partición. Proporciona el nombre de la partición activa.
3. Para cada partición:
   a) **Boot block**: contiene el módulo de arranque del sistema operativo de esa partición.
   b) **Superblock**: contiene todos los parámetros clave acerca del sistema de archivos.
   c) Administración del **espacio libre**: información acerca de los bloques libres, en forma de *bitmap* o una lista de punteros.
   d) **Nodos-i**: arreglo de estructuras de datos, uno por archivo, que indica todo acerca del archivo. La cantidad de nodos-i es **fija**, y se establece al crear la partición.
   e) Directorio raíz.
   f) Todos los otros directorios y archivos.

4 Implementación de archivos

Objetivo: mantener un registro acerca de qué bloques del disco van con cuál archivo.
Formas:

1. **Asignación contigua**: almacenar cada archivo como una serie contigua de bloques de disco. Cada archivo comienza al inicio de un nuevo bloque, y puede desperdiciarse espacio en el último bloque de cada archivo.

   Ventajas:

   a) Es simple de implementar. Sólo hay que **guardar**, por cada archivo, **la dirección de disco del primer bloque y la cantidad de bloques que ocupa**.
   b) El **rendimiento de lectura** es excelente. El archivo completo se puede leer en una sola operación.

   Desventajas:

a) Con el tiempo, el disco se **fragmenta**. Cuando se quita un archivo quedan bloques libres, y no se pueden compactar en el momento.

Solución posible: mantener una lista de bloques libres.

Problema de esta solución: cuando se cree un archivo nuevo será necesario conocer su tamaño final para poder elegir el hueco apropiado.

2. **Asignación de lista enlazada**: cada archivo es una lista enlazada de bloques de disco. La primera palabra del primer bloque del archivo se usa como puntero al próximo bloque, hasta llegar a un puntero inválido (fin de archivo).

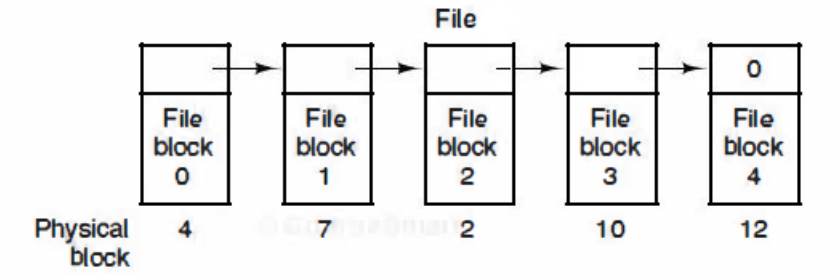


Figura 4: Asignación de lista enlazada.

Ventajas:

a) **No se produce fragmentación**, porque cada bloque se puede aprovechar.

b) La entrada del directorio se reduce a almacenar, para cada archivo, la **dirección de disco del primer bloque**.

Desventajas:

a) La lectura aleatoria es **lenta**. Para llegar al bloque n, se necesita leer los $n-1$ bloques anteriores.

b) El puntero ocupa mucho **espacio**.

3. **FAT / Asignación de lista enlazada utilizando una tabla en memoria**: misma estructura anterior, pero los punteros de cada bloque del disco se almacenan en una tabla en memoria. Esta tabla contiene tantas entradas como bloques de disco haya. Cada entrada posee un puntero al bloque siguiente del archivo.

   Además se tiene un directorio que, para cada archivo, tiene el número del bloque inicial.

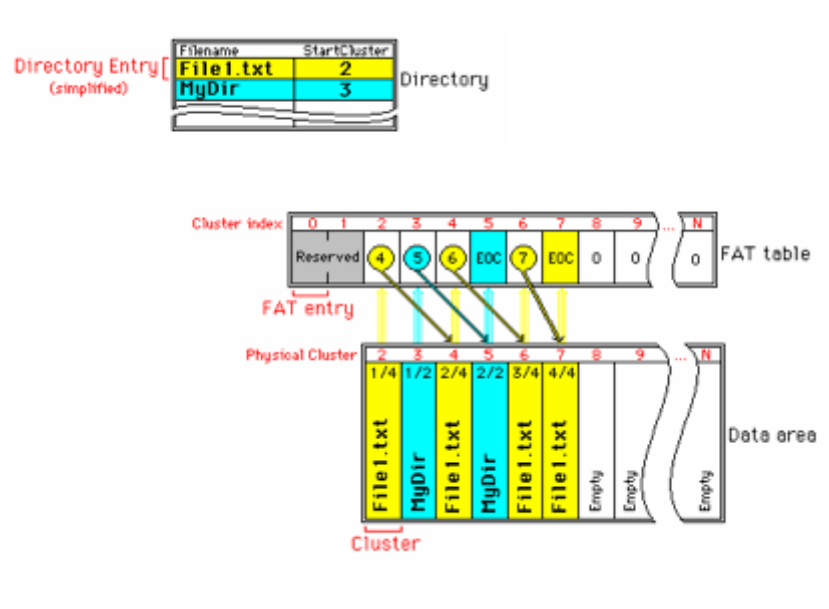


Figura 5: FAT.

Ventajas:

a) El bloque completo contiene datos.

b) La lectura aleatoria es mucho más rápida, porque la lista enlazada está en memoria.

Desventajas:

a) Toda la tabla debe estar en la memoria todo el tiempo. Con un disco de 200 GB, se requiere una FAT que ocupa 600 MB en memoria.

4. **Nodos-i**: cada archivo y cada directorio tiene asociado una estructura de datos llamada **nodo-i**. Estos son registros de **longitud fija** con los siguientes campos[3]:

   a) Propietario y grupo del propietario,

   b) Tipo de archivo (regular, directorio, de caracteres, de red, de bloque)

   c) Permisos de acceso (9 bits),

   d) Fechas de acceso y modificación,

   e) Cantidad de registros del directorio que referencian al nodo-i (contador de *hardlinks*)

   f) Tamaño del archivo en bytes,

   g) Lista de bloques de almacenamiento del archivo:

      1) Bloques 0 a 9 contienen direcciones directas
      2) Bloque 10 contiene dirección indirecta simple
      3) Bloque 11 contiene dirección indirecta doble
      4) Bloque 12 contiene dirección indirecta triple

[3] El nombre del archivo NO es un campo del nodo-i. Se guarda en la entrada del directorio.

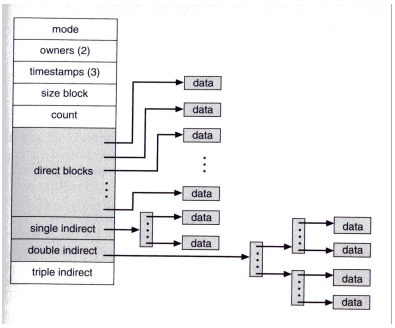


Figura 6: Nodo-i

Cada vez que un archivo cambia, cambia su correspondiente nodo-i. Pero si el contenido del nodo-i cambia, no necesariamente cambiará el archivo también.

La copia de un nodo-i a memoria contiene, además de los campos ya mencionados, los siguientes:

- Estado del nodo-i en memoria ("bloqueado/desbloqueado", "proceso esperando por el nodo-i", "la copia en memoria es distinta a la que está en el disco por un cambio en el nodo-i", "la copia en memoria es distinta a la que está en el disco por un cambio en los datos del archivo", "el archivo es un punto de montaje"),
- Número de dispositivo que contiene al archivo,
- Número de nodo-i,
- Punteros a otros nodos-i en memoria (el sistema encadena los nodos-i en memoria de igual forma que los buffers de bloques),
- Contador que indica el número de instancias del archivo que están activas.
  Cuando este contador llega a 0, el nodo-i se libera y se lo agrega a la lista de nodos-i libres.

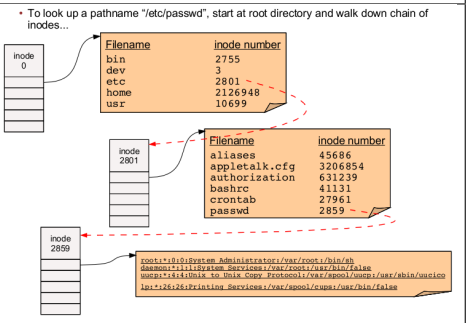
Ventajas:

a) El nodo-i necesita estar en memoria **solo cuando está abierto el archivo**. Si cada nodo-i ocupa n bytes, y el máximo de archivos abiertos es k, la memoria total ocupada por los nodos-i es nk bytes.

Desventajas:

a) El número del nodo-i es simplemente un índice a un sector del disco que contiene al nodo-i en sí. Por lo tanto, hay una ca**ntidad fija de nodos-i posibles**.

5 Implementación de directorios

Implementación de archivos mediante...	**Asignación contigua**	**Asignación de lista enlazada (en disco y en memoria)**	**Nodos-i**
Entrada de directorio debe contener...	Nombre de archivo + Dirección de disco del primer bloque + Cantidad de bloques que ocupa el archivo	Nombre de archivo + Dirección de disco del primer bloque	Nombre de archivo + **Número** del nodo-i del archivo. El directorio raíz suele ser el nodo-i 0.
Para leer un archivo...			

6 Asignación de espacio en disco

El disco se divide en bloques. Cada archivo puede ocupar uno o más bloques. Hay que tomar ciertas decisiones:

1. *¿Bloques contiguos o no?* Si son contiguos se genera fragmentación externa, aunque la lectura es más veloz.
2. *¿Tamaño del bloque?* Si son muy grandes, perdemos espacio por fragmentación interna. Si son muy chicos, perdemos tiempo al buscar cada bloque.

   En general se eligen bloques desde 1 KB hasta 4 KB.
3. *¿Cómo registramos los bloques libres?*

   a) **Lista enlazada de bloques libres**, donde cada bloque contiene tantos números de bloques de disco libres como pueda.
   Ejemplo: un bloque de disco de 1 KB puede contener 256 números de bloque de disco de 32 bits.

   b) **Lista enlazada de bloques libres contiguos**, donde cada bloque contiene un número de bloque libre + la cantidad de bloques libres que le siguen a ese.

   c) **Mapa de bits de bloques libres**. Un 1 representa "bloque libre", un 0 representa "bloque ocupado".
   El mapa de bits requiere menos espacio excepto cuando hay pocos bloques libres.
4. *¿Como aseguramos la consistencia del sistema de archivos?*

   Se pueden realizar dos tipos de verificaciones de consistencia:

   a) **Consistencia de bloques**: se crean dos tablas, con una entrada por cada bloque de disco. Una tabla contiene los bloques en uso (hay que recorrer todos los nodos-i), la otra tabla contiene los bloques libres (hay que recorrer todo el mapa de bits o la lista enlazada de bloques libres).
   Si el sistema de archivos es consistente, cada bloque tendrá un 1 en la primera tabla o en la segunda. Si no es consistente, puede deberse a varios motivos:
   - *Bloque faltante* (contador 0 en ambas tablas): no está en uso ni libre. Solución: marcarlo como libre.
   - *Bloque doblemente libre* (contador 2 en tabla de libres). Solución: reconstruir la lista de bloques libres.

- *Bloque doblemente ocupado* (contador 2 en tabla de ocupados). Solución: copiar el bloque en otro, e insertar la copia en uno de los archivos. Es muy probable que uno de los archivos termine con basura, así que se lo debe reportar al usuario.

b) **Consistencia de directorios**: se crea una tabla de contadores por archivo. Se inspecciona cada directorio, aumentando el contador cada vez que aparece el archivo. (Los hard links cuentan pero los soft links no). Cuando el verificador termina, la tabla está indexada por número de nodo-i. Después se comparan estos contadores con las cuentas de vínculos almacenadas en los mismos nodos-i.

Si el sistema de archivos es consistente, ambas cuentas concordarán. Si no es consistente, puede deberse a dos motivos:

- Cuenta de vínculos del nodo-i $>$ valor en tabla: reestablecer valor en el nodo-i.
- Cuenta de vínculos del nodo-i $<$ valor en tabla: reestablecer valor en el nodo-i.

7 Concurrencia

Los pedidos de acceso al disco se especifican mediante el número de bloque que se necesita, y los niveles más bajos del sistema de archivos se encargan de convertir este número en "cilindro, cara, sector".

El **algoritmo del ascensor** es un **técnica de optimización de los accesos al disco** que consiste en **reducir los movimientos de cambio de pista**.

Su nombre se deriva del comportamiento de un ascensor: mientras está ocupado, el ascensor viaja en una sola dirección (arriba o abajo); solo se detiene para dejar gente afuera o recoger nuevas personas.

Funcionamiento:

1. Si el disco está desocupado y si hay una petición de lectura o escritura:

   a) El brazo del disco se mueve hacia dentro o hacia afuera, hacia el cilindro buscado.

   b) Si llega otra petición:

      1) Si el nuevo cilindro buscado se encuentra en la dirección a la que viaja el brazo del disco, se lo lee / escribe.
      2) Si no, se espera a que el brazo llegue al borde del plato, se le invierte la dirección, y se procede con la petición.

8 Optimización de rendimiento: buffers y cachés[4]

Buffer: sección de la memoria principal disponible para almacenar copias de bloques del disco, que luego se escribirán en él. Esta sección puede no ser suficiente para almacenar todos los bloques de un archivo, por lo que se necesita administrar el buffer.

- Es muy común que el sistema utilice varios buffers para realizar la E/S de un programa (al menos dos: uno para la lectura, y otro para la escritura).
- ***Multiple buffering***: procedimiento mediante el cual dos buffers intercambian sus roles luego de una tarea de lectura o escritura. Esto permite al sistema operativo trabajar con un buffer mientras el otro se está cargando del disco o vaciando hacia el disco.
- Reemplazo de buffers: cuando se necesita una parte de buffer y no hay lugar, hay varias técnicas para mandar bloques al disco y hacer lugar:
  - **LRU** (*Least Recently Used*) para mandar el bloque que se utilizó primero al disco.
  - **MRU** (*Most Recently Used*) para mandar el bloque que se utilizó último al disco.

[4] *Bases de Datos: Conceptos. Silberschatz.* Capítulo 11, sección 4 y 5.

- Bloques sujetados (*pinned blocks*): son bloques que están en el buffer y que no tienen permitido escribirse en el disco.

El acceso al disco es un millón de veces más lento que a la memoria. Por ello, existen optimizaciones para mejorar el rendimiento:

1. **Uso de caché**

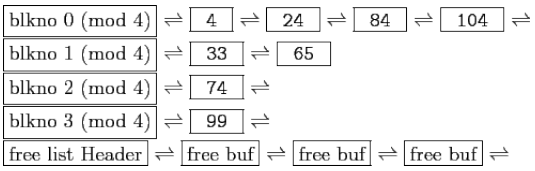
   Buffer caché[5]: estructura de software cuyo objetivo es minimizar los accesos a disco. Agrupa una colección de bloques que se leyeron del disco y se mantienen en memoria.

   El sistema de *buffers* de UNIX consta de dos partes:

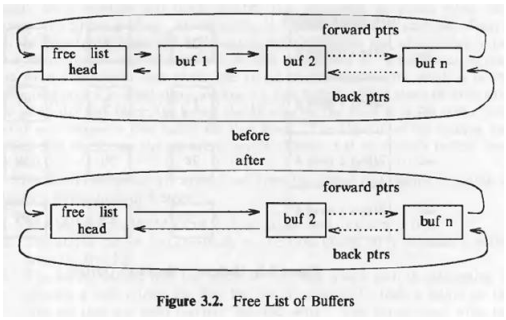
   a) Secuencia de tamaño fijo de bloques de memoria (los *buffers*). La cantidad de *buffers* es limitada y se especifica al inicializar el sistema. Un buffer se corresponde con uno y solo un bloque de disco. El tamaño del buffer no puede ser menor al tamaño de un bloque de disco.

   b) Secuencia de encabezados de buffers (*buffer headers*) que identifican a cada *buffer*. Contienen:

      1) Número lógico de dispositivo[6]
      2) Número de bloque en disco
      3) Banderas: "bloqueado/desbloqueado", "datos válidos/inválidos", "escribiendo/leyendo", "otro proceso está esperando por este buffer", "escritura diferida" (el buffer contiene datos válidos que aún no fueron escritos en el disco)
      4) Puntero al *buffer* en sí
      5) Puntero a los *buffers* anterior y siguiente en la cola de dispersión
      6) Puntero a los *buffers* libres anterior y siguiente en la lista de *buffers* libres



(a) **Buffer caché**



(b) **Lista de buffers libres**

Figura 7: **Buffer cache**

Los buffer headers siempre están en alguna cola de dispersión, y pueden o no estar en la lista de *buffers* libres.

[5] *The Design of the UNIX Operating System*. Capítulo 3.
[6] El número físico del disco lo calcula la controladora del disco.

Los **buffers libres** se almacenan mediante una lista circular doblemente enlazada de buffer headers. Esta lista preserva el orden de LRU. Cuando se necesita un buffer, se lo saca del principio de la lista (o del medio, si se necesitaba un buffer específico). Cuando se agrega un buffer libre, se lo agrega al final de la lista.

Los **buffers en uso** se dividen en varias colas de dispersión (parámetro configurable), que pemiten que la búsqueda de un buffer en particular sea rápida. Cada cola es una lista doblemente enlazada de buffer headers. Cuando el kernel necesita un buffer, lo pone en una y solo una de las colas de dispersión, mediante la aplicación de una función de hash.

Cuando el kernel usa un buffer, lo marca como "bloqueado", y si no está en uso, lo desbloquea. Cuando se desbloquean, se los coloca en la lista de buffers libres (si tiene datos válidos, al final de la cola, y sino, al principio), pero no se los saca de las colas de dispersión.

Algoritmo 1 Algoritmo `brelse` para liberar un buffer

```
1  salida: ninguna
2  {
3    despertar todos los procesos que quieren un buffer libre
4    despertar todos los procesos que querian este buffer
5    bloquear interrupciones
6    si (buffer tiene datos validos y buffer no es viejo)
7      agregar buffer al final de la lista de buffers libres
8    si no:
9      agregar buffer al principio de la lista de buffers libres
10   permitir interrupciones
11   desbloquear(buffer)
12 }
```

Algoritmo 2 Algoritmo `bread` para leer un bloque

```
1  salida: buffer que contiene datos
2  {
3    obtener buffer del bloque (algoritmo getblk)
4    si (buffer tiene datos validos)
5      devolver buffer
6    iniciar lectura de disco a buffer
7    sleep (evento: lectura finalizada)
8    devolver buffer
9  }
```

Algoritmo 3 Algoritmo `getblk` para obtener un bloque

```
salida: buffer bloqueado que se puede usar para manejar un bloque de disco
{
  buffer no encontrado = true
  mientras (buffer no encontrado)
  {
    cola de dispersion = cola((num Dispositivo * num Bloque) mod cant Colas)
    si (bloque en cola de dispersion)
    {
      bloquear interrupciones
      si (buffer bloqueado)
      {
        marcar el buffer con "hay un proceso esperando"
        sleep (evento: buffer desbloqueado)
        continuar
      }
      marcar el buffer como "bloqueado"
      permitir interrupciones
      remover el buffer de la lista de libres
      devolver buffer
    }
    si no
    {
      si (no hay buffers en la lista de libres)
      {
        sleep (evento: se libera algun buffer)
        continuar
      }
      remover primer buffer de la lista de libres
      si (buffer marcado como "escritura diferida")
      {
        remover buffer de la lista de libres
        marcar buffer como "escribiendo"
        escribir el buffer al disco asincronicamente
        // asinc = el kernel NO espera que la escritura termine
        continuar
      }
      remover buffer de la vieja cola de dispersion
      insertar buffer a la nueva cola de dispersion
      devolver buffer
    }
  }
}
```

Algoritmo 4 Algoritmo bwrite para escribir un bloque de disco

```
salida: ninguna
{
  iniciar escritura a disco
  si (I/O sincronica)
    sleep (evento: escritura completa)
    liberar buffer (algoritmo brelse)
  si (buffer marcado como "escritura diferida")
    marcar el buffer para ponerlo al principio de la lista de libres
}
```

Ventajas	*Desventajas*
Acceso uniforme al disco.	El sistema es vulnerable a caidas; lo que se escribió en el buffer pero no en el disco se pierde.
El coódigo es más prolijo.	La copia de datos muy grandes es más lenta, porque se realiza dos veces (del proceso al kernel y del kernel al disco).
No hay restricciones de alineación de bytes.	
Reduccion de accesos al disco.	
Algoritmo `getblk` previene corrupción de datos.	

2. **Lectura adelantada de bloque**

   Tratar de colocar bloques en la caché antes de que se necesiten. Si se necesita el bloque k, se planifica una lectura **asincrónica** al bloque $k+1$. Esta estrategia es más rápida para los archivos que se leen en forma secuencial.

3. **Reducción del movimiento del brazo del disco**

   Al asignar bloques al disco, se intenta ponerlos de forma consecutiva (y en el mismo cilindro), utilizando el *bitmap* de bloques libres como referencia.

   También la colocación de los nodos-i se puede optimizar, poniéndolos todos al principio del disco, o todos en el medio, o distribuidos en grupos de cilindros junto con la lista de bloques libres y los bloques.

9 Ejemplos de sistemas de archivos

9.1 Linux: ext2

Estructura del sistema de archivos:

- Linux admite varias docenas de sistemas de archivos mediante la capa ***Virtual File System*** (VFS). VFS oculta de los procesos y las aplicaciones de nivel superior las diferencias entre muchos tipos de sistemas de archivos que Linux acepta. VFS define un conjunto de abstracciones básicas del sistema de archivos y las operaciones que se permiten en estas abstracciones.

  VFS acepta sistemas de archivos con cuatro estructuras principales:

Superbloque	Contiene información sobre el sistema de archivos.
Dentry	Representa una entrada de directorio. Se crea mediante el sistema de archivos al instante.
Nodo-i	Cada nodo-i describe un archivo, directorio o dispositivo.
Archivo	Representación en memoria de un archivo abierto.

- Nombres de archivos de 255 caracteres.
- **Archivo**: consiste en una secuencia de 0 o más bytes. No se distingue entre archivos ASCII o binarios, o cualquier otro tipo.
- Es un sistema de archivos jerárquico.

Sistema de archivos ext2:

- Estructura:

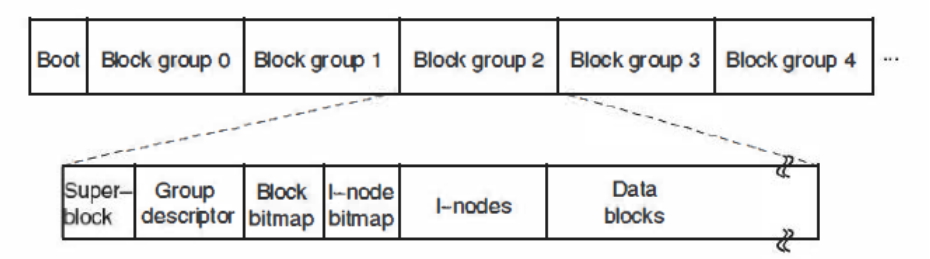


Figura 8: Sistema de archivos ext2.

- *Superblock*: contiene la cantidad de nodos-i, la cantidad de bloques de disco, y el principio de la lista de bloques libres.
- *Descriptor de grupo*: contiene la localización de los bitmaps, la cantidad de bloques y nodos-i libres, y la cantidad de directorios.
- *Bitmap de bloques*, de 1 KB.
- *Bitmap de nodos-i*, de 1 KB.
- *Nodos-i*, de 128 bytes cada uno.
- *Bloques de datos*.

- Una entrada de directorio contiene:
  - Número de nodo-i,
  - Tamaño de la entrada,
  - Tipo de entrada (archivo, directorio, dispositivo),
  - Longitud del nombre del archivo,
  - Nombre el archivo.
- Cada proceso tiene su propia *file descriptor table* (tabla descriptora de procesos). El sistema tiene una única *open file descriptor table*.

Seguridad:

- Cada usuario tiene un **User ID** (UID), un número entero desde 1 a 65.525. El usuario con UID 0 es el **root** (supervisor).
- Los usuarios se pueden organizar en grupos. Cada grupo tiene su propio **Group ID** (GID).
- Cada proceso lleva el UID y el GID de su dueño.
- Los permisos de los **archivos** se heredan del proceso que los creó. Cada archivo tiene **9 bits de seguridad**: 3 bits **rwx** (r=*read*, w=*write*, x=*execute*) para el dueño, 3 bits rwx para el grupo del dueño, y 3 bits rwx para el resto.
- Los directorios son archivos, y por lo tanto también tienen 9 bits de seguridad, pero el bit **x** se refiere al permiso de búsqueda.

9.2 Linux: ext3

Es el mismo que ext2 pero incorpora la funcionalidad de *journaling*, por lo que ext3 es más lento que ext2.

9.3 UNIX: BSD (Berkeley Software Distribution)

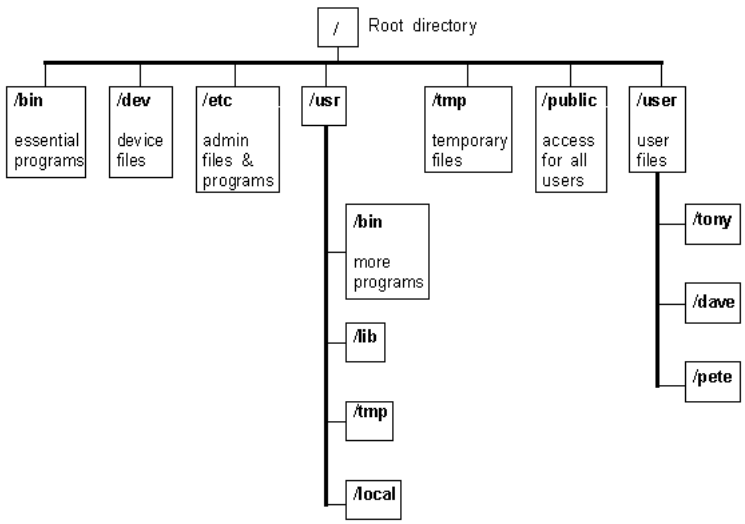


Figura 9: UNIX File System

Estructura del sistema de archivos:

- Los **registros de directorio** son de longitud variable. Cada uno contiene:
  - Nombre del archivo o subdirectorio,
  - Número de nodo-i del archivo o subdirectorio.
- El volumen se organiza en **bloques** (en general de 4 KB cada uno), pero para minimizar la fragmentación interna, el sistema maneja **fragmentos de bloque** que deben tener de tamaño submúltiplo del tamaño de bloque (por ejemplo, de 1 KB). Los fragmentos de bloque se utilizan para asignárselos a bloques finales de archivos. Entonces, un bloque puede tener fragmentos de distintos archivos, pero un archivo no puede tener sus fragmentos en distintos bloques.
- Un **volumen** se compone de varias secciones:
  1. **Boot block**: contiene el módulo de arranque del sistema operativo de esa partición.
  2. **Superblock**: contiene todos los parámetros clave acerca del sistema de archivos. Se encuentra replicado en distintas partes del sistema, para asegurar su integridad. Contiene:
     a) Tamaño del sistema de archivos,
     b) Tamaño de los bloques,
     c) Bitmap para para control de fragmentos de bloque libres
     d) Cantidad de nodos-i libres,
     e) Lista de nodos-i libres,
     f) Puntero al primer nodo-i libre,
     g) Campos para bloquear los mapas de bloques y la lista de nodos-i libres,
     h) Campo que indica que el superblock ha sido modificado.
  3. **Inode list**: lista de nodos-i del sistema de archivo. El tamaño de esta lista debe ser especificado por el administrador al configurar el sistema.
  4. **Data blocks**. Un block de datos puede pertenecer a uno y solo un archivo.
     Los datos se dividen en áreas conformadas por cilindros adyacentes. Cada área tiene su superblock propio, para aumentar la seguridad. Cada grupo de cilindros tiene su propia lista de nodos-i. La política de asignación de espacio procura que los archivos queden localizados en una misma área. El sistema trata de ubicar los nodos-i de un directorio en una misma área.

- Cada proceso tiene su propia *user file descriptor table*. El sistema tiene una única *file table*. Cuando un proceso abre o crea un archivo, el kernel devuelve un *file descriptor* que es un índice a la *user file descriptor table* de ese proceso.
    - *User file descriptor table*: para cada archivo abierto por el proceso, contiene un puntero a la *file table* correspondiente a ese archivo.
    - *File table*: estructura global del kernel, que para cada archivo abierto, contiene el desplazamiento en bytes en el archivo donde comenzará la próxima lectura o escritura.
    - *Inode table*:

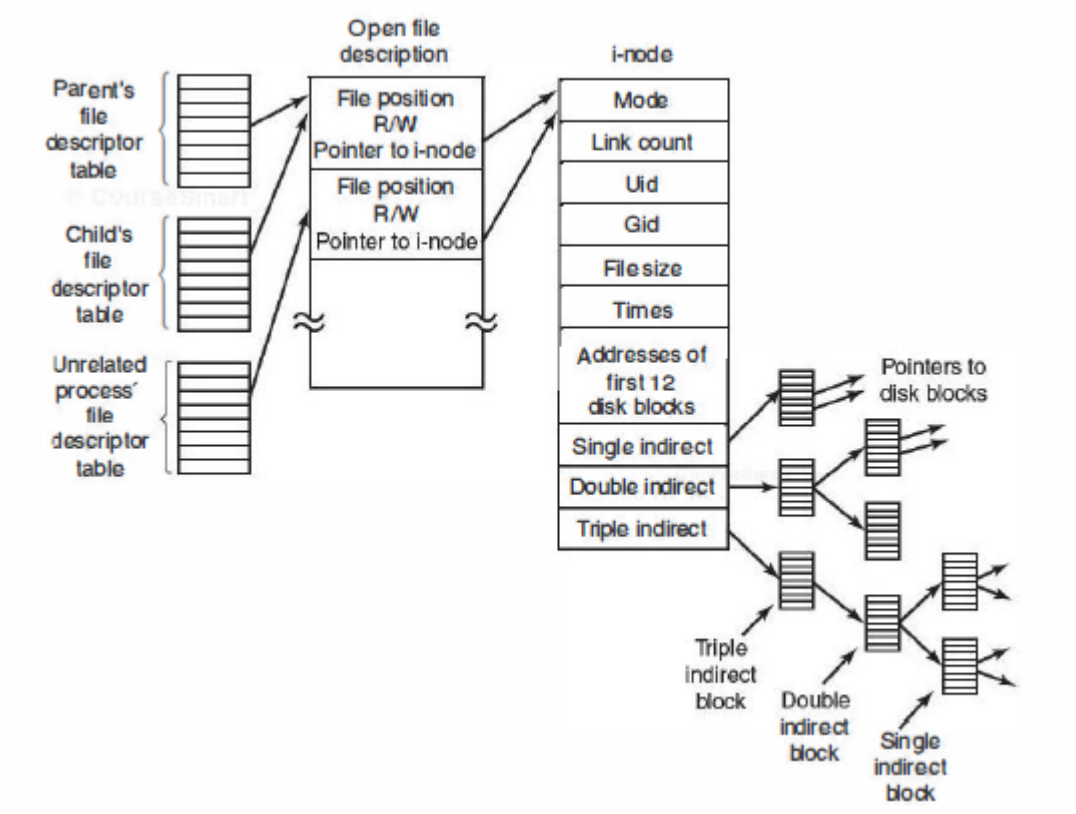


Figura 10: Relación entre las tablas

Cuando se llama a una *system call* [7] que involucra un *file descriptor*, el *file descriptor* se utiliza como un índice a la *file descriptor table* para localizar el puntero a la *open file descriptor table* y a su vez al nodo-i correspondiente al archivo.

9.4 Microsoft: DOS (Disk Operating System)

Estructura del sistema de archivos:

- Los **directorios son archivos** cuyos registros contienen datos sobre los archivos del directorio, y localización del archivo en el disco:
    - 8 bytes para el nombre del archivo + estado del archivo (nunca usado, borrado)

[7] *System call*: función que le pide un servicio al kernel del sistema operativo.

- 3 bytes para la extensión del nombre del archivo
- 1 byte para los atributos del archivo (normal, sólo lectura, oculto, de sistema, subdirectorio, modificado)
- 10 bytes reservados
- 2 bytes para la **hora** en que el archivo fue creado o actualizado
- 2 bytes para la **fecha** en que el archivo fue creado o actualizado
- 2 bytes para el bloque inicial del archivo en el disco
- 4 bytes para el tamaño del archivo en bytes

Seguridad: no tiene mecanismos de seguridad más que la opción de ocultar archivos mediante el atributo correspondiente.
Asignación de espacio en dispositivos de almacenamiento:

1. **Boot record**: contiene instrucciones para ayudar al sistema a cargar del disco a memoria programas para controlar la E/S de datos y para interpretar comandos. También contiene: nombre del fabricante, versión del sistema, cantidad de bytes por sector, cantidad de sectores por clúster, sectores reservados, sectores por pista, número de cabezas de lectoescritura.
2. **FAT**: contiene una entrada de 16 o 32 bits por cada clúster del disco. El registro de directorio de un archivo provee el clúster donde comienza el archivo, y la entrada de la FAT provee el número del siguiente clúster.
   a) Valor 0: clúster libre,
   b) Valor 65.527: clúster dañado
   c) Valores 65.528 y 65.535: fin de archivo.
3. **Sectores del directorio raíz**: el directorio raíz tiene capacidad limitada.
4. **Resto de los directorios**

9.5 Microsoft: NTFS (New Technology File System)

Estructura del sistema de archivos:

- Es un sistema de archivos jerárquico que se desarrolló específicamente para la versión Windows NT.
- Utiliza direcciones de disco de 64 bits, y por lo tanto acepta particiones de hasta 16 exabytes.
- **Archivo**: consiste en varios atributos, y cada uno se representa mediante un flujo de bytes.
  - Atributos residentes, se almacenan en la MTF
  - Atributos no residentes
- Hay soporte para *hard links*, pero sólo se pueden aplicar a archivos del mismo volumen dado que al registro del archivo en la MFT se le agrega otro registro. Además, si se le cambia el tamaño o los atributos a un archivo, las entradas de los demás links pueden no ser actualizadas hasta que se abran.
- Hay soporte para *soft links*.
- Cada volumen se organiza como una secuencia lineal de **clústeres** (conjuntos de **sectores**) de tamaño fijo (por lo general, 4 KB). Los **fragmentos** son conjuntos contiguos de clusters, y están especificados por:
  - *Virtual Cluster Number* (VCN): número de cluster relativo respecto del archivo.
  - *Logical Cluster Number* (LCN): número real de cluster lógico respecto del disco.

- Cantidad de clusteres contiguos a partir de LCN.

- **Master File Table (MFT)**: principal estructura de datos de cada volumen.
  - Es un **archivo** extensible hasta 2^{48} registros. Es una secuencia lineal de **registros** de 1 KB cada uno. Cada registro consiste en un **encabezado de registro** (número mágico para comprobar la validez del archivo, puntero al primer atributo, puntero al primer byte de espacio libre en el registro, número de registro base del archivo) más una secuencia de **pares** <encabezado de atributo (definición del atributo, longitud del valor), valor del atributo>.

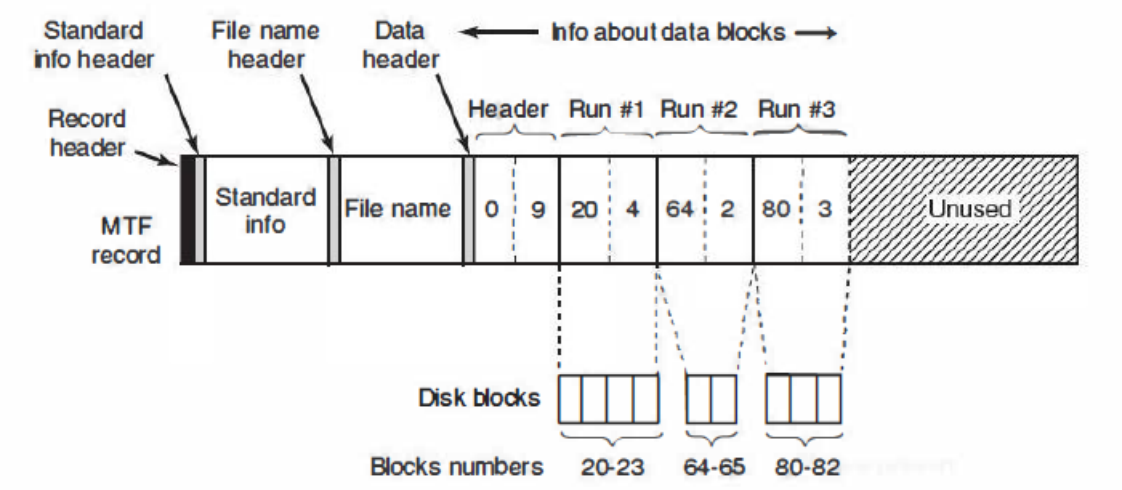


Figura 11: Ejemplo de registro de la MFT.

- Cada registro describe a un archivo o un directorio:
  - Atributos del archivo.

Attribute	Description
Standard information	Flag bits, timestamps, etc.
File name	File name in Unicode; may be repeated for MS-DOS name
Security descriptor	Obsolete. Security information is now in $Extend$Secure
Attribute list	Location of additional MFT records, if needed
Object ID	64-bit file identifier unique to this volume
Reparse point	Used for mounting and symbolic links
Volume name	Name of this volume (used only in $Volume)
Volume information	Volume version (used only in $Volume)
Index root	Used for directories
Index allocation	Used for very large directories
Bitmap	Used for very large directories
Logged utility stream	Controls logging to $LogFile
Data	Stream data; may be repeated

Figura 12: Atributos de un registro de la MFT.

    - Lista de direcciones de disco donde se encuentran sus bloques.
- Los directorios chicos son registros con varias entradas de directorios, cada una de las cuales describe a un archivo o directorio. Los directorios grandes utilizan árboles B+ para listar archivos, facilitando la búsqueda alfabética y la inserción.
- Si el archivo ocupa muchos bloques en disco, el primer registro de la MFT (**registro base**) de ese archivo apunta a los otros registros de la MFT del mismo archivo.
- Como es un archivo, se lo puede colocar en cualquier parte del disco.
- Un **bitmap** lleva el registro de las entradas libres en la MFT.
- Los primeros 16 registros de la MFT se reservan para describir los archivos de metadatos de NTFS.

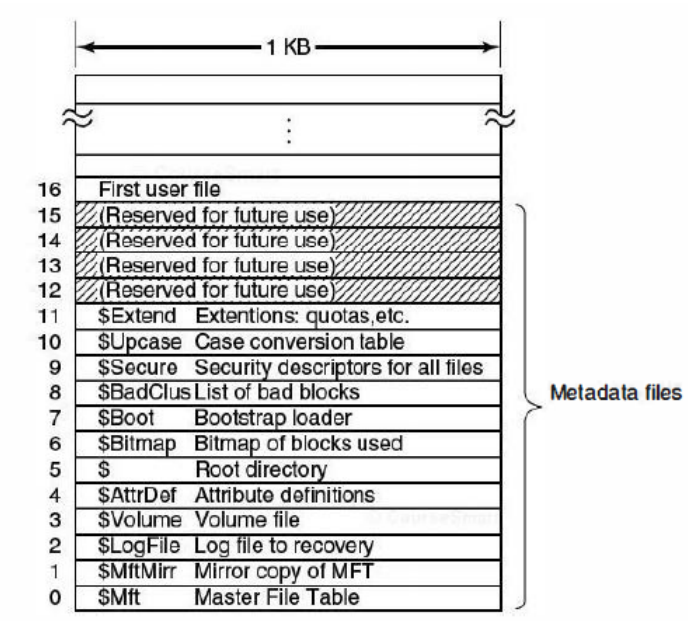


Figura 13: MFT de NTFS.

Seguridad:

- NTFS proporciona compresión transparente de archivos.
- NTFS cuenta con una opción para cifrar directorios (y todos los archivos que hay en él).
- NTFS utiliza un diario de cambios (*journal*): es el archivo de metadatos `$LogFile`. Este archivo registra, cada 5 segundos, los cambios que se producen en todas las estructuras de disco, y si se completan con éxito, se elimina el registro correspondiente. Este archivo tiene dos tipos de registros:
    - UNDO: para deshacer acciones que no se pudieron completar.
    - REDO: para rehacer acciones que no se pudieron completar.
- Cada usuario y grupo se identifica mediante un **SID** (*Security ID*). Cada SID es único en el mundo. Cuando un usuario comienza un proceso, el proceso y sus hilos corren bajo el SID del usuario.
- Cada proceso tiene un **token de acceso**, que indica quién es el propietario del proceso y qué valores y poderes están asociados con él.
- Cada objeto tiene asociado en la MFT un **descriptor de seguridad**, que indica quién puede realizar operaciones sobre él.

Encabezado	
SID del propietario	
SID del grupo	
DACL (*Discretionary Access List*)	Especifica quién puede usar el objeto. Contiene una o más ACL (*Access Control Entries*), y cada una de ellas tiene: un SID, un elemento binario ("permitir/denegar"), y un mapa de bits que indica qué tipo de operaciones tiene permitidas ese SID (*execute, read, write, delete, write DAC, write owner*).
SACL (*System Access Control List*)	Especifica qué operaciones en el objeto se registran en el registro de eventos de seguridad del sistema. Contiene una o más ACL (*Access Control Entries*), y cada una de ellas tiene: un SID, un elemento ("auditar"), y un mapa de bits que indica qué tipo de operaciones se registrarán (*execute, read, write, delete, write DAC, write owner*).

Cuadro 1: Descriptor de seguridad.

9.6 IBM OS/2: HPFS (High Performance File System)

Estructura del distema de archivos:

- ***File Node*** (FNODE): registros de especificación de archivos y directorios.

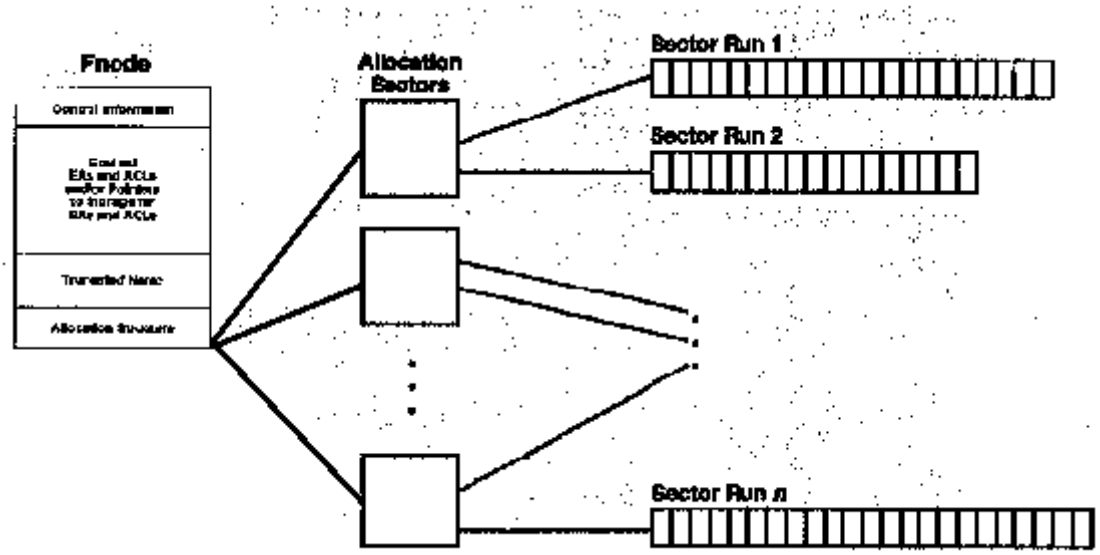


Figura 14: FNODE.

- Longitud del nombre del archivo
- Primeros 15 caracteres del nombre del archivo
- Puntero al directorio que contiene este archivo (LSN - Logical Sector Number)
- Información de control
- Historia de accesos
- Atributos extendidos
- Listas de control de acceso

- Punteros al archivo o puntero al bloque del directorio raíz: hasta ocho **punteros a fragmentos** de archivo de hasta 16 MB cada uno. Cada puntero emplea 32 bits para la dirección inicial de cada fragmento y otros 32 bits para la cantidad de sectores contiguos de fragmentos. Si el archivo es muy grande y ocho punteros no son suficientes, se forma una estructura de árbol donde en lugar de guardar punteros se guardan punteros a **allocation nodes** (alnodes), con capacidad para 40 punteros a fragmentos cada uno.

- **Directorios** formados por uno o más bloques de directorio de 2 kb. Cada **entrada de directorio** contiene: longitud de entrada, fechas, longitud del nombre del archivo o subdirectorio, nombre del archivo o subdirectorio, puntero al FNODE (contiene información del archivo u otro puntero), puntero al bloque de directorio sucesor.
- Si un directorio es muy grande para ser descrito en un bloque, se le asignan más bloques y se organiza como un **árbol B**.
- Directorios ordenados alfabéticamente.
- Estructura del volumen:

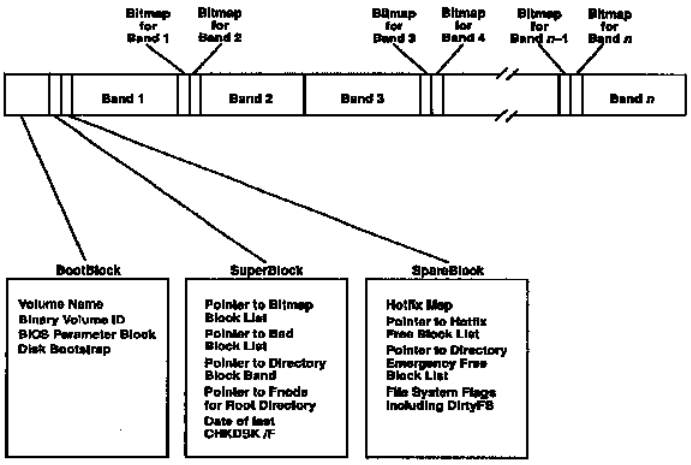


Figura 15: HPFS.

- **Boot block** (sectores 0 a 15), volume ID, bootstrap.
- **Super block** (puntero al FNODE del directorio raíz)
- **Spare block** (tabla de hotfixes + Bloque de Directorio Vacío)
- **Bandas**. Las bandas miden 8 MB. Entre cada banda hay un *bitmap* de 8 KB de sectores libre. Una de las bandas es la de bloques de directorio.

Seguridad:

- Los FNODES contienen listas de control de acceso.

Parte II

Organización de archivos

10 Archivos planos - *streams*

En C/C++ hay dos formas posibles de manejar un archivo: utilizando rutinas de biblioteca, incluídas en `stdio.h`, o utilizando *system calls*.

Utilizando rutinas	*Utilizando system calls*
La entrada/salida se realiza en *registros*.	La entrada/salida se realiza en *bytes*.

Primitivas para el manejo de archivos:

	Utilizando rutinas	*Utilizando system calls*
Apertura	`FILE* fopen (char* nombreArchivo, char* modoApertura);`	`int open (char* nombreArchivo, int flag_apertura [,int permisos])`
Lectura	`unsigned int fread (void* bufferDestino, unsigned tamReg, unsigned cantidadRegs, FILE* punteroAlArchivo);`	`int read (int fileDescriptor, void* bufferDestino, unsigned cantidadBytes);`
Escritura	`unsigned int fwrite (void* bufferFuente, unsigned tamReg, unsigned cantidadRegs, FILE* punteroAlArchivo);`	`int write (int fileDescriptor, void* bufferFuente, unsigned cantidadBytes);`
Posicionamiento	`int fseek (FILE* punteroAlArchivo, long offset, int origen);`	`int lseek (int fileDescriptor, long offset, int origen);`
Cierre	`int fclose (FILE* punteroAlArchivo);`	`int close (int fileDescriptor);`

11 Medios de almacenamiento físicos

Nombre	*Descripción*	*Costo $*	*Capacidad*	*¿Es volátil?*	*Velocidad de transferencia*
Caché	Manejado por hardware.	Muy caro	Muy poca	Sí	Muy alta
Memoria principal	Desde varios megabytes a varios gigabytes. Permite acceso aleatorio. Compuestos por celdas de un byte cada una.	Caro	Poca	Sí	Muy alta (20 segundos)
Memoria flash	Para sobreescribir memoria, hay que borrar todo. Soporta un numero limitado de ciclos de borrado.	Caro	Poca	No	Alta
Discos magnéticos	Cintas magnéticas, discos duros, diskettes. Cada **plato** del disco está dividido en **pistas**, que son conjuntos de **sectores**. El conjunto de todas las pistas i de todos los platos conforman el **cilindro** i. Un **bloque** es un conjunto de sectores.	Barato	Grande	No	Baja (58 días)
Discos ópticos	CD (700 MB), DVD (4,7 GB), BluRay (25 GB)	Barato	Grande	No	Baja
Cintas	Se usa para crear copias de respaldo. Solo permite acceso secuencial.	Muy barato	Grandísima	No	Muy baja

Cuadro 2: Tipos de almacenamiento.

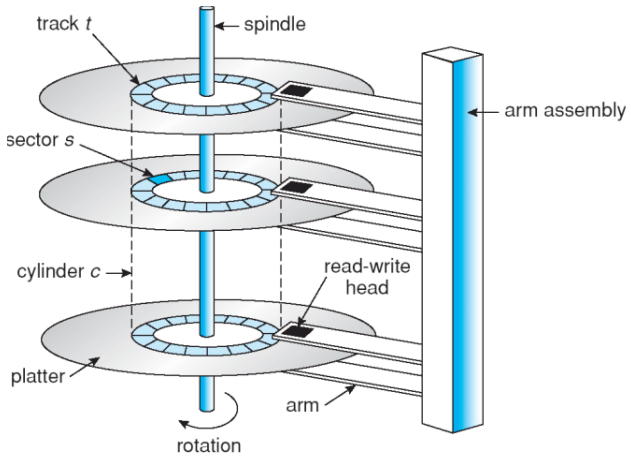


Figura 16: Disco magnético.

12 Organización de archivos

Historia:

1. Archivos secuenciales, uso de cintas magnéticas
   a) *Mucho tiempo para hacer un seek!*

2. Índices que tenían una clave y el offset al archivo secuencial
    a) *El índice podía ser muy grande!*
3. Árboles n-arios para almacenar el índice
    a) *Podían desbalancearse fácilmente* $\implies$ *Mucho tiempo para hacer un seek!*
4. Árboles n-arios balanceados (AVL)
    a) *Aún requerían muchos accesos a disco!*
5. Árboles B
    a) *Los archivos no se pueden acceder de forma secuencial!*
6. Árbol B+
    a) *Aún requerían varios accesos a disco!*
7. Hashing (tablas de dispersión)
    a) *El archivo no debe crecer de tamaño demasiado!*
8. Hashing extensible

Objetivo: cómo organizar datos en archivos para optimizar la eficiencia en almacenamiento en disco, recuperación de disco y resguardo de los datos.
Lo ideal es 1 acceso a disco, o la cantidad mínima que se pueda, para obtener toda la información que necesitamos en una sola lectura.

- ¿Dónde almacenar nuevos registros?
- ¿Cómo encontrar registros para borrarlos, modificarlos, o recuperarlos?

Archivo: colección de **registros** formados por **campos**, que se mapean a **bloques** del disco. El tamaño de los bloques es fijo y es determinado por el sistema operativo, pero el tamaño de los registros puede variar.
Fragmentación interna: espacio perdido dentro de un registro, generalmente al final.
Fragmentación externa: espacio perdido entre registros.

13 Estructura de campos

¿Cómo representar campos de manera que no pierdan su identidad?

- Campos de longitud limitada
    - Pierdo espacio
    - Puedo no estar contemplando valores muy grandes
- Campos comienzan con su longitud en bytes
    - El tamaño del *campo que representa la longitud del campo* debe elegirse con cuidado! Con 1 campo de 1 byte podemos decir, como máximo, "registro de 256 bytes a continuación".
- Campos separados por un delimitador
    - El delimitador tiene que ser elegido con cuidado!
- Campos identificados por el par "clave, valor"

14 Estructura de registros

Registros de longitud fija: producen fragmentación interna.

1. *¿Cómo borramos un registro y luego reclamamos el espacio?*

   a) **Estático**: marcar el registro como borrado. Luego de que se borraron suficientes registros, se realiza una reconstrucción del archivo, eliminando los registros vacíos.

      1) Ventaja: borrar un registro es rápido.
      2) Desventaja: reorganizar el archivo es lento.

      Para insertar un registro nuevo, se busca secuencialmente hasta encontrar un registro borrado.

   b) **Dinámico**: usar una lista enlazada de registros libres. Le agregamos un **file header** al archivo que apunte al primer registro libre, el cual apunta al segundo registro libre, y así sucesivamente.

      Para borrar un registro: copiamos lo que dice el *file header* al registro. Y en el *file header* ponemos el número de registro que queremos borrar.

      Para insertar un registro: mantenemos una copia del número de registro apuntado por el registro apuntado por el *file header*. En el registro apuntado por el *file header* copiamos el registro. En el *file header* ponemos la copia de antes.

      Estructura de archivo necesaria:

      1) *File header*: número del primer registro libre, -1 si no hay libres.
      2) Para cada registro:
         a' Bit bandera ("libre/ocupado")
         b' Datos. Si está "libre": puntero a próximo registro libre.

2. *¿Como nos aseguramos que un registro requiera solo un acceso a disco?*

   Hay que asegurarse de que el tamaño del bloque sea un múltiplo del tamaño del registro.

Registros de longitud variable: producen fragmentación externa.

- *¿Cómo los implementamos?*

  1. Usar **delimitadores** que marquen el fin de un registro.
  2. Agregar el **tamaño del registro** al comienzo del mismo.
  3. Usando **registros de longitud fija**.
     - Si hay un **tamaño máximo** de registro que nunca se sobrepasa, usamos registros de ese tamaño para albergar registros de longitud variable.
       Problema: desperdicia mucho espacio.
     - Un registro de longitud variable (RV) es representado por una lista enlazada de registros de longitud fija (RF).
       Problema: El primer RF del RV no desperdicia espacio, pero el resto sí (porque solo almacenan la información que no cabe en el primer RF).
       Solución: usar **bloques ancla** que contienen el primer RF, y usar **bloques de overflow** que contienen la continuación de la lista.
  4. Si los registros tienen una **clave**, usando un **índice**.
     - El índice posee la clave primaria de cada registro, ordenados alfabéticamente para poder hacer búsquedas binarias. Para cada clave se guarda el offset al archivo de datos.
     - El archivo de datos posee un encabezado que guarda el offset al primer registro libre. Luego, para cada registro de longitud variable, se guarda:
       - La longitud de ese registro (para saber cuánto ocupa)
       - Un bit bandera "libre/ocupado" (para poder borrar un registro)

- La cantidad de bytes libres del registro (en una modificación de un registro, puede que el nuevo registro sea más pequeño que el viejo; también se utiliza para encadenar registros libres)
- Datos

Algoritmo de borrado: buscar el registro en el índice, buscar el registro en el archivo de datos, marcarlo como "libre". Si el registro anterior o el siguiente están libres también, fundirlos en un solo registro libre. Borrar la clave del índice.
Algoritmo de agregado: Si hay registros libres, seleccionar uno[8] y guardar el registro ahí, y actualizar la lista de registros libres. Si no hay registros libres, agregarlo al final del archivo.

15 Modos de acceso a archivos

- **Secuencial**: los registros se acceden en orden, empezando por el principio, sin poder saltearse ninguno ni volver atrás.

  Formas de organización recomendadas si se prefiere este modo de acceso:

  - Organización secuencial (tira de bytes)
  - Organización balanceada secuencial (árbol B+ o B#)
  - Organización secuencial indexada (archivo secuencial + índice)

- **Relativo**: los registros se acceden con posicionamiento.

  Formas de organización recomendadas si se prefiere este modo de acceso:

  - Organización balanceada no secuencial (árbol B o B*)
  - Organización directa (mediante hashing)
  - Organización indexada

15.1 Modos de búsqueda

Sea n la cantidad de registros de un archivo.

- Búsqueda **secuencial**: $O(n)$
- Búsqueda **secuencial por bloques**: $O(n)$
- Búsqueda **directa**: $O(1)$
- Búsqueda **binaria**: $O(\log_2 n)$. El archivo debe estar ordenado.

16 Archivos secuenciales

Algoritmo 5 Primitivas de la organización secuencial

```
1 int S_OPEN (const char* nombreArchivo, int modoApertura);
2 int S_CLOSE (int fileHandler);
3 int S_READ (int fileHandler, void* regDestino);
4 int S_WRITE (int fileHandler, const void* regFuente, unsigned long cantBytes);
5 int S_DESTROY (const char* nombreArchivo);
```

[8] Estrategias de selección:
- First-fit: devolver el primer registro vacío libre tal que $tam(reg\,vacío) \geq tam(reg)$
- Best-fit: ordenar la lista de registros vacíos en orden **ascendente** por tamaño, devolver el primer registro vacío libre tal que $tam(reg\,vacío) \geq tam(reg)$
- Worst-fit: ordenar la lista de registros vacíos en orden **descendente** por tamaño, devolver el primero.

17 Archivos relativos

Algoritmo 6 Primitivas de la organización relativa

```
1  int R_OPEN (const char* nombreArchivo, int modo);
2  int R_CLOSE (int fileHandler);
3  int R_SEEK (int fileHandler, int numReg);
4  int R_READ (int fileHandler, int numReg, void* regDestino);
5  int R_READNEXT (int fileHandler, void* regDestino);
6  int R_WRITE (int fileHandler, int numReg, const void* regFuente);
7  int R_UPDATE (int fileHandler, int numReg, const void* regFuente);
8  int R_DELETE (int fileHandler, int numReg);
9  int R_DESTROY (const char* nombreArchivo);
10 int R_GETMAXREGS (int fileHandler);
```

18 Archivos directos y hashing

Archivo directo estático:

- El archivo tiene longitud fija y **no extensible**.
- Los **registros tienen longitud fija** y tienen una clave primaria.
- Cada registro del archivo posee un atributo extra que indica si el mismo está "libre", "ocupado", o fue "borrado" del archivo.
- Para realizar altas, bajas y modificaciones de registros en el archivo, se debe calcular su posición en el mismo con una **función de hashing**. Si dos registros hashean a la misma posición, la colisión debe resolverse.

Factor de carga del archivo: $\frac{\#\,registros\,ocupados}{\#\,registros\,disponibles}$

Algoritmo 7 Primitivas de la organización directa

```
1 int D_OPEN (const char* nombreArchivo, int modoApertura);
2 int D_CLOSE (int handler);
3 int D_READ (int fileHandler, void* regDestino);
4 int D_WRITE (int fileHandler, const void* regFuente);
5 int D_UPDATE (int fileHandler, const void* regFuente);
6 int D_DELETE (int fileHandler, const void* regFuente);
7 int D_DESTROY (const char* nombreArchivo);
```

Función de hashing: función f tal que $f(k) = offset$.

- Espacio de claves
- Espacio de direcciones

Claves sinónimas producen **colisión**: evento en el cual, para dos claves k_1 y k_2 ($k_1 \neq k_2$), se da que $f(k_1) = f(k_2)$.

18.1 Funciones de hashing perfectas

No produce colisiones. Se pueden construir si conocemos de antemano las claves a hashear.

18.2 Funciones de hashing no perfectas

Pueden producir colisiones.

- **Función módulo**: $f(k) = k \, mod \, N$. El valor de N debería ser primo para que la función disperse uniformemente.
  - Espacio de direcciones: $[0..N-1]$
- **Función multiplicación**: $f(k) = \lfloor N \cdot [k\,(x-1)] \rfloor$. El valor de x debería ser irracional.
  - Espacio de direcciones: $[0..N-1]$
- **Función *fold&add***: dividir la clave en partes de 2 o 4 bytes, aplicar la función de hashing a cada parte, y luego aplicar una operación que concatene los resultados (suma, XOR, etc.)

18.3 Aplicaciones

- **Rolling hashing** es una función de hash donde la entrada se hashea en una ventana que se mueve a lo largo de la entrada.
  - Algoritmo de Rabin-Karp: es un método que usa una función de hashing para buscar un patrón de string en un texto.

    $$H = c_1 a^{k-1} + c_2 a^{k-2} + \ldots + c_k a^0$$

    , donde a es constante y $c_1 \ldots c_k$ son los caracteres de la entrada.
- **Filtros de Bloom**: para verificar la pertenencia de un elemento en un conjunto.

18.4 Funciones de hashing unidireccionales

Una **función de hashing unidireccional** es de la forma $H(M) = k$. Tienen ciertos requisitos:

- Toma un valor de longitud variable M y devuelve un valor de longitud fija k (generalmente la longitud es de 128 bits).
- Dado M debe ser sencillo calcular k.
- Dado k debe ser muy difícil encontrar un M tal que $H(M) = k$.
- Dado M debe ser muy difícil encontrar un M' tal que $H(M) = H(M')$

Aplicaciones:

- Autenticación de documentos
- Firmas digiales
- Códigos de verificación

Algunas funciones de hashing unidireccionales:

1. **MD5**: devuelve un valor de 128 bits.
2. **SHA**: devuelve un valor de 160 bits.

18.5 Resolución de colisiones en archivos directos estáticos

1. **Búsqueda de posición libre**

   Sean las funciones de hashing $f(k) = offset$ y $g(k) = offset_g$ y el número de iteración i. P_i es la posición en el archivo relativo donde se debe comprobar la inserción o búsqueda.

   a) Linealmente y cíclicamente:

   $$P_i = f(k) + i$$

   Se inserta en el primer lugar disponible (registro "libre" o "borrado") a partir de $offset$. Si se llega al final del archivo, se vuelve al principio del mismo y se busca hasta llegar a $offset$. Si llegó a $offset$ es porque no hay lugar disponible.

   Las búsquedas de un registro terminan cuando:

   - Se encuentra el registro, o
   - Se encuentra un registro "libre", o
   - Se vuelve a la posición inicial $offset$ (luego de hacer un ciclo).

   Para borrar: buscar y marcar como "libre".

   Desventaja: aglomeramiento de sinónimos en porciones del archivo.

   b) Cuadrático y cíclico:

   $$P_i = f(k) + i^2$$

   Ídem anterior.

   c) Doble hashing:

   $$P_i = f(k) + i \cdot g(k)$$

2. **Uso de área de overflow**

   Es el lugar donde se guardan los registros sinónimos.

   a) En el mismo archivo de datos.

   1) Área de overflow distribuida. Los registros de overflow se intercalan con los archivos de datos (b registros de overflow por cada a registros de datos).
   $Posicion(k) = k + b(x \, div \, a)$
   $PosicionOverflow(k) = b(x \, div \, a) + a \, [(x \, div \, a) + 1]$
   2) Área de overflow al final.

   b) En otro archivo.

3. **Encadenamiento de claves sinónimos (direccionamiento cerrado)**

   a) En el mismo archivo: cada registro posee, además del flag antes mencionado ("libre", "ocupad", "borrado"), un puntero a otro registro sinónimo. Si el registro no tiene sinónimos, este puntero es -1.

   Método de doble pasada para cargar el archivo:

   1) Guardar las claves que no producen colisiones.
   2) Guardar las claves que producen colisiones, encadenando los sinónimos.

   b) En otro archivo: cada registro posee, además del flag antes mencionado, un puntero a un registro del archivo de overflow. Este a su vez posee encadenamiento de sinónimos (fin marcado con un puntero -1).

4. **Uso de buckets**: un bucket almacena una cantidad fija de registros. Cuando hay overflow en el bucket, se debe utilizar otra técnica de resolución de colisiones.

5. **Uso de varias funciones de hashing**

18.6 Hashing extensible

Permite que el archivo crezca sin cambiar la función de hashing.
Se necesitan 4 estructuras:

1. Archivo de tabla de dispersión, que guarda su **tamaño de tabla** (tt). La tabla posee punteros a bloques.
2. Archivo con bloques. Cada bloque soporta una **cantidad fija de registros de longitud fija**. Cada bloque tiene su **tamaño de dispersión** (td).

$$bloque.td = \frac{tabla.tt}{\#refs\, bloque\, en\, tabla} = distancia\, en\, tabla\, a\, otra\, ref$$

3. Archivo con lista de bloques libres.
4. Función de hashing f tal que $f(clave)$ =posición en la tabla que nos dará el puntero al bloque.

Métodos:

- **Mediante bit sufijos**:
  - Las claves se representan en base 10.
  - $f(clave) = clave \bmod tabla.tt$

Algoritmo 8 Alta mediante bits sufijos.

```
1   int posTabla = fhash(registro.clave)
2   int numeroBloque = tabla[posTabla]
3   Bloque bloque = bloques[numeroBloque]
4
5   if (bloque.contiene(registro.clave))
6     return RES_REGISTRO_DUPLICADO
7
8   if (bloque.cantRegsLibres > tabla.cantRegsPorBloque)
9     bloque.insertar(registro)
10    return RES_OK
11
12  // La clave no esta en el bloque y no se puede insertar en el
13  // El bloque aparece referenciado en la tabla solo 1 vez
14  if (bloque.td == tabla.tt)
15    tabla.duplicar()
16    tabla.tt *= 2
17
18    Bloque nuevoBloque // Uno nuevo o el primero de bloquesLibres
19    bloques.agregar(nuevoBloque) // Si cree uno nuevo
20
21    bloque.td *= 2
22    nuevoBloque.td = bloque.td
23
24    // Actualizo la referencia en la tabla
25    tabla[posTabla] = numNuevoBloque
26
27  // La clave no esta en el bloque y no se puede insertar en el
28  // El bloque aparece referenciado en la tabla mas de 1 vez
29  else:
30    Bloque nuevoBloque // Uno nuevo o el primero de bloquesLibres
31    bloques.agregar(nuevoBloque) // Si cree uno nuevo
32
33    bloque.td *= 2
34    nuevoBloque.td = bloque.td
35
36    // Actualizo las referencias en la tabla
37    int punteroInicial = numBloque
38    while (tabla[punteroInicial] == numBloque)
39      tabla[punteroInicial] = numNuevoBloque
40      punteroInicial = (punteroInicial + bloque.td) mod (tabla.tt)
41
42  bloque.redispersar() // Aplicar otra vez la funcion de hashing a sus elementos
43
44  insertar(registro, tabla, bloques)
```

Algoritmo 9 Baja mediante bits sufijos

```
1   int posTabla = funcionMod(registro.clave)
2   int numeroBloque = tabla[posTabla]
3   Bloque bloque = bloques[numeroBloque]
4
5   if (! bloque.contiene(registro.clave))
6     return RES_REGISTRO_NO_EXISTE
7
8   if (bloque.cantRegsLibres > 1)
9     bloque.eliminar(registro)
10    return RES_OK
11
12  // El bloque queda vacio, podemos liberarlo?
13  else:
14    int posTabla_atras = (posTabla - bloque.td / 2) mod (tabla.tt)
15    int posTabla_adelante = (posTabla - bloque.td / 2) mod (tabla.tt)
16
17    if (tabla[posTabla_atras] == tabla[posTabla_adelante):
18      // El bloque se lo puede agregar a "bloques libres"
19
20      bloquesLibres.agregar(numBloque)
21      numBloqueReemplazo = tabla[posTabla_atras]
22      Bloque reemplazo = bloques[numBloqueReemplazo]
23
24      // Se reemplazan sus referencias en la tabla por otro bloque
25      int punteroInicial = numBLoque
26      while (tabla[punteroInicial] != numBloqueReemplazo)
27        tabla[punteroInicial] = numBloqueReemplazo
28        punteroInicial = (punteroInicial + reemplazo.td) mod (tabla.tt)
29
30      reemplazo.td /= 2
31      if (tabla.mitadesIguales())
32        tabla.cortarPorMitad()
33
34    else:
35      // No se puede marcar como libre, queda vacio
36      return RES_OK
```

- **Mediante bits prefijos**
  - Las claves se representan de forma binaria.
  - La función de hash devuelve los $\log_2(tabla.tt)$ bits menos significativos de la clave, en base 10.

$$f(clave) = \left(clave_{[\log_2(tabla.tt),\ldots,0]}\right)_{10}$$

Algoritmo 10 Alta mediante bits prefijos.

```
1   int posTabla = fhash(registro.clave)
2   int numeroBloque = tabla[posTabla]
3   Bloque bloque = bloques[numeroBloque]
4
5   if (bloque.contiene(registro.clave))
6     return RES_REGISTRO_DUPLICADO
7
8   if (bloque.cantRegsLibres < tabla.cantRegsPorBloque)
9     bloque.insertar(registro)
10    return RES_OK
11
12  // La clave no esta en el bloque y no se puede insertar en el
13  // El bloque aparece referenciado en la tabla solo 1 vez
14  if (bloque.td == tabla.tt)
15    tabla.duplicar() // Elemento a elemento
16    tabla.tt *= 2
17
18    Bloque nuevoBloque // Uno nuevo o el primero de bloquesLibres
19    bloques.agregar(nuevoBloque) // Si cree uno nuevo
20
21    bloque.td *= 2
22    nuevoBloque.td = bloque.td
23
24    // Actualizo la referencia en la tabla
25    tabla[posTabla] = numNuevoBloque
26
27  // La clave no esta en el bloque y no se puede insertar en el
28  // El bloque aparece referenciado en la tabla mas de 1 vez
29  else:
30    int viejoTT = tabla.tt
31    Bloque nuevoBloque // Uno nuevo o el primero de bloquesLibres
32    bloques.agregar(nuevoBloque) // Si cree uno nuevo
33
34    bloque.td *= 2
35    nuevoBloque.td = bloque.td
36
37    // Actualizo las referencias en la tabla
38    int punteroInicial = 0
39    while (punteroInicial != viejoTT)
40      if (tabla[punteroInicial] == numBloque)
41        tabla[punteroInicial] = numNuevoBloque
42      punteroInicial += 1
43
44  bloque.redispersar()
45
46  insertar(registro, tabla, bloques)
```

Algoritmo 11 Baja mediante bits prefijos

```
1   int posTabla = funcionMod(registro.clave)
2   int numeroBloque = tabla[posTabla]
3   Bloque bloque = bloques[numeroBloque]
4
5   if (! bloque.contiene(registro.clave))
6     return RES_REGISTRO_NO_EXISTE
7
8   if (bloque.cantRegsLibres > 1)
9     bloque.eliminar(registro)
10    return RES_OK
11
12  // El bloque queda vacio, ¿podemos liberarlo?
13  else:
14    // Calculo el vector donde posiblemente haya bloques de reemplazo
15    vector<int> posicionesTabla
16    int cantBitsEspacioBusqueda = log(tabla.tt,2)
17    int cantBitsPrefijoClave = log(bloque.td,2)
18    claveConUltimoBitPermutado = clave.permutarUltimoBit()
19
20    int cantidadVariantes = cantBitsEspacioBusqueda - cantBitsPrefijoClave
21    for (int i = 0; i < 2; i ++)
22      variante = getNextPermutacion (cantidadVariantes)
23      posicionesTabla.agregar(claveConUltimoBitPermutado + variante)
24
25    if (bloque.td == tabla.tt)
26      // El bloque se lo puede agregar a "bloques libres"
27
28      bloquesLibres.agregar(numBloque)
29      numBloqueReemplazo = tabla[posicionesTabla(1)]
30      Bloque reemplazo = bloques[numBloqueReemplazo]
31
32      // Se reemplaza su referencia en la tabla por otro bloque
33      tabla[numBloque] = numBloqueReemplazo
34      reemplazo.td /= 2
35
36    else:
37      if (tabla[posicionesTabla(1..etc)].elementosIdenticosEntreSi()):
38        // El bloque se lo puede agregar a "bloques libres"
39
40        bloquesLibres.agregar(numBloque)
41        numBloqueReemplazo = tabla[posicionesTabla(1)]
42        Bloque reemplazo = bloques[numBloqueReemplazo]
43
44        // Se reemplazan sus referencias en la tabla por otro bloque
45        int punteroInicial = numBLoque
46        while (tabla[punteroInicial] != numBloqueReemplazo)
47          tabla[punteroInicial] = numBloqueReemplazo
48          punteroInicial = (punteroInicial + reemplazo.td) mod (tabla.tt)
49
50        reemplazo.td /= 2
51        if (tabla.mitadesIguales())
52          tabla.cortarPorMitad()
53
54      else:
55        // No hay bloques de reemplazo, queda vacio.
56        return RES_OK
```

19 Archivos indexados

Clave: expresión derivada de uno o más campos de un registro que se utiliza para localizar el mismo. Debe ser:

- Canónicos: debe haber una sola representación para dos claves similares. Ejemplo: "The Beatles", "THE BEATLES", "the beatles" deberían mapear a "beatles".

- No cambiante: si el registro cambia y la clave cambia, luego tendrá que reorganizarse el archivo.

Clave primaria: debe ser:

- Sin datos: no deben contener información, porque no sabemos si esa información podría repetirse en el futuro.
- Única: no puede haber dos registros con la misma clave primaria.

Índice: herramienta para **buscar registros**. Se basan en los conceptos de "clave,referencia".

- Un índice permite imponer orden en un archivo sin tener que ordenar el archivo.
- Un índice permite buscar un registro en un archivo de registros variables de forma rápida.
- Varios índices sobre un archivo permiten tener varias formas de acceder a un registro.

Árbol balanceado: la distancia más corta a un nodo hoja difiere en 1 con la distancia más larga.

Árbol completamente balanceado: todos los caminos desde la raíz hasta un nodo hoja son del mismo largo.

Para manejar archivos indexados con registros de longitud variable, en vez de contar la cantidad de registros que caben en un nodo, se cuenta la cantidad de bytes que caben.

Algoritmo 12 Primitivas de la organización indexada

```
int I_DESTROY (char* nombreArchivo);
int I_OPEN (const char* nombreArchivo, int modoApertura);
int I_CLOSE (int fileHandler);
int I_ADD_INDEX(int handler, const campo* clave);
int I_DROP_INDEX (int fileHandler, int indexId);
int I_IS_INDEX (int fileHandler, campo* clave);
int I_START (int fileHandler, int indexId, char* operador, const void* valorReferencia);
int I_READ (int fileHandler, void* regDestino);
int I_READNEXT (int fileHandler, int indexId, void* regDestino);
int I_WRITE (int fileHandler, const void* reg);
int I_UPDATE (int fileHandler, const void* reg);
int I_DELETE (int fileHandler, const void* reg);
```

19.1 Árbol B (indexado)

Árbol B: estructura que intenta acceder y mantener un índice que es muy grande para mantener en memoria.

- Todos los nodos mantienen elementos almacenados. El **orden** de un árbol B es la cantidad máxima de nodos descendientes que puede tener un nodo interno.
- Cantidad de *seeks* necesarias para acceder por referencia a una clave:

$$1 + \log_{\lceil \frac{k}{2} \rceil} \left(\frac{n+1}{2} \right)$$

  donde $k =$ cantidad de registros lógicos, $n =$ factor de bloqueo.
- Son árboles anchos y poco profundos. Todas las ramas tienen igual profundidad.
- Cada **nodo** del árbol posee:
  - Claves: k como máximo, $\lfloor \frac{k}{2} \rfloor$ como mínimo.
  - Punteros a nodos descendientes: $k+1$ como máximo, $\lceil \frac{k+1}{2} \rceil$ como mínimo.
- La **raíz** puede no tener claves (si el árbol está vacío), y puede tener 0 descendientes, o más de 2.

- Los nodos **hoja** no tienen descendientes.
- Entre la cantidad mínima de claves y la cantidad mínima de descendientes, hay una diferencia de 1.
- El árbol está **ordenado** por clave. Cada clave en un nodo tiene dos referencias: una a un nodo con elementos de clave menor, y otra a un nodo con elementos de clave mayor.

```
  BTreeNode root
  root.isLeaf = true
  root.elements = 0
  disk-write(root)
  this->root = root

BTreeNode BTree::search (motherNode, key):
  if motherNode.contains(key):
    return motherNode
  if motherNode.isLeaf():
    return NULL // Key not found

  BTreeNode nextToFind = NULL
  for each keyi in motherNode:
    if motherNode[keyi] > key:
      // Element with next higher key
      nextToFind = motherNode[keyi].leftPointer

  // Element with node's max key
  nextToFind = motherNode[maxKey].rightPointer

  return (search (nextToFind, key))
```

```
  BTreeNode leafNode = search(key)
  if (leafNode.isFull())
    // Split
    BTreeSuperNode supernode = leafNode
    supernode.insert(key)
    supernode.split()  // 2-3 o 3-2
    supernode.promote(median) //More splits may be necessary

  else:
    leafNode.insert(key)
```

```
  BTreeNode node = find(this->root,key)
  if (node == NULL)
    return false
  if (node.isLeaf())
    node.delete(key)
    if (node.underflow())
      if (node.sibling.hasExtraKeys())
        // Bajar uno del padre al nodo y
        // subir uno del sibling al padre.
        // El sibling es siempre el izquierdo o siempre el derecho
        node.redistributeWithSibling(node.sibling)
      else:
        // Juntar dos nodos en uno y bajar el separador del padre
        node.concatenateWith(node.sibling, node.father)
        if (node.father.underflow())
          node.father.redistributeWithSibling()
  else:
    BTreeNode immediate = node.immediatelySuperiorORinferior()
    swap(node[key], immediate[key])
    immediate.delete (key)
```

19.2 Árbol B* (indexado)

Es una variante del árbol B.

- Cada **nodo** del árbol posee:
  - Claves: k como máximo, $\lfloor \frac{2}{3}k \rfloor$ como mínimo.
  - Punteros a nodos descendientes: $k+1$ como máximo, $\lceil \frac{2k+1}{3} \rceil$ como mínimo.
- Se realizan redistribuciones en el alta.
- Los split se realizan tomando 2 *siblings* y dividiéndolos en 3 nodos, donde cada uno tiene una carga de $\frac{2}{3}$.
  - La **raíz** es un **caso especial**, porque no tiene *siblings*. Entonces, para hacer un split de la raíz, puede elegirse entre las siguientes alternativas:
    - Manejar la raíz como un nodo de un árbol B
    - Raíz con más capacidad que los demás nodos, para que cuando hagamos un split, los dos nodos resultantes estén $\frac{2}{3}$ llenos,
    - Raíz con mucha más capacidad que los demás nodos, para que cuando hagamos un split, los tres nodos resultantes estén $\frac{2}{3}$ llenos

Ventajas con respecto a árbol B	*Desventajas con respecto a árbol B*
Menor altura del árbol, y por ende menor tiempo de búsqueda Aprovecha mejor el espacio en el nodo	Altas y bajas más costosas, porque se redistribuye

19.3 Árbol B+ (indexado secuencial)

- Es una combinación entre un set indexado y un set secuencial. Por lo tanto, provee acceso **secuencial** e **indexado**.
- El set secuencial está formado por **bloques** (no necesariamente contiguos físicamente en disco) que contienen registros ordenados, y dos punteros (uno al bloque anterior, y uno al siguiente).
- El set indexado es un índice al set secuencial. Es un árbol B que contiene:
  - Registros de **longitud fija** (usualmente del mismo tamaño de los bloques del set secuencial) que tienen **claves** para distinguir entre dos bloques. Cada nodo contiene N separadores y $N+1$ descendientes.

  ó

  - Registros de longitud variable que tienen:

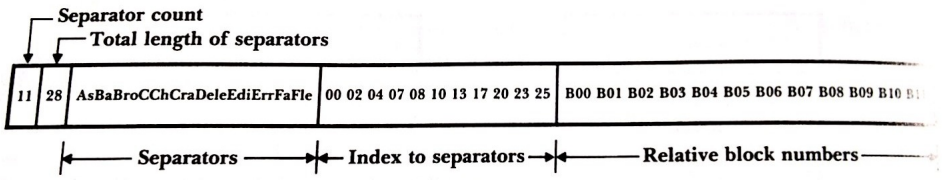


Figura 17: Index set block

    - Contador de separadores (para poder hacer una búsqueda binaria sobre los mismos)
    - Longitud total de los separadores (para poder acceder al índice de separadores)

- Separadores concatenados
- Índice a los separadores (la longitud de cada separador se obtiene con el puntero siguiente
- Puntero a otros bloques

```
BPlusTreeBlock block = findSequentialSetBlock(key)
if (block + key = overflow):
  block.split() // no se promueven claves, se distribuyen en 2 bloques
  // Actualizar index set
  if (node of index set in overflow)
    // resolver como en arbol B.
else:
  block.insert(key)
  // Actualizar index set si corresponde.
```

```
BPlusTreeBlock block = findSequentialSetBlock(key)
if (block - key = underflow):
  if (block.rightNeighbour.load =  1/2)
    concat(block, block.rightNeighbour)
    markDeleted(block) // Con un mapa de bits
    // Actualizar index set
    indexset.remove(separator)
  if (block.neighbors.load > 1/2)
    redistributeLoad(block,block.neighbors)
    indexset.update(separator)
else:
  block.delete(key)
  // El index set no cambia.
```

19.4 Árbol B# (indexado secuencial)

20 Ordenamiento de archivos

Se tiene un archivo formado por registros con claves. Los registros están desordenados. Se lo quiere ordenar.

1. **Métodos Internos**: el tamaño del archivo no puede ser mayor al tamaño de la memoria disponible.
   a) Heapsort
   b) Quicksort
   c) Keysort: la suma de los tamaños de las claves de los registros no puede ser mayor al tamaño de la memoria disponible.

```
{
  vector<pair<Clave,Puntero>> claves = archivoDesordenado.getClaves()
  claves.sort() // Ordenar por Clave
  for parClavePuntero in claves:
    Registro registro = archivoDesordenado.getRegistro(parClavePuntero.second)
    archivoOrdenado.append(registro)
}
```

   Problema: Hay que leer el archivo dos veces, y la segunda lectura no es en forma secuencial.

2. **Métodos Externos**: se carga el archivo parcialmente en memoria, se ordena, se graban copias ("particiones") y luego se hace *merge* de estas particiones.
   a) **Mergesort**: genera particiones del tamaño de la memoria.

Algoritmo 13 Mergesort

```
{
  Registro registroLeido = archivoDesordenado.getNextRegistro()
  registrosLeidos.append(registroLeido)
  memoriaLibre = memoriaLibre - registroLeido.tamanio
  cantRegsLeidos ++
}

archTempOrdenado _generarParticion(archivoDesordenado, cantRegsArchivo, cantRegsLeidos, tamanioMemoria)
{
  vector<Registro>* registrosLeidos = new Vector()
  int memoriaLibre = tamanioMemoria

  while (cantRegsLeidos < cantRegsArchivo)
  {
    agregarRegistroAmemoria (archivoDesordenado, *registrosLeidos, &cantRegsLeidos, &memoriaLibre)

    while (cantRegsLeidos < cantRegsArchivo && memoriaLibre > registroLeido->tamanio())
    {
        agregarRegistroAmemoria (archivoDesordenado, *registrosLeidos, &cantRegsLeidos, &memoriaLibre)
    }
  }

  registrosLeidos.sort()
  archTempOrdenado.agregar(registrosLeidos)
  delete(registrosLeidos)  // Libero la memoria
}

void mergesort (archivoDesordenado, archivoOrdenado)
{
  int cantRegsArchivo = archivoDesordenado.getCantRegs()
  int cantRegsLeidos = 0
  vector<archTempOrdenado> particiones //referencias a archivos

  // Genero las particiones
  while (cantRegsLeidos < cantRegsArchivo)
  {
    archTempOrdenado = _generarParticion(archivoDesordenado,cantRegsArchivo, &cantRegsLeidos, tamanioMemoria)
    particiones->append(archTempOrdenado)
  }

  mergePolifasico(particiones, archivoOrdenado)
}
```

b) **Mergesort con Replacement Selection**: usa dos *heaps*.
Mejor caso: archivo ordenado. Genera una partición del tamaño del archivo.
Peor caso: archivo ordenado en forma inversa. Particiones de tamaño $tamaño\,memoria$ como un sort interno.
Caso medio: archivo con orden aleatorio. Particiones de tamaño $2 * tamaño\,memoria$

ARCHIVO = 2,1,3,5,7,8,2,6,4,1,2,0,3,5
Tamaño registro = 1 byte, Tamaño RAM = 3 bytes

Lectura	**Heap primario en memoria**	**Heap secundario en memoria**	**Salida**	***Comentarios***
2,1,3	1,2,3	-	1	
5	2,3,5	-	2	
7	3,5,7	-	3	
8	5,7,8	-	5	
2	7,8	2	7	2 es menor a 5, va al heap secundario
6	8	2,6	8	6 es menor a 7, va al heap secundario. Se vació el heap primario
	-	2,6		
	2,6	-		El heap 2 pasa a memoria
4	2,4,6	-	2	
1	4,6	1	4	1 es menor a 2, va al heap secundario
2	6	1,2	6	2 es menor a 4, va al heap secundario. Se vació el heap primario
	-	1,2		
	1,2	-		El heap 2 pasa a memoria
0	0,1,2	-	0	
3	1,2,3	-	1	
5	2,3,5	-	2	Se terminó el archivo
	3,5		3	Vaciamos el heap primario
	5		5	Vaciamos el heap primario

Cuadro 3: Ejemplo de Replacement Selection

Algoritmo 14 Replacement Selection

```
{
  vector<archivoOrdenado> particiones
  vector<Registro> registros = llenarMemoria(tamanioMemoria, archivoDesordenado)
  Heap heapPrimario (registros)
  Heap heapSecundario () // Aca se guardan los registros congelados
  archivoOrdenado
  particiones.agregar(archivoOrdenado)

  bool hayMasRegistros = !archivoDesordenado.eof()
  mientras (hayMasRegistros && ! heapPrimario.vacio())
  {

    Registro regMinimo = heapPrimario.quitarMinimo()
    archivoOrdenado.agregar(regMinimo)

      si (heapPrimario.vacio()
      {
        archivoOrdenado.close()
        archivoOrdenado = open(nuevo archivoOrdenado)
        particiones.agregar(archivoOrdenado)

        vector<Registro> registros = llenarMemoria(tamanioMemoria, heapSecundario)
        Heap heapPrimario (registros)
        continue
      }

    Registro regLeido = archivoDesordenado.getNextRegistro()
    si (regLeido.clave < regMinimo.clave)
      heapSecundario.insertar(regLeido)

    si (regLeido.clave > regMinimo.clave)
      heapPrimario.insertar(regLeido)
  }
  return particiones
}

void mergesort(archivoDesordenado, particiones, archivoOrdenado, tamanioMemoria)
{
  vector<archivoOrdenado> particiones = replacementSelection(archivoDesordenado, tamanioMemoria)
  mergepolifasico(particiones,archivoOrdenado)
}
```

c) **Mergesort con Natural Selection:**

Mejor caso:

Peor caso:

Caso medio: archivo con orden aleatorio. Particiones de tamaño $3 * tamaño memoria$

ARCHIVO = 2,1,3,5,7,8,2,6,4,1,2,0,3,5
Tamaño registro = 1 byte, Tamaño RAM = 3 bytes

Lectura	**Memoria**	**Freezer (archivo)**	**Salida**	***Comentarios***
2,1,3	2,1,3	-	1	Llenamos la memoria
5	2,5,3	-	2	
7	7,5,3	-	3	
8	7,5,8	-	5	
2	7,8	2		2 es menor a 5, va al freezer
6	6,7,8	2	6	
4	7,8	2,4		4 es menor a 6, va al freezer
1	7,8	2,4,1		1 es menor a 6, va al freezer (que se llena)
		-	7	Vaciamos la memoria ordenadamente
		-	8	Vaciamos la memoria ordenadamente
	2,4,1	-	1	El freezer pasa a memoria
2	2,4,2	-	2	
0	2,4	0		0 es menor a 2, va al freezer
3	2,4,3	0	2	
5	5,4,3	0	3	
	5,4	0	4	
	5	0	5	
	-	0		El archivo se terminó.
	0	-	0	Copiamos lo del freezer a memoria. Nueva partición.

Cuadro 4: Ejemplo de Natural Selection

Algoritmo 15 Natural Selection

```
{
  vector<archivoOrdenado> particiones
  vector<Registro> registros = llenarMemoria(tamanioMemoria, archivoDesordenado)
  Heap heap (registros)
  archivoOrdenado archivoOrdenado
  particiones.agregar(archivoOrdenado)

  archivoOrdenado freezer (tamanioMemoria) // Aca se guardan los registros congelados

  bool hayMasRegistros = !archivoDesordenado.eof()
  mientras (hayMasRegistros)
  {
    Registro regMinimo = heap.quitarMinimo()
    archivoOrdenado.agregar(regMinimo)

    Registro regLeido = archivoDesordenado.getNextRegistro()
    si (regLeido.clave < regMinimo.clave)
      heap.insertar(regLeido)

    si (regLeido.clave > regMinimo.clave)
      freezer.agregar(regLeido)
      si (freezer.lleno())
      {
        while (! heap.vacio())
          regMinimo = heap.quitarMinimo()
          archivoOrdenado.agregar(regMinimo)
          registros = llenarMemoria(tamanioMemoria, freezer)
          heap (registros)
      }

  }
  si (! freezer.vacio()) //Quedaron registros sin grabar a disco
  {
    vector<Registro> registros = llenarMemoria(tamanioMemoria, freezer)
    registros.sort()
    particiones.agregar(archivoOrdenado(registros))
  }
  return particiones
}

void mergesort(archivoDesordenado, particiones, archivoOrdenado, tamanioMemoria)
{
  vector<archivoOrdenado> particiones = naturalSelection(archivoDesordenado, tamanioMemoria)
  mergepolifasico(particiones,archivoOrdenado)
}
```

d) **Radix Sort**: solo sirve para ordenar números enteros.
Mejor caso = peor caso = caso medio = $O(nk)$, donde n es la cantidad de elementos y k es la cantidad máxima de dígitos de los números.

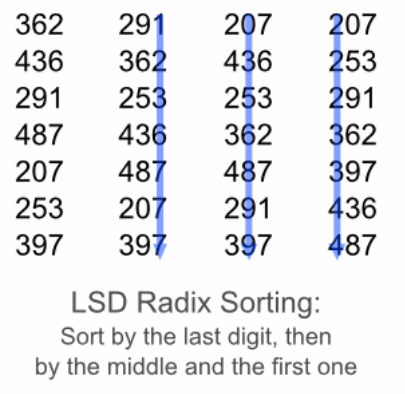


Figura 18: Radix Sort

e) **Bucket Sort**: ordena eficientemente n números, donde cada número pertenece al rango $[0..N-1]$, y N no es mucho más grande que n, o $N < n$.

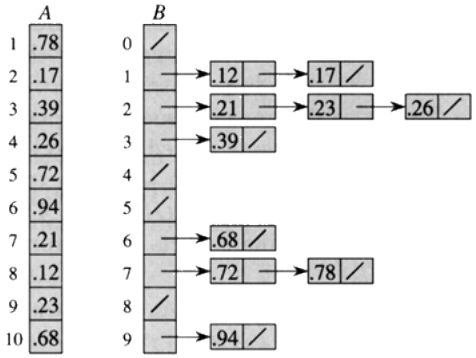


Figura 19: Bucket Sort

21 Operaciones entre 2 o más archivos

21.1 Intersección (*match*)

Se tienen dos archivos ordenados por clave. Se quiere obtener un archivo que contenga los registros que están en ambos archivos.

```
1  {
2    boolean masElementos = proximoItem(lista1) && proximoItem(lista2)
3    mientras (masElementos):
4      si (item(lista1) < item(lista2)):
5        masElementos = proximoItem(lista1)
6      si (item(lista1) > item(lista2)):
7        masElementos = proximoItem(lista2)
8      si (item(lista1) == item(lista2)):
9        salida.agregar(item(lista1))
10     masElementos = proximoItem(lista1) && proximoItem(lista2)
11 }
```

21.2 Unión (*merge*)

Se tienen dos archivos ordenados por clave. Se quiere obtener un archivo que contenga los registros que están en uno o en ambos archivos.

```
{
  boolean masElementosLista1 = proximoItem(lista1)
  boolean masElementosLista2 = proximoItem(lista2)
  mientras (masElementosLista1 || masElementosLista2):
    si (item(lista1) < item(lista2)):
      salida.agregar(item(lista1))
      masElementos = proximoItem(lista1)
    si (item(lista1) > item(lista2)):
      salida.agregar(item(lista2))
      masElementos = proximoItem(lista2)
    si (item(lista1) == item(lista2)):
      salida.agregar(item(lista1))
      masElementosLista1 = proximoItem(lista1)
      masElementosLista2 = proximoItem(lista2)
}
```

21.3 *Merge* de k vías

Objetivo: hacer la unión de k archivos, donde $k \leq 8$.

Entrada: k listas ordenadas, cada una de tamaño n.

Salida: lista de tamaño nk

Orden: $O(nk^2)$

```
  // Crear lista de indices
  int indices[1..k] = 0

  int contador = 0
  mientras (contador < kn):
    // Obtener el numero de lista
    // cuyo proximo elemento sin procesar es el minimo
    // (ignora listas para las cuales indices[j] > n)
    para cada j entre 1 y k:
      int minimo = min (listas[j][indices[j]])

    salida[contador] = listas[p][indices[p]]
    indice[p] += 1
    contador += 1
```

21.4 Apareo

Se tiene un archivo "maestro" y un archivo "transaccional" que reporta cambios sobre el "maestro". Los archivos están ordenados por la misma clave y tienen el mismo formato de registro, con la excepción de que los registros de los archivos "transaccionales" deben tener un campo extra que indique si es una transacción "alta", "baja", o "modificación".

```
  boolean masRegistrosMaestros = proximoItem(listaMaestra)
  boolean masRegistrosTransacc = proximoItem(listaTransacciones)
  mientras (masRegistrosMaestros || masRegistrosTransacc)
  {
    si (item(listaMaestra) < item(listaTransacciones))
    {
      salida.agregar(item(listaMaestra))
      masRegistrosMaestros = proximoItem(listaMaestra)
    }
    si (item(listaMaestra) > item(listaTransacciones))
    {
      si (operacion(item(listaTransacciones)) == "alta"):
        salida.agregar(item(listaTransacciones))
      si (operacion(item(listaTransacciones)) == "baja"):
        mostrar(ERROR EN BORRADO POR REGISTRO INEXISTENTE)
      si (operacion(item(listaTransacciones)) == "modificacion"):
        mostrar(ERROR EN MODIFICACION POR REGISTRO INEXISTENTE)
```

```
18        masRegistrosTransacc = proximoItem(listaTransacciones)
19      }
20      si (item(listaMaestra) == item(listaTransacciones))
21      {
22        si (operacion(item(listaTransacciones)) == "alta"):
23          mostrar(ERROR EN ALTA POR REGISTRO DUPLICADO)
24        si (operacion(item(listaTransacciones)) == "modificacion"):
25          salida.agregar(item(listaTransacciones)
26
27        masRegistrosMaestros = proximoItem(listaTransaccionesMaestra)
28        masRegistrosTransacc = proximoItem(listaTransacciones)
29      }
30    }
```

Parte III

FTRS (*Full Text Retrieval System*)

Objetivo: encontrar material de naturaleza no estructurada, que satisface una necesidad de información, en una colección muy grande. La necesidad de información se expresa en una *query*, y se quiere ordenar los documentos de más a menos relevantes, y presentarle al usuario un subconjunto de los más relevantes.

- Los documentos no tienen estructura clara. No hay campos definidos.
- No existe una respuesta correcta para una consulta.
- Cada documento puede ser más o menos relevante a una consulta. La relevancia es subjetiva.
- No sólo importa la velocidad de respuesta sino la calidad de la respuesta.
- Se intenta buscar una aproximación a lo que el usuario pide.

Término: unidad mínima de información consultable.
Documento: unidad mínima de información recuperable. Puede ser un un capítulo o un párrafo, una página web, o un libro completo.
Colección: puede ser homogénea (es decir, todos los documentos tratan de un tema) o no homogéneas (por ejemplo, la Web).
Etapas para alcanzar el objetivo:

1. Elegir el **modelo** que permita calcular la relevancia de un documento frente a una consulta.

   "Cómo adivinar lo que el usuario quiso preguntar"

2. Diseñar algoritmos y estructuras de datos que lo implementen de forma eficiente (**índice**).

   "Cómo no tardar demasiado"

22 Indexación

Modelos clásicos:

- Booleano
  - Conjuntos difusos
  - Extendido
- Vectorial
  - Generalizado
  - LSI (Latent Semantic Indexing)
  - Redes Neuronales
- Probabilístico
  - Redes bayesianas
  - Redes de inferencia bayesiana

Índice invertido: índice de clasificación de documentos por término. En la versión básica, por cada término se tiene la lista de documentos en los que aparece el término. El conjunto de términos es el **vocabulario** o **léxico**. Las listas de cada término son **listas invertidas**.
Un índice tiene **granularidad**, que es la precisión con la que se guardan las posiciones de cada término en cada documento (*word level, paragraph level, etc.*).
Como regla general, los índices invertidos son mejores a los *signature files* y los *bitmaps* tanto en espacio ocupado como en velocidad de resolución de consultas.

22.1 Modelo booleano

Formas de implementación:

1. **Índice invertido**. Formadas por dos estructuras:

   a) Vocabulario: Estructura de búsqueda. Suele ser un árbol B.

   $$< \text{i(termino)}, freq_t, \text{offset lista invertida} >$$

   La frecuencia de cada término se almacena aquí para resolver las consultas más eficientemente (sin tener que acceder a sus listas invertidas), del término menos a más frecuente.

   b) Listas Invertidas: una por cada término, contiene una lista **ordenada de forma ascendente** de los **números** de documentos donde se encuentra ese término.

   $$< d_1, d_2, \ldots, d_{freq_t} >$$

   Si el índice es posicional:

   $$< \left\{ d_1, freq_{t_{d_1}}, \left[pos_1, \ldots, pos_{freq_{t_{d_1}}} \right] \right\}; \ldots; \left\{ d_{freq_t}, freq_{t_{d_{freq_t}}}, \left[pos_1, \ldots, pos_{freq_{t_{d_{freq_t}}}} \right] \right\} >$$

Algoritmo 16 Fase 1 de 2: *scanning*

```
indice de archivos = {}
archivo temporal = {}
archivo de lexico = {}

para cada "doc" en coleccion:

  indice de archivos += <num doc, doc>

  para cada palabra en doc:
    si palabra es termino indexable:
      si termino no esta en lexico:
        archivo de lexico += termino

      num termino = posicion de "termino" en "lexico"
      archivo temporal += <num doc, num termino, pos termino>

  num doc ++

return archivo temporal // Este archivo esta ordenado por numero de doc
return lexico // Esto deberia estar en memoria al resolver consultas
```

Algoritmo 17 Fase 2 de 2: *indexing*

```
archivo indice = {}
archivo invertido de documentos = {}
archivo invertido de posiciones = {}

ordenar "archivo_temporal" por "num_termino" y luego por "num_doc" y luego por "pos_termino"

num termino actual = primer "num_termino"

mientras ("archivo_temporal" tenga registros):

  leer <num term, num doc, pos termino>
  frecuencia = 0

  mientras (num termino actual == "num_term"):
    lista posiciones += pos termino
    lista documentos += <num doc, offset lista posiciones>
    frecuencia ++

  puntero a indice invertido = ftell(archivo invertido)
  archivo invertido de documentos += lista documentos
  archivo indice += <lexico[num term], frecuencia, puntero a indice invertido>
```

2. **Índices de firmas** (*bitstring signature file*): se representa a cada documento mediante una firma única de B bits. Se utilizan N funciones de hash (si $N = 1$, debería ser $B =$ tamaño del léxico). Cada función de hash devuelve un valor entre 0 y $B - 1$.

   Sean:

   - b: cantidad máxima de bitslices a procesar para resolver una consulta
   - p: probabilidad de que un bit de una firma valga "1"

     $$p = \sqrt[b]{\left(\frac{z}{N}\right)}$$

   - z: cantidad tolerable de falsos positivos en una consulta

     $$z = p^b \cdot N$$

   - f: cantidad de pares "término, documento" únicos
   - q: cantidad de términos en una consulta
   - N: cantidad de documentos en la colección
   - B: cantidad de funciones de hashing a usar

     $$B = \frac{f}{N} \cdot \frac{b}{q}$$

   - W: cantidad de bits de la firma

     $$W = \frac{1}{1 - \sqrt[B]{1-p}}$$

   Receta: $W = \frac{V \times B}{0{,}8}$

Algoritmo 18 Indexar mediante índices de firmas

```
para cada doc en coleccion:
  firma de doc = B bits en cero
  para cada term en doc:
    para cada funcion en funcionesHash:
      numero bit = funcion(term)
      encender "numero_bit" en la firma de doc //puede haber colisiones!
  guardar firma de doc en archivo secuencial
```

Algoritmo 19 Consultar mediante índices de firmas

```
documentos candidatos = {}
firma de consulta = B bits en cero

para cada term de consulta:
  para cada funcion en funcionesHash:
    numero bit = funcion(term)
    encender "numero_bit" en la firma de consulta //puede haber colisiones!
para cada doc de coleccion:
  si (firma(doc) AND firma de consulta == firma de consulta)
    documentos candidatos += doc

resultados = revisar(documentos candidatos) // descarto docs que tengan falsos positivos
devolver resultados
```

Ventajas	Desventajas
No se necesita mantener en memoria el léxico durante la resolución de consultas.	Produce falsos positivos. (Para disminuir la cantidad hay que agregar más bits a la firma, o agregar más funciones de hash).
Son eficientes para consultas conjuntivas de muchos términos.	No soportan consultas rankeadas.
	2 o 3 veces más grande que un índice invertido.
	Se necesita leer todo el archivo para resolver una consulta.
	La función de hash debe generar valores distribuidos uniformemente.
	Son ineficientes para colecciones donde el tamaño de cada documento es muy variable.

3. **Índices de porciones de firmas** (*bitslice signature file*): cada registro i del archivo índice (secuencial) representa los bits en la $i-$ésima posición de la firma de todos los documentos.

Algoritmo 20 Indexar mediante índices de porciones firmas

```
archivo indice = {}

para cada doc en coleccion:
  para cada term en doc:
    para cada funcion en funcionesHash:
      numero bit = funcion(term)
      archivo temporal += <numero bit, id doc>

ordenar archivo temporal por "numero bit + id doc"

para cada "numero bit" en archivo temporal:
  porcion = {}
  para cada "id doc":
    encender bit n "id doc" en la porcion
  archivo indice += porcion

devolver archivo indice
```

Algoritmo 21 Consultar mediante índices de porciones de firmas

```
documentos candidatos = {}
firma de consulta = B bits en cero
para cada term de consulta:
  para cada funcion en funcionesHash:
    numero bit = funcion(term)
    encender "numero bit" en la firma de consulta // puede haber colisiones
para cada bit encendido en la firma de la consulta:
  chequear el registro n "bit encendido" del archivo indice
  documentos candidatos += documentos que tienen 1 en el registro

resultados = revisar(documentos candidatos) // descarto docs que tengan falsos positivos
devolver resultados
```

Ventajas	Desventajas
Comparado con índice de firma: solo hay que leer FN registros, donde N es la cantidad de términos de la consulta y F la cantidad de funciones de hash.	

4. **Bitmaps**: cada registro i del archivo índice (secuencial) representa un término i. Cada registro tiene tantos bits como documentos halla (j):

$$p(t_i, d_j) = \begin{cases} 1 & \text{si } t_i \in d_j \\ 0 & \text{si } t_i \notin d_j \end{cases}$$

Algoritmo 22 Indexación con bitmaps.

```
N = cantidad de docs
para cada doc en coleccion:
  para cada term en doc:
    si no esta en vocabulario:
      agregar <term, id term> a vocabulario
    si esta:
      obtener su id term de vocabulario
    agregar <id term, id doc> a archivo temporal

ordenar archivo temporal por "id term + id doc"
para cada id term en archivo temporal:
  firma term = N bits en 0
  para cada id doc:
    encender bit "id doc " de firma term

  agregar firma term a bitmap
```

Algoritmo 23 Consultas con bitmaps.

```
candidatos = {}
para cada term en consulta:
  obtener su id term de vocabulario
  docs = obtener registro "id term" del bitmap
  candidatos += docs

revisar candidatos para descartar falsos positivos
```

Ventajas	Desventajas
No produce falsos positivos.	Desperdicia mucho espacio (si hay N documentos y n términos, ocupa Nn bits).
Eficiente para consultas booleanas.	
Pueden armarse índices secundarios sobre él.	

Los bitmaps se pueden comprimir con árboles de derivación binaria.

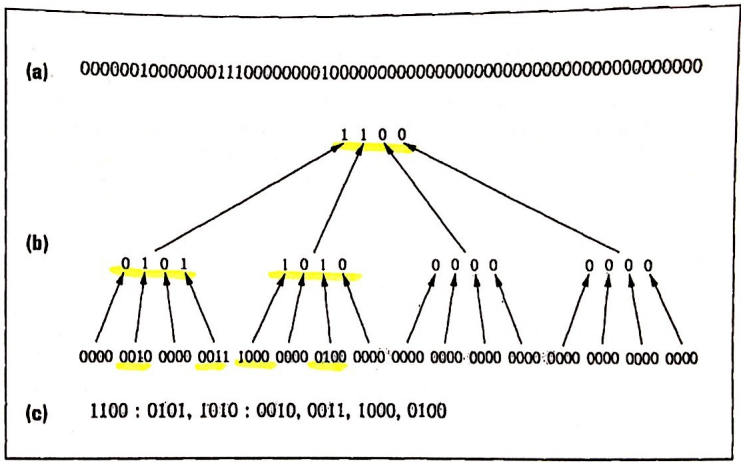


Figure 3.8 Hierarchical bitvector compression: (a) original bitvector; (b) hierarchical structure; (c) flattened tree as a string of bits.

Figura 20: Árboles de derivación binaria.

22.2 Modelo vectorial

Dados los documentos $D = \{d_1, d_2, \ldots, d_N\}$ (hay N documentos) y los términos $T = \{t_1, t_2, \ldots, t_H\}$ (hay H términos), cada documento d_j se modela como un vector

$$\overrightarrow{d_j} = [p(t_1, d_j), p(t_2, d_j), \ldots, p(t_H, d_j)]$$

donde $p(t_i, d_j)$ es el peso del término i en el documento j. Hay varias formas para representar estos pesos, una de ellas es:

$$p(t_i, d_j) = \underbrace{ft_{ij}}_{\text{peso local}} \cdot \underbrace{\log_{10}\left(\frac{N}{n_i}\right)}_{\text{peso global}}$$

donde:

- ft_{ij} = frecuencia del término i en el documento j (si aumenta ft_{ij}, aumenta $p(t_i, d_j)$)
- n_i = cantidad de documentos en los que aparece el término i (si aumenta n_i, disminuye $p(t_i, d_j)$).

Forma de implementación:

- **Listas invertidas**. Formadas por dos estructuras:
  1. Vocabulario: Estructura de búsqueda. Suele ser un árbol B.

     $$<$$ i(termino),peso global, ft, offset lista invertida $>$

     La frecuencia de cada término se almacena aquí para resolver las consultas más eficientemente (sin tener que acceder a sus listas invertidas), del término más a menos frecuente.
  2. Entradas de listas invertidas, una por cada término:

     $$< doc_1, \gamma\left(\frac{freq_{t,d_1}}{\left|\overrightarrow{doc_1}\right|}\right); \ldots; doc_{ft}, \gamma\left(\frac{freq_{t,d_{ft}}}{\left|\overrightarrow{doc_{ft}}\right|}\right) >$$

a) Si la colección de documentos es muy grande, en orden **decreciente** de frecuencia **normalizada**.

b) Si la colección de documentos es chica, en orden **creciente** de documento.

22.3 Modelo probabilístico

$$Sim\,(d,q) = \frac{P\,(d \text{ es relevante para la consulta } q)}{P\,(d \text{ no es relevante para la consulta } q)}$$

23 Consultas

Hay dos tipos:

- **Booleanas**: la consulta es una lista de términos combinados con conectivas AND (+), OR, NOT (-).
  Relevancia: binaria. "1" es relevante, "0" no lo es. Un documento es relevante para un término sí y sólo sí contiene a ese término.

Ventajas	Desventajas
Muy usado por expertos.	Pequeñas variaciones en la consulta producen resultados muy distintos.
	No discrimina entre documentos más o menos relevantes.
	No tiene en cuenta la frecuencia de aparición de las palabras.
	No considera *matchs* parciales.
	No permite ordenar los resultados.
	El usuario promedio no entiende reglas lógicas.
	Usar ANDs produce mucha precisión pero bajo recall. Usar ORs produce mucho recall pero baja precisión.

- **Rankeadas**: la consulta es una lista de términos. Se aplica una heurística para decidir la similitud entre cada documento con la consulta.

Ventajas	Desventajas
Mayor precisión que con consultas booleanas	Se requiere más información que para resolver consultas booleanas
	Requiere mayor procesamiento que para resolver consultas booleanas

23.1 Booleanas

Algoritmo 24 Consultas **conjuntivas** en el modelo booleano.

```
consulta(q):
  para cada term de q:
    obtener del arbol B# su frecuencia y la direccion en disco de su lista invertida

  identificar el term de q con la menor frecuencia
  leer la lista invertida del termino con la menor frecuencia
  C = {lista invertida del termino con la menor frecuencia}

  para cada term restante de q:
    leer su correspondiente lista invertida
    para cada "num_doc" en C:
      si ("num_doc" no esta en la lista invertida de term):
        C = C - {"num_doc"}
    si (|C| = 0):
      devolver no hay resultados

  para cada "num_doc" en C:
    devolver doc correspondiente
```

- La consulta debe ser modificada con las leyes de De Morgan para llevarla a la forma normal disyuntiva.
- Se realizan las operaciones de conjuntos correspondientes sobre las listas invertidas. Como estas están ordenadas, se puede operar recorriéndolas secuencialmente. Si una lista es corta y la otra larga, puede buscarse binariamente los elementos de la lista corta en la lista larga.
- Si las listas estuvieran codificadas con $d-gaps$, se ahorra espacio pero no se puede realizar la búsqueda binaria, y conviene almacenar, cada tanto, un valor absoluto.
- Costo de procesar la consulta q:

$$O\left(\sum_{\forall t \in q} f_t\right)$$

  Si las listas se codifican con $d-gaps$, el orden es el mismo pero la constante es mayor.
- Ventaja: para una consulta conjuntiva, la cantidad de documentos resultado siempre es menor o igual que la frecuencia del término de la consulta menos frecuente.

23.2 Rankeadas

A diferencia del modelo booleano, en este modelo, agregar un término a la consulta agranda la búsqueda en vez de achicarla.

En las consultas de este modelo, los documentos devueltos podrían no contener a todos los términos de la consulta (pero deben contener al menos uno). Se necesita devolver, por ejemplo, los 100 documentos más similares a la consulta, y luego seleccionar los más relevantes, en lo que se conoce como el **proceso de ranking**.

23.2.1 Coordinate matching

Para cada documento, se cuenta la cantidad de términos de la consulta que posee. Muestra primero los documentos que tienen más términos de la consulta.

Representa a cada documento d y consulta como un vector v de H componentes.

$$v_i = \begin{cases} 1 & \text{si } t_i \in d \\ 0 & \text{si } t_i \notin d \end{cases}$$

Algoritmo 25 Resolución de consulta con coordinate matching.

```
vector consulta = (0,0,0,...,0) // vector de longitud H
vector consulta [termino] = frecuencia termino en consulta

para cada documento en coleccion:
  vector documento = (0,0,0,...,0) // vector de longitud H
  para cada termino en documento:
    vector documento [termino] = frecuencia termino en documento

para cada documento en coleccion:
  ranking documento = vector consulta x vector documento // producto interno

ordenar los rankings en orden descendente
```

Ventajas	Desventajas
	No tiene en cuenta la frecuencia de los términos en los documentos
	Asume que todos los términos tienen igual importancia
	Favorece a los documentos más largos porque tienen más términos

23.2.2 Producto interno

Representa a cada documento d y consulta como un vector v de H componentes.

$$v_i = freq_{i,d}$$

Ventajas	Desventajas
	Asume que todos los términos tienen igual importancia
	Favorece a los documentos más largos porque tienen más términos

23.2.3 Producto interno mejorado

Reconoce que cada término tiene una importancia determinada.

$$I_t = \log_{10}\left(\frac{N}{ft}\right)$$

Representa a cada documento con un vector v de H componentes.

$$v_i = \text{frecuencia termino i en documento} \times I_t$$

Representa a cada consulta con un vector c de H componentes.

$$c_i = \text{frecuencia termino i en consulta}$$

Algoritmo 26 Resolución de consulta con producto interno mejorado.

```
para cada documento en coleccion:
  ranking documento = v x c // producto interno

ordenar los rankings en orden descendente
```

Ventajas	Desventajas
Reconoce que cada término tiene una importancia determinada.	Favorece a los documentos más largos porque tienen más términos.

23.2.4 Distancia coseno

Representa a cada documento y consulta como un vector v de H componentes.

$$\begin{array}{rcl} v_i & = & [p(t_1,d),\ldots,p(t_H,d)] \\ p(t_i,d) & \underset{\text{norma euclídea}}{=} & ft_{i,d} \times \log_{10}\frac{N}{n_i} \end{array}$$

El vector de la consulta tiene en cuenta los pesos de cada término.

La **similitud** entre dos documentos d_i y d_j se calcula mediante la **distancia coseno**[9]:

$$sim(d_i,d_j) = \frac{\overrightarrow{d_i} \times \overrightarrow{d_j}}{\left|\overrightarrow{d_i}\right| \times \left|\overrightarrow{d_j}\right|} \underset{\textbf{norma euclídea}}{=} \frac{\sum_t p(t,d_i) \times p(t,d_j)}{\sqrt{\sum_t p(t,d_i)^2} \times \sqrt{\sum_t p(t,d_j)^2}}$$

$$\underset{\textbf{norma infinito}}{=} \frac{\sum_t p(t,d_i) \times p(t,d_j)}{\textbf{max freq}_{d_i} \times \textbf{max freq}_{d_j}}$$

Si se usa la norma euclídea, esta similitud es un valor entre 0 y 1. Si los documentos son iguales, tienen similitud 1, y si no comparten términos, la similitud es 0.

Para calcular la similitud entre una consulta q y un documento d, se transforma la consulta en un vector con H componentes. Para resolver más rápidamente la consulta, se pueden realizar dos aproximaciones:

- Las consultas poseen términos distintos (con lo cual $ft_q \approx 1$ para todo t),
- $|\overrightarrow{q}|$ es constante para cualquier consulta (afecta el valor de la similitud pero no el orden del ranking)

$$sim(q,d) \approx \sum_{t \in q} \underbrace{\frac{ft_{i,d}}{\left|\overrightarrow{d}\right|}}_{\text{peso local}} \underbrace{\cdot \log_{10}\left(\frac{N}{n_i}\right)}_{\text{peso global}}$$

- $\left|\overrightarrow{d}\right|$ = norma del vector. Es el peso del documento en toda la colección.
- $\frac{ft_{ij}}{\left|\overrightarrow{d_j}\right|}$ = frecuencia normalizada del término i en el documento j. Este valor se debe almacenar en el índice para evitar tener que hacer el cálculo en cada consulta.
- $p(t_i,d_j) = ft_{i,j} \cdot \log_{10}\left(\frac{N}{n_i}\right)$ = peso del término i en el documento j.
  - $ft_{i,j}$ es la frecuencia del término i en la el documento j.

Ventajas	Desventajas
Reconoce que cada término tiene una importancia determinada.	
Al incorporar en el cálculo $\left\|\overrightarrow{d}\right\|$, no favorece a los documentos más largos.	

[9] **Teorema: similitud** $= \cos\theta = \frac{X \cdot Y}{\|X\| \cdot \|Y\|}$. Geométricamente, θ es el ángulo entre los vectores X e Y

Algoritmo 27 Cálculo de ranking en el modelo vectorial para colecciones chicas

```
// la lista invertida de un termino esta formada por <cantidad docs, (num doc, freq norm) +>
// la lista invertida esta ordenada por "num doc" creciente
para cada doc en coleccion:
  acumulador[doc]= 0
para cada term en consulta:
  recuperar lista de invertida para "term"
  para cada entrada de la lista invertida <doc, freq norm>:
    acumulador[doc] += peso global term * freq norm

ordenar los acumuladores en forma decreciente
devolverlos en ese orden
```

Algoritmo 28 Cálculo de ranking en el modelo vectorial para colecciones grandes (devuelve k resultados)

```
// la lista invertida de un termino esta formada por <long, (num doc, freq norm) +>
// la lista invertida esta ordenada por "freq norm" decreciente

para cada term en consulta:
  leer su lista invertida

ordenar las listas invertidas por "peso_global" decreciente

para cada lista invertida:
  pivote lista = primer doc lista

contador resultados = 0
mientras (contador resultados < k)
  para cada lista invertida:
    sim (pivote lista, term) = peso global term * freq norm doc

  devolver doc pivote con mayor valor de "sim"
  en la lista que tenia el pivote que acabamos de devolver:
    avanzar en 1 el pivote

  contador resultados ++
```

23.2.5 Retroalimentación en el modelo vectorial

Puede preguntársele al usuario cuáles documentos recuperados son relevantes y cuáles no, y volver a pesar todo en función de la respuesta.

Sea V el conjunto de documentos recuperados para una consulta. El usuario los clasifica en V_r (resultados relevantes) y V_{nr} (resultados no relevantes). La fórmula de Rochio modifica el vector de la consulta de la siguiente manera:

$$\vec{c}_m = \alpha \vec{c} + \frac{\beta}{|V_r|}\left(\sum_{d_i \in V_r} \vec{d_i}\right) - \frac{\gamma}{|V_{nr}|}\left(\sum_{d_i \in V_{nr}} \vec{d_i}\right)$$

Para no tener que pedirle al usuario los documentos no relevantes, puede asumirse que $\gamma = 0$.

24 Consultas especiales

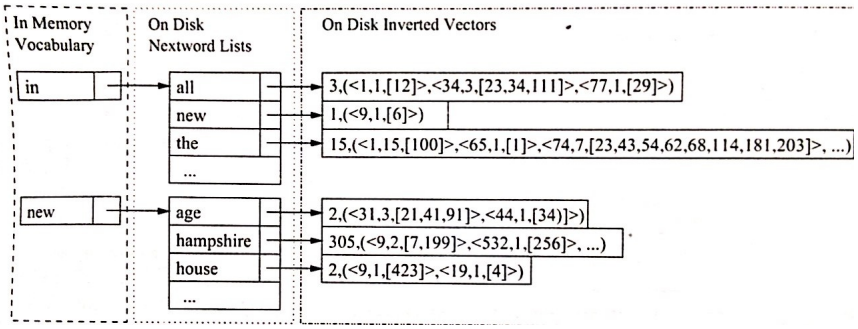
24.1 Consultas de frases

Por ejemplo: "Segunda Guerra Mundial". La distancia entre los términos debe ser exactamente la indicada en la frase. Si hay un documento que posee todos los términos, hay que, además, conocer la posición relativa de cada término en el documento.

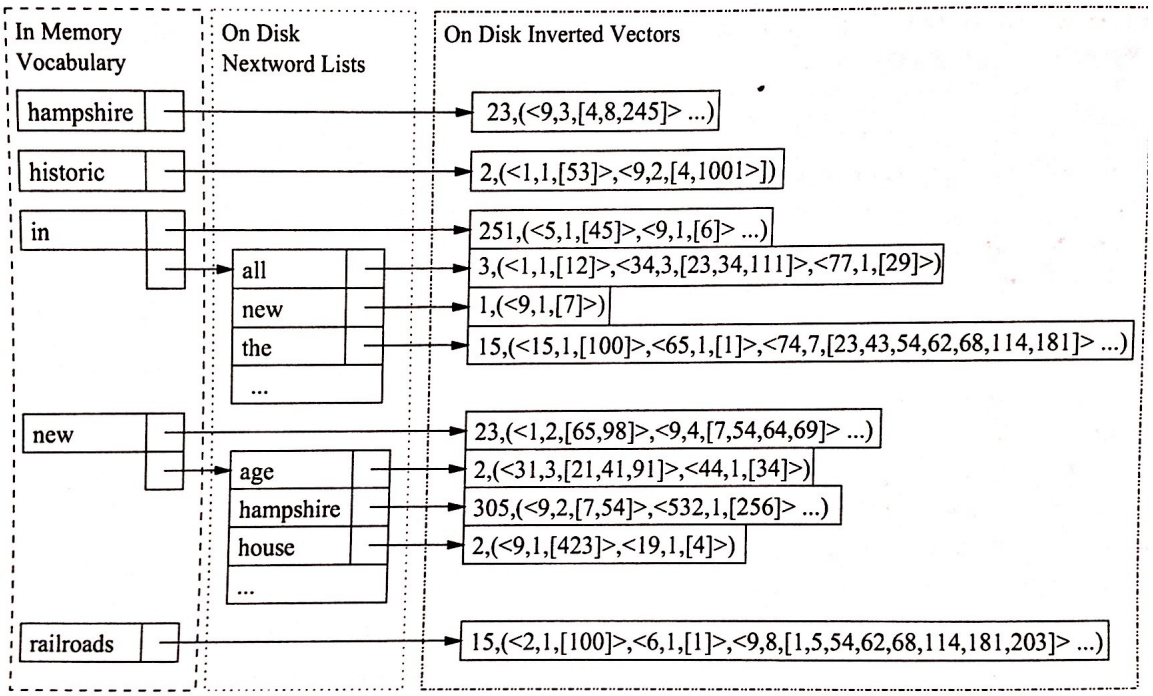
Hay dos formas de resolver estas consultas:

- **Índice invertido booleano posicional**: con tres estructuras:
  - Vocabulario:
    $$< \text{i(termino)}, \text{id termino, cantidad docs, offset lista invertida} >$$
  - Listas de documentos: para un término, se almacenan "cantidad docs" registros del siguiente formato:
    $$< id\, doc, \text{freq termino en } doc, \text{offset lista posiciones } doc >$$
  - Listas de posiciones: para un término y un documento, se almacena el siguiente registro:
    $$< p_1, p_2, \ldots, p_{\text{freq termino en doc}} >$$
- ***Biword index***: los términos se indexan de a pares. Para cada *firstword* hay una lista de *nextwords* que lo siguen, los documentos y la posición del par en ese documento.

  Si se combina un *biword index* para términos comunes con un índice invertido para términos raros, se reduce a la mitad el tiempo de resolución de una consulta, con un *space overhead* del 10 % del índice invertido.



(a) Biword index.



(b) Biword index con índice invertido.

Figura 21: Biword index.

La resolución de una consulta se resuelve descomponiendo la misma en pares de términos.

```
resolver consulta (c):
  ordenar los terminos de c de mas raros a mas comunes, guardando sus posiciones originales en c
  buscar la lista invertida del termino mas raro, y guardarla temporalmente.
  para cada termino restante de c:
    buscar su lista invertida
    hacer merge con la lista temporal
```

24.2 Consultas de términos próximos / cercanos

Por ejemplo: "algoritmos árboles ~4". Si hay un documento que posee todos los términos, hay que, además, conocer la posición relativa de cada término en el documento.

La distancia entre los términos no debe superar la distancia dada.

Para reducir el espacio ocupado por el índice, se puede tener un índice de **bloques**: se almacena el bloque donde ocurre cada término.

24.3 Consultas con *wildcards*

Ejemplos: "cas*", "am?r". El *wildcard* * significa "uno o más caracteres desconocidos". El *wildcard* ? significa "un caracter desconocido".

- Métodos de resolución:

1. **Fuerza bruta**: recorrer el léxico secuencialmente.

Ventajas	Desventajas
No se requieren estructuras adicionales.	Es muy lento si los wildcards aparecen al principio.
No se requiere que el léxico esté ordenado.	No permite búsquedas de tipo "*x".

2. **Índice secundario de léxico rotado**: para cada término del léxico se almacenan todas sus rotaciones. Por ejemplo: *frozen* se convierte en *frozen|*, *rozen|f*, *ozen|fr*, *zen|fro*, *en|froz*, *n|froze*, *|frozen* (el caracter "|" indica el comienzo del término). Para buscar *fro** se buscan los términos correspondientes a todas las rotaciones que comiencen con *|fro*.

   Cada rotación apunta a un solo elemento del índice invertido de términos, porque cada rotación pertenece a un sólo término. El término al que refiere la rotación se obtiene rotando la rotación hasta llevar el "|" a un extremo, para luego quitarlo.

Algoritmo 29 Indexación con léxico rotado.

```
para cada termino distinto:
  obtener sus rotaciones
  para cada rotacion:
    agregar <i(rotacion)> a archivo de lexico rotado (arbol B)
```

Algoritmo 30 Resolución de consultas con léxico rotado.

```
si hay un solo wildcard:
  rotar consulta y llevar el wildcard al extremo derecho
  candidatos = buscar todos los lexicos rotados que comiencen igual, y devolver el termino asociado
  hacer OR de candidatos
  buscar documentos que tienen los candidatos

si hay mas de un wildcard:
  rotar consulta y llevar un wildcard al extremo derecho
  1era opcion:
    candidatos = buscar todos los lexicos rotados que comiencen igual, hasta el primer wildcard
  2da opcion:
    separar el termino en tantas partes como wildcards haya
    candidatos = buscar los lexicos rotados de cada parte
    devolver AND de candidatos
```

Ventajas	Desventajas
Permite resolver consultas de tipo "*x", "x*", "*x*", "x*y".	El léxico rotado debe estar ordenado.
Se pueden realizar búsquedas binarias sobre el léxico rotado.	Se requiere un índice secundario que ocupa espacio (un término de n caracteres aporta $n+1$ rotaciones).

3. **Índice secundario de n-gramas**: es un índice invertido de n-gramas. Para cada término del léxico se almacenan todos sus n-gramas, para cada n-grama se guarda la lista de punteros a términos que tienen ese n-grama. Por ejemplo: *frozen* contiene los 2-gramas *fr, ro, oz, ze, en*. Para reducir los falsos positivos en los resultados, también puede guardarse el caracter de fin de término: *|f, fr, ro, oz, ze, en, n|*, aunque esta técnica no sirve para 1-gramas.

   Para buscar el patrón *fro** se buscan las listas de términos correspondientes a los digramas *|f, fr, ro* y se hace la intersección de las listas.

   Cuanto más grande es n, más tamaño ocupa el índice secundario, pero las consultas devuelven menos falsos positivos. Si n es chico, hay muchos más falsos positivos y hay un alto costo de procesamiento.

   ¿Por qué se producen falsos positivos? Ejemplo: si la consulta es **ten***, se descompone en **te** y **en**, y eso puede devolver como resultado **tense** (resultado OK) y **enter** (falso positivo, contiene los digramas **te** y **en** pero no en forma consecutiva).

Algoritmo 31 Indexación con n-gramas.

```
archivo temporal = {} // secuencial
archivo listas de terminos = {} // secuencial
archivo de n-gramas = {} // arbol B

para cada documento en coleccion:
  para cada termino en documento:
    si termino NO esta en vocabulario:
      agregarlo y obtener su id termino
      descomponer termino en sus n-gramas
      para cada n-grama:
        agregar <n-grama, id termino> a archivo temporal


ordenar archivo temporal por "n-grama␣+␣id␣termino"
para cada n-grama:
  offset = agregar lista de id_terminos a "archivo␣listas␣de␣terminos"
  agregar <n-grama, offset> a "archivo␣de␣n-gramas"
```

Algoritmo 32 Resolución de consultas con n-gramas.

```
separar consulta en sus n-gramas

para cada n-grama de consulta:
  terminos candidatos (n-grama) = buscar cada n-grama en índice y devolver sus terminos asociados

hacer AND de todos los terminos candidatos
candidatos = verificar cada termino candidato por falsos positivos contra la consulta
devolver OR de candidatos
```

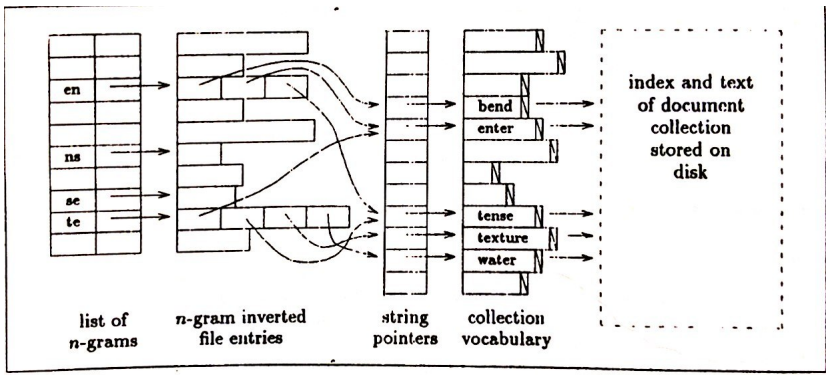


Figura 22: Índice de n-gramas en memoria.

Si el léxico está desordenado (para colecciones dinámicas), se pueden aplicar algunas optimizaciones:

a) **Thresholding**: usar un valor límite fijo. Cuando la cantidad de candidatos a una consulta cae por debajo de este valor, directamente se accede a los documentos para eliminar falsos positivos.

b) **Blocking**: ablocar el léxico. La ventaja es que disminuye el tamaño del índice secundario, por dos motivos: algunos n-gramas aparecerán varias veces en un bloque, pero solo se necesita una referencia, y los valores de los punteros serán menores y podrán ser representados en menos bits. Cuanto más grandes son los bloques, menos tamaño requiere el índice, pero se necesita más tiempo para acceder al léxico.

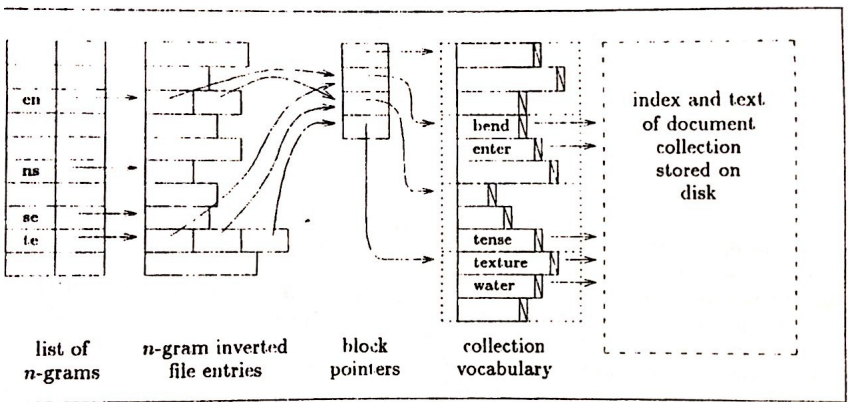


Figura 23: Índice de n-gramas en memoria.

Si el léxico está ordenado (para colecciones estáticas), se pueden aplicar algunas optimizaciones:

1) Comprimir el índice secundario mediante *run-length encoding*.
2) Ablocar el índice secundario.
3) Usar *front coding*.
4) Usar búsquedas binarias.

Ventajas	Desventajas
Ocupa menos espacio que un índice de léxico rotado.	Pueden haber falsos positivos.

24.4 Consultas de términos parecidos

Por ejemplo: "~Swarzeneger". Se utiliza, por ejemplo, cuando no se está seguro de cómo se escribe el término.

Distancia de Levenshtein: métrica para medir la diferencia entre dos cadenas de texto. La distancia entre dos *strings* a y b es la cantidad mínima de ediciones de un caracter (inserciones, supresiones y sustituciones) necesarias para cambiar de una palabra a la otra.

Para reducir el cálculo de distancias en la resolución de consultas, se utilizan **índices de espacios métricos**. Estos calculan muchas distancias al momento de construir el índice.

Un espacio métrico se define como el par (U, d). U es el universo de objetos, y d es una función de distancia tal que

- $d(x, y) \geq 0$ para todo $(x, y) \in U$,
- $d(x, x) = 0$ para todo $x \in U$,
- $d(x, y) = d(y, x)$ para todo $(x, y) \in U$,
- $d(x, y) \leq d(x, z) + d(z, y)$ para todo $(x, y, z) \in U$ (**desigualdad triangular**).

Objetivo: dada una consulta $q \in U$, se pueden querer dos cosas.

1. Los objetos (términos) de U que estén, como máximo, a una distancia r de q. Existen dos formas de resolver esta consulta.

Algoritmo 33 Algoritmo de selección de pivotes.

```
1  pivotes = {}
2  M = maxima longitud de termino en el vocabulario
3  pivotes += {primer termino del vocabulario}
4  para cada termino en vocabulario:
5    si para cada pivote se cumple (d(termino, pivote) >= coef * M ): //coef entre 0.4 y 0.6
6      pivotes += {termino}
7  devolver pivotes
```

a) Mediante **clusters** de objetos: para cada cluster, se le asocian términos y un radio.

(Cluster (pivote,radio,(termino, d(termino,pivote)+)) +)

Algoritmo 34 Indexar mediante clusters de objetos.

```
1 para cada termino:
2   para cada pivote:
3     calcular d(termino,pivote)
4
5 para cada termino:
6   cluster asignado = min(pivote : d(termino, pivote)) // politicas de desempate?
7
8 para cada cluster:
9   radio = max (d(termino,pivote))
```

Algoritmo 35 Resolver consultas mediante clusters de objetos.

```
C = {}
para cada cluster:
  si ( d(q,pivote ) > r + radio):
    continuar // Descartamos clusters cuyo radio no interseque con el radio de consulta
  si no:
    para cada term en cluster:
      si ( |d(q,pivote) - d(term,pivote)| > r):
        continuar // Descartar terminos que no esten en el radio de la consulta
      si no:
        si (d(q,term) <= r)
          C = C + {term}
devolver C
```

b) Mediante **SSS** (*Sparse Spatial Selection*): se eligen algunos términos pivotes, y para cada término y para cada pivote, se almacena la distancia del término al pivote.

SSS(Firma(d(termino,pivote1),..,d(termino,pivoteK))+)

Algoritmo 36 Indexar mediante SSS.

```
para cada termino:
  para cada pivote:
    guardar d(termino,pivote)
```

Algoritmo 37 Resolver consultas mediante SSS.

```
C = {}
para cada pivote:
  calcular d(q,pivote)
para cada termino de vocab:
  si para algun pivote se cumple ( |d(q,pivote) - d(termino, pivote)| > r):
    continue // Siguiente termino
  si no:
    C = C + {termino}

R = {}
para cada Candidato en C:
  si ( 0 < d(q,Candidato) <= r)
    R = R + {Candidato}
devolver R
```

2. Los k objetos de U que estén más cercanos a q.

   a) Mediante **árboles de aproximación espacial** $(K - NN)$.

25 Evaluación de un sistema de consultas

Para evaluar la efectividad de un sistema de consultas, se utilizan dos estadísticas acerca de los resultados devueltos ante una *query*:

Precisión: "cuántos documentos recuperados son relevantes". Mide cuán cerca estamos del objetivo.

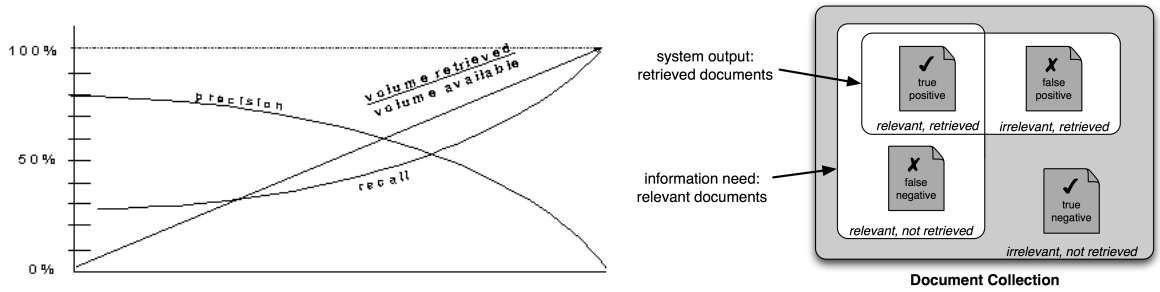
$$P = \frac{\{\text{Relevantes}\} \cap \{\text{Devueltos}\}}{\{\text{Devueltos}\}}$$

Ejemplo: si de 50 documentos devueltos 35 son relevantes a la consulta, la precisión es $\frac{35}{50} = 70\,\%$.

Recuperación (*recall*): "cuántos documentos relevantes se recuperaron". Mide el grado de exhaustividad del sistema.

$$R = \frac{\{\text{Relevantes}\} \cap \{\text{Devueltos}\}}{\{\text{Relevantes}\}}$$

Ejemplo: si había 70 documentos relevantes a la consulta, y si se devuelven 50 de los cuales solo 35 son relevantes, el recall es de $\frac{35}{70} = 50\,\%$.



(a) Cuando aumenta la precisión, baja el recall, y viceversa.

(b) Precisión y recall.

Figura 24: Evaluación de consultas.

26 Implementación

Cantidad de documentos	N
Cantidad de términos	F
Cantidad de términos distintos	n
Cantidad de punteros a documentos	f
Cantidad de espacio requerida como máximo	$f \cdot \lceil \log_2 n \rceil$ bits

Ley de Zipf: modela la distribución de frecuencias de las palabras de un léxico. Unas pocas palabras (las *stopwords*) aparecen muchas veces, y muchas palabras aparecen pocas veces.

La palabra más frecuente aparece el doble de veces que la segunda palabra más frecuente, el triple de veces que la tercera palabra más frecuente, etc.

Sean:

- N términos distintos,
- k la frecuencia,
- s el valor del exponente que caracteriza la distribución

$$f(k; s, N) = \frac{1/k^s}{\sum_{n=1}^{N}(1/n^s)}$$

Ley de Heaps: el número de términos únicos V de una colección con N términos es aproximadamente $\sqrt{N}$. De forma más general:

$$V = C \times N^{\beta}$$

donde C y $\beta \in (0, 1)$ dependen del tipo de texto. En la práctica, $C \in [5, 50]$ y $\beta \in (0{,}4; 0{,}6)$. Además, V suele ser menos del 1 % de N. Es decir: el vocabulario entra en memoria RAM (5 MB para 1 GB de texto).

26.1 Compresión del léxico

Transformar las palabras antes de indexarlas.

1. Usar una lista de ***stopwords***, para no indexar palabras que se repiten mucho.

   Ejemplos: "el", "la", "que", "ellos", "nosotros", y frases como "sin embargo", "a pesar de", etc.

Ventajas	Desventajas
El índice ocupa menos espacio.	Muchas consultas incluyen *stopwords*, y solo podrían resolverse analizando todos los documentos.
	No tiene sentido eliminar *stopwords* si el índice invertido se almacena comprimido, porque los punteros de las palabras más frecuentes pueden almacenarse con muy pocos bits.
	No tiene sentido eliminar *stopwords* si se usan índices de firmas o bitmaps.

   Las *stopwords* se deben guardar en un archivo, y cargarlas en memoria al momento de indexar.

2. Usar *case folding* y eliminar signos de puntuación, acentos, etc.

3. Usar ***stemming***: guardar solo el tallo de las palabras.

Ventajas	Desventajas
Aumenta el recall.	Baja la precisión.

4. Reducción de plural a singular: "*person#*" representa "persona" y "personas".

5. Reducción de género y número de adjetivos: "*buen?*" representa "buen", "bueno", "buena", "buenos", "buenas".

6. Sustituir palabras por sus sinónimos. (Luego hay que modificar la consulta también).

Métodos para almacenar el léxico ordenado:

1. **Términos de longitud fija**: se destina un espacio fijo para cada término, equivalente a la longitud del término más largo x.

   (término: x bytes, puntero a lista invertida: 4 bytes)

Ventajas	Desventajas
Puede hacerse búsqueda binaria	Ocupa mucho espacio

2. **Concatenación de términos**: todos los términos se almacenan de forma contigua, y se utilizan punteros para acceder a ellos.

   (puntero a archivo de términos: 4 bytes, puntero a lista invertida: 4 bytes)

3. **Front coding**: aprovecha el hecho de que el vocabulario está ordenado alfabéticamente y que los comienzos se repiten.

   En el léxico (hojas del árbol B#) se almacenan registros con la siguiente estructura:

   (# chars iguales al anterior: 1 byte, # chars distintos al anterior : 1 byte, chars distintos, puntero a lista invertida: 4 byt

   El primer registro de cada hoja debe almacenar el término completo:

   (longitud del término: 1 byte, término: longitud, puntero a lista invertida : 4 bytes)

   Ejemplo:

codazo, codearse, codera, codicia, codiciar, codiciosa, codicioso, codificar, codigo
6|codazo#|3|5|earse#|4|2|ra#|3|4|icia#|7|1|r#|6|3|osa#|8|1|o#|4|5|ficar#|4|2|go#

Ventajas	Desventajas
El léxico ocupa menos espacio.	La búsqueda de un término implica descomprimir secuencialmente la hoja.

4. **Front coding parcial o por secciones**:

   En el léxico (hojas del árbol B#) se almacenan registros con la siguiente estructura:

   (# chars iguales al anterior, # chars distintos al anterior, chars distintos, puntero a lista invertida)+)

   El primer registro de cada hoja, y cada n registros ($n \approx \sqrt{\#\text{ registros en hoja}}$) debe almacenar el término completo:

   (longitud del término,término,puntero a lista invertida)

   Ejemplo:

codazo, codearse, codera, codicia, codiciar, codiciosa, codicioso, codificar, codigo
6|codazo#|3|5|earse#|4|2|ra#|<p1>|7|codicia#|7|1|r#|6|3|osa#|<p2>|9|codicioso#|4|5|ficar#|4|2|go#|<p3>|9|3

   Las búsquedas comienzan desde la derecha. Hay 3 campos de control:

   a) 3: cantidad de secciones en la hoja

   b) 9: cantidad de términos en la hoja

   c) <p3>: posición relativa de <p2>

Ventajas	Desventajas
Se puede hacer una búsqueda $n-$aria del léxico.	Desperdicia más espacio que el front coding.

5. **Hashing perfecto y mínimo**: no se guardan los términos.

   a) Se utiliza una función de hashing que es:

   1) Perfecta: no produce colisiones.
   2) Mínima: genera todos los valores posibles del espacio de direcciones.
   3) Preservadora de orden: sean los strings s_1 y s_2. Entonces $s_1 < s_2 \iff f(s_1) < f(s_2)$.

26.2 Compresión de listas invertidas

1. $d-gaps$: la lista invertida para el término t de la forma $\langle f_t; d_1, d_2, \ldots, d_{f_t}\rangle$ se almacena en forma ascendente mediante diferencias sucesivas:

$$\langle f_t; d_1, d_2 - d_1, d_3 - d_2, \ldots, d_{f_t} - d_{f_t - 1}\rangle$$

   Como son más comunes las $d-gaps$ pequeñas, se puede utilizar codificación de longitud variable.

2. Codificación alineada a bytes: representa números naturales con la mínima cantidad de bytes, usando los 2 bits más significativos del código para indicar la cantidad de bytes empleados.

Rango de números representable	Codificación en bytes
$0 \ldots 2^6 - 1$	00xxxxxx
$2^6 \ldots 2^{14} - 1$	01xxxxxx xxxxxxxx
$2^{14} \ldots 2^{22} - 1$	10xxxxxx xxxxxxxx xxxxxxxx

Métodos que describen la distribución de probabilidad de las $d-gaps$:

1. **Globales**: un mismo modelo para comprimir todas las listas invertidas.

   a) **Parametrizados**

   1) **Bernoulli**: utiliza como parámetro la densidad de punteros del archivo invertido.

      "La probabilidad de cada distancia $d-gap$ es la distribución geométrica $pr[x] = (1-p)^{x-1}p$, donde $p = \frac{f}{N \cdot n}$ es la probabilidad de que un documento al azar posea un término al azar".

   2) **Frecuencia observada**: utiliza como parámetro la densidad exacta de punteros del archivo invertido.

   b) **No parametrizados**

   1) **Binario**:

      "La probabilidad de cada distancia $d-gap$ es uniforme ($pr[x] = \frac{1}{N}$)".

      Un entero x se codifica como x en binario de $\lceil \log_2 N \rceil$ bits (N es la cantidad de documentos).

   2) **Unario**:

      "La probabilidad de cada distancia $d-gap$ es $pr[x] = 2^{-x}$".

      Un entero $x \geq 1$ se codifica como $x-1$ bits en cero (uno) seguido de un bit en uno (cero).
      Ejemplo:

      $$(x = 5) \text{ en unario} = \underbrace{0000}_{4 \text{ bits en cero}}1$$

   3) **Gamma** (γ):

      "La probabilidad de cada distancia $d-gap$ es $pr[x] \approx \frac{1}{2x^2}$".

      Un entero x se codifica como el número $(1 + \lfloor \log_2 x \rfloor)$ en **unario**, concatenado con el número $(x - 2^{\lfloor \log_2 x \rfloor})$ en binario, en $\lfloor \log_2 x \rfloor$ bits. (La parte en unario especifica cuántos bits se necesitan para codificar a x, y la parte en binario codifican x en esa cantidad de bits).
      Receta:

      a' Escribir el número en binario.

      b' Restarle 1 a la cantidad de bits escritas en el paso anterior, y agregarle esa cantidad de ceros al principio.

      Ejemplo:

      $$(x = 5) \text{ en gamma} = 00 \underbrace{101}_{5 \text{ en binario}}$$

      Decodificación: extraer el código unario c_u, y tratar los próximos $c_u - 1$ bits como un número binario, para obtener el segundo valor c_b. El valor de x se calcula como

      $$x = 2^{c_u - 1} + c_b$$

      Ejemplo anterior: $2^{3-1} + 1 = 4 + 1 = 5$

   4) **Delta** (δ):

      "La probabilidad de cada distancia $d-gap$ es $pr[x] \approx \frac{1}{2x(\log x)^2}$".

      Un entero x se codifica como el número $(1 + \lfloor \log_2 x \rfloor)$ en **gamma**, concatenado con el número $(x - 2^{\lfloor \log_2 x \rfloor})$ en binario, en $\lfloor \log_2 x \rfloor$ bits. (La parte en gamma especifica cuántos bits se necesitan para codificar a x, y la parte en binario codifican x en esa cantidad de bits).
      La codificación en δ ocupa más espacio que la codificación γ cuando $x < 15$. Cuando $x \geq 15$, δ ocupa menos espacio.
      El 0 no se puede representar.

Ejemplo:

$$\begin{aligned}(x=5) \text{ en delta} &= (1+2) \text{ en gamma, } (5-2^2) \text{ en binario de 2 bits} \\ &= 011,01\end{aligned}$$

Decodificación: extraer el código gamma c_g, y tratar los próximos $c_g - 1$ bits como un número binario, para obtener el segundo valor c_b. El valor de x se calcula como

$$x = 2^{c_g - 1} + c_b$$

2. **Locales**: las listas invertidas de cada término se comprimen de acuerdo a un parámetro (generalmente, la frecuencia del término). Comprimen mejor que los métodos globales, pero son más difíciles de implementar.

   a) **Bernoulli**: implementado con **códigos de Golomb**, que aproxima una codificación con Huffman: usa un parámetro ajustable g para codificar un entero $x > 0$ en dos partes: el cociente q y el residuo r.

   - $q+1$ en unario, donde $q = \left\lfloor \frac{x}{g} \right\rfloor$.
   - $r = x - qg$ en binario de $\lceil \log_2 g \rceil$ bits.
     Puede demostrarse que si g satisface

     $$(1-p)^g + (1-p)^{g+1} \leq 1 < (1-p)^{g-1} + (1-p)^g$$

     el código es óptimo. (Es decir, g se relaciona con la probabilidad p de que un término se encuentre en un documento). Entonces el parámetro g óptimo es, para cada término:

$$\begin{aligned} g_{optimo_{term}} &= round\left(\frac{\log_2(2-p_{term})}{-\log_2(1-p_{term})}\right) \\ p_{term} &= \frac{\#\text{ docs que tienen el term}}{\#\text{ docs}} \end{aligned}$$

$$\begin{aligned} b &= \lceil \log_2 g \rceil \\ y &= 2^b - g \\ \text{Si } r < y &\implies r \text{ en } b-1 \text{ bits} \\ \text{Si } r \geq y &\implies y + r \text{ en } b \text{ bits} \end{aligned}$$

Nota: si $p > 0{,}5$, el código de Golomb es más efectivo si el archivo invertido se complementa antes de comprimir.
El parámetro g se debe almacenar al comienzo de cada lista invertida, para poder descomprimirla.
Problema: si a un término le agrego un nuevo documento, hay que volver a codificar la lista afectada (porque cambió p_{term} y por ende g).
Decodificación: Se tiene la siguiente cadena

$$110|100000010|10|00001101$$

y si sabe que contiene 2 números con $g = 385$.

$$\begin{aligned} b &= \lceil \log_2 g \rceil = 9 \\ y &= 2^b - g = 127 \end{aligned}$$

Primer número x_1:

- Prefijo: $q + 1 = 110 = 3 \implies q = 2$

- Sufijo: $b-1$ bits siguientes al prefijo $\implies 10000001 = 129$. Como es mayor/igual a y, tomamos un bit más $\implies 100000010 = 258$.
  $r = sufijo - y = 131$

$$x_1 = g \times q + r = 901$$

Segundo número x_2:

- Prefijo: $q + 1 = 10 = 2 \implies q = 1$
- Sufijo: $b-1$ bits siguientes al prefijo $\implies 00001101 = 13$. Como es menor a y, ese es el sufijo.
  $r = sufijo = 13$

$$x_2 = g \times q + r = 398$$

Table 3.6 Example codes for integers.

Gap x	Coding Method				
	Unary	γ	δ	Golomb	
				$b=3$	$b=6$
1	0	0	0	0 0	0 00
2	10	10 0	100 0	0 10	0 01
3	110	10 1	100 1	0 11	0 100
4	1110	110 00	101 00	10 0	0 101
5	11110	110 01	101 01	10 10	0 110
6	111110	110 10	101 10	10 11	0 111
7	1111110	110 11	101 11	110 0	10 00
8	11111110	1110 000	11000 000	110 10	10 01
9	111111110	1110 001	11000 001	110 11	10 100
10	1111111110	1110 010	11000 010	1110 0	10 101

Figura 25: Unario, delta, gamma y Golomb

26.3 Colecciones dinámicas

La organización de las listas invertidas en el archivo secuencial depende de la política de actualización del índice:

- Actualizaciones por reconstrucción total (para actualizaciones poco frecuentes):
  - Las listas invertidas son **registros de longitud variable**.
  - El offset en el archivo del léxico es el offset del comienzo del registro de longitud variable.
- Actualizaciones incrementales (para actualizaciones muy frecuentes):
  - Las listas invertidas son **bloques de tamaño fijo**, con un campo extra que dice el término de la lista.

    $Bloque\,(cantListas : 1, (offsetLista)\,cantListas, Lista\,(idT : 2, cantDocs : 2, (doc : 2)\,cantDocs)\,cantListas)$

    - Si el término leído no estaba en el índice:
      - El archivo secuencial de términos se actualiza normalmente.
      - Se inserta la lista invertida en un bloque. Para esto, podría ser necesario relocalizar listas ya existentes en un bloque, para hacer lugar a la nueva.
    - Si el término leído estaba en el índice:
      - Se actualiza la lista invertida del término en el bloque.
  - El offset en el archivo del léxico es el offset del comienzo del bloque.
  - Los bloques pueden enlazarse, para permitir almacenar listas invertidas muy grandes.

27 Google y las consultas por Internet

Un documento es una página web.
Problemas de las búsquedas en Internet:

1. Internet es una colección heterogénea no controlable, de enormes proporciones. Hay que eliminar enlaces muertos, páginas duplicadas, o páginas que tengan spam.
2. Hay que indexar contenido no indexable (imágenes, etc.).
3. El proceso de *crawling* puede aumentar el tráfico de los servidores de las páginas web
4. No es posible indexar todas las páginas web existentes, por dos razones:
   a) El espacio de almacenamiento es limitado,
   b) En cierto punto el *crawler* deberá visitar páginas ya indexadas para buscar cambios.
5. Hay muchos errores en los documentos y en las consultas
6. Usar la información presente en hipertexto para producir mejores resultados
7. Se debe automatizar la indexación, porque los índices armados y mantenidos por personas son subjetivos, caros para armar, lentos para mejorar, y no cubren todos los temas
8. No se puede medir el *recall*
9. Usuarios inexpertos

Se necesitan tres cosas:

1. Algoritmos de *crawling* rápidos,
2. Mecanismos que almacenen eficientemente los índices y, opcionalmente, los documentos mismos
3. Algoritmos de resolución de consulta rápidos y que devuelvan resultados precisos.

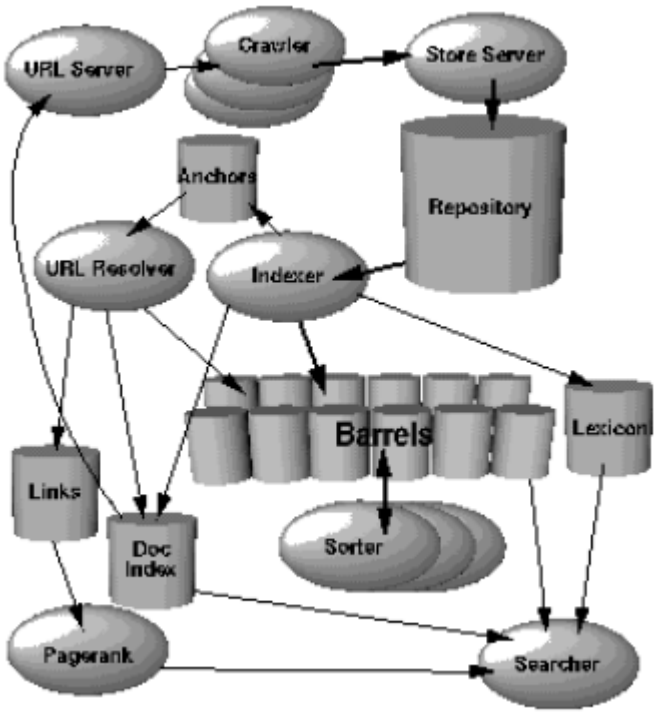


Figura 26: Arquitectura de Google.

27.1 *Crawling*

Crawling: recorrer páginas web recursivamente para agregarlas al índice, obteniendo las páginas más importantes primero, y visitando páginas frecuentemente para estar al tanto de los cambios.

Algoritmo 38 Algoritmo de crawling

```
cola de prioridad = {} // la cabeza de esta cola tiene mas prioridad que el resto

indexar(seed url):
  si (url se puede indexadar) // verifica robots.txt
    si (url no fue indexada) // verifica duplicados usando checksums
      guardar url
      para cada url dentro de seed url
        agregar url a la cola de prioridad

  remover url de cola prioridad e indexarla
```

El proceso de *crawling* se hace con cientos de máquinas distribuidas. Una función de hashing determina qué máquina debe indexar qué URL.

27.2 *Storing*

Google guarda los documentos que indexa en forma comprimida.

Google usa datos de proximidad para las búsquedas.

Google guarda datos de presentación visual, como las **negritas** y el subrayado de palabras para modificar el peso de las palabras.

27.3 *Searching*

La web, como conjunto de páginas relacionadas entre sí mediante links, puede ser representada mediante un grafo dirigido, en el que cada nodo es una página web, y cada arista es un link.

Para rankear una página se utilizan dos cosas:

1. Estructura de links,
2. *Anchor text* (lo que aparece en los links).
   a) Proveen una descripción muy buena de las páginas a las que apuntan
   b) Proveen una descripción de documentos no indexables (imágenes, programas, etc.)

Dada una página web x, puede definirse su importancia, $I(x)$, en alguna de las siguientes formas:

1. Similitud con una consulta q.
2. Contador de citaciones (*backlink count*), es la cantidad de links en la Web hacia x. Este método asume que todos los links son iguales.
3. *Forward link count*, es la cantidad de links que emanan de x. Bajo esta métrica, los sitios webs como ser los directorios son muy importantes.
4. Localización. $I(x)$ es una función de la localización de x, no de sus contenidos. Por ejemplo, páginas que terminan en ".com" son muy importantes, o páginas que contengan el string "home".

5. PageRank, es la suma ponderada de los *backlinks*.

$$PR_i(x) = (1-\alpha) + \alpha \cdot \sum_{i=1}^{n} \left(\frac{PR(T_i)}{C(T_i)} \right)$$

donde:

- $PR_i(x)$ es el PageRank de la página web x para la iteración i ($PR_0(x) = 1$ para todo x).

$$\sum_{\forall x} PR(x) = 1$$

El algoritmo es iterativo hasta que los valores converjan.

- $\alpha \in (0,1)$ es el factor de amortiguación, es constante. En general, $\alpha = 0{,}85$.
- T_i es una página que posee un link a x, una "citación" de x.
- $C(x)$ es la cantidad de links hacia afuera de una página x.

PageRank es una medida objetiva de la "importancia" de una página web, contando las citaciones de la misma.

PageRank puede pensarse como un modelo del comportamiento de un usuario. Asumir que hay un usuario que recibe una página web al azar, y hace clics continuamente, hasta que se aburre y comienza en otra página aleatoria. La probabilidad de que el usuario visite la página x es $PR(x)$. La probabilidad de que el usuario se aburra y comience el proceso de nuevo es α.

Algoritmo 39 Resolución de consultas.

```
Parse the query.
Convert words into wordIDs.
Seek to the start of the doclist in the short barrel for every word.
Scan through the doclists until there is a document that matches all the search terms.
Compute the rank of that document for the query.
If we are in the short barrels and at the end of any doclist, seek to the start of the doclist  in the full ba
If we are not at the end of any doclist go to step 4.
Sort the documents that have matched by rank and return the top k.
```

28 Algoritmo de construcción de índice

```
construir_indice (coleccion, bool consultas_exactas, bool con_posiciones, bool con_ngramas, bool con_lexrot):
  // Archivos
  archivo de registros "tabla_id_docs.dat" // <documento> o <ruta al documento>
  arbol b-sharp "lexico.dat" // <i(termino), id termino, offset lista invertida>
  archivo secuencial "terminos_por_aparicion.dat" // <termino>

  if (con_posiciones):
    archivo temporal secuencial "posiciones_terms_en_docs.dat" // <id term, id doc, pos>
    archivo secuencial "lista_posiciones.dat" // <m, pos1, pos2, ..., pos m>
    if (consultas_exactas):
      archivo secuencial "listas_invertidas.dat" // <cantidad docs, <id doc, offset lista pos>+ >
    else:
      archivo secuencial "listas_invertidas.dat" // <cantidad docs, <id doc, freq norm, offset lista pos>+ >

  if (con_posiciones and not consultas_exactas):
    archivo temporal "frecuencias_terminos_en_docs.dat" // <id term, id doc, freq, offset lista posiciones>
    archivo temporal "frecuencias_normalizadas_terminos_en_docs.dat" // <id term, id doc, freq norm, offset li
    archivo temporal "normas.dat" // <norma>

  if (con_ngramas):
    archivo temporal secuencial "lista_n_gramas.dat" // <n-grama, id term>
    archivo b-sharp "n-gramas.dat" // <i(n grama), freq, offset lista terminos por ngrama"
    archivo secuencial "lista_terminos_por_ngrama.dat" // <id term 1, ..., id term freq>
```

```
24
25  if (con_lexrot):
26    archivo b-sharp "lexico_rotado.dat" // <i(rotacion), id term>
27
28  // Algoritmo
29  para cada doc en coleccion:
30    "id_doc" = agregar doc a "tabla_id_docs.dat"
31    para cada term en doc:
32
33      pos = posicion de term en doc
34      if (term esta en "lexico.dat"):
35        obtener "id_term" de "lexico.dat"
36      else:
37        agregar term al final de "terminos_por_aparicion.txt"
38        "id_term" = posicion de term en "terminos_por_aparicion.txt"
39        agregar "i(term), id_term, NULL" a "lexico.dat"
40        if (con_lexrot):
41          crear rotaciones de term
42          para cada rotacion:
43            agregar "i(rotacion), id_term" a "lexico_rotado.dat"
44
45        if (con_ngramas):
46          descomponer term en n-gramas
47          para cada n-grama:
48            agregar "n-grama, id_term" a "lista_n_gramas.dat"
49
50      if (con_posiciones):
51        agregar "id_doc, id_term, pos" a "posiciones_terms_en_docs.dat"
52
53  ordenar "posiciones_terms_en_docs.dat" por "id_doc + id_term + pos"
54
55  para cada id doc en "posiciones_terms_en_docs.dat":
56    max freq term = 1
57    para cada id term en "posiciones_terms_en_docs.dat":
58      offset = agregar "(freq_term, (pos)+)" a "lista_posiciones.dat"
59      agregar "(id_doc, id_term, freq_term, offset)" a "frecuencias_terminos_en_docs.dat"
60
61      max freq term = (max freq term > freq term) ? max freq term, freq term
62
63    agregar "max_freq_term" a "normas.dat"
64
65  ordenar "frecuencias_terminos_en_docs.dat" por "id_doc + id_term"
66
67  para cada id doc en "frecuencias_terminos_en_docs.dat"
68    norma = leer_registro "id_doc" de "normas.dat"
69    para cada id term en "frecuencias_terminos_en_docs.dat"
70      freq_norm = freq term / norma
71      agregar "(id_term, id_doc, freq_norm, offset)" a "frecuencias_normalizadas_terminos_en_docs.dat"
72
73  if (consultas_exactas):
74    ordenar "frecuencias_normalizadas_terminos_en_docs.dat" por "id_term + id_doc"
75  else:
76    ordenar "frecuencias_normalizadas_terminos_en_docs.dat" por "id_term + (-freq_norm)"
77
78  para cada term en "terminos_por_aparicion.dat":
79    para cada id term en "frecuencias_normalizadas_terminos_en_docs.dat": // <id term, id doc, freq norm, offs
80      if (consultas_exactas):
81        posLD = agregar "(freq, (id_doc, offset)+)" a "listas_invertidas.dat"
82      else:
83        posLD = agregar "(freq, (id_doc, freq_norm, offset)+)" a "listas_invertidas.dat"
84      finally:
85        actualizar "lexico.dat" con "posLD" para term
86
87  if (con_ngramas):
88    ordenar "lista_n_gramas.dat" por "n_grama + id_term"
89    para cada n grama de "lista_n_gramas.dat":
90      offset = agregar todos los "id_term" a "lista_terminos_por_ngrama.dat"
91      agregar "n_grama, cantidad_id_terms, offset" a "n-gramas.dat"
```

Parte IV
Compresión

29 Introducción a la compresión

Teoría de la información: estudio de los procesos que se realizan sobre la información. Provee una medida de la información.

En un mensaje, hay más información cuando la probabilidad de un mensaje es baja. Por ejemplo, si leemos el *string* "Argentin", la información que recibimos es la misma que si leemos "Argentina".

Compresión: representación de información utilizando menos bits. Hay dos tipos de algoritmos compresores:

- *Con pérdida / lossy / compactadores*: se pierde información, se gana tasa de compresión. Se utiliza para comprimir imágenes, audio o video, en donde la pérdida de calidad no es notoria.
- *Sin pérdida / lossless*: estos compresores permiten, a partir del archivo comprimido, recrear el archivo original tal cual era.
  - *Compresores estadísticos*: se basan exclusivamente en la probabilidad de aparición de los símbolos.
  - *Compresores no estadísticos*
    - RLE (Run Length Encoding): codifican secuencias repetidas.
    - Predictores
    - Por sustitución

	Compresión estática	**Compresión dinámica**
Cantidad de pasadas al archivo	Una pasada para obtener estadísticas del archivo, y otra pasada para comprimir.	Una pasada para obtener estadísticas y para comprimir.
Velocidad de compresión	Lento.	Rápido.
Nivel de compresion	Muy bueno.	Bueno.

Cuadro 5: Compresión estática y dinámica.

Entropía: grado de desorden de una fuente.

Sea una fuente F, la entropía de F se define como

$$H(F) = \sum_{i=0}^{n} P_i \cdot (-\log_2(P_i))$$

Códigos prefijos: código con la "propiedad de prefijo": ninguna palabra de código es prefijo de cualquier otra palabra de código del conjunto. Ejemplos:

1. A=0, B=10, C=11 tiene la propiedad de prefijo.
2. A=0, B=1, C=10, D=11 no tiene la propiedad de prefijo, porque "1" es prefijo de "10" y de "11".

30 Marca de fin de archivo

1. Comprimir el archivo. Agregar un "1" al final, y completar con ceros hasta el final del byte. El descompresor empezará por el final, leyendo los bits en cero y eliminándolos. Cuando llega al "1" lo elimina, se detiene, y comienza a descomprimir por el principio.
2. Indicar la longitud del archivo comprimido al inicio. Este método no sirve para los compresores dinámicos, puesto que la longitud del archivo no se conoce hasta procesarlo todo.
3. Utilizar un caracter extra como EOF.

31 Compresión por Huffman

Es un compresor estadístico. Se construye un árbol de símbolos, y a cada símbolo se le asigna un código binario que se deduce recorriendo el árbol.

La codificación Huffman es una técnica óptima para construir códigos prefijo de longitud variable.

31.1 Huffman estático

Algoritmo 40 Armado de árbol Huffman.

```
1 Crear un nodo hoja para cada símbolo. Agregar cada nodo a la cola de prioridad.
2 Mientras haya más de 1 nodo en la cola:
3 Quitar los 2 nodos de menor frecuencia de la cola.
4 Crear un nuevo nodo con los 2 nodos anteriores como hijos. Su frecuencia es igual a la suma de las frecuencias
5 Agregar el nuevo nodo a la cola de prioridad.
6 El nodo restante es la raíz; el árbol está completo.
```

Algoritmo 41 Huffman estático: compresor

```
1 Primera pasada al archivo para armar una tabla de frecuencias que contiene como clave el símbolo, y como valor
2 Generación del árbol.
3 Emitir la tabla de frecuencias al archivo comprimido.
4 Segunda pasada al archivo: mientras queden caracteres sin leer del archivo:
5 Leer el caracter siguiente,
6 Emitir su código, que se obtiene del árbol.
```

Algoritmo 42 Huffman estático: descompresor.

```
1 Leer la tabla de frecuencias.
2 Generar el árbol a partir de la tabla.
3 Mientras queden códigos sin leer del archivo comprimido:
4 Leer el código siguiente
5 Buscarlo en el árbol
6 Emitir su símbolo correspondiente.
```

31.2 Huffman dinámico

Algoritmo 43 Huffman dinámico: compresor.

```
Armar una tabla de frecuencias que incluye todos los caracteres posibles, todos con frecuencia igual a 1.
Generar el árbol.
Mientras queden caracteres sin procesar:
Leer el caracter siguiente.
Comprimirlo.
Actualizar la tabla de frecuencias.
Rearmar el árbol.
```

Parte V
Criptografía

32 Introducción

Esteganografía: esconder información dentro de otra información.

Criptografía: ciencia que estudia la transformación de un mensaje en un código, de forma tal que a partir de ese código solo algunas personas puedan recuperar el mensaje original. Para ello se utiliza una palabra clave. El código resultante solo puede ser descifrado por las personas que conozcan la palabra clave.

Usos de la Criptografía:

1. Encriptación de información crítica para
   a) almacenarla en computadoras
   b) enviarla por una red insegura
2. Certificación de identidad

Objetivos de la Criptografía:

- **Confidencialidad**: poder decidir quién puede acceder a cierta información.
- **Integridad**: poder asegurar que un mensaje no fue corrompido en la transmisión (por error o con intención), o poder darse cuenta cuando lo fue.
- **Autenticación**: poder asegurar que algo es lo que dice ser.

  Sistemas de autenticación:
  - Basados en algo conocido (ejemplo: una contraseña).
  - Basados en algo poseído (ejemplo: una tarjeta de identidad).
  - Basados en una característica del usuario (ejemplo: huellas dactilares).
- **No repudio**: evitar que el emisor o el receptor de un mensaje nieguen, respectivamente, la emisión o recepción del mismo.

$$\text{No repudio} \implies \text{Autenticación}$$

Criptosistema: formado por cinco elementos

1. M es el conjunto de todos los posibles mensajes o textos planos (*plaintext*) que podemos querer encriptar.
   - Cuanta más entropía tenga el mensaje ($H(M) \approx 1$), más difícil será romper el criptosistema.
2. C es el conjunto de todos los posibles códigos que podemos obtener al encriptar (*cyphertext*).
3. K es el conjunto de todas las claves que podemos usar. El tamaño de este conjunto es el *keyspace* o $|K|$.
   - Pueden existir ciertas **claves débiles** tales que $E_k\left(E_k(m)\right) = m$
   - Pueden existir ciertas **claves semi débiles** tales que $E_{k_1}\left(E_{k_2}(m)\right) = m$

   $|K| = |M|$ implica seguridad perfecta, y el reuso de claves destruye esta propiedad (porque son vulnerables a ataques por replay).
4. E es la función de encriptación. Existe una transformación diferente E_k para cada valor posible de la clave k.

5. D es la función de desencriptación.

Todo criptosistema debe cumplir:

$$D_k\left(E_k(m)\right) = m$$

Principio de Kerckhoff: un criptosistema debería ser seguro incluso cuando se conoce todo el algoritmo, excepto la clave. No se debe confiar en "seguridad basada en obscuridad".

Criptoanálisis: romper un criptosistema, obtener el mensaje a partir del código cifrado.

33 Algoritmos de encriptación

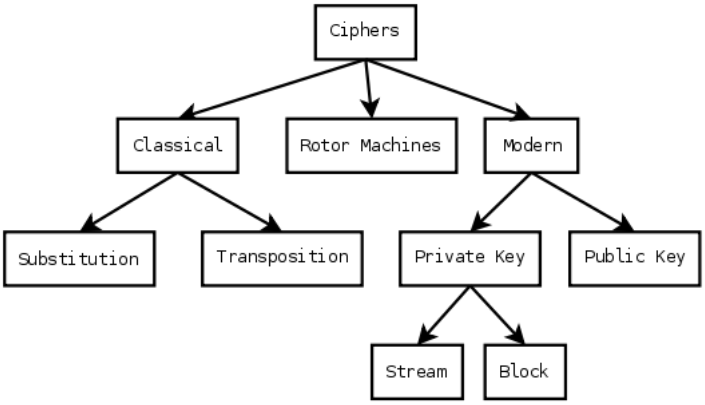


Figura 27: Taxonomía de algoritmos.

Se los divide en dos grupos:

- Algoritmos de clave privada o **simétricos**. Se utiliza una sola clave para encriptar y desencriptar.
  - Encriptación: mensaje + clave = código
  - Desencriptación: código + clave = mensaje

  Desventajas:
  - ¿Cómo transmitir la clave de forma segura?
  - ¿Qué pasa si se roban la clave?
  - Para establecer una comunicación segura entre n personas requiere una clave para cada par de usuarios: en total, se necesitan $\frac{n(n-1)}{2}$ claves.

  Son algoritmos rápidos.

  Se basan en la confusión y la difusión de la redundancia del mensaje.
- Algoritmos de clave pública o **antisimétricos**. Se utilizan dos claves para encriptar, una es privada y la otra es pública. Cada persona tiene un par de estas claves.

  El conocimiento de la clave pública no debería permitir calcular la clave privada.

  Estos algoritmos suelen basarse en problemas matemáticos cuyo tiempo de resolución sea muy alto.

  Los métodos criptográficos de este tipo garantizan que un par de claves no se puede generar más de una vez.

Los métodos de encriptación y desencriptación son de público conocimiento. La fortaleza del criptosistema debe residir solamente en la clave.

Resuelven el problema de la comunicación entre desconocidos sin tener que ponerse de acuerdo en una clave.

- Encriptación con clave privada y desencriptación con clave pública: provee autenticación (una persona puede encriptar, pero cualquiera puede desencriptar).
- Encriptación con clave pública y desencriptación con clave privada: provee confidencialidad y no repudio del receptor (cualquiera puede encriptar, pero solo una puede desencriptar).

Desventajas:

- ¿Cómo obtener la clave pública de forma segura?
- ¿Qué pasa si convenientemente "perdemos" nuestra clave privada? Podríamos repudiar una firma digital... Solución: diseñar el algoritmo de tal forma que sea imposible para los usuarios acceder a sus claves privadas.
- La encriptación y desencriptación son lentas (1000 más lentas que un algoritmo simétrico).
- Son vulnerables a ataques por texto plano escogido. Si hay n textos planos posibles, y se conoce $C = E_{k_{pub}}(M)$, para obtener M solo hay que encriptar los n textos planos posibles y compararlos con C (recordar que la clave pública es conocida). Cuando son iguales, se descubrió M.

- Algoritmos **híbridos**.

  En la vida real se suelen utilizar algoritmos simétricos, enviando la clave utilizando mecanismos asimétricos (porque la clave es corta). La clave de sesión es la que se utiliza como clave del algoritmo simétrico, y una vez que se termina la comunicación, la misma se desecha.

  1. Bob le envía a Alice su clave pública, $k_{pub_{Bob}}$.
  2. Alice genera una clave de sesión **aleatoria** K, la encripta con $k_{pub_{Bob}}$, y se la envía a Bob.
  3. Bob desencripta el mensaje $E_{k_{pub_{Bob}}}(K)$ con su clave privada, recuperando la clave K.
  4. Alice y Bob se comunican con K.

Estructura de grupo: propiedad de un algoritmo de encriptación que satisface:

$$\forall k_1, k_2 (k_1 \neq k_2)\, \exists k_3 \,:\, E_{k_2}(E_{k_1}(m)) = E_{k_3}(m)$$

Es decir, que encriptar con una clave k_1 y luego con una clave k_2 no es más seguro que encriptar sólo con k_1. Esta propiedad **no** es deseable.

34 Ejemplos de criptosistemas de clave privada

Cifrados por bloque:

- Transposiciones
  - Común: el caracter i del mensaje m se pondrá en una posición k en el mensaje encriptado.
    - Tiene estructura de grupo.
  - Por columnas: el caracter i del mensaje m se pondrá en una posición k en el mensaje encriptado, donde k depende de una clave de longitud l. La clave **no** puede tener caracteres repetidos.
    - El ataque por análisis de frecuencias no sirve.

- $|K| = l!$

Clave = (6,3,4,7,5,2,1)
Mensaje = ESTABLECENLASSSIGUIENTESCARAC

Mensaje en columnas:

1	2	3	4	5	6	7
E	S	T	A	B	L	E
C	E	N	L	A	S	S
I	G	U	I	E	N	T
E	S	C	A	R	A	C

Mensa encriptado:

6	3	4	7	5	2	1
E	L	S	T	B	E	A
S	S	E	N	A	C	L
T	N	G	U	E	I	I
C	A	S	C	R	S	E

Figura 28: Ejemplo de transposición por columnas.

- Sustituciones
  - **Monoalfabéticas**: un caracter original se encripta con un caracter.
  - **Polialfabéticas**: n caracteres originales se encriptan en m caracteres ($n \neq m$).
  - **Homofónica**: un caracter original x se encripta en varios caracteres. Cuanto más frecuente sea x en el mensaje, más cantidad de símbolo se le asignarán. De esta forma se evita el ataque por análisis de frecuencias.
  - **Poligráfica**: n caracteres originales se encriptan en n caracteres. La sustitución aplicada a cada caracter varía en función de la posición que éste ocupe dentro del mensaje.
- Productos: generar "rondas" donde en cada una se realiza una sustitución o una transposición. La clave puede variar entre cada ronda, y la misma determina el orden de las operaciones.

34.1 Scytale

- Es un palo en el cual se escribe un mensaje, en tiras.
- La clave es el tamaño del palo.
- Cifrado por transposición.

34.2 CAESAR

- Sustitución monoalfabética
- Muy malo
- Es el sistema mas viejo conocido
- $k \in K = \{1, 2, \ldots, 26\}$ (cantidad de caracteres del alfabeto)
- $|K| = 26$
- $E_k(char) = (char + k) \mod 26$
- $D_k(char) = (char - k) \mod 26$
- Criptoanálisis: ataque por fuerza bruta.

34.3 Vigenere

- Sustitución polialfabética.
- La clave k es un conjunto de l letras, $\{k_1, k_2, \ldots, k_l\}$.
- Para encriptar:
  - Suponemos un alfabeto de 26 caracteres.
  - Separar el mensaje m en grupos de l caracteres.
  - $E_k(m_i) = \left[m_i + k_{(i \mod l)}\right] \mod 26$

```
Plaintext:  CRYPTOISSHORTFORCRYPTOGRAPHY
Key:        ABCDABCDABCDABCDABCDABCDABCD
Ciphertext: CSASTPKVSIQUTGQUCSASTPIUAQJB
```

- El método de Vigénere tiene estructura de grupo si y sólo si k_1 y k_2 son tales que la longitud de una es múltiplo de la longitud de la otra.
- Desventajas:
  - La naturaleza repetitiva de la clave. Si el criptoanalista descubre la longitud de la clave, se puede tratar al *cyphertext* como un conjunto de cifrados por CAESAR. El test de Kasiski pueden ayudar a determinar la longitud de la clave.

34.4 Hill

- Sustitución poligráfica
- Tiene estructura de grupo
- $k \in K$ =matrices de $d \times d$
  - elementos deben ser enteros entre 0 y 25 (el tamaño del alfabeto) puestos en forma aleatoria
  - matriz debe ser inversible en aritmética módulo 26. $[(\det K) \mod 26] = 1$
- $|K| = 26^{d^2}$
- Sea un mensaje m con d caracteres. A cada caracter de m lo reemplazamos por su posición en el alfabeto.
  - $E_k(m) = (K \cdot m) \mod 26 = c$
  - $D_k(c) = \left[\left(K^{-1} \mod 26\right) \cdot c\right] \mod 26 = m$
- Criptoanálisis:
  - si d es chica, ataque por fuerza bruta probando $|K|$ claves.
  - si d es grande, ataque por texto plano conocido (si se conocen d^2 textos planos y sus criptogramas, se puede armar un sistema de ecuaciones lineales y despejar los valores de la matriz K).

34.5 Playfair

- Sustitución monoalfabética de orden 2 (grupos de 2 caracteres siempre son encriptados de la misma forma)
- $k \in K$ =matrices de 5×5
  - elementos deben ser las letras del alfabeto puestas en forma aleatoria (una posición de la matriz puede tener 2 letras)
- $|K| = 26^{25} \approx 2 \times 10^{35}$
- Encriptación:
  - Sea un mensaje M. Lo dividimos en grupos de 2 caracteres (c_1 y c_2).
  - A cada grupo le aplicamos la siguiente reglas:
    - Si $c_1 = c_2$, reemplazamos c_2 por una letra predeterminada (por ejemplo, 'X').
    - Si c_1, c_2 se encuentran en la misma fila de la matriz k:
      - Encriptar cada caracter con el caracter que está a la derecha del mismo en la matriz (si se termina la fila seguir en la de abajo).
    - Si c_1, c_2 se encuentran en la misma columna de la matriz k:
      - Encriptar cada caracter con el caracter que está abajo del mismo en la matriz (si se termina la columna seguir en la de la derecha).
    - Si c_1, c_2 no se encuentran en la misma fila ni en la misma columna de la matriz k:
      - Encriptar cada caracter con los caracteres que están en las esquinas del rectángulo formado por $c1, c2$, y en la misma fila que el caracter a encriptar.
- Desencriptación:
  - Ignorar la primera regla. En las reglas 2 y 3 hace shift arriba y a la izquierda en vez de abajo y a derecha. La regla 4 permanece igual.
  - Al finalizar, borrar las X sobrantes y localizar alguna I que debe ser J.
- Desventajas: $D(E(M)) \neq M$ porque dos letras iguales se encriptan como dos caracteres distintos.
- Criptoanálisis:
  - Análisis de frecuencias: analizar la frecuencia de aparición de los digramas y compararlas con los digramas más frecuentes del idioma en el cual se supone que se escribió el mensaje original.

34.6 DES (Data Encryption Standard)

- Cifrado de producto.
- Utiliza una clave de 56 bits (en realidad es de 64 bits, pero 8 bits son de paridad así que no los tenemos en cuenta).$|K| = 2^{56} \approx 7 \times 10^{16}$
- No tiene estructura de grupo.
- Fue creado por IBM, y modificada por NSA.
- En 1999 se logró romperlo en 1 día con una computadora valuada en U$S 100.000.
- Desventajas:
  - Tiene 4 claves débiles y 16 claves semi débiles.
  - La clave es muy corta.

- Encriptación:
  - Entran 64 bits de *plaintext* y salen 64 bits de *cyphertext*.
  - Redes de Feistel:
    - Dividir un bloque de longitud n en dos mitades, L y R.
      - $L_i = \begin{cases} R_{i-1} & \text{si } i < n \\ L_{i-1} \oplus f\left(R_{i-1}, K_i\right) & \text{si } i = n \end{cases}$
      - $R_i = \begin{cases} L_{i-1} \oplus f\left(R_{i-1}, K_i\right) & \text{si } i < n \\ R_{i-1} & \text{si } i = n \end{cases}$

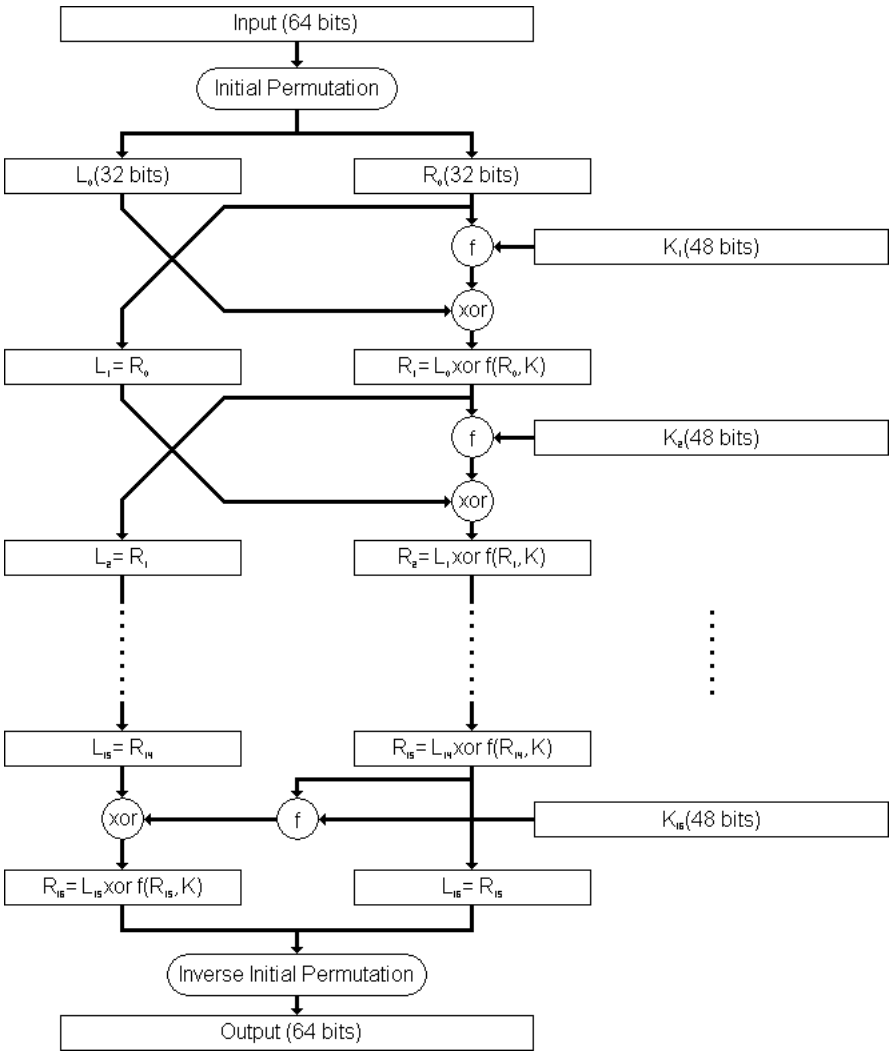


Figura 29: DES

- Criptoanálisis: ataque por fuerza bruta.

34.7 Triple-DES

- Triple DES con 3 claves independientes ($k_1 \neq k_2 \neq k_3$) tiene una longitud de clave de 168 bits, pero debido al ataque *meet-in-the-middle*, la seguridad efectiva es de 112 bits.

- Triple DES con 2 claves independientes ($k_3 = k_1$) tiene una longitud de clave de 112 bits, pero debido a los ataques de texto plano escogido o texto plano, la seguridad efectiva es de 80 bits.
    - Este modo es más seguro que encriptando con DES 2 veces, porque protege ante ataques *meet-in-the-middle*.
- Es tres veces más lento que DES.
- Encripta bloques de 64 bits.
- Si se elige el modo "2 claves",
    - $E_{k_1,k_2}(m) = E_{k_3}\left(D_{k_2}\left(E_{k_1}(m)\right)\right)$
    - $D_{k_1,k_2}(m) = D_{k_3}\left(E_{k_2}\left(D_{k_1}(m)\right)\right)$
- Criptoanálisis: ataque por fuerza bruta.

34.8 AES (*Advanced Encryption Standard*)

- Tiene dos partes:
    - Discreta: aplicar sustituciones y transposiciones
    - Continua
- Es el heredero moderno de DES.
- Fue diseñado en una competencia pública.
- Utiliza claves de 128, 192 o 256 bits.

35 Modos de encriptación por bloques

Problema: utilizando un algoritmo con una clave de longitud l, ¿cómo encriptar un mensaje de longitud $m > l$? Soluciones: dividir el mensaje en bloques de l caracteres, y encriptar los bloques.

- **ECB** (*Electronic Code Block*): cada bloque se encripta en forma independiente.
    - Desventaja: si b_1 y b_2 son dos bloques idénticos, $E\left(b_1\right) = E\left(b_2\right)$. Con lo cual, no esconde bien los patrones del *plaintext*.

```
// Encriptacion
while true:
  codigo n = encriptar (bloque n)

// Desencriptacion
while true:
  desencriptar (codigo n)
```

- **CBC** (*Cipher Block Chaining*): a cada bloque de *plaintext* se le aplica XOR con el bloque *cyphertext* anterior antes de ser encriptar. De esta manera, cada *cyphertext* depende de todos los *plaintext* anteriores. Para el primer bloque se usa un vector de inicialización.

```
// Encriptacion
codigo 0 = encriptar (bloque 0 XOR vector de inicializacion)
while true:
  codigo n = encriptar (bloque n XOR codigo n-1)

// Desencriptacion
bloque 0 = desencriptar (codigo 0) XOR vector de inicializacion
while true:
  bloque n = desencriptar (codigo n) XOR codigo n-1
```

- Desventajas:
  - La encriptación debe hacerse en forma secuencial; no se puede paralelizar.
  - Desencriptar con el vector equivocado hará que el primer bloque de *plaintext* esté corrupto (pero el resto no).

- **CFB** (*Chiper Feedback*): similar a CBC.

```
// Encriptacion
codigo 0 = encriptar (vector de inicializacion) XOR bloque 0
while true:
  codigo n = encriptar (codigo n-1) XOR (bloque n)

// Desencriptacion
bloque 0 = encriptar (vector de inicializacion) XOR codigo 0
while true:
  bloque n = encriptar(codigo n-1) XOR codigo n
```

- **OFB** (*Output Feedback*)

```
// Encriptacion
block cipher encryption = encriptar (vector de inicializacion)
codigo 0 = block cipher encryption XOR bloque 0
while true:
  block cipher encryption = encriptar (block cipher encryption)
  codigo n = block cipher encryption XOR bloque n

// Desencriptacion
block cipher encryption = encriptar (vector de inicializacion)
bloque 0 = block cipher encryption XOR codigo 0
while true:
  block cipher encryption = encriptar (block cipher encryption)
  bloque n = block cipher encryption XOR codigo n
```

36 Cifrado de flujo

- El emisor y el receptor tienen tiene una tira de bits (la misma) generadas aleatoriamente.
- Debe cumplirse que $l(mensaje) = l(clave)$.
- Es un método incondicionalmente seguro. Es el algoritmo de cifrado "perfecto".
- *Encrypting two different random "plain texts" with the same "pad" is indistinguishable from using two different random one time pads for encrypting the same plain text.*
- Desventajas:
  - Si el mensaje es muy largo, la clave también lo será.
  - Hay que transmitir la clave por cada mensaje y por cada remitente.
  - Dada la naturaleza determinista de las computadoras, generar una clave aleatoria es prácticamente imposible. Se deben utilizar mecanismos pseudoaleatorios, que cumplan tres propiedades:
    - Cada bit de la clave no se debe poder predecir, aún sabiendo los bits anteriores.
    - Debe pasar los tests estadistícos de aleatoriedad (chi cuadrado).
    - Si se corre el generador dos veces, las salidas deben ser distintas.
  - No proporciona autenticación.
- $E_k(m) = m \text{XOR} k$
- $D_k(m) = m \text{XOR} k$
- Criptoanálisis:

- Ataque por texto plano conocido o escogido no sirven porque las claves solo se utilizan para un solo mensaje, luego se descartan.
- Ataque por fuerza bruta no sirve porque solo generaría todos los mensajes que existen de esa longitud.
- Ataque por análisis de frecuencia no sirve porque el criptograma es aleatorio.

37 Ejemplos de criptosistemas de clave pública

37.1 Knapsacks

- No provee autenticación.

  Problema matemático de la mochila: se tiene una mochila con capacidad para k kilos. Se tienen n objetos cuyos pesos son $P = (p_1, p_2, \ldots, p_n)$. ¿Qué objetos seleccionamos para guardar en la mochila para que ésta quede llena?
  Solución computacionalmente lenta: probar las 2^n combinaciones.

Criptosistema de clave privada:

- Encriptación:
  1. La clave es $K = P$.
  2. Dado un mensaje, lo pasamos a binario.
  3. Para cada bloque de n bits del mensaje:
     a) Sea α el vector de bits del bloque.
     b) Se calcula $S = \sum_{i=1}^{n} \alpha_i \cdot K_i$
     c) S es la encriptación del bloque.
- Desencriptación: dado el código S, y la clave K, hay que resolver el problema de la mochila (hallar los α_i tal que $S = \sum_{i=1}^{n} \alpha_i \cdot K_i$).

Criptosistema de clave pública: se encripta con la clave pública y se desencripta con la clave privada.

- Generación de claves:
  - Elegir un vector V super incrementante ($V_i > \sum_{j=0}^{i-1} V_j$ para todo i)
  - Elegir dos números t ("multiplicador") y m ("módulo") tal que t y m no tengan factores en común (i.e. deben ser relativamente primos)
  - Calcular t', el inverso multiplicativo de t usando aritmética módulo m.
  - La clave **privada** es $< t', m >$.
  - Obtener el vector $V' = (p_i \times t) \mod m$. V' **no** es super incrementante. Esta es la clave **pública**.
- Encriptación:
  - Dado un mensaje con l caracteres, lo pasamos a binario.
  - Para cada bloque de n bits del mensaje:
    1. Sea α el vector de bits del bloque.
    2. Se calcula $S = \sum_{i=1}^{n} \alpha_i \cdot V'_i$
    3. S es la encriptación del bloque.
  - La encriptación del mensaje resulta $(S_1, S_2, \ldots, S_l)$
- Desencriptación:
  - Dado el código $(S_1, S_2, \ldots, S_l)$, el vector público V' y la clave privada $< t', m >$:
    - Obtener el vector A haciendo $A_i = (S_i \times t') \mod m$
    - Obtener el vector V haciendo $V_i = (V'_i \times t') \mod m$
    - Resolver el problema de la mochila (hallar los α_i tales que $A_i = \sum_{i=1}^{nl} \alpha \cdot V_i$).

37.2 RSA (Rivest - Shamir - Adleman)

- Provee confidencialidad (se encripta con la pública y se desencripta con la privada).

  Problema: se tiene un número k que es el producto de dos números primos p y q de aproximadamente 100 dígitos cada uno (desconocidos). ¿Cómo obtener los números p y q a partir de k?
  Solución computacionalmente lenta: probar todas las combinaciones de p y q.

Generación de claves:

- Elegir dos números primos p y q distintos, muy grandes, de tamaño en bits similar.
- Calcular:
  - $n = p \times q$
  - $\phi(n) = (p-1) \times (q-1)$
- Elegir un número e tal que $1 < e < \phi$ y $mcd(e, \phi(n)) = 1$ (es decir, e y $\phi(n)$ son coprimos).
- Calcular d como $d^{-1} \equiv e(\mod \phi(n))$. Es decir, d es el inverso multiplicativo de e en aritmética módulo $\phi(n)$. Utilizar el algoritmo de Euclides.
- La **clave pública** es el par (n, e).
- La **clave privada** es el par (ϕ, d).

Encriptación:

- Pasar el mensaje a binario.
- Dividir el mensaje en bloques de tamaño i $(10^{i-1} < n < 10^i)$
- Para cada bloque b:
  - Calcular $c = (b^e) \mod n$[10]

Desencriptación:

- Para cada bloque c:
  - Obtener $b = (c^d) \mod n$

Desventajas:

- El criptograma resultante es de tamaño fijo.
- Es lento.

  Suppose Alice uses Bob's public key to send him an encrypted message. In the message, she can claim to be Alice but Bob has no way of verifying that the message was actually from Alice since anyone can use Bob's public key to send him encrypted messages. In order to verify the origin of a message, ***RSA can also be used to sign a message****.*

Criptoanálisis:

- Ataque por fuerza bruta no sirve porque habría que probar una cantidad enorme de combinaciones de números.

[10] Para calcular esto rápidamente, las computadoras usan exponenciación por cuadrados. Cualquier número x puede representarse como $x = \sum_i \alpha_i 2^i$. Entonces $(a^b \mod q)$ implica hacer $a_i = a_{i-1}^2$ tantas veces como bits tenga b. El resultado es $\prod_i \text{bit}_i(b) a_i \mod q$

38 Criptoanálisis

Tipos de ataques:

- **Por fuerza bruta**: dado C, probar el espacio K de claves para obtener M.
- ***Shoulder surfing***: utilizando *keyloggers*, cámaras, etc.
- **Caballos de Troya**: programa que cada vez que se ejecuta, le pide la contraseña al usuario, y la graba en un archivo.
- **Ingeniería social**: es el método más útil. Consiste en conseguir que la persona que conoce la clave privada nos la diga.
- **Ataque por texto plano**: el atacante conoce textos planos y sus correspondientes criptogramas, pero no conoce E. Quiere descubrir la clave.
- **Ataque por texto plano escogido**: el atacante puede encriptar textos planos que él escogió y obtener sus correspondientes criptogramas. Quiere descubrir la clave.

  - Ataque *meet-in-the-middle*: afecta solo a cifrados de producto (DES, 3DES, etc.). Consiste en encriptar de un lado, desencriptar del otro, y matchear los resultados intermedios.
    El criptoanalista conoce M_1, C_1 y M_2, C_2, y quiere descubrir k_1 y k_2.

$$\begin{aligned} C_1 &= E_{k_2}\left(E_{k_1}(M_1)\right) \\ C_2 &= E_{k_2}\left(E_{k_1}(M_2)\right) \\ D_{k_2}(C_1) &= E_{k_1}\left(C_1\right) \end{aligned}$$

    1. Para cada posible j, calcular $E_j\left(M_1\right)$ y guardar en memoria el par $j, E_j\left(M_1\right)$ en una tabla de hash.
    2. Para cada posible k, calcular $D_k\left(C_1\right)$ y buscar este resultado en la memoria. Si se lo encuentra, es posible que $k = k_2$ y que $j = k_1$.
    3. Si $E_k\left(E_j(M_2)\right) = C_2$, es muy probable que descubierto k_1y k_2. Si no son iguales, tendrá que seguir buscando coincidencias.

    Este ataque logra disminuir la cantidad de intentos necesarios de 2^{2n} a 2^{n+1}, pero requiere muchísima memoria (2^{56} bloques de memoria de 64 bits).

- **Ataque por criptograma**: el atacante conoce criptogramas generados por el criptosistema, e intenta recuperar los mensajes asociados (no la clave).
- **Análisis de frecuencias**: estudiar la frecuencia con que aparecen los distintos símbolos en un lenguaje determinado, y compararla con la frecuencia de los símbolos de los criptogramas.
- **Ataque del intermediario** (*man-in-the-middle*): solo es posible en criptosistemas asimétricos y en medios inseguros.

  1. Alice le envía un mensaje a Bob, el cual es interceptado por Mallory.

```
Alice "Hi␣Bob,␣it's␣Alice.␣Give␣me␣your␣key"--> Mallory Bob
```

  2. Mallory le envía este mensaje a Bob; Bob no sabe que en realidad el mensaje provino de Mallory.

```
Alice Mallory "Hi␣Bob,␣it's␣Alice.␣Give␣me␣your␣key"--> Bob
```

  3. Bob responde con su clave pública.

```
Alice Mallory <--[Bob's␣key]␣Bob
```

4. Mallory reemplaza la clave de Bob con la suya, y se la envía a Alice, haciéndole creer que es la clave de Bob.

```
Alice <--[Mallory's key] Mallory Bob
```

5. Alice encripta un mensaje con lo que cree es la clave de Bob.

```
Alice "Meet me at the bus stop!"[encrypted with Mallory's key]--> Mallory Bob
```

6. Sin embargo, como en realidad fue encriptado con la clave de Mallory, Mallory puede desencriptarla, leerla, modificarla si quiere, reencriptarla con la clave de Bob, y enviársela a Bob.

```
Alice Mallory "Meet me at 22nd Ave!"[encrypted with Bob's key]--> Bob
```

7. Bob cree que este mensaje significa que ha establecido una comunicación segura con Alice.

Este ataque se puede evitar con el uso de certificados digitales que prueben la autenticidad de una clave pública.

- **Análisis diferencial**: dados dos mensajes m_1 y m_2 con un bit de diferencia, estudiar las diferencias entre $E_k(m_1)$ y $E_k(m_2)$.
- **Ataque por replay**: *a form of network attack in which a valid data transmission is maliciously or fraudulently repeated or delayed. This is carried out either by the originator or by an adversary who intercepts the data and retransmits it.*

  Suppose Alice wants to prove her identity to Bob. Bob requests her password as proof of identity, which Alice dutifully provides (possibly after some transformation like a hash function); meanwhile, Mallory is eavesdropping on the conversation and keeps the password (or the hash). After the interchange is over, Mallory (posing as Alice) connects to Bob; when asked for a proof of identity, Mallory sends Alice's password (or hash) read from the last session, which Bob accepts.

  - Soluciones:
    - Uso de claves de sesión generados por un algoritmo aleatorio.
    - Uso de claves desechables.
    - Timestamping, pero requiere de sincronización entre las partes.

Resultados de un ataque según la severidad del mismo (en orden decreciente):

1. Ruptura total: el atacante descubre la clave.
2. Deducción global: el atacante descubre un algoritmo funcionalmente equivalente al cifrado y descifrado de mensajes.
3. Deducción local: el atacante descubre el texto plano de un criptograma.
4. Deducción de información: el atacante recibe alguna información acerca del texto plano o de la calve.
5. Distinción del algoritmo: el atacante descubre un patrón en los criptogramas.

39 Firma digital y certificados digitales

Una firma digital es un mecanismo criptográfico que permite al receptor de un mensaje firmado digitalmente determinar la entidad originadora de dicho mensaje (autenticación de origen y no repudio).

Las firmas digitales suelen incluir *timestamps*, para impedir que alguien las reutilice.

Propiedades necesarias:

1. Únicas: Las firmas deben poder ser generadas solamente por el firmante.

2. Infalsificables: Para falsificar una firma digital el atacante tiene que resolver problemas matemáticos de una complejidad muy elevada, es decir, las firmas han de ser computacionalmente seguras. Por tanto la firma debe depender del mensaje en sí.
3. Verificables: Las firmas deben ser fácilmente verificables por los receptores de las mismas y, si ello es necesario, también por los jueces o autoridades competentes.
4. Innegables: El firmante no debe ser capaz de negar su propia firma.
5. Viables: Las firmas han de ser fáciles de generar por parte del firmante.

39.1 Firma digital simétrica

- Provee confidencialidad, autenticación y no repudio.
- Es un protocolo arbitrado: debe existir una autoridad que almacene las claves privadas de todos (el *big brother*). Esta autoridad debe tener su propia clave privada.
- Encriptación:
  - Alice encripta m mediante $E_{k_{privada_{Alice}}}(m) = c$. Le envía al *big brother* c, junto con el nombre de Bob.
  - El *big brother* desencripta c mediante $D_{k_{privada_{Alice}}}(c) = m$. Encripta m con la clave privada del Bob,

$$E_{k_{privada_{Bob}}}\left(Bob, m, \underbrace{E_{k_{privada_{bigbrother}}}(Bob, m)}_{\text{garantiza no repudio de Alice}}\right)$$

    y se lo envía a Bob.
- Desencriptación:
  - Bob desencripta lo que le envió el *big brother* con su clave privada.
- Desventajas:
  - Todo el sistema depende del buen funcionamiento del *big brother*.

39.2 Firma digital asimétrica

- Provee autenticación y no repudio (solo el emisor puede firmar el mensaje, pero cualquiera puede verificar la firma).
- Encriptación:
  - El emisor desencripta m con su clave privada.
  - El emisor encripta m con la clave pública del receptor.
- Desencriptación:
  - El receptor desencripta m con su clave privada.
  - El receptor encripta m con la clave pública del emisor.
- Desventajas:
  - Es lento.

39.3 Firma digital con *message digests*

- Para lograr integridad de las transmisiones.
- Se utiliza una función de hash para hashear el documento[11]. La firma consiste en encriptar el hash, que es mucho más rápido que el caso anterior, que hay que desencriptar y encriptar.
    - Características de la función de hashing f:
        - Dado x, $f(x)$ es fácil de calcular.
        - Dado $f(x)$, x es muy difícil de calcular.
        - Dado x, es difícil obtener un x' tal que $f(x) = f(x')$.
        - Si le cambiamos un bit a x para obtener $\widetilde{x}$, $f(\widetilde{x})$ será muy distinta a $f(x)$.
        - $f(x)$ tiene longitud fija (128 o 256 bits).

Firma digital con resumen que **no** brinda confidencialidad:

- Provee autenticación, no repudio, integridad.
- Emisión:
    1. Alice aplica la función resumen al mensaje, obteniendo md.
    2. Alice desencripta md con su clave privada: $D_{k_{priv_{Alice}}}(md)$.
    3. Alice envía $m + D_{k_{priv_{Alice}}}(md)$.
- Recepción:
    1. Bob encripta $D_{k_{priv_{Alice}}}(md)$ con la clave pública del Alice obteniendo m y md.
    2. Bob compara $MD(m) = md$. Si son iguales, sabe que el emisor es quien dice ser, y además que el mensaje no fue modificado en tránsito.

Firma digital con resumen que **sí** brinda confidencialidad:

- Provee confidencialidad, autenticación, no repudio e integridad.
- La encriptación se aplica después de la firma, y se debe encriptar tanto la firma como el mensaje final (mensaje + firma). Es como enviar una carta por correo: hay que firmar tanto la carta como el sobre.
- Emisión:
    1. Alice aplica la función resumen a m, obteniendo md.
    2. Alice encripta md con su clave privada, obteniendo $E_{k_{priv_{Alice}}}(md)$.
    3. Alicer encripta el mensaje m y lo anterior con la clave pública de Bob. Obtiene $E_{k_{pub_{Bob}}}\left(m + E_{k_{priv_{Alice}}}(md)\right)$.
- Recepción:
    1. Bob desencripta lo recibido con su clave privada. Obtiene $m + E_{k_{priv_{Alice}}}(md)$.
    2. Bob desencripta la segunda parte con la clave pública de Alice, obteniendo md.
    3. Compara $MD(m) = md$. Si son iguales, sabe que el emisor es quien dice ser, y además que el mensaje no fue modificado en tránsito.

[11] Una función de hash no se puede utilizar para encriptar, porque nadie podría desencriptarlo.

39.4 Certificados digitales

El objetivo de un certificado digital es relacionar inequívocamente una persona con su clave pública. Si la clave no estuviese certificada, no podría asegurar autenticación o no repudio, a menos que hubiesen sido intercambiadas personalmente.

- Certificación:
  1. Se le aplica la función resumen a "datos de la persona + clave pública de la persona", obteniendo $md(persona + k_{pub_{persona}})$.
  2. La autoridad certificante recibe $md(persona + k_{pub_{persona}})$. La encripta con su clave privada y se genera $E_{k_{priv_{autoridad}}}\left(md(persona + k_{pub_{persona}})\right)$.
  3. El certificado digital es:
     - $persona + k_{pub_{persona}}$
     - $E_{k_{priv_{autoridad}}}\left(md(persona + k_{pub_{persona}})\right)$
- Verificación:
  1. Se desencripta $E_{k_{priv_{autoridad}}}\left(md(persona + k_{pub_{persona}})\right)$ con la clave pública de la autoridad certificante. Se obtiene $md(persona + k_{pub_{persona}})$.
  2. Se calcula la función resumen de $persona + k_{pub_{persona}}$, y se compara con lo anterior. Si son iguales, la firma es auténtica.

40 Protocolos criptográficos

Protocolo: conjunto finito de pasos que deben realizar las partes involucradas para llevar a cabo un objetivo. El mismo debe ser de previo conocimiento de las partes participantes. Además debe ser completo (no existen situaciones que no abarque) y no ambiguo (para cada situación un único resultado).

En criptografía se utilizan protocolos para impedir o detectar fraudes y/o que otras partes intercepten la comunicación.

Los protocolos pueden ser:

- **Arbitrados**: existe una tercera parte confiable y neutral que controla que las 2 partes cumplan el protocolo.
- **Adjudicados**: como el anterior, pero no está controlando todo el tiempo. Sólo se lo "llama" para que determine si un protocolo fue ejecutado de forma justa.
- **Autosuficientes**: no existen terceras partes.

40.1 Kerberos

- Provee autenticación.
- Alice (cliente) y Bob (servidor) comparten la clave con Authentication Server.
- La mayor parte del trabajo lo hace Alice.
- Alice quiere generar una clave de sesión para comunicarse con Bob.

1. Alice le envía un mensaje a Authentication Server con su nombre, y un mensaje encriptado con la clave privada de Alice (éste tiene un período de validez).
2. **Authentication Server** trata de desencriptar el mensaje con la clave privada de Alice (la cual posee). Si lo logra, la autenticación es correcta.

3. **Authentication Server** genera un **TGT** (*Ticket Granting Ticket*) con un *timestamp*, una vida útil, una clave de sesión aleatoria K_1, y la identidad de **Alice**. La encripta con la clave de **Ticket Granting Server**.

   Authentication Server genera un mensaje con la clave de sesión K_1. La encripta con la clave de **Alice**.

   Envía ambas cosas a Alice.

4. Alice le envía a **Ticket Granting Server** el TGT junto con el nombre de **Bob**.

   Alice le envía a **Ticket Granting Server** el authenticator (cliente + timestamp) encriptado con la clave de sesión K_1.

5. **Ticket Granting Server** le envía un Ticket a Alice:

   a) Nombre de **Bob**,

   b) Mensaje que contiene el nombre de **Alice**, la validez, y la clave de sesión K_2, encriptados con la clave de **Bob**.

   Ticket Granting Server envía otro mensaje a Alice: la clave de sesión K_2 para que pueda comunicarse con el servidor. Esta está encriptada con la clave K_1.

6. Alice le envía el Ticket a Bob.

   Alice le envía un **authenticator** al servidor (cliente + timestamp) encriptado con la clave K_2.

7. Bob genera un mensaje con el *timestamp+1*, lo encripta con K_2, y se lo envía a Alice. Esto se hace para evitar el ataque *man-in-the-middle*.

Los *tickets* son reutilizables (por 8 horas).

Los *authenticators* tienen un tiempo de duración corto.

Desventajas:

- Se depende de Authentication Server.
- Los relojes de todos deben estar sincronizados.

40.2 PGP

- Algoritmo híbrido.
- Trabaja con RSA con claves de 256, 512 y 1024 bits.

PGP emisión:

1. El emisor aplica una función de hash al mensaje, obteniendo md.
2. El emisor desencripta md con su clave privada, obteniendo $D_{k_{priv_{emisor}}}(md)$.
3. El emisor comprime $m + D_{k_{priv_{emisor}}}(md)$ y obtiene $C\left[m + D_{k_{priv_{emisor}}}(md)\right]$
4. El emisor encripta $C\left[m + D_{k_{priv_{emisor}}}(md)\right]$ con la **clave de sesión** (clave desechable, generada mediante un algoritmo pseudoaleatorio que utiliza tipeos de teclado o movimientos de mouse por parte del usuario) obteniendo $E_{k_{sesion}}\left(C\left[m + D_{k_{priv_{emisor}}}(md)\right]\right)$.
5. El emisor encripta la clave de sesión con la clave pública del receptor, obteniendo $E_{k_{pub_{emisor}}}(k_{sesion})$.
6. El emisor envía $E_{k_{sesion}}\left(C\left[m + D_{k_{priv_{emisor}}}(md)\right]\right) + E_{k_{pub_{emisor}}}(k_{sesion})$.

PGP recepción:

1. El receptor desencripta $E_{k_{pub_{emisor}}}(k_{sesion})$ con su clave privada, y obtiene la clave de sesión.

2. El receptor desencripta $E_{k_{sesion}}\left(C\left[m + D_{k_{priv_{emisor}}}(md)\right]\right)$ con la clave de sesión que obtuvo en el paso anterior, y obtiene $C\left[m + D_{k_{priv_{emisor}}}(md)\right]$.
3. El receptor descomprime lo anterior y obtiene $m + D_{k_{priv_{emisor}}}(md)$.
4. El receptor encripta $D_{k_{priv_{emisor}}}(md)$ con la clave pública del emisor y obtiene md.
5. Ahora que el receptor tiene m y md: si $MD(m) = md$, entonces sabe que el mensaje es auténtico. Si no son iguales, entonces o bien el mensaje fue modificado en tránsito, o bien el emisor no es quien dice ser.