# Resumen: Fundamentos de Organización de Datos

## Archivos:

**¿Qué es una base de datos?** Conjunto de datos interrelacionados con un propósito específico vinculado a la resolución de un problema del mundo real.

A su vez, un **Sistema de Gestión de Bases de Datos (SGBD)** consiste en un conjunto de programas necesarios para acceder y administrar una base de datos. Para manipular datos, se utiliza un **Lenguaje de Manipulación de Datos (LMD)**, mediante el cual se puede recuperar información, agregar nueva información y modificar o borrar datos existentes. Existen dos tipos de LMD:

* **LMD procedimentales:** el usuario debe especificar qué datos requiere y cómo obtener esos datos.
* **LMD no procedimentales:** el usuario debe especificar qué datos requiere, sin detallar cómo obtener esos datos.

**¿Qué es un archivo?** Los datos que necesitan ser preservados más allá de la ejecución de un algoritmo deben residir en archivos. Para que esto suceda, un archivo no puede residir en la memoria RAM, sino que debe estar en dispositivos de almacenamiento permanente de información, como un disco rígido. Hay 3 grandes definiciones:

* Colección de registros semejantes, guardados en almacenamiento secundario.
* Colección de datos del mismo tipo, almacenados en dispositivos secundarios de memoria.
* Colección de registros que abarcan entidades con un aspecto común y originadas para algún propósito particular.

**¿Cómo se organiza un archivo?** Hay 2 formas:

* Por una secuencia de **bytes**
* Archivos de texto.
* Se leen o recuperan caracteres.
* No hay formato previo.
* Una palabra se termina por un conjunto de caracteres que termina con blanco. Pero eso es una “convención”.
* Por un conjunto de **registros**
* Hay estructura.
* Los registros pueden estar conformados por campos.

Hay que diferenciar 2 conceptos diferentes (pero relacionados) de archivos:

* **Archivo físico:**
* Es el archivo residente en la memoria secundaria.
* Es administrado por el sistema operativo.
* **Archivo lógico:**
* Es el archivo utilizado desde el algoritmo.
* Cuando este necesita operar con un archivo, genera una conexión con el sistema operativo, el cual será el responsable de la administración. Esta acción se denomina independencia física.

Formas de acceso a los datos de un archivo:

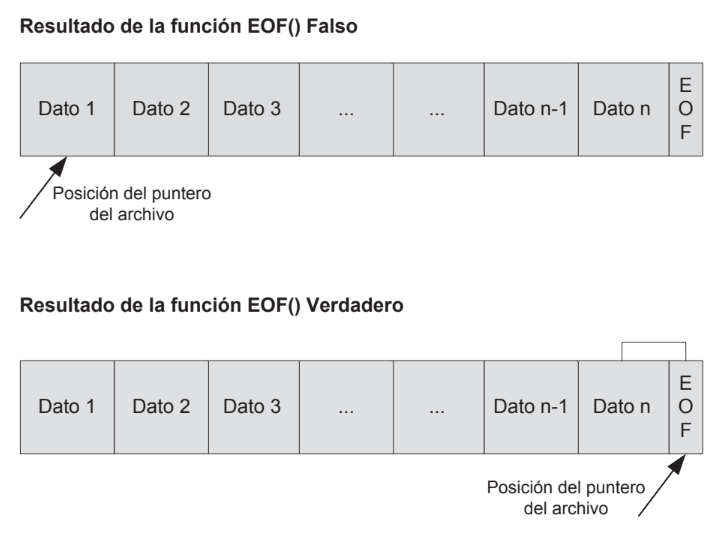
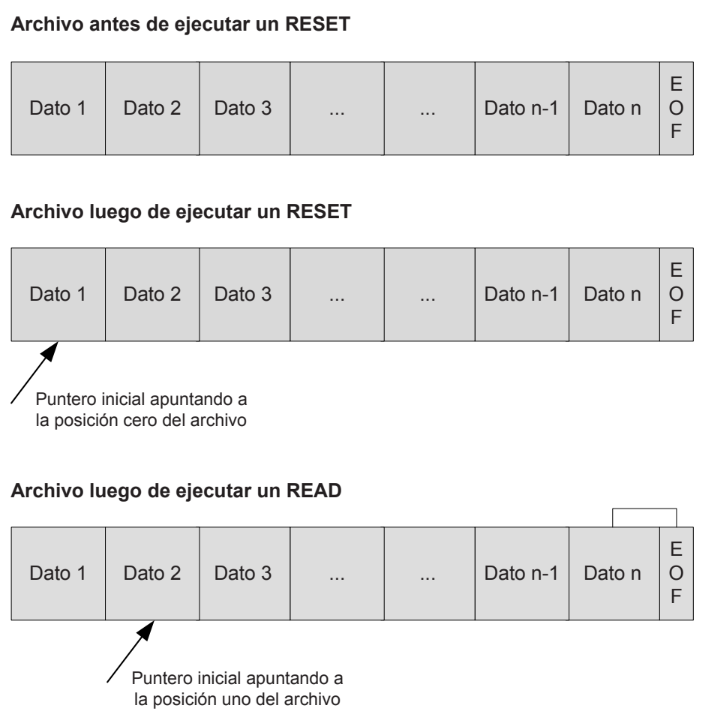
* **Secuencial físico:**
* El acceso a cada elemento de datos se realiza luego de haber accedido a su inmediato anterior.
* El recorrido es desde el primero hasta el último de los elementos, siguiendo el orden físico de estos.
* Archivos serie.
* **Secuencial indizado (lógico):**
* El acceso a los elementos de un archivo se realiza teniendo presente algún tipo de organización previa, sin tener en cuenta el orden físico.
* Por ejemplo, si disponemos de un archivo con los nombres de los empleados de una empresa: los datos de dicho archivo están almacenados en el orden en el cual fueron ingresados; sin embargo, es posible recuperar los datos en forma ordenada como si hubiesen sido ingresados alfabéticamente.
* Árboles.
* **Directo:**
* Es posible recuperar un elemento de dato de un archivo con un solo acceso, conociendo sus características, más allá de que exista un orden físico o lógico predeterminado.
* Hashing.

**Buffers** de memoria:

* Las lecturas y escrituras desde o hacia archivo se realizan sobre buffers.
* Un buffer es una memoria intermedia (ubicada en RAM) entre un archivo y un programa, donde los datos residen provisoriamente hasta ser almacenados definitivamente en la memoria secundaria.
* Esto se debe a cuestiones de *performance*, ya que mientras que una operación sobre la memoria RAM tiene una demora del orden de nanosegundos, una operación sobre disco tiene una demora del orden de milisegundos.

Puntero de trabajo de un archivo (**File pointer**):

* En el momento de ejecutarse la instrucción reset, el sistema operativo coloca un puntero direccionando al primer registro disponible dentro del archivo. Este puntero, relacionado al nombre lógico del archivo, indicará en todo momento la posición actual de trabajo. Tanto la instrucción read como la write avanzan en forma automática el puntero una posición luego de ejecutarse.
* Un algoritmo debe controlar el fin de archivo antes de realizar una operación de lectura. La función EOF(nombre\_logico) es la encargada de resolver este control. La función retornará verdadero si el puntero del archivo referencia a EOF, y falso en caso contrario.

**Operaciones esenciales sobre archivos:**

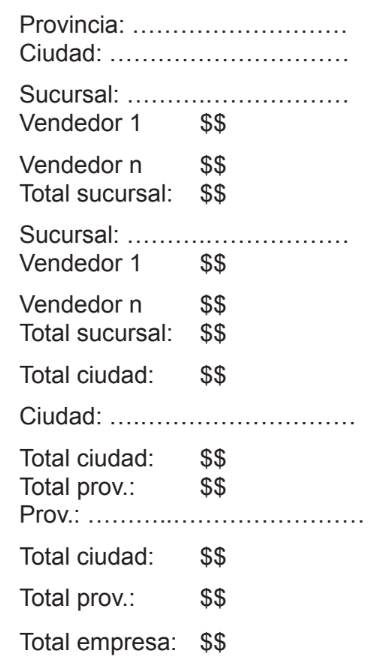
* **Alta:** ingresar nuevos datos al archivo.
* **Modificación:** alterar el contenido de algún dato del archivo.
* **Consulta:** presentar el contenido total o parcial del archivo.
* **Baja:** quitar información del archivo. Sin embargo, cuando se opera con bases de datos, suelen ser muy infrecuentes las circunstancias donde se desea quitar información de un archivo.

Condiciones para realizar una **actualización maestro-detalle:**

* Existe un archivo maestro.
* Existe al menos un archivo detalle que modifica al maestro.
* Cada registro del detalle modifica a un registro del maestro. Esto significa que solamente aparecerán datos en el detalle que se correspondan con datos del maestro. Se descarta la posibilidad de generar altas en ese archivo.
* No todos los registros del maestro son necesariamente modificados.
* **Todos los archivos están ordenados por igual criterio**. Esta precondición, considerada esencial, se debe a que hasta el momento se trabaja con archivos de datos de acuerdo con su orden físico.
* La información es **disjunta**; esto significa que un elemento puede aparecer una sola oportunidad en todo el problema.
* Se utiliza un proceso leer para no perder datos por la comprobación del EOF.

Condiciones para realizar un **Merge 🡪** Mismas que en maestro-detalle. La principal diferencia es que el **maestro no existe** y se crea a partir de la conjunción de los datos de los detalles.

**Corte de control** 🡪 Proceso mediante el cual la información de un archivo es presentada en forma organizada de acuerdo con la estructura que tiene el archivo.

Por ejemplo, se almacena en un archivo la información de ventas de una cadena de electrodomésticos, y es necesario informar al gerente de ventas de la empresa el total de ventas producidas de acuerdo con el siguiente formato:

Deben tenerse en cuenta las siguientes **precondiciones**:

* El archivo se encuentra ordenado por algún criterio. Por ejemplo, provincia, ciudad,

sucursal y vendedor.

* Se debe informar el total de vendido en cada sucursal, ciudad y provincia, así como el

total final.

* En diferentes provincias pueden existir ciudades con el mismo nombre, o en diferentes

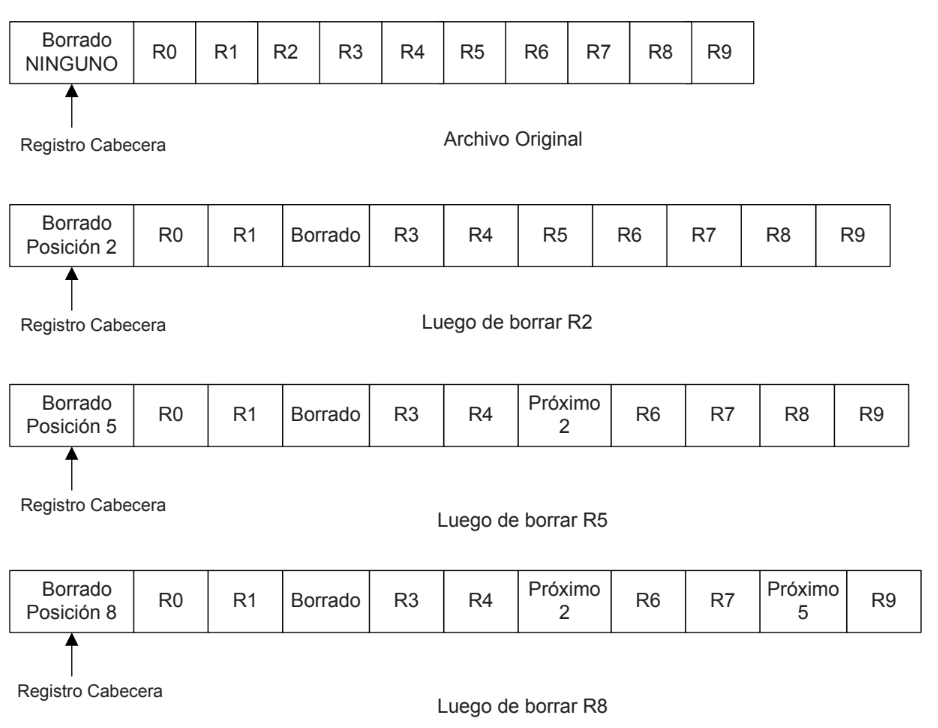
ciudades pueden existir sucursales con igual denominación.

**Bajas** 🡪 Proceso que permite quitar información de un archivo.

Históricamente, el proceso de baja era necesario para no mantener información poco útil en un archivo y recuperar espacio en dispositivos de almacenamiento secundario.

En la actualidad, las organizaciones que disponen de bases de datos consideran la información como su bien más preciado. En general, el conocimiento adquirido (*información*) no se quita, sino que se preserva en archivos o repositorios históricos. Teniendo en cuenta esta situación, el proceso de baja se relativiza en importancia.

El proceso de baja puede llevarse a cabo de dos modos diferentes:

* **Baja física**: consiste en borrar efectivamente la información del archivo, recuperando el espacio físico.
* Se realiza baja física sobre un archivo cuando un elemento es **quitado** del archivo y es reemplazado por otro elemento del mismo archivo, decrementando en uno su cantidad de elementos.
* La **ventaja** de este método consiste en administrar siempre un archivo de datos que ocupe **el mínimo lugar necesario**.
* La **desventaja** tiene que ver con la **performance** de los algoritmos que implementan esta solución. Ya que, suponiendo que el archivo tiene *n registros*, *n* *lecturas* y *n-1 escrituras*, todas las operaciones se realizan en forma secuencial.
* Para realizar el proceso de baja física existen, básicamente, dos técnicas algorítmicas:
* Generar un **nuevo** archivo con los elementos válidos (sin copiar los que se desea eliminar).
* Utilizar el **mismo** archivo de datos, generando los reacomodamientos que sean necesarios y truncando el archivo tantas posiciones atrás como elementos se quieran borrar.
* **Baja lógica**: consiste en borrar la información del archivo, pero sin recuperar el espacio físico respectivo.
* Se realiza una baja lógica sobre un archivo cuando el elemento que se desea quitar es **marcado como borrado**, pero sigue ocupando el espacio dentro del archivo.
* La **ventaja** del borrado lógico tiene que ver con la **performance**. Basta con localizar el registro a eliminar y colocar sobre él una marca que indique que se encuentra no disponible. Entonces, el costo es de tantas lecturas sean necesarias hasta encontrar el elemento a borrar, y una sola escritura que deja la marca de borrado lógico sobre el registro.
* La **desventaja** de este método está relacionada con el **espacio** en disco. Al no recuperarse el espacio borrado, el tamaño del archivo tiende a crecer continuamente. Pero esto puede ser compensado con técnicas alternativas para el alta de nuevos elementos. Estas son:
* **Recuperación de espacio** 🡪 Periódicamente utilizar el proceso de baja física para realizar un proceso de **compactación** del archivo, quitando todos aquellos registros marcados como borrados, utilizando cualquiera de los algoritmos de borrado físico.
* • **Reasignación de espacio** 🡪 otra alternativa posible consiste en recuperar el espacio, utilizando los lugares indicados como borrados para el alta de nuevos elementos al archivo.
* Esto puede realizarse buscando los lugares libres desde el comienzo del archivo, pero es poco eficiente.
* La alternativa consiste en que, a medida que los datos se borran del archivo, se genera una lista encadenada invertida con las posiciones borradas. Al comienzo, en el registro cabecera del archivo solamente se indica que no hay registros borrados. A medida que se quitan elementos, este proceso se reitera en la lista de elementos borrados que se va construyendo.

**Registros con longitud fija y variable:**

La información en un archivo siempre es **homogénea**. Todos los elementos almacenados en él son del mismo tipo. De esta forma, cada uno de los datos es del mismo tamaño, generando lo que se denomina archivos con registros de longitud fija.

La longitud de cada registro está determinada por la información que se guarda.

* Por ejemplo, si se define un archivo que contiene números enteros, cada elemento ocupa 2 bytes.
* Si se define un archivo que contiene registros con Nombre y apellido (String[40]), DNI (String[8]), Edad (integer), cada elemento ocupa 50 bytes (la suma de cada uno de los campos).

Administrar archivos con **registros** **de** **longitud** **fija** tiene algunas importantes **ventajas**:

* El proceso de entrada y salida de información, desde y hacia los buffers, es responsabilidad del sistema operativo.
* Los procesos de alta, baja y modificación de datos son (relativamente) sencillos.

*No obstante*, hay determinados problemas donde **no es posible**, o no es deseable, **trabajar con registros de longitud fija**. Supóngase el siguiente ejemplo:

* Se define un registro empleado con la siguiente estructura:
* Type Empleado = Record;

Nombre: String [50];

Dirección: String [50];

Documento: String [12];

Edad: integer;

Observaciones: String [200];

End;

* El registro ocupa 314 bytes.
* Cada vez que se ingresan los datos de un empleado, en promedio se ocupan 30 lugares para el nombre, otro tanto para la dirección y, en algunos casos, 50 bytes para observaciones.
* En promedio, cada registro utiliza 124 de los 314 bytes disponibles. Esto significa desperdicio de espacio y mayor tiempo de procesamiento. Se transfieren 314 bytes de los cuales solo 124 representan información útil; el resto son espacios de relleno.

Para evitar estas situaciones, es de interés contar con alguna organización de archivos que solo utilice el espacio necesario para almacenar la información. Este tipo de soluciones se representan con archivos donde los **registros utilicen longitud variable**. En estos casos, como el nombre lo indica, el **espacio** utilizado por cada elemento del archivo **no está determinado** a priori.

Pero para poder arribar a soluciones de este tipo, es necesario que los archivos de datos tengan una estructura diferente de lo visto hasta el momento.

* Cada elemento de dato debe descomponerse en cada uno de sus elementos constitutivos y así, elemento a elemento, guardarse en el archivo.
* En el caso de necesitar transferir un String, debe hacerse carácter a carácter
* En caso de tratarse de un dato numérico, cifra a cifra.
* Etc.
* Es necesario colocar marcas que delimiten cada elemento. Estas marcas se denominan **marcas de fin de campo** o **marcas de fin de registro**. Hay algunas *alternativas* a la marca de fin de registro:
* Si un registro dispone de 4 campos, luego de obtener 4 marcas de fin se asume la finalización de un registro y el comienzo de uno nuevo.
* Utilizar indicadores de longitud de campo y/o registro. De esta manera, antes de almacenar un registro, se indica su longitud; luego, los siguientes bytes corresponden a elementos de datos de dicho registro.

**¿Cómo se realiza una baja en un registro de longitud variable**? El proceso de baja lógica **no tiene diferencias sustanciales** con respecto a los registros de longitud fija. Sin embargo, cuando se desea recuperar el espacio borrado lógicamente con nuevos elementos, deben tenerse en cuenta nuevas consideraciones relacionadas con el **espacio disponible**.

* Mientras que con registros de longitud fija los elementos a eliminar e insertar son del mismo tamaño, utilizando registros de longitud variable esta precondición no está presente.
* Para insertar un elemento no basta con disponer de lugar; es necesario, además, que el lugar sea del tamaño suficiente.
* La filosofía para administración del lugar disponible se plantea similar a la de los registros de longitud fija. Se genera una lista invertida donde a partir de un registro cabecera se dispone de las direcciones libres dentro del archivo. Pero Ahora es necesario indicar la cantidad de bytes disponibles en cada caso para su reutilización. Existen tres formas para seleccionar el espacio:
* **Primer ajuste:** consiste en seleccionar el primer espacio disponible donde quepa el registro a insertar. (Rápido).
* **Mejor ajuste:** consiste en seleccionar el espacio más adecuado para el registro. Se considera el espacio más adecuado como aquel de menor tamaño donde quepa el registro. (Menor espacio posible desperdiciado).
* **Peor ajuste:** consiste en seleccionar el espacio de mayor tamaño, asignando para el registro solo los bytes necesarios. (Pueden guardarse varios registros en el mismo espacio).
* El desperdicio de espacio cuando se inserta un registro y no ocupa el 100% del lugar disponible se denomina **fragmentación**. Existen dos tipos:
* Se denomina **fragmentación interna** a aquella que se produce cuando a un elemento de dato se le asigna mayor espacio del necesario.
* Los registros de longitud fija tienden a generar fragmentación interna.
* Se asigna tanto espacio como lo necesario de acuerdo con la definición del tipo de dato. Pero este espacio no siempre se condice con lo que realmente utiliza el registro.
* Se denomina **fragmentación externa** al espacio disponible entre dos registros, pero que es demasiado pequeño para poder ser reutilizado.
* Las dos fragmentaciones tienen que ver con la utilización del espacio.
* En el caso de la fragmentación interna, se reserva lugar que no se utiliza
* La fragmentación externa se genera por dejar espacios tan pequeños que no pueden ser utilizados.
* *Fragmentación de las técnicas de recuperación de espacio:*
* Primer y mejor ajuste suelen implementar una variante que genera fragmentación interna. Una vez seleccionado el lugar libre el espacio asignado corresponde a la totalidad de lo disponible. De esa forma, al nuevo registro se le puede asignar más del espacio necesario.
* Por el contrario, la técnica de peor ajuste solo asigna el espacio necesario. Peor ajuste consiste en buscar sobre la lista de registros borrados el espacio disponible de mayor tamaño y, una vez localizado, se asignan solo los bytes necesarios. Así, el resto del espacio libre sigue figurando en la lista de disponibles. De esta forma, es posible que la técnica de peor ajuste genere fragmentación externa dentro del archivo. Es deseable disponer de un algoritmo que se ejecute periódicamente para la recuperación de estos espacios no asignados: *garbage collector.*
* Desde un punto de vista de la fragmentación generada, solo la técnica de peor ajuste mantiene la filosofía de trabajo de longitud variable, donde cada registro ocupa solamente el espacio que necesita.

También es importante el proceso de **modificación** de información de un archivo con registros de longitud variable:

* Cuando se trabaja con archivos que soportan registros de longitud fija, una modificación consiste en **sobrescribir** un registro con el nuevo dato.
* Sin embargo, cuando los archivos soportan registros de longitud variable, surge un nuevo **problema**:
* Modificar un registro existente puede significar que el nuevo registro requiera el mismo espacio en disco, que ocupe menos espacio o que requiera uno de mayor tamaño. Evidentemente, el problema no se genera cuando ambos registros requieren el mismo espacio.
* Se podría decir que, si el nuevo elemento ocupa menos espacio, no se produce una situación problemática dado que el espacio disponible es suficiente, aunque se generaría fragmentación interna.
* El problema surge cuando el nuevo registro ocupa mayor espacio que el anterior. En este caso, no es posible utilizar el mismo espacio físico, y el registro necesita ser reubicado.
* En general, para evitar todo este análisis y para facilitar el algoritmo de modificación sobre archivos con registros de longitud variable, se estila dividir el proceso de modificación en dos etapas:
* En la primera se da de baja al elemento de dato viejo.
* Y en la segunda etapa el nuevo registro es insertado de acuerdo con la política de recuperación de espacio determinada.

Búsqueda de información:

Hay que hacer especial énfasis en las operaciones en disco, dado que son relativamente lentas, a diferencia de las realizadas en memoria principal, que son relativamente rápidas. Por ejemplo, si un acceso a memoria principal representa 5 segundos, un acceso a memoria secundaria representaría 29 días. Por lo tanto, tomamos en cuenta solo el costo de acceso a memoria secundaria para los análisis de *performance*.

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

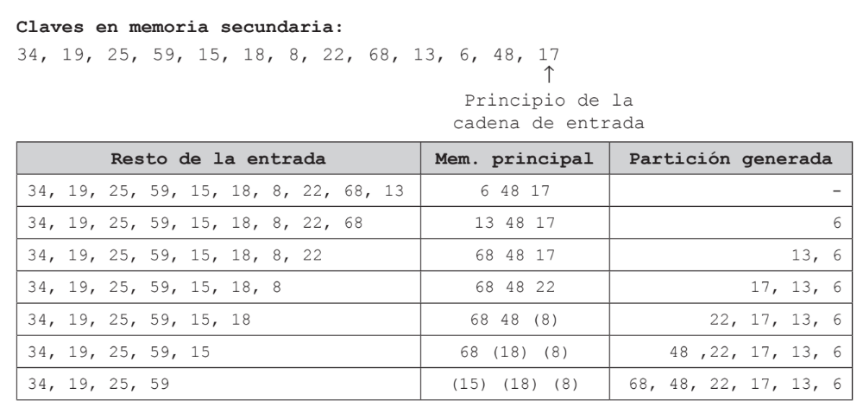
* El caso más simple consiste en disponer de un archivo **serie** (no ordenado). Donde:
* El mejor caso es ubicar el dato deseado en el primer registro (1 lectura).
* El peor caso, en el último registro (N lecturas, siendo N la cantidad de registros).
* El caso **promedio** consiste, por o tanto, en realizar **N/2 lecturas**. Es decir, la performance depende de la cantidad de registros del archivo, siendo entonces de **orden N**.
* Si el archivo está físicamente **ordenado** y el argumento de búsqueda coincide con el criterio de ordenación, la variación con el caso anterior se produce si el dato buscado no se encuentre en dicho archivo. En ese caso, el proceso de búsqueda se detiene cuando el dato es “mayor” al buscado. De ese modo, no existe la necesidad de recorrer todo el archivo. No obstante, la performance sigue siendo de **orden N**, y a la estrategia de búsqueda utilizada en ambos casos se la denomina **secuencial**.
* Si se dispone de un archivo con registros de longitud fija y además físicamente ordenado, es posible mejorar esta performance de acceso si la búsqueda se realiza con el mismo argumento que el utilizado para ordenar este archivo.
* La primera comparación del dato que se pretende localizar es contra el registro medio del archivo, es decir, el que tiene NRR = N/2.
* Si el registro no contiene ese dato, se descarta la mitad mayor o menor, según corresponda, reduciendo el espacio de búsqueda a los N/2 registros restantes (mitad restante).
* Nuevamente, se realiza la comparación, pero con el registro medio de la mitad restante, repitiendo así este proceso hasta reducir el espacio de búsqueda a un registro.
* A esta estrategia se la llama **búsqueda binaria**. Y es de **orden log2(N).** No obstante, se debe considerar el costo adicional de mantener el archivo ordenado, y, si bien el número de accesos disminuye, aún dista bastante de recuperar la información en un acceso a disco.

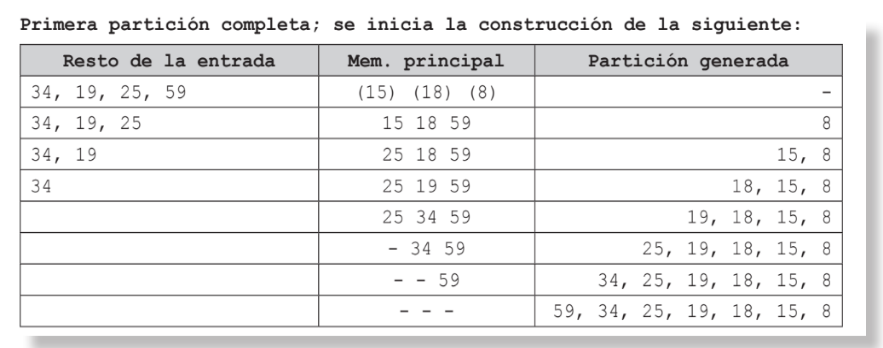
A priori, la búsqueda binaria es una gran opción en base a performance. Pero, ¿y el costo de **ordenar el archivo**?

* Si se aplicara cualquier algoritmo de ordenación de vectores, por ejemplo, para realizar la ordenación de un archivo, habría que tener en cuenta que se debe leer y escribir varias veces cada elemento, con su consiguiente desplazamiento. Estas operaciones realizadas en memoria secundaria serían excesivamente lentas.
* Supóngase que se dispone de un archivo con 1.000 elementos y que el método de ordenación utilizado es el de N pasadas. Este método recorre el archivo buscando el elemento menor y lo posiciona en el primer lugar, luego repite la misma operación buscando el segundo menor y así sucesivamente hasta ordenar todo el archivo.
* En este caso, serán necesarias 1.000 lecturas para determinar el menor elemento, y 2 escrituras que intercambien el primer elemento menor con el elemento que está en la posición 1 del archivo (1.002 accesos).
* El proceso se repite, siempre con un elemento menos. Esto significa que la cantidad de operaciones requeridas será: 1.002 + 1.001 + 1.000 + (…) + 4 operaciones
* Supóngase que el archivo contiene 1.000.000 de registros. El número final de accesos es demasiado elevado.
* Si el archivo a ordenar puede almacenarse en forma completa en memoria RAM, una alternativa muy atrayente es trasladar el archivo completo desde memoria secundaria hasta memoria principal, y luego ordenarlo allí.
* Si bien esto implica leer los datos de memoria secundaria, su acceso es secuencial sin requerir mayores desplazamientos, por lo que su costo no es excesivo.
* Luego, la ordenación efectuada en memoria principal será realizada con alta performance. La operatoria finaliza escribiendo nuevamente el archivo ordenado en memoria secundaria.
* Esta posibilidad constituye la mejor alternativa en cuanto a performance, para ordenar físicamente un archivo, pero solamente puede ser utilizada para archivos pequeños.
* Si el archivo no cabe en RAM, una segunda alternativa constituye transferir a memoria principal de cada registro del archivo solo la clave por la que se desea ordenar, junto con los NRRs, a los registros correspondientes en memoria secundaria. Y así ordenar un archivo de mayor cantidad de datos al ocupar menos RAM en datos “innecesarios”.
* Parecería muy buena opción, ¿para qué transferir cada registro en forma completa si finalmente se considerará la clave como argumento para hacer la ordenación?
* El algoritmo ordena en memoria principal solo las claves. Posteriormente, se debe leer nuevamente cada registro del archivo y escribirlo sobre el archivo ordenado.
* Entonces, se requiere leer el archivo en forma completa dos veces y reescribirse ordenado una vez. Esto implica muchos desplazamientos en memoria secundaria. Por lo tanto, es alto el costo a pagar para poder utilizar la memoria principal como medio de ordenación.
* Aun así, la idea de reorganizar solo las claves no deja de ser interesante.
* La tercera alternativa surge cuando el archivo es realmente grande, de modo tal que las claves no se pueden ordenar en memoria principal porque no caben. Descartando la posibilidad de ordenar el archivo directamente sobre memoria secundaria (por el costo) se llega a la siguiente estrategia, que consiste en:

1. Dividir el archivo en particiones de igual tamaño, de modo tal que cada partición quepa en memoria principal.
2. Transferir las particiones (de a una) a memoria principal. Esto implica realizar lecturas secuenciales sobre memoria secundaria, pero sin ocasionar mayores desplazamientos.
3. Ordenar cada partición en memoria principal y reescribirlas ordenadas en memoria secundaria. También en este caso la escritura es secuencial (estos tres pasos anteriores se denominan **sort interno**).
4. Realizar el merge o fusión de las particiones, generando un nuevo archivo ordenado. Esto implica la lectura secuencial de cada partición nuevamente y la reescritura secuencial del archivo completo.

* Desventajas:
* Para realizar el merge de K formas de K porciones, donde cada partición es del tamaño disponible de memoria principal, se requieren k desplazamientos para leer todos los registros en cada partición individual. Además, la operación de merge requiere k^2 desplazamientos. Por lo tanto, la ordenación evaluada en términos de desplazamientos tiene una performance del orden K^2. Y como k es directamente proporcional a los N registros del archivo, se puede concluir que la ordenación es una operación de **O(N^2)** evaluada en términos de desplazamiento.
* La performance del merge está ligada de manera inversa a la cantidad de particiones generadas. A mayor número de particiones, menor la performance final.
* Si pudiera incrementarse de algún modo el tamaño de las particiones habría menos de estas, y decrecería la cantidad de trabajo requerida durante el merge para el proceso de ordenar un archivo. Pero ¿cómo crear particiones que sean, por ejemplo, dos veces más grandes que la cantidad de registros que se pueden almacenar en RAM? Para suplir esta necesidad se usa una nueva estrategia implementada con un algoritmo conocido como **selección por reemplazo**. Los pasos de este algoritmo serían:

1. Leer desde memoria secundaria tantos registros como quepan en memoria principal.
2. Iniciar una nueva partición.
3. Seleccionar, de los registros disponibles en memoria principal, el registro cuya clave es menor.
4. Transferir el registro elegido a una partición en memoria secundaria.
5. Reemplazar el registro elegido por otro leído desde memoria secundaria. Si la clave de este registro es menor que la clave del registro recientemente transferido a memoria secundaria, este nuevo registro se lo guarda como no disponible.
6. Repetir desde el Paso 3 hasta que todos los registros en memoria principal estén no disponibles.
7. Iniciar una nueva partición activando todos los registros no disponibles en memoria principal y repetir desde el Paso 3 hasta agotar los elementos del archivo a ordenar.



🡪

* El método de **selección** **natural** en relación con selección por reemplazo es una variante que reserva y utiliza memoria secundaria en la cual se insertan los registros no disponibles. De este modo, la generación de una partición finaliza cuando este nuevo espacio reservado está en overfow (completo).
* **Ventajas** de cada método:
* **Sort interno:**
* Produce particiones de igual tamaño.
* Algorítmica simple.
* Las particiones tienen igual tamaño, y cualquier variante del método de merge es más eficiente si todos los archivos que unifica son de igual tamaño.
* **Selección natural** **y selección por reemplazo:**
* Producen particiones con tamaño promedio igual o mayor al doble de la cantidad de registro, que caben en memoria principal.
* **Desventajas** de cada método:
* **Sort interno:**
* Es el más costoso en términos de performance.
* **Selección natural y selección por reemplazo:**
* Tienden a generar muchos registros no disponibles.
* Las particiones no quedan, necesariamente, de igual tamaño.
* En pos de buscar mejoras aún mayores, está la posibilidad de realizar el **merge en más de un paso**. Esto requiere leer 2 veces cada registro; no obstante, se analiza a modo de ejemplo un archivo de 800.000 registros y 1 Mb de memoria principal, donde cada registro ocupa 100 bytes y la clave, 10 bytes).
* En vez de realizar el merge entre 80 particiones, se realizará el merge de 10 conjuntos de 8 particiones cada una. Se generan, así, 10 archivos intermedios, los cuales deberán sufrir otro merge para obtener el archivo original ordenado.
* El primer paso consiste en tomar 8 particiones, asignando un buffer a cada una. Estos 8 buffers de entrada pueden almacenar 1/8 de cada partición, requiriendo 8 desplazamientos cada uno para recuperar una partición. y como se dispone de 8 particiones, serán necesarios (8 \* 8 =) 64 desplazamientos en memoria secundaria para generar el archivo intermedio.
* Este proceso debe repetirse 9 veces más para generar los 10 archivos intermedios (10 \* 64 = 640 desplazamientos).
* Además, cada una de las 10 particiones es 8 veces más grande; esto implica 80 desplazamientos más por partición, y como son 10 particiones, 10 \* 80 = 800 desplazamientos más. En total, se requieren 640 + 800 = 1.440 desplazamientos.
* Si se compara con los 6.400 desplazamientos que serían requeridos en el merge en un paso, la mejora es clara.

¿Qué es la **indización**? 🡪 Hasta ahora la mejor performance para búsqueda de datos es de O(log2(n)), pero puede ser mejorada.

Supóngase el problema de buscar un tema en un libro; independientemente de si este estuviera ordenado por temas, lo lógico sería buscar el material deseado en el índice del libro y acceder directamente a la página que se incluya en dicho índice. Es de notar que en este caso se busca la información en una fuente de datos adicional (el índice), que es de tamaño considerablemente menor, para luego acceder directamente a la información.

Analizando este ejemplo, puede intuirse que, con el uso de una fuente de información organizada adicional, se puede lograr mejorar la performance de acceso a la información deseada.

* Un **índice** es una estructura de datos adicional que permite *agilizar* el acceso a la información almacenada en un archivo. En dicha estructura se almacenan las claves de los registros del archivo, junto a la referencia de acceso a cada registro asociado a la clave. Es necesario que las claves permanezcan ordenadas.
* Esta estructura de datos es *otro* archivo con registros de longitud fija, independientemente de la estructura del archivo original. La característica fundamental de un índice es que posibilita imponer orden en un archivo sin que realmente este se reacomode.
* El índice es un nuevo archivo ordenado, con registros de longitud fija, con la diferencia de que contiene solo un par de datos del archivo original, y, por ende, es mucho más pequeño.
* Una vez disponible el índice, la búsqueda de un dato se realiza primero en este (a partir de la clave), de allí se obtiene la dirección efectiva del archivo de datos y luego se accede directamente a la información buscada.
* Solo se realiza una búsqueda con potencial bajo costo en el índice y luego un acceso directo al archivo de datos, por lo que la cantidad de accesos a memoria secundaria está condicionada por la búsqueda en el índice.

*Operaciones*:

* **Creación de índice primario**
* Al crearse el archivo, se crea también el índice asociado, ambos vacíos, solo con el registro encabezado.
* **Altas en índice primario**
* Consiste simplemente en agregar dicho registro al final del archivo.
* A partir de esta operación, con el NRR o la dirección del primer byte más la clave primaria, se genera un nuevo registro de datos a insertar en forma ordenada en el índice.
* Costo:
* En caso de que el índice se encuentre en RAM, la inserción implica un costo relativamente bajo.
* Si se encuentra en memoria secundaria, el costo es mayor.
* **Modificaciones en índice primario**
* Se considera la posibilidad de cambiar cualquier parte del registro excepto la clave primaria.
* Si el archivo está organizado con registros de longitud fija, el índice no se altera.
* Si el archivo está organizado con registros de longitud variable y el registro modificado no cambia de longitud, el índice no se altera.
* Si el registro modificado agranda su tamaño debe reubicarse. En este caso, esta nueva posición del registro es la que debe quedar asociada en la clave primaria respectiva del índice.
* **Bajas en índice primario**
* Implica borrar la información asociada en el índice primario. Así, se debe borrar física o lógicamente el registro correspondiente en el índice.
* No tiene sentido recuperar el espacio físico en el índice con otro registro que se inserte, debido a que este archivo índice está ordenado y el elemento a insertar no debe alterar este orden.
* **Ventajas del índice primario**
* Al ser de menor tamaño que el archivo asociado y tener registros de longitud fija, posibilita mejorar la performance de búsqueda.
* Se pueden realizar búsquedas binarias, mientras que en el archivo original esto quedaba condicionado a que contuviera registros de longitud fija.

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

El tratamiento de un índice que soporte una clave candidata es similar al definido anteriormente para un índice primario.

**Índices secundarios** 🡪 Cuando se desea buscar, por ejemplo, una canción, no es natural ni intuitivo solicitar un dato por clave primaria, sino por el nombre de la canción o eventualmente por autor, que son atributos mucho más fáciles de recordar.

Sin embargo, estos atributos intuitivos podrían contener valores repetidos en el archivo original. Por este motivo, no es posible pensarlos como parte de una clave primaria. **La clave que soporta valores repetidos se denomina clave secundaria.**

Por lo tanto, es necesario crear otro tipo de índice mediante el cual se pueda acceder a la información de un archivo, pero con datos fáciles de recordar. De esta manera surge el uso de índices secundarios.

* Un **índice secundario** es una estructura adicional que permite relacionar una clave secundaria con una o más claves primarias, dado que varios registros pueden contener la misma clave secundaria. Para acceder a un dato deseado, primero se accede al índice secundario por clave secundaria; allí se obtiene la clave primaria, y se accede con dicha clave al índice primario para obtener finalmente la dirección efectiva del registro que contiene la información buscada.
* ¿Por qué motivo el índice secundario no posee directamente la dirección física de elemento de dato? Porque al tener la dirección física solamente definida para la clave primaria, si el registro cambia de lugar en el archivo, solo debe actualizarse la clave primaria.

*Operaciones:*

* **Creación de índice secundario**
* Al implantarse el archivo de datos, se deben crear todos los índices secundarios asociados, naturalmente vacíos, solo con el registro encabezado.
* Esta operación es similar a la definida para índices primarios.
* **Altas en índice secundario**
* Cualquier alta en el archivo de datos genera una inserción en el índice secundario, que implica reacomodar el archivo en el cual se almacena.
* Esta operación es de bajo costo en términos de performance si el índice puede almacenarse en memoria principal, pero resulta muy costosa si está en memoria secundaria.
* **Modificaciones en índice secundario**
* Dos alternativas:
* *Se produce un cambio en la clave secundaria* 🡪 Se debe reacomodar el índice secundario, con los costos que ello implica.
* *Cambia el resto del registro de datos* (excepto la clave primaria) 🡪 No se genera ningún cambio en el índice secundario.
* **Bajas en índice secundario**
* Cuando se elimina un registro del archivo de datos, implica eliminar la referencia a ese registro del índice primario más todas las referencias en índices secundarios.
* La eliminación a realizarse en el índice secundario, almacenado en un archivo con registros de longitud fija, puede ser física o lógica.
* Una alternativa interesante es borrar solo la referencia en el índice primario.
* El índice primario actúa como una especie de barrera de protección, que aísla a los índices secundarios de este tipo de cambios en el archivo.
* El beneficio principal es que el índice secundario no se altera.
* Sin embargo, existe un costo adicional representado en el espacio ocupado de datos que ya no existen.
* Si el archivo fuese poco volátil, este no sería un gran problema.
* Pero si es volátil, se deberían programar borrados físicos de los índices secundarios.

**Alternativas de organización de índices secundarios 🡪** Sabemos que el índice secundario debe reacomodarse con cadaalta realizada sobre el archivo de datos, aun si se ingresa una clave secundaria ya existente. Esto se debe a que existe un segundo nivel de ordenación por la clave primaria asociada al registro ingresado.

La organización de índices que conocemos hasta el momento consiste enalmacenar la misma clave secundaria en distintos registros, tantascomo ocurrencias haya. Esto implica mayor espacio, generando unamenor posibilidad de que el índice quepa en memoria.

* La primera alternativa de mejora sobre el espacio ocupado por el índice secundario implica **almacenar en un mismo registro todas las ocurrencias de la misma clave secundaria**. Esta nueva organización consiste en que cada registro esté formado por la clave secundaria, más un arreglo de claves primarias correspondientes a dicha clave. Ejemplo:

|  |  |
| --- | --- |
| **A-ha** | SON15 VIR1323 WAR23 |

* En este caso, al agregar un nuevo registro al archivo de datos con la misma clave secundaria, no genera una inserción en el índice secundario. Solo se debe insertar en forma ordenada la clave primaria en el vector de claves primarias respectivo.
* El **problema** inherente a esta alternativa es la elección del tamaño del registro, ya que conviene que este sea de longitud fija. Esto implica decidir el tamaño del vector, por lo que puede haber casos en que resulte insuficiente o que sobre espacio (provocando fragmentación).
* Otra alternativa es pensar en una **lista de claves primarias asociada a cada clave secundaria**. De esta manera, no se debe realizar reserva de espacio y puede existir cualquier número de claves primarias por cada clave secundaria.
* Para acceder a las claves primarias asociadas a las secundarias, se accede al archivo que contiene solo las claves secundarias, y allí se obtiene la dirección (NRR) de acceso a la lista de claves primarias asociadas.
* Dicha lista(denominada *lista invertida*), se encuentra almacenada en otro archivo, el cual se recorre de acuerdo con el camino establecido por el enlace, que indica cuál es el próximo registro asociado (en caso de no existir contendrá un puntero nulo).
* Ambos archivos están organizados con registros de longitud fija.
* Esta opción tiene las siguientes ventajas:
* El único reacomodamiento en el índice se produce cuando se agrega una nueva clave primaria.
* Igualmente, el índice es más pequeño, por lo que dicho reacomodamiento resulta menos costoso.
* Si se agregan o borran datos de una clave secundaria ya existente solo se debe modificar el archivo que contiene la lista. Y en cada caso solo se debe modificar una lista.
* Dado que se generan dos archivos, uno de ellos podría residir en memoria secundaria, liberando así memoria principal.
* No obstante, si existiesen muchos índices secundarios, el mantenimiento de dos archivos por cada índice podría resultar muy costoso.
* Existe otra posibilidad que consiste en disponer de **índices selectivos** que incluyan solo claves asociadas a una parte de la información existente, es decir, aquella información que tenga mayor interés de acceso.
* En el ejemplo, se podría tener un índice con las canciones del grupo musical A-ha solamente.
* De esta forma, el índice incluye solo un subconjunto de datos asociado a los registros del archivo que interesa consultar.
* Obviamente solo se deben considerar las modificaciones al archivo, vinculadas con los datos presentes en el índice.

## Árboles:

Teniendo en cuenta que, sobre una BD, aproximadamente 80% de las operaciones consisten en **búsqueda** de información, es de gran importancia poder recuperar un dato requerido con la menor cantidad de accesos sobre disco. Es así que la organización física que se determine para el archivo resultará de suma importancia para la performance final de cada una de las operaciones de consulta.

El principal problema para poder administrar un índice en disco rígido lo representa la cantidad de accesos necesarios para recuperar la información. El proceso de búsqueda de información es costoso y el mantenimiento de esta ante operaciones de altas, bajas y modificaciones también debe ser considerado como parte del proceso de actualización del archivo. Las estructuras tipo árbol presentan algunas mejoras tanto para la búsqueda como para el mantenimiento del orden de la información.

¿Qué es un **árbol binario**? 🡪 Es una estructura de datos dinámica no lineal, en la cual cada nodo puede tener a lo sumo dos hijos. En general tienen sentido cuando están ordenados (a la izquierda de un elemento se agregan los elementos menores que él, y a la derecha, los mayores).

* La búsqueda en este tipo de estructuras se realiza a partir del nodo raíz y se recorre, explorando hacia los nodos hoja. Se chequea un nodo; si es el deseado, la búsqueda finaliza o, en su defecto, se decide si la búsqueda continua a izquierda o a derecha descartando la mitad de los elementos restantes. Por este motivo, la búsqueda en un árbol binario es de orden **log2(n).**
* Tiene el mismo orden de performance que el algoritmo de búsqueda sobre un archivo de índices ordenado por clave. Entonces ¿Cuál es la utilidad del árbol?
* Una **ventaja** de la organización mediante árboles binarios está dada en la inserción de nuevos elementos.
* Un archivo se desordena cuando se agrega un nuevo dato.
* Pero en un árbol binario, la operatoria resulta más sencilla en términos de complejidad computacional.
* Definición de datos necesaria para generar un archivo de índices utilizando árboles binarios sobre disco:

*Type* ***registroarbolbinario*** *= record*

***elemento\_de\_dato****: tipo\_de\_dato;*

***hijo\_izquierda****,* ***hijo\_derecha****: integer;*

*end;*

***indicebinario*** *= file of registroarbolbinario;*

* El campo *elemento*\_de\_dato contendrá la clave del índice.
* Los campos *hijo\_izquierda* e *hijo\_derecha* definen la dirección del hijo menor y mayor, respectivamente.
* Mientras que sobre memoria RAM estos hijos se representaban mediante punteros, ahora el valor de cada uno es representado mediante un valor entero que indica el NRR del hijo dentro del archivo.
* Pasos para insertar un nuevo elemento:

1. Agregar el nuevo elemento de datos al final del archivo.
2. Buscar al padre de dicho elemento, recorriendo el archivo desde la raíz hasta llegar a un nodo terminal.
3. Actualizar el padre, haciendo referencia a la dirección del nuevo hijo.

* La operación de borrado presenta un análisis similar:
* Para quitar un elemento de un árbol, este debe ser necesariamente un elemento terminal.
* Si no lo fuera, debe intercambiarse el elemento en cuestión con el menor de sus hijos mayores.
* En caso de tener diferentes criterios de ordenación, deberán crearse tantos árboles binarios como criterios haya.
* Pueden **paginarse**:
* El árbol se divide en páginas y cada página contiene un conjunto de nodos, los cuales están ubicados en direcciones físicas cercanas. Estas páginas pueden ser recogidas por **buffers**.
* Así, cuando se transfieren datos, no se accede al disco para transferir unos pocos bytes, sino que se transmite una página completa.
* Al dividir un árbol binario en páginas, es posible realizar búsquedas más rápidas de datos almacenados en memoria secundaria.
* Suponiendo que una página contiene 255 nodos del árbol, la altura relativa decrece rápidamente, permitiendo recuperar un elemento del árbol en muchos menos accesos que los planteados originalmente (log2(n)).
* La performance final de búsqueda resultaría ser del orden de log256(n), es decir, log k+1(n), siendo n la cantidad de claves del archivo y k la cantidad de nodos por página.
* ¿Cómo generar un árbol binario paginado?
* Para que el uso de las páginas tenga sentido, los elementos correspondientes a la misma deberían llegar consecutivos para almacenarse en el mismo sector del disco.
* Dividir el árbol en páginas implica un costo extra necesario para su reacomodamiento y para mantener su balanceo interno.
* Un algoritmo que soporte esta construcción será muy costoso de implementar y luego también en cuanto a performance.
* La solución para este problema consiste en adoptar la idea de manipular más de un registro (la página) y tratar de disponer de algoritmos a bajo costo para construir un árbol balanceado.
* El gran **problema** de los árboles binarios:
* El desempeño de la búsqueda en un árbol binario es bueno porque el árbol se encuentra **balanceado** (se entiende por árbol balanceado a aquel árbol donde todos los nodos hoja se encuentran a igual distancia del nodo raíz).
* Si el árbol se encuentra **desbalanceado**, la performance de búsqueda ya no puede considerarse más en el orden logarítmico. El caso degenerado de un árbol binario transforma en una estructura tipo lista, y la performance de búsqueda decae hasta orden **lineal**.
* La correcta elección de la raíz del árbol determinará si el mismo permanecerá balanceado o no.
* No obstante, cuando se genera un archivo que implanta un índice de búsqueda, es imposible a priori determinar cuál es la mejor raíz, dado que dependerá de los elementos de datos que se inserten.
* Para solucionarlo están los **árboles AVL**

**Árboles AVL** 🡪 árboles balanceados en altura donde el máximo desbalanceo posible es uno. Al insertar la primera clave que desbalancea el árbol, este árbol aún se mantiene balanceado como AVL (1). Sin embargo, cuando se intenta insertar otra clave hija de la previamente ingresada, se viola el precepto establecido. Se debe llevar a cabo un **rebalanceo** de este árbol.

Si bien este algoritmo es relativamente sencillo de implementar, los **costos** computacionales de acceso a disco **aumentan** considerablemente. Entonces:

* Se dispone de una estructura capaz de mantener el balanceo acotado
* *Pero* asumiendo mayores costos en las operaciones de inserción y borrado.

Por lo tanto, los árboles binarios y los AVL **no representan una solución viable** para los índices del archivo de datos.

**Árboles multicamino** 🡪 estructura de datos en la cual cada nodo puede contener **k elementos y k+1 hijos**. El **orden** de un árbol multicamino es la máxima cantidad de descendientes posibles de un nodo (en un árbol binario sería 2, en uno ternario 3 y en un multicamino será m).

Estos árboles siguen manteniendo el problema del desbalance, por lo que, a partir de estos árboles multicamino, surgen los **árboles balanceados**.

**Árboles B (balanceados)** 🡪 son árboles multicamino con una construcción especial que permite mantenerlos balanceados a bajo costo. Sus propiedades son:

* Cada nodo del árbol puede contener, como **máximo**, **M** descendientes y **M-1** elementos.
* La raíz no posee descendientes directos o tiene al menos dos.
* Un nodo con x descendientes directos contiene x-1 elementos.
* Los nodos terminales (hojas) tienen, como mínimo, [M/2] – 1 elementos, y como máximo, M-1 elementos.
* Los nodos que no son terminales ni raíz tienen, como mínimo, [M/2] elementos.
* Todos los nodos terminales se encuentran al **mismo nivel**. De aquí el nombre de balanceados.

**Definición** de la estructura:

*Const* ***orden*** *= 255;*

*Type* ***reg\_arbol\_b*** *= record;*

***Hijos****: array [0 .. orden] of integer;* //nótese que el orden realmente es 256

***Claves****: array [1 .. orden] of tipo\_de\_dato;*

***Nro\_registros****: integer;*

*End;*

Operaciones:

* **Creación de árboles B:**
* Para poder lograr que todas las hojas estén a la misma distancia de la raíz, no es posible concebir un árbol B de la misma forma que un árbol “tradicional”. Cuando no se dispone de suficiente espacio y para preservar el principio de altura constante, es la raíz la que debe alejarse de los nodos terminales.
* Los elementos a insertar comenzarán siempre el proceso de inserción **a partir del nodo raíz**.
* Los elementos deben insertarse siempre en nodos terminales. Por lo cual, si la raíz no es el nodo terminal, debe recuperar nodos desde memoria secundaria hasta encontrar el lugar adecuado.
* Si el nodo tiene lugar, se procederá a insertarlos en este teniendo en cuenta que los elementos de datos deben quedar ordenados dentro de él.
* Véase que este proceso **no resulta costoso**, dado que el registro que contiene al nodo está almacenado en memoria RAM.
* Si no, la llegada de un nuevo elemento provocará un **overflow** en el nodo. Este overflow significa que en el nodo no hay capacidad disponible para almacenar un nuevo elemento de datos. Para tratarlo, el proceso es el siguiente:

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

* El overflow se trata de igual manera en todos los nodos, incluyendo la raíz.
* **Búsqueda en árboles B:**
* El primer paso de la inserción consiste en localizar el nodo que debería contener al nuevo elemento. El proceso de búsqueda realiza la misma operación 🡪 Comenzando desde el nodo raíz, se procede a buscar el elemento en cuestión.
* En caso de encontrarlo en dicho nodo, se retorna una condición de éxito (esto implica retornar la dirección física en memoria secundaria, asociada al registro que contiene la clave encontrada).
* Si no se encuentra, se empieza a buscar en el nodo inmediato siguiente que debería contener al elemento, procediendo de esta manera hasta encontrar el dato buscado, o hasta encontrar un nodo sin hijos que no lo incluya.
* El proceso algorítmico de búsqueda de un elemento en un árbol B no difiere demasiado del mismo proceso en un árbol binario. Comienza la búsqueda en el nodo raíz y se va bifurcando hacia los nodos terminales en la medida en que el elemento no sea localizado.
* La **eficiencia** de búsqueda en un árbol B consiste en contar los accesos al archivo de datos que se requieren para localizar un elemento o para determinar que el elemento no se encuentra.

El resultado es un valor acotado en el rango entero [1, H], siendo H la altura del árbol.

* Si el elemento se encuentra ubicado en el nodo raíz, la cantidad de accesos requeridos es 1.
* En caso de localizar al elemento en un nodo terminal (o que el elemento no se encuentre), serán requeridos H accesos.
* **Eficiencia de inserciones en árboles B:**
* Dado que los elementos solo pueden insertarse en nodos terminales, siempre serán necesarias **H** (altura) **lecturas** para encontrar el nodo dónde almacenar el elemento.
* Pueden surgir dos alternativas:
* **Si el nodo dispone de lugar**, la inserción del nuevo elemento no produce overflow y solo será necesaria una escritura en el nodo terminal con el nuevo elemento. Se da por finalizado el proceso de alta.
* **Si no, se produce overflow** y debe tratarse. **El peor caso** es aquel donde el overflow se propaga hacia la raíz, **generando dos escrituras por nivel** y **haciendo aumentar** en uno **el nivel del árbol**.
* Entonces, la performance de la inserción en un árbol B está compuesta por:
* H lecturas | 1 escritura (**mejor** **caso**)
* H lecturas | (2 \* H) + 1 escrituras (**peor** **caso**)
* Estudios empíricos realizados han demostrado que:
* Generando árboles B de orden M = 10, la cantidad de overflows es de 25% aproximadamente.
* Y si el orden se eleva a M = 100, la cantidad de overflows se reduce a 2%.
* 98 de cada 100 inserciones se tratan por el mejor caso
* Solamente dos no están dentro de esta consideración, lo que no significa necesariamente que se trate del peor caso.

Por lo tanto, la inserción de nuevos elementos en el árbol B requiere un número considerablemente bajo de operaciones de entrada-salida para lograr mantener el orden en la estructura.

* **Eliminación en árboles B:**
* Consideración heredada de los árboles binarios 🡪 **Para poder borrar un elemento, el mismo debe estar localizado en un nodo terminal**.
* Por lo tanto, si se desea eliminar un dato de un nodo interno, el mismo deberá ser **intercambiado** por otro elemento residente en un nodo terminal, sin que esto afecte el recorrido del árbol generado (para hacerlo, se intercambia con la menor de las claves mayores). Hecho esto **se elimina del árbol.**
* Si al nodo que se le quita un elemento deja de cumplir la condición de contener al menos [M/2]-1 elementos se produce un **underflow.** En estos casos existen dos alternativas:
* **Redistribución** 🡪 plantea utilizar algún nodo adyacente hermano del nodo conflictivo, permitiendo que dicho nodo adyacente hermano ceda elementos al nodo que presenta insuficiencia. Así **se redistribuyen los elementos entre los dos hijos y el padre** (que también tiene que redistribuirse ya que el elemento separador no sirve más, dado que se pasaron claves entre nodos).
* Una **ventaja** que presenta la redistribución es que **la cantidad de nodos no se ve afectada**. Entonces, con solo cambiar de posición elementos entre tres nodos.
* No obstante, **la redistribución no siempre es posible**. Para redistribuir se debe disponer de un nodo adyacente hermano con suficientes elementos para compartir, si no quedará en insuficiencia este otro nodo.
* En el caso de que no se pueda redistribuir, la única operación viable resulta la **concatenación** o **fusión**, es decir, tomar los elementos del nodo A, juntarlos con los elementos del nodo B y traer a ese nuevo nodo el elemento del padre, dado que al juntar A y B no será necesario utilizar un separador.
* La operación de concatenación **puede propagar cambios** a lo largo del árbol. Puede ocurrir que el nodo padre genere insuficiencia. Este caso deberá ser resuelto de la misma manera que cualquier otro underflow.
* Si hubiera insuficiencia en el nodo padre, se procederá a redistribuir con algún adyacente hermano de este.
* Si no se pudiera realizar la operación de redistribución, se aplicará la operación de concatenación.
* Así, **la concatenación puede propagarse hacia arriba** en el árbol B, obligando a modificar la raíz, **disminuyendo en uno la cantidad de niveles** del árbol.
* **Eficiencia de eliminación en árboles B:**
* El **mejor** **caso** es cuando se intenta eliminar un elemento que está en un nodo terminal y cuyo borrado no genera insuficiencia. Entonces serán necesarias **H (niveles del árbol) lecturas y una sola escritura**.
* El **peor** **caso** quedará representado cuando la operación de borrado necesite concatenar, lo cual implica leer un nodo adyacente hermano. **Por cada nivel** que se deba concatenar **habrá dos lecturas**, salvo en el nodo raíz, el cual carece de hermanos. **La cantidad de escrituras se limita a una por** **nivel**.
* Resumiendo:
* H lecturas | 1 escritura (**mejor** **caso**)
* 2 \* H -1 lecturas | H - 1 escrituras (**peor** **caso**)
* **Modificación en árboles B:**
* La opción más simple para tratar un caso de modificación es proceder tomando un cambio como una **baja del elemento anterior y un alta del nuevo elemento**.
* Esta consideración no requiere mayor análisis de operatividad y tiene como resultado, en el análisis de performance, el obtenido de sumar una operación de baja seguida de una de alta.
* En el peor caso, dicha situación se mantiene acotada en el tiempo de respuesta.
* **Conclusiones**:
* Los árboles balanceados representan una **buena solución** como estructura de datos, para implementar el manejo de índices asociados a archivos de datos.
* Cualquier operación se realiza en **términos aceptables de performance**. Puede decirse que este tipo de estructura representa una **solución viable**.
* Se debe tener siempre en cuenta que las estructuras de árboles B serán utilizadas para administrar los índices asociados a claves primarias, candidatas o secundarias. El archivo de datos original se plantea como un archivo serie donde cada elemento se inserta siempre al final.
* Tanto estos índices como aquellos que almacenan claves candidatas **referencian a la clave primaria**, en lugar de referenciar directamente a la posición física en el archivo de datos. Esto se debe a cuestiones de performance.
* En caso de modificar la posición física de un registro, solo debe modificarse un índice, el correspondiente a la clave primaria.
* De modificarse la clave primaria, deben modificarse el resto de los índices, pero al diseñar un modelo de datos, en general las claves primarias se definen bajo la pauta de “nunca serán modificadas”.

**Árboles B\*** 🡪 Representan una *variante* sobre los árboles B. La consideración efectuada para avanzar sobre esa estructura parte del análisis realizado en el proceso de *baja* de información:

* El tratamiento de una saturación en un árbol B genera una sola operación, la división.
* El tratamiento de una insuficiencia genera dos operaciones, redistribución y/o concatenación.
* Puede notarse que división y concatenación son operaciones **opuestas**.
* Entonces, el proceso de tratamiento de saturaciones en un árbol B no plantea una operación equivalente a la redistribución en el proceso de baja.

Precisamente, la algorítmica de estos nuevos árboles define una **alternativa** para los casos de **overflow**:

* Antes de dividir y generar nuevos nodos se **redistribuye también ante una saturación**.
* Esta acción **demorará la generación de nuevos nodos.**
* Por ende, tendrá el efecto de **aumentar en forma más lenta la cantidad de niveles** del árbol.
* Si la cantidad de niveles del árbol crece más lentamente, **la performance final de la estructura es mejor**.
* Si se aplica el concepto de redistribuir, cuando un nodo se completa, reubica sus elementos utilizando un nodo adyacente hermano.
* Cuando no sea posible esta redistribución, se estará ante una situación donde tanto el nodo que genera overfow como su adyacente hermano están completos.
* Esto permite dividir partiendo de dos nodos completos y **generando tres nodos completos en dos terceras partes** (2/3), a diferencia de nodos con solo la mitad de los elementos como en árboles B.

**Propiedades**:

* Cada nodo del árbol puede contener, como **máximo, M descendientes y M-1 elementos**.
* La raíz no posee descendientes o tiene al menos dos.
* Un nodo con x descendientes contiene x-1 elementos.
* Los nodos terminales tienen, como mínimo, [(2M - 1) / 3] -1 elementos, y como máximo, M-1 elementos.
* Los nodos que no son terminales ni raíz tienen, como mínimo, [(2M - 1) / 3] descendientes.
* Todos los nodos terminales se encuentran al mismo nivel.

En base a estas propiedades, se puede observar que cuando el árbol B\* tiene un solo nivel (la raíz) y este nivel se completa, no está disponible la posibilidad de redistribuir, dado que la raíz no tiene hermanos. Entonces, la propiedad que indica la mínima cantidad de elementos tiene una **excepción de tratamiento**, al dividir el nodo raíz y generar dos hijos. En este caso, los dos nodos hijos generados solamente completarán su espacio a la mitad. Esta situación se considera como la única excepción admitida para el tratamiento de las propiedades de árboles B\*.

Operaciones:

* **Búsqueda en árboles B\*** 🡪 Es similar a la usada en árboles B. La naturaleza de ambos árboles para localizar un elemento no presenta diferencias. La búsqueda comienza en el nodo raíz y se avanza hasta encontrar el elemento, o hasta llegar a un nodo terminal y no poder continuar con el proceso.
* **Bajas en árboles B\*** 🡪 Resulta nuevamente similar, tanto para los casos donde no se genera underflow como para aquellos en los que sí se genera insuficiencia, donde:
* La primera opción consiste en redistribuir.
* Si no, se debe concatenar, basados en los principios definidos en árboles B (aunque existen algunas variantes).
* **Inserción en árboles B**\* 🡪 puede ser regulado de acuerdo con tres políticas básicas, que determinan el hermano adyacente a tener en cuenta:
* **Política de un lado**:
* El nodo adyacente hermano utilizado será **uno solo**, definiendo la política de izquierda o de derecha.
* En caso de completar un nodo, **intenta** **redistribuir** con el hermano indicado.
* En caso de no ser posible porque el hermano también está completo, tanto el nodo que genera overfow como dicho hermano **son divididos en tres nodos** dos tercios llenos.
* **Política de un lado u otro lado**:
* En caso de producirse una saturación en un nodo, se intenta primero redistribuir con un adyacente hermano.
* De no ser posible la redistribución, **se intenta con el otro** adyacente hermano.
* Si nuevamente la redistribución no es posible, la alternativa es **dividir de dos nodos** **llenos a tres nodos** dos tercios llenos.
* El hermano elegido lo es siempre según diga la política (izquierda o derecha y viceversa).
* **Política de un lado y otro lado**:
* La forma de trabajo es **similar** al caso anterior. Primero, redistribuir hacia un lado y, si no es posible, con el otro hermano.
* La diferencia aparece cuando los tres nodos están completos. Aquí **se toman los tres nodos y se generan cuatro nodos** con tres cuartas partes completas cada uno.
* Si bien esta alternativa completa más cada nodo del árbol y **genera árboles de menor altura**, necesita de **mayor número de operaciones** de entrada-salida sobre disco para poder ser implementada.
* **Eficiencia de inserción en árboles B\*:**
* La performance resultante **dependerá** **de cada política**.
* Ante la ocurrencia de overfow, **como mínimo cada una de las políticas requiere dos lecturas** (el nodo que se satura y un adyacente hermano) y tres escrituras (cada nodo hijo y el padre).
* No se está contabilizando la lectura en el nodo padre, ya que una inserción se realiza sobre nodos terminales y para acceder a los mismos necesariamente se hace a través del padre.
* En caso de necesitar realizar una división, **la política de un lado necesita a su vez realizar cuatro escrituras** (el nodo padre, los dos nodos que estaban completos y el nuevo que se genera).
* Es la **misma situación generada por la política de un lado o el otro lado**, dado que en este caso se divide nuevamente en tres nodos.
* Por último, **la política de un lado y el otro lado genera cinco escrituras**, dado que, además de involucrar a los tres nodos terminales, se deberán tener en cuenta el nodo padre y el nuevo generado.

**Árboles B virtuales en buffer** 🡪 Un árbol B que administre un índice de acceso muy frecuente significará para el SO que el buffer que contenga el nodo raíz sea muy utilizado. Entonces, el SO tenderá a mantener dicho buffer en memoria, disminuyendo en uno el número de accesos a disco, requerido para recuperar la información.

En general, empíricamente es posible afirmar que, si se dispone de un árbol de tres niveles, la primera búsqueda de un elemento demandará a lo sumo tres accesos a disco; la segunda búsqueda demandará a lo sumo dos accesos a disco (en este caso, la raíz ya se mantiene en memoria principal). Ahora, buscar cinco elementos requerirá en promedio 1.71 accesos al disco, disminuyendo la cantidad de accesos requeridos a medida que se intente localizar mayor información.

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

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

Supóngase que se dispone de un archivo con N registros que fueron organizados físicamente por orden de llegada, pero que se creó una estructura de índice utilizando un árbol B o B\* que permite la localización rápida de la información.

* Si se desea un tratamiento secuencial de los elementos del archivo **utilizando el orden de la llave**, la única forma de extraerlos en orden es utilizando el índice. Para recuperar los N registros en orden, es necesario **recorrer todos los nodos del árbol** cada vez a través de los punteros definidos. Esto resulta en un algoritmo **mucho** **menos eficiente** que tener la misma estructura ordenada físicamente.
* Por otro lado, si se plantea la alternativa de **mantener el archivo físicamente ordenado**, se soluciona el problema de acceso anterior, pero resulta inaceptable, al momento de acceder para recuperar un dato, realizar una inserción o una eliminación.

Estos problemas necesitan una solución alternativa que permita compatibilizar ambos accesos. *Los árboles B+*.

**Árboles B+** 🡪 Son una modificación de los árboles B. Incorporan las características discutidas para estos y además el tratamiento secuencial ordenado del archivo. Así, se podrán realizar búsquedas aleatorias rápidas de información, en conjunto con acceso secuencial eficiente.

**Propiedades**:

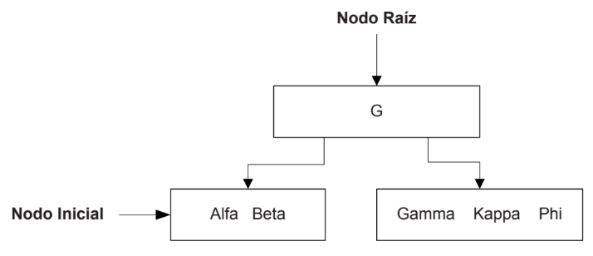
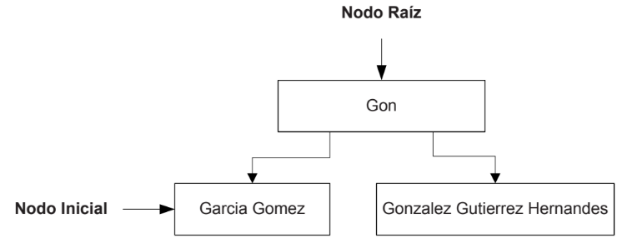
* Cada nodo del árbol puede contener, como máximo, M descendientes y M-1 elementos.
* La raíz no posee descendientes o tiene al menos dos.
* Un nodo con x descendientes contiene x-1 elementos.
* Los nodos terminales tienen, como mínimo, ([M / 2] – 1) elementos, y como máximo, M-1 elementos.
* Los nodos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

**Creación del árbol B+:**

1. Al comenzar, el nodo raíz, actúa tanto como punto de partida para búsquedas como para acceso secuencial.
2. En el momento en que el nodo se satura, se produce una división. Se genera un nuevo nodo terminal donde se redistribuyen los elementos.
3. Ahora se dispone de tres nodos, los nodos terminales contendrán todos los elementos y el nodo raíz contendrá el elemento que actúa como separador. La clave utilizada como separador es la misma que está contenida en el nodo terminal, es decir, **se utiliza una copia del elemento y no el elemento en sí**.

* El proceso de creación del árbol B+ **sigue los lineamientos para árboles B**.
* Los elementos siempre se insertan en nodos terminales.
* Si se produce una saturación, el nodo se divide y se **promociona una copia** (aquí está la diferencia) del menor de los elementos mayores, hacia el nodo padre.
* Si el padre no tiene espacio para contenerlo, se dividirá nuevamente.
* En caso de dividir un nodo no terminal, se debe promocionar hacia el padre el elemento en sí y no una copia del mismo, es decir, **solo ante la división de un nodo terminal se debe promocionar una copia**.
* Para borrar un elemento de un árbol B+, siempre se borra de un nodo terminal.
* Si hubiese una copia de ese elemento en un nodo no terminal, esta copia se mantendría porque sigue actuando como separador.
* A menos que se produzca underflow, dónde si será necesario modificarla porque las claves habrán cambiado de nodo.

Particularmente, si bien siguen los mismos preceptos, las operaciones de inserción y eliminación en un árbol B+ **pueden** (no siempre) **requerir más tiempo** que en un árbol B, debido a la promoción de copias.

**Árbol B+ de prefijos simples** 🡪 es un árbol B+ donde **los separadores están representados por la mínima expresión** posible de la clave, que permita decidir si la búsqueda se realiza hacia la izquierda o hacia la derecha. Esto se hace intentando **aprovechar mejor el espacio físico.** Ejemplo:

Alcanza con la letra G como separador 🡪 Es necesario que el separador sea Gon 🡪

Conclusiones de los árboles balanceados:

* Son estructuras **muy poderosas y flexibles** para la administración de índices asociados a archivos de datos, con un **nivel de performance muy interesante**.
* Sin embargo, **no deben considerarse como la única solución** posible a todos los problemas.
* La mejor estructura para un archivo va a depender del archivo en sí y del propósito de uso de dicho archivo.
* Es decir, si el archivo de datos se limita a unos pocos registros que pueden estar contenidos en un nodo, utilizar un árbol no es necesario, dado que agrega un nivel de indirección por cada índice.
* Tampoco es necesario si el archivo no requiere consultarse con frecuencia.
* Hay metodologías para organizar archivos que presentan un mejor tiempo de respuesta. Entonces, ante determinadas necesidades de performance, puede ocurrir que los árboles balanceados no sean la mejor opción.
* Sin embargo, los árboles B tienen una gran aplicabilidad en situaciones en las que resulta necesario acceder a un archivo tanto para búsquedas aleatorias eficientes como para acceso secuencial.
* Las **características** **compartidas** entre los árboles de la familia B son las siguientes:
* *Manejo de bloques o nodos de trabajo* 🡪 Esto facilita las operaciones de entrada-salida y mantiene acotado su número de accesos.
* Todos los nodos terminales están a la misma distancia del nodo raíz, es decir, estos árboles *están balanceados*.
* Crecen de abajo hacia arriba, a diferencia de los árboles binarios comunes.
* Son “bajos” y “anchos”, a diferencia de los binarios, que son “altos” y “delgados”.
* Es posible implementar algoritmos que manipulen registros de longitud variable para mejorar aún más la performance y el uso del espacio.
* Los **árboles B\* mejoran la eficiencia de los árboles B**, aumentando la cantidad mínima de elementos en cada nodo y así **ayudando a disminuir la altura** final del árbol. El costo a pagar es que las operaciones de inserción son un poco más lentas.
* En los **árboles B+**, toda la información de las claves está contenida en los nodos terminales. Los nodos no terminales actúan como separadores, y poseen una copia de los elementos contenidos en los nodos terminales.

## Hashing o Dispersión:

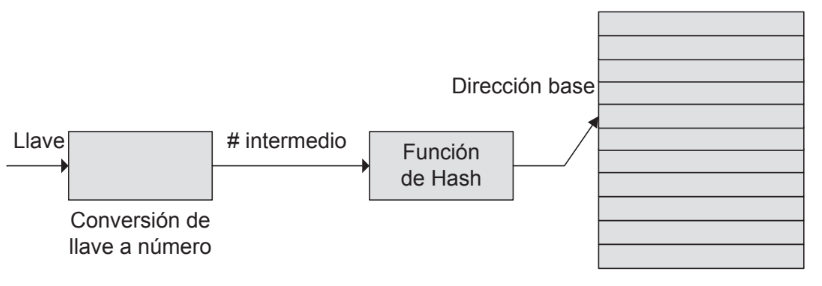
Un **archivo directo**, o con acceso directo, es un archivo en el cual **cualquier registro puede ser accedido directamente**, sin acceder antes a sus predecesores. En un archivo serie, en cambio, un registro está disponible solo cuando el registro predecesor fue procesado.

Un archivo directo podría pensarse como un arreglo de registros, en tanto que un archivo serie puede pensarse como una lista de registros.

La forma en la que se almacenen los archivos directos debe permitir una rápida recuperación de la información contenida en el este. **El mecanismo que trata de asegurar una recuperación rápida de registros** **lleva el nombre de dispersión o** **hashing**. Este método mejora la eficiencia obtenida con árboles balanceados, asegurando, en promedio, un acceso para recuperar la información.

Hashing tiene tres **definiciones**:

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

**Atributos** de la técnica de dispersión:

* **No se requiere almacenamiento adicional** 🡪 Esto significa que cuando se elige la opción de dispersión como método de organización de archivos, es el archivo de datos el que resulta disperso. Por lo que **no es necesario tener una estructura auxiliar** que actúe como soporte para poder acceder rápidamente a la información.
* **Facilita la inserción y eliminación rápida de registros en el archivo** 🡪 el proceso de inserción o borrado de información se realiza de una manera más **eficiente** en términos de accesos a disco.
* **Localiza registros dentro del archivo con un solo acceso a disco** 🡪 Como corolario del punto anterior, otra ventaja del método de dispersión consiste en ubicar cada elemento de datos, en promedio, con un acceso a disco. Si bien algún registro puede requerir más de un solo acceso.

Sin embargo, el método de dispersión presenta **limitaciones**:

* Pueden ocurrir situaciones en las que el método no sea aplicable.
* Otras donde, a partir de su aplicación, no es posible lograr otras características que podrían ser deseadas para el archivo de datos. Estas limitaciones son las siguientes:
* **No es posible aplicarlo en archivos con registros de longitud variable** 🡪 Esto se debe a que cada dirección física obtenida debe tener capacidad para almacenar un registro de tamaño conocido.
* **No es posible obtener un orden lógico de los datos** 🡪 Utilizando índices como metodología de acceso a datos, no solo la búsqueda es eficiente, sino que además presenta la característica de mantener los registros ordenados. Bajo la técnica de hashing, los registros son “esparcidos” en el archivo de datos.
* **No es posible tratar con claves duplicadas** 🡪 Así, no es aplicable la función de hash sobre una clave secundaria. Ya que esta podría repetirse y dos registros diferentes con la misma clave secundaria, aplicando la función de hash, tendrían como resultado la misma dirección de memoria.

Hay dos **tipos de dispersión**:

* **Hashing con espacio de direccionamiento estático** 🡪 El espacio disponible para dispersar los registros de un archivo de datos *está fijado* previamente.
* **Hashing con espacio de direccionamiento dinámico** 🡪 El espacio disponible para dispersar los registros de un archivo de datos *aumenta o disminuye* en función de las necesidades de espacio que en cada momento tiene el archivo.

La función de hash da como resultado un valor intermedio, que será utilizado para obtener una dirección física posible para el archivo. Estas direcciones físicas no están establecidas a priori y son generadas de manera dinámica.

**Parámetros de la dispersión**

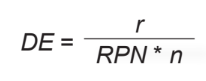
El método de dispersión, cuando utiliza espacio de direccionamiento estático, presenta cuatro parámetros esenciales que definen su comportamiento:

* **Función de hash** 🡪 caja negra que transforma un valor (llave primaria de un registro) en otro dentro de un determinado rango, que se utiliza como dirección física de acceso para insertar/buscar un registro en un archivo.
* Como la función de hash podría retornar cualquier valor, es necesario mapear dicho valor dentro del rango de valores posibles antes de que efectivamente sea retornado.
* La función de hash aplicada sobre cualquier llave resolverá rápidamente la dirección física donde almacenar el registro asociado a la clave.
* Sin embargo, esta sencillez tiene asociado un inconveniente 🡪 Si se ingresan dos claves diferentes: DACA y CADA. Aplicando la función de hash sumando los valores de cada carácter, *el valor resultante será el mismo*; por lo tanto, los registros asociados a ambas claves se deberían almacenar en el mismo lugar de memoria. Esta situación **genera una colisión** entre ambos registros. En este caso, las llaves mencionadas se denominan **sinónimos**.
* Las **colisiones o desbordes ocasionan problemas**. *No es posible almacenar dos registros en el mismo espacio* *físico*. Así, es necesario encontrar una solución a este problema. Las alternativas son dos:
* **Elegir un algoritmo de dispersión perfecto**, que no genere colisiones. Este tipo de algoritmo debe asegurar que dadas dos claves diferentes siempre se obtendrán dos direcciones físicas diferentes.

Dado que obtener este tipo de funciones resulta extremadamente difícil, intentar generar estos algoritmos no suele ser una opción válida.

* **Minimizar el número de colisiones** a una cantidad aceptable, y de esta manera tratar dichas colisiones como una condición excepcional.

Existen diferentes **modos de reducir el número de colisiones**; las alternativas disponibles son:

* **Distribuir los registros de la forma más aleatoria posible**.
* **Utilizar más espacio de disco** 🡪 Si se intentan distribuir 10 registros en 10 lugares, las posibilidades de colisión son altas al haber un solo lugar por registro. Si se aumenta el tamaño del archivo, la probabilidad de colisión disminuye a costa de desperdiciar espacio.
* **Ubicar o almacenar más de un registro por cada dirección física en el archivo** 🡪 Cada dirección obtenida por la función de hash es la dirección de un sector del disco donde es posible almacenar más de un registro, para sí poder guardar dos claves en situación de colisión juntas.
* **Tamaño de cada nodo de almacenamiento** 🡪 Queda determinado por la posibilidad de transferencia de información en cada operación de entrada/salida desde RAM hacia disco, y viceversa. Mismo análisis que con árboles balanceados.
* **Densidad de empaquetamiento** 🡪 Relación entre el espacio disponible para el archivo de datos y la cantidad de registros que integran dicho archivo.
* El espacio disponible se define como la cantidad de nodos direccionales (**n**) por la función de hash, y la cantidad de registros (**r**) que cada nodo puede almacenar, Registros por Nodo (**RPN**).
* Si se debieran esparcir 30 registros entre 10 direcciones con capacidad de cinco registros por cada dirección, la DE sería de 0.6 o 60% (30 / 5 \* 10).
* Cuanto mayor sea la DE, mayor será la posibilidad de colisiones, dado que en ese caso se dispone de menos espacio para esparcir registros. Por el contrario, si la DE se mantiene baja, se dispone de mayor espacio para esparcir registros y, por ende, disminuye la probabilidad de colisiones.
* Por otra parte, cuando la DE se mantiene baja, se desperdicia espacio en el disco, dado que se utiliza menor espacio que el reservado, generando fragmentación.
* La DE no es constante. Al inicio, la DE será baja, e irá aumentando a medida que se agreguen registros.
* Métodos de tratamiento de desbordes (overflow) 🡪 Ocurre cuando un registro es direccionado a un nodo que no dispone de capacidad para almacenarlo. Deben realizarse dos acciones (para las cuales existen diferentes métodos) cuando pasa esto:
* Encontrar lugar para el registro en otra dirección.
* Asegurarse de que el registro posteriormente sea encontrado en esa nueva dirección.

**Resolución de colisiones con overflow** 🡪 Aunque la función de hash sea eficiente y aún con DE relativamente baja, es probable que las colisiones produzcan overflow o saturación. Por este motivo, se debe contar con algún método para reubicar a aquellos registros que no pueden ser almacenados en la dirección base obtenida a partir de la función de hash:

* **Saturación progresiva:**
* Consiste en **almacenar el registro en la dirección siguiente más próxima** al nodo donde se produce saturación.
* El proceso de búsqueda de información debe, ahora, sufrir algún cambio:
* Si se busca un dato que fue desplazado, la dirección base que determina la función de hash seguirá siendo la misma.
* Como el elemento no se encuentra en dicho lugar y el nodo está completo, se debe continuar la búsqueda en los nodos subsiguientes hasta encontrar el elemento, o hasta encontrar un nodo que no esté completo.
* El método podría requerir **chequear todas las direcciones** disponibles en el peor caso.
* La **simpleza** del método viene acompañada por su **limitada** **eficiencia**.
* Además, el método necesita indicar que, si una dirección estuvo completa anteriormente, debe ser marcada como dirección ya saturada a fin de no impedir la búsqueda potencial de registros.
* Si no podría detenerse la búsqueda por encontrar una cubeta vacía, donde se produjo una baja en algún otro momento.
* **Saturación progresiva encadenada:**
* En líneas generales, el método funciona igual a su predecesor 🡪 Un elemento que se intenta ubicar en una dirección completa es direccionado a la inmediata siguiente con espacio disponible.
* La diferencia radica en que, una vez localizada la nueva dirección, esta **se enlaza con la dirección base inicial**, generando una cadena de búsqueda de elementos.
* Cabe considerar que el enlace entre nodos sirve tanto para la búsqueda como para la inserción (y hasta bajas) de las claves.
* Si, por ejemplo, se desea insertar una clave a la cual la función de hash le asigna el nodo “50”.
* Al estar este nodo lleno se intenta insertar en el siguiente según el enlace (supongamos nodo 52).
* Nuevamente, al estar completo el nodo 52, se busca en el próximo nodo indicado en el enlace.
* En este caso no está enlazado, y hace que se deba buscar el nodo siguiente más próximo con espacio libre.
* **Doble dispersión:**
* Consiste en disponer de **dos funciones de hash**:
* La primera obtiene a partir de la llave la dirección de base, en la cual el registro será ubicado.
* *De producirse overflow*, se utilizará la segunda función de hash 🡪 Esta segunda función no retorna una dirección, sino que su resultado es un desplazamiento.
* Este desplazamiento se suma a la dirección base obtenida con la primera función, generando así la nueva dirección donde se intentará ubicar al registro.
* En caso de generarse nuevamente overflow, se deberá sumar de manera reiterada el desplazamiento obtenido hasta encontrar una dirección con espacio suficiente.
* Viene a solucionar el problema de que los registros tienden a esparcirse en nodos cercanos, provocando un exceso de *saturación sectorizada.*
* La doble dispersión tiende a esparcir los registros en saturación a lo largo del archivo de datos, pero tiene un **problema**.
* Los registros en overflow tienden a ubicarse “lejos” de sus direcciones de base.
* Esto produce un mayor desplazamiento de la cabeza lectora/grabadora del disco rígido, **aumentando** la latencia entre pistas y, por consiguiente, **el tiempo de respuesta**
* **Área de desbordes por separado:**
* Ante la ocurrencia de overflow, los registros son dispersados en nodos que no se corresponden con su dirección base original.
* Así, a medida que se completa un archivo por dispersión, pueden existir *muchos registros ocupando direcciones que originalmente no les correspondían*.
* Esto **disminuye** **la** **performance** final del método de hashing.
* Para evitar estas situaciones, está el uso del *área de desbordes por separado*. Aquí se distinguen dos tipos de nodos:
* Aquellos direccionables por la función de hash.
* Aquellos de reserva, que solo podrán ser utilizados en caso de saturación pero que no son alcanzables por la función de hash.
* Cuando se produce saturación, el registro es reubicado en la primera dirección disponible dentro del área de desbordes separada y la dirección base original se encadena con la dirección de reserva.

**Hash asistido por tabla:**

El método de hash resultó ser **el más eficiente en términos de recuperación de información**; permite conseguir **un acceso para recuperar un dato** en más de **99.9%** de los casos, cuando la **DE** es **inferior** a **75%**. Asimismo, las operaciones de altas y bajas se comportan con el mismo nivel de eficiencia para los mismos porcentajes.

Sin embargo, *restan aún situaciones en las que puede ser necesario utilizar más de un acceso* para recuperar o almacenar un registro. Si bien las alternativas son eficientes, a medida que se generan situaciones de saturación, el número de accesos requeridos aumenta.

*Existe una alternativa* que sigue utilizando espacio de direccionamiento estático y que, aun así, asegura acceder en un solo acceso a un registro de datos. Esta variante para hash estático se denomina *hash asistido por tabla*, y para poder ser implementada requiere que una de las propiedades del hashing *no sea cumplida* 🡪 **Necesita una estructura adicional**.

* El método utiliza tres funciones de hash:
* **La primera** de ellas **retorna la dirección** física del nodo donde el registro debería residir (**F1H**).
* **La segunda retorna un desplazamiento**, similar al método de doble dispersión, (**F2H**).
* **La tercera retorna una secuencia de bits** que no pueden ser todos unos (**F3H**).
* El método comienza con una tabla con tantas entradas como direcciones de nodos se tengan disponibles. *Cada entrada tendrá todos sus bits en uno.*
* Ante cada inserción, la primera función de hash retorna la dirección del nodo donde se debe almacenar el registro. Si no se produce saturación, el registro es almacenado en ese lugar.
* *Si se produce saturación,* el procedimiento es:

1. Se obtiene el valor de la F3H para todos los registros en el nodo saturado.
2. Se determina la clave que genera el valor mayor.
3. Se escribe el nodo saturado con todos los registros menos el que posee F3H mayor.
4. En la tabla en memoria, para la posición del nodo saturado queda el valor de F3H del registro con mayor valor. Similar a los punteros de saturación progresiva encadenada.
5. Para la clave seleccionada en el paso 2 se obtiene la F2H.
6. Se suma, a la F1H de la clave seleccionada, el desplazamiento determinado por F2H.
7. Se intenta insertar el registro seleccionado en la dirección resultante, en caso de que no haya saturación. Si la hubiera, se procederá nuevamente con los pasos indicados desde 1.

* *Para localizar la clave desplazada*, el proceso es el siguiente:

1. Se genera la F1H para la clave a buscar.
2. Se genera la F3H para la misma clave.
3. Se chequea el contenido de la posición de la tabla indicada por F1H, comparándolo con el resultado de F3H.

* Si el resultado de F3H tiene valor menor que el encontrado en la tabla, se busca la clave en cuestión en la dirección del nodo indicada por F1H y el proceso termina.
* Si la F3H tiene valor mayor o igual que el encontrado en la tabla, el proceso continúa como se indica en el paso 4.

1. Se obtiene el valor de F2H y se suma al valor indicado por F1H, obteniéndose una nueva posición.
2. Se comienza nuevamente desde el Paso 3, utilizando ahora como posición el resultado de F1H más el desplazamiento.

* El método **asegura encontrar la clave buscada en un solo acceso a disco**.
* Pero **el costo asociado presenta dos aristas**:
* La primera tiene que ver con la necesidad de utilizar espacio extra para la administración de la tabla en memoria principal.
* La segunda tiene que ver con la complejidad adicional en caso de una inserción que produzca saturación. En este caso, la cantidad de accesos para terminar el proceso está vinculada con la cantidad de overflows sucesivos que se produzcan.
* La conclusión final de esta variante para hash con espacio de direccionamiento estático es que este método **pondera la búsqueda sobre las otras operaciones**.

**Bajas en hashing asistido con tabla:**

A fines prácticos, se puede establecer el siguiente proceso para dar de baja un registro del archivo:

1. Se localiza el registro de acuerdo con el proceso de búsqueda definido.
2. Si el paso anterior encontró la llave buscada, se reescribe el nodo en cuestión sin el elemento a borrar.

Es de notar la simplicidad del proceso de eliminación. Hay un caso especial 🡪 Supongamos que se borra una clave de un nodo que se encontraba completo.

* Cuando se intente insertar un nuevo elemento en el nodo, habrá lugar.
* Si el nodo ya había producido overflow, la tabla en memoria contiene un valor correspondiente a la F3H de la llave que produjo saturación.
* En ese caso, el nuevo elemento a insertar debe cumplir la siguiente propiedad:

F3H (nuevo elemento) < Tabla (dirección del nodo)

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

**Hashing con espacio de direccionamiento dinámico:**

El *costo de redispersar un archivo que ha quedado pequeño es muy alto*. El tiempo utilizado es importante, y mientras se realiza esta operatoria no es posible que el usuario final de la información acceda al archivo. Por este motivo, cuando se piensa en la creación de un archivo, es importante:

* Analizar la tasa de crecimiento posible del archivo.
* Determinar la cantidad de nodos que optimice el uso de espacio vs. la periodicidad de la redispersión.

Otra alternativa posible es trabajar con archivos que administren el espacio de direccionamiento de manera **dinámica**. Esto es, *no establecer a priori la cantidad de nodos disponibles*, sino que esta **crezca a medida que se insertan nuevos registros**.

Surge así la necesidad de utilizar *hash con espacio de direccionamiento dinámico.* Este hash dispersa las claves en función de las direcciones disponibles en cada momento, y **la cantidad de direcciones puede crecer**, a priori sin límites, en función de las necesidades de cada archivo particular. Hay varias alternativas. Consideramos la de **hash extensible**.

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

* Este método, al igual que el resto de los que trabajan con espacio dinámico, no utiliza el concepto de DE. Dado que el tamaño aumenta dinámicamente según los registros que se insertan.
* El principal **problema** que se tiene con los métodos dinámicos en general y con el hash extensible en particular es que **las direcciones de nodos no están prefijadas a priori**, y por lo tanto **la función de hash no puede retornar una dirección fija**.
* Entonces, se cambia la política de trabajo. Para el método extensible, la función de hash retorna un String de bits. La cantidad de bits que retorna determina la cantidad máxima de direcciones a las que puede acceder el método.
* Por ejemplo, si la función de hash retorna 32 bits, es posible direccionar 2^32 direcciones de nodos diferentes
* **El método necesita**, para su implementación, **de una estructura auxiliar**. Esta estructura es **una tabla** que se administra en memoria principal.
* A diferencia del hash asistido por tabla, esta estructura **contiene la dirección física** de cada nodo.
* Se debe tener en cuenta que **la función de hash no retorna una dirección física, sino una secuencia de bits**.
* Dicha secuencia **permite obtener de la tabla en memoria la dirección física** del nodo para almacenar la llave.
* Además, la tabla **será utilizada posteriormente para recuperar cada registro en un solo acceso** a disco.
* El tratamiento de dispersión con la política de hash extensible comienza con un solo nodo en disco, y una tabla que solamente contiene una dirección, la del único nodo disponible.
* Para realizar inserciones, el método trabaja de la siguiente forma:

1. Se calcula la función de hash de la clave a dispersar y se toman tantos bits como indica el valor asociado a la tabla.

* El valor asociado a la tabla indica la cantidad de bits que es necesario tomar de la función de hash. A partir de que su valor sea mayor o igual que uno se deberá tomar n (siendo n el valor asociado a la tabla), los menos significativos.

1. Puede ocurrir que:

* Si el valor asociado es 0 y no provoca overflow se inserta directamente.
* Si no es 0 el valor, se toman los n bits menos significativos de la clave y se comparan con la dirección de la tabla. Si no genera overflow, se inserta en la dirección que coincida.
* Si se produce overflow:

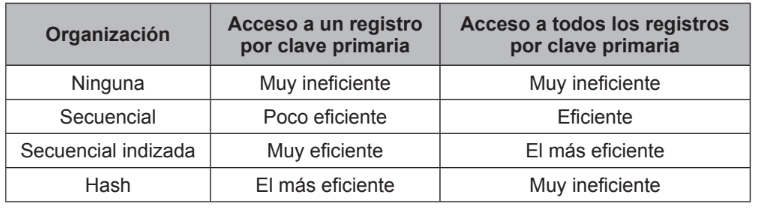
1. Se aumenta en uno el valor asociado al nodo saturado.
2. Se genera un nuevo nodo con el mismo valor asociado al nodo saturado y el valor asociado al nodo se aumenta en 1 para ambos implicados.
3. Se reacomodan las claves según corresponda, comparando las claves con las direcciones de la tabla.
4. Si el valor asociado al nodo es mayor que el asociado a la tabla, significa que la tabla no dispone de entradas suficientes para direccionar al nuevo nodo. Entonces, la cantidad de celdas de la tabla se duplica, y el valor asociado a la tabla se incrementa en uno.

* El proceso de búsqueda **asegura encontrar cada registro en un solo acceso**.

1. Se calcula la secuencia de bits para la llave.
2. Se toman tantos bits de esa llave como indique el valor asociado a la tabla.
3. La dirección del nodo contenida en la celda respectiva debería contener el registro buscado.

* En caso de no encontrar el registro en dicho nodo, el elemento no forma parte del archivo de datos.
* El **costo** que debe asumirse con esta técnica es **mayor procesamiento cuando una inserción genera overflow**.

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



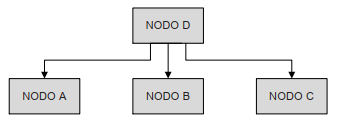
**Conceptos**:

**NRR** 🡪 Número Relativo de Registro

**BD** 🡪 Base de Datos

**Nodos** **Hermanos** 🡪 nodos que tienen el mismo nodo padre.

**Nodos Adyacentes Hermanos o Nodos Hermanos Adyacentes** 🡪 nodos que, siendo hermanos, son además dependientes de punteros consecutivos del padre.

* Los tres nodos hijos A, B y C dependen del mismo padre, D; por ende, son **nodos** **hermanos**.
* el nodo A y el nodo B son apuntados por punteros consecutivos, entonces son **nodos** **adyacentes** **hermanos**.
* Lo mismo ocurre entre B y C.
* Los nodos A y C **solo** **son** **hermanos**, puesto que, al no ser apuntados por punteros consecutivos en el padre, **no** **pueden** **considerarse** **adyacentes**.