**­Fundamentos de Organización de Datos**

Contenido

[**Base de datos** 3](#_Toc139884612)

[**ARCHIVOS** 3](#_Toc139884613)

[**Tipos de almacenamiento:** 3](#_Toc139884614)

[**Definiciones**: 3](#_Toc139884615)

[**Acceso a registros desde archivos:** 4](#_Toc139884616)

[**CLAVES** 5](#_Toc139884617)

[Forma Canoníca: 5](#_Toc139884618)

[**Estudio de performance:** 6](#_Toc139884619)

[**Operaciones sobre archivos:** 6](#_Toc139884620)

[**Bajas:** 6](#_Toc139884621)

[**Altas:** 7](#_Toc139884622)

[**Modificaciones:** 7](#_Toc139884623)

[**Consultas:** 7](#_Toc139884624)

[**Tipo de altas:** 7](#_Toc139884625)

[**Fragmentación de archivos:** 8](#_Toc139884626)

[**Fragmentación interna:** 8](#_Toc139884627)

[**Fragmentación externa:** 8](#_Toc139884628)

[**Estrategias de colocación en registros de LONGITUD VARIABLE:** 8](#_Toc139884629)

[**Eficiencia**: 9](#_Toc139884630)

[**ÍNDICES** 10](#_Toc139884631)

[**Persistencia:** 10](#_Toc139884632)

[**Tipo de indices:** 10](#_Toc139884633)

[**Índices primarios:** 10](#_Toc139884634)

[**Índices secundarios:** 10](#_Toc139884635)

[**ARBOLES** 12](#_Toc139884636)

[**Arbol binario:** 12](#_Toc139884637)

[**Arboles AVL (BA) (arboles balanceados en altura):** 12](#_Toc139884638)

[**Árbol binario paginado:** 12](#_Toc139884639)

[**Arboles multicamino (Arboles generales):** 12](#_Toc139884640)

[**Arbol Balanceado** 12](#_Toc139884641)

[**Arboles B\*** 14](#_Toc139884642)

[**Políticas de remplazo** 14](#_Toc139884643)

[**Arboles B+** 15](#_Toc139884644)

[**Arboles B+ de prefijos simples:** 15](#_Toc139884645)

[**HASHING** 17](#_Toc139884646)

[**Definiciones:** 17](#_Toc139884647)

[**Propiedades:** 17](#_Toc139884648)

[**Función hash:** 17](#_Toc139884649)

[**Tamaño nodos / cubetas:** 18](#_Toc139884650)

[**Densidad de empaquetamiento (DE):** 18](#_Toc139884651)

[**Tratamiento de overflow:** 18](#_Toc139884652)

[**Saturación progresiva:** 18](#_Toc139884653)

[**Saturación progresiva encadenada:** 19](#_Toc139884654)

[**Dispersión doble:** 19](#_Toc139884655)

[**Área de desborde/dispersión separada:** 19](#_Toc139884656)

[**Tipos de dispersión** 19](#_Toc139884657)

[**Estática:** 19](#_Toc139884658)

[**Dinámica:** 19](#_Toc139884659)

# **Base de datos**

Una base de datos es una colección de datos relacionados, estos conjuntos de datos que almacena se llaman archivos.

La base de datos se caracterizan por almacenar los datos coherentes entre sí de manera persistente, es decir que dichos datos van a perdurar más allá de la ejecución del programa.

Las BD son diseñadas, construidas y completadas para un conjunto de usuarios en concreto y con ciertas aplicaciones pre-concebidas, logrando de esta manera almacenar una colección coherente de datos.

# **ARCHIVOS**

Los archivos son una colección de registros/datos, que tienen un aspecto común y que son almacenados de manera persistente en memoria secundaria.

## **Tipos de almacenamiento:**

Almacenamiento primario:   
 Capacidad limitada, mayor costo, más rápida.  
 Velocidad del rango de los NS (nanosegundos).  
 Memoria volátil, es decir no persistente.  
 Ej: RAM.

Almacenamiento secundario:   
 Acceso lento por lo que se busca métodos de acceder lo menor posible  
 Velocidad del rango de los MS (milisegundos)  
 Memoria persistente  
 Ej: HDD / SSD, dispositivos externos (USB, etc.).

## **Definiciones**:

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

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

Buffer:  
 Estructura de información intermedia entre programas y MS (Memoria Secundaria). Permitiendo reducir la cantidad de accesos a MS.  
 Aquí se almacenan temporalmente los datos a guardar / cargar en MS.

**Niveles de archivos:**

Físicos:  
 Cuando los archivos se encuentran en MS.

Lógicos:  
 Cuando los archivos ya fueron cargados al programa.  
 Permite al programa efectuar operaciones sobre el archivo.  
 No se sabe ubicación real del archivo físico.

**Viaje del byte:** El sistema operativo debe actuar para obtener la información. Para ello debe hacer uso del administrador de archivos, el buffer de E/S, el procesador de E/S y el controlador de disco.

Administrador de archivos:  
 Es un conjunto de programas del S.O. que se encarga de gestionar los archivos y dispositivos E/S. Para esta gestion se separa en 2 partes / niveles.

Capas superiores: aspectos lógicos de datos (tabla).  
 Establece si las características del archivo permiten la operación deseada. Es decir determina la operatividad del archivo, quien lo creo, cuando, etc. Determina por ej. Si X usuario puede escribir, leer, etc.

Capas inferiores: aspectos físicos (FAT file-alocation-table).  
 Guarda donde esta almacenada la ubicación (Plato, sector, etc.)

Controlador de disco:  
 Es el encargado de controlar la operación del disco, es decir coloca la pista, sector, etc. Necesarios para poder realizar la operación de transferencia E/S.

## **Acceso a registros desde archivos:**

**Secuencia Físico: (Serie)** Lectura secuencial en orden físico en el que están almacenados.

**Secuencial Indizado (Lógico): (Secuencial)** Lectura secuencial pero de una estructura con cierto ordenamiento lógico.

**Directo: (Directo)** Acceso directo al registro, mediante índice.

**Tipos de archivos:**

**Archivos como secuencia de byte** Trabaja almacenando una secuencia de byte, caracteres, uno después de otro. Difícil de determinar el comienzo y fin de cada palabra.

**Archivos de Texto**  
 Usando por convención el espacio como separador entre datos.

**Archivos Estructurados:**

**Registros:** Longitud fija o variables

**Campos:** Longitud fija o variables

**Archivos Tipados:**

**Archivos de Bytes:**

**Tamaños del campo / registro:** (Operatividad vs Uso de espacio)

**Longitud fija:**(Se usa hoy en día por que sobra espacio) El tamaño del campo es fija y predefinida. Ej: Int, Char, String[20], etc.  
 -Se sabe dónde empieza o termina cada campo.  
 -Se desperdicia espacio, ya que queda espacio reservado pero no usado.  
 -Es apto para realizar una búsqueda binaria / dicotómica.

**Longitud variable:**  
 El campo pasa a usar únicamente lo que necesite.  
 No se sabe dónde empieza o termina cada campo. Por lo que se requiere alguno de estos metodos:  
 Indicador de longitud: Es decir al principio de cada campo se almacena el tamaño de campo, Ej. Cuantos caracteres va a tener.

Delimitador al final: Se establece un carácter especial para marcar fin del campo.

Segundo archivo: Se usa otro archivo que va determinando la longitud de cada campo, por lo que se va obteniendo la longitud desde el.

Cabe recordar que los registros pueden ser de longitud fija y los campos de longitud variable.

# **CLAVES**

El objetivo del acceso a los archivos es buscando cierta información específica, para ello se utiliza una **clave** con el cual logremos identificar al archivo que buscamos.

Ej: Si buscamos la información de una persona, solemos buscar por su DNI, su nombre, etc.

Es decir una clave permite identificar un registro y también debe permitir ordenar el archivo por dicha clave. Se separan en 2 tipos de claves

Primaria:  
Identifica un elemento particular del archivo. Dicha clave es univoca es decir que no se repite. Ej: numero DNI / legajo.

Secundaria:  
 Reconoce un conjunto de elementos con igual valor. Ej: Nombre y apellido.

### Forma Canoníca:

Determina reglas que definen como va a ser tomada la llave, de esta forma permitiendo la correcta comparación entre llaves, es decir por ejemplo regularizando las mayúsculas en todas las claves, o los espacios en blanco.

Cabe aclarar que estas modificaciones se deberán aplicar a las llaves, mas no al archivo de donde se obtienen las llaves.

# **Estudio de performance:**

Se va a evaluar dependiendo 2 parámetros, la cantidad de accesos a memoria y cuantos ciclos de maquina usa el algoritmo, teniendo en cuenta el mejor caso y el peor caso. Teniendo como un resultado final el promedio entre ambos 2.  
Hay que tener en cuenta que el costo real son los accesos a disco, ya que cada uno consume 1ms. Una relación de igualdad (1 acceso a disco = 1Millon ciclos de maquina)

Acceso a disco: es la cantidad de veces que se lee la MS (Mem. Secundaria).

Lectura: Es la cantidad de veces que se lee los archivos.

Cabe aclarar que por cada lectura se llenara el buffer de archivos, haciendo que cantidad de accesos a disco <> cantidad de lecturas.

Los Accesos Directos son muy rápidos para el accesos a registros específicos es decir cuando se va a acceder a un único archivo de un bloque.

Pero por otro lado si se requiere el acceso a todos los registros de un archivo, el acceder uno por uno a ellos resulta ineficiente. Pudiendo llegar a ser más eficiente para estos casos el acceso serie, ya que con la ayuda del buffer se reducen los accesos.

Clasificación de archivos según cantidad de cambios:

Estáticos:  
 Son aquellos archivos que requieren casi nada de actualización, casi que desde su creación conservan los mismos valores

No requieren de mucha estructura para agilizar cambios.  
 Se lo puede considerar estáticos a los archivos que cambian cada mucho y que además su tamaño es reducido. Ya que en caso de necesitar ser actualizado con uno o pocos accesos a memoria se logra.

Volátiles:  
 Son la mayoría de archivos, aquellos los cuales deberán ser actualizados con mas frecuencia.  
 Se deberá prestar más atención a la estructura para lograr realizar los cambios rápidamente. Necesitara estructuras adicionales para mejor TE.

## **Operaciones sobre archivos:**

### **Bajas:**

Borramos datos de un archivo.  
No es una operación realizada muy frecuentemente, la información siempre es muy útil. Se debe corroborar que realmente desee borrarla.

Bajas físicas: (Libera espacio)  
 Cuando se desea borrar un archivo se elimina dicho espacio de memoria, haciendo que se deba reacomodar los registros.  
 Para reacomodarlo se puede:  
 -Generar un nuevo archivo sin los archivos que se desean borrar  
 -En el mismo archivo se desplazan para adelante los archivos y se pisan a los borrados.

Bajas lógica: (Rápido)  
 Cuando se requiere eliminar un registro se marca dicho archivo como borrado, pero sigue estando y ocupando espacio en memoria.  
Las bajas lógicas además permiten en muchos casos recuperar archivos dados de baja.  
La marca de borrado es un carácter o código que se pone en alguno de los campos del registro.   
Por Ej: Registro.Nombre:= ‘#’ + Registro.Nombre.

### **Altas:**

Alta es la creación de un nuevo registro / campo. Para esto se debe buscar donde tenemos espacio y guardar este nuevo dato ahí. Pero esto no es tan fácil ya que hay que tener en cuenta o no los espacios creados por baja lógica y/o física.

### **Modificaciones:**

Agregamos datos al archivo ya existente.  
No hay demasiado problema en la modificación de archivos de longitud fija. Pero por el otro lado en registros de longitud variable se generan algunos inconvenientes.

Si ocupa menos espacio se generará fragmentación interna. Si ocupa más puede no entrar en el archivo teniendo que implementar métodos complicados para su inserción.

Por otro lado también tenemos que tener en cuenta que podemos llegar a modificar una clave, pudiendo ser que esta rija el orden del registro en el archivo y/o pudiendo duplicar una clave.

### **Consultas:**

Imprimimos los datos de un archivo.  
No tiene demasiada magia más que el consumo de acceso a disco para obtener el registro (Si es que no esta en memoria).

### **Tipo de altas:**

**Recuperar espacio secuencial:**  
Se debe recorrer todo el archivo de manera secuencial buscando los espacios libre, existiendo la posibilidad de que no haya ningún espacio disponible y se deba almacenar el registro al final, habiendo recorrido sin sentido todo el archivo.

Archivos de longitud variable: Al ser espacios variables además está la desventaja de que se encuentre un lugar pero dicho espacio no sea suficiente para el dato requerido. Dando pie a generar FRAGMENTACION.

**Recuperar espacio indexada:**  
Se crea una lista a partir del nodo principal al inicio del archivo, creando nodos en cada espacio libre del archivo. Cada nodo almacena la dirección del próximo espacio disponible. El ultimo nodo guarda dirección -1 para indicar fin

Archivos longitud fija: Cada nodo almacena el inicio de la próxima dirección libre (NRR numero relativo de registro)

Archivos longitud variable: Cada nodo almacena la dirección de memoria del próximo campo libre. Cada Espacio puede tener una longitud más chica que la buscada por lo que se puede requerir recorrer todo el archivo incluso con la posibilidad de que no haya espacio requerido y tener que insertar al final. Puede generar FRAGMENTACION.

## **Fragmentación de archivos:**

Pasa cuando el archivo ocupa más lugar que el que necesita,

### **Fragmentación interna:**

Cuando es un espacio reservado por un registro, pero dicho registro utiliza menos espacio del que reservo.

El espacio de fragmentación interna ya no se puede ocupar.

Ej: Strings [30] pero utiliza únicamente 20 dejando 10 bytes asignados pero libres dentro del mismo registro.

Ej2: Cuando se realiza una reasignación de espacio con estrategia de primer o mejor ajuste.

### **Fragmentación externa:**

Se genera con los archivos de **longitud variable** con baja lógica cuando se inserta un registro de menor dimensión que el espacio libre. Generando esta fragmentación la cual depende su tamaño puede ser inútil. Es decir, cuando la estrategia es de peor ajuste

El espacio de fragmentación externa puede ser reutilizado dependiendo sus dimensiones.

### **Estrategias de colocación en registros de LONGITUD VARIABLE:**

Estos son los sistemas que se pueden elegir para insertar los datos en la posición más adecuada posible.

Estas características son elegidas por el DBMS[[1]](#footnote-1).

**Primer ajuste**  
Almacena el archivo en el primer espacio que sea mayor igual al archivo a almacenar.   
El Primero que encuentre sin importar el tamaño.

Utiliza la totalidad del espacio disponible. Haciendo que sobre espacio sin utilizar, generando fragmentación interna.

Es el más rápido ya que utilizar el primer lugar aceptable.

**Mejor ajuste**  
 Busca que la diferencia entre el tamaño a insertar y el disponible sea el menor posible. Busca el mejor lugar posible.

Va a encontrar el más ajustado, pero también se le asigna el espacio completo generando fragmentación interna.

Requiere recorrer todos los espacios para encontrar el mejor. Salvo que encuentre un lugar de dimensión exactamente igual al elemento a insertar, en este caso lo inserta y deja de recorrer.

**Peor ajuste** Busca el espacio más grande para almacenar el registro. Requiere recorrer todo el archivo.

Utiliza el espacio justo necesario por el archivo, generando fragmentación externa.   
Es la más eficiente respecto a espacio en memoria, ya que permite y busca la utilización completa de memoria.

## **Eficiencia**:

Búsqueda de información:

Cantidad de comparaciones (NanoSegundos - RAM).

Cantidad de accesos (MiliSegundos - disco).

**Acceso serie:** eficiencia secuencial, tiene accesos lineal siendo eficiencia n/2 lecturas, muy lento.

**Acceso directo:** permiten acceder directo al registro, consumiendo 1.

**Acceso secuencial:**

En caso de que sean archivos ordenados se puede hacer una búsqueda binaria (o dicotómica) Consiguiendo una eficiencia de Log2 N.

La ventaja es mucha y el costo es tener que mantener el archivo ordenado. En el caso de archivos estáticos este coste es muy poco, ahora en archivos volátiles este costo pasa a ser muy elevado.

Como resultado este método no es eficiente para archivos volátiles.

Métodos para ordenar un **archivo secuencial**:

Ordenar en memoria recorriendo todo el archivo, resultado N!.

Copiar en RAM, ordenarlo y cargarlo. Resultado 2 N, si entra todo en RAM.[[2]](#footnote-2)

Copiar a RAM las claves a ordenar. Resultado 3 N, si entran las claves.

Se parte el archivo en fracciones que entren en RAM, se ordenan y se hace un merge de las partes ordenadas (Maestro – detalle). [[3]](#footnote-3)

Eficiencia de árbol binario de búsqueda Log2 N

# **ÍNDICES**

Con el objetivo de tener la eficiencia de archivos ordenados, pero sin tener el gran coste de mantenimiento del mismo, se introduce los índices.

Los índices se resumen en un archivo ordenado de claves, donde cada clave además guarda la referencia a la dirección del archivo de datos donde está almacenado el resto de información de la clave.

De esta forma se consigue tener los archivos físicamente desordenados, pero lógicamente ordenados a través de un índice.

Un índice entonces está compuesto por una clave y la dirección del archivo (longitud variable) o el NRR (numero relativo registro) (longitud fija).

Para mayor eficiencia los índices se ordenan en estructuras de tipo árbol. Permitiendo la búsqueda binaria.

Los índices van a estar ordenados por la llave primaria. La llave primaria va a ser de Forma Canoníca:, pudiendo ser conformada por un conjunto de claves secundarias / primaria.

Debemos tener en cuenta que si realizamos bajas / altas debemos modificar el archivo de datos y también sus índices. Para esto se puede seguir utilizando las técnicas previamente vistas (bajas físicas / lógicas, etc.).

## **Persistencia:**

Lo ideal para la eficiencia seria tener los índices en RAM, pero esto genera alto riesgo de persistencia.

Por lo tanto la solución para mayor persistencia es tener los índices en memoria, para esto se deberán utilizar estructuras de datos eficientes.

Como ARBOLES en memoria (Logrando log2 en disco (MS)). O utilizando dispersión (HASHING).

## **Tipo de indices:**

### **Índices primarios:**

Está conformada por alguna clave univoca[[4]](#footnote-4).

Se genera un archivo con llave primaria. Que guarda la dirección directa al archivo de datos.

### **Índices secundarios:**

Guarda la dirección de la clave primaria en el índice primario. Es decir que guarda de manera indirecta su camino al archivo de datos. De esta forma simplificando los procedimientos de modificación de archivos.

Desventaja de los índices secundarios, la duplicación de claves secundarias genera problemas para la identificación y búsqueda de la clave primaria requerida.

Para esto se propone generar para cada clave secundaria un vector con todas las claves primaria con igual clave secundaria, esto genera problemas ya que requiere registros de longitud variable y por lo tanto se pierde la posibilidad de búsqueda dicotómica.

La solución más eficiente seria manteniendo registros de longitud fija. Para esto se genera 2 archivos de longitud fija donde en uno se almacenan las claves secundarias y la dirección a la primera clave primaria en el segundo archivo. En este segundo archivo se genera una lista invertida de claves secundarias.

Esto demuestra que los índices secundarios son más ineficientes. Pero más naturales, es decir más fácil para el usuario.

# **ARBOLES**

## **Arboles binario:**

Muy costoso de mantener balanceado

Búsqueda peor caso Log²(N + 1)

## **Arboles AVL (BA) (arboles balanceados en altura):**

Es algorítmicamente complicado, mucho código.

Árbol binario casi balanceado

Para TODO nodo su altura de ambas ramas varia en N. Para un BA(1) la diferencia de altura para cada nodo es <= a 1.

El coste de mantener el árbol balanceado requiere de muchos accesos a almacenamiento,

Búsqueda para peor caso: 1.44 Log² (N + 2)

## **Árbol binario paginado:**

Se separa el árbol en páginas (de árboles) donde el tamaño de cada página cambia dependiendo el tamaño RAM

Permite llevar el árbol a RAM. Todas las transferencias son de tamaño de página.

Muy costosa de implementar.

Requieren alto nivel computacional.

Es complejo mantener la relación de que una página tenga le tamaño de cada buffer

La búsqueda va a seguir siendo de Log2, pero los accesos van a ser mucho más reducidos.

## **Arboles multicamino (Arboles generales):**

Son arboles balanceados.

Son arboles de claves primarias.

Se crean de arriba.

Como mínimo tienen 2 datos / claves / registros (N) y con 3 hijos para: (orden M = 3) 1 menor a ambos datos, 1 intermedio entre los 2 datos, 1 mayor a ambos datos.

## **Arboles B** (Balanceado)

El tamaño de cada nodo en un principio estaría determinado por el tamaño del buffer, como los árboles paginados, pero con mayor eficiencia.

Se construyen de abajo hacia arriba, se llenan de izq. a derecha los nodos

Se agregan los datos SIEMPRE en hojas.

Su coste de mantenimiento es bajo.

Grado del árbol = M, claves del nodo = N y hijos por nodo = M.

Ningún nodo tiene más de M hijos, máximo M-1.

Todos los nodos internos tienen como mínimo [M/2] hijos.

La raíz tiene como mínimo 2 hijos (o ninguno).

Las hojas como mínimo [M / 2] – 1 datos.

Todos los nodos terminales (hojas) están a mismo nivel.

**Altas:**

Cuando inserta un dato y se llena el nodo, ocurre overflow.

Cuando hay overflow se debe dividir y luego promocionar.

Se promociona el dato en la posición [N/2]+1.

**Bajas:**

Si la clave a borrar no está en una hoja:

-Se permuta a un nodo terminal (hoja), se permuta el elemento inmediato superior / inferior (que se encuentran en el extremo derecho del hijo izquierdo o extremo izquierdo del hijo derecho).

Se borra la clave desde la hoja:

Mejor de los casos:

-La cantidad de elementos de la hoja es > [M/2] – 1. En este caso se borra y se reacomodan los elementos.

Peor de los casos: (Cuando se produce underflow, elementos < [M/2] – 1)

Se puede redistribuir elementos con los hermanos adyacentes. (Es decir se pasa un elemento de un hermano a otro logrando que no haya underflow).

Si no se puede redistribuir, se debe concatenar. Es decir cuando no se puede conseguir elementos del hermano adyacente. Se concatenan los nodos juntando los datos en un solo nodo.

La concatenación se realiza con 2 hermanos y el nodo padre. Juntando 3 elementos en un nodo

**Eficiencia:**

La eficiencia de búsqueda de un árbol balanceado igual a la altura del arbol.

La altura del arbol se puede calcular, para el peor caso H <= [1 + log[M/2] ((N+1)/2) ].

Siendo H entonces la cantidad máxima de accesos a disco.

PD: El peor caso para la eficiencia es que un nodo tenga la cantidad mínima de datos posible.

Eficiencia de inserción:

Mejor caso (sin overflow):  
 H lecturas, 1 escritura.

Pero caso (con overflow): (en todos los niveles)  
 H lecturas, 2h+1 escrituras (dos por niveles mas la raiz).

Pd: para un arbol de orden 10, un 25% de las veces hay overflow, para orden 100 2% de las veces.

Eficiencia de eliminación:

Mejor caso: (borra de nodo terminal)  
 H lecturas, 1 escritura.

Peor caso: (genera underflow en todos los niveles)  
 2H-1 lecturas y H+1 escrituras

### **Políticas de remplazo**

**Izquierda / derecha:**  
 Intenta redistribuir en la izquierda. Si no se puede genera una división.  
 En caso de los nodos extremos (extremo derecho / izquierdo) redistribuyen con su único hermano.

**Izquierda o derecha / derecha o izquierda:**  
 Busca redistribuir en el nodo izquierda, en caso de estar lleno intenta con el derecho.  
 Si ambos nodos están llenos procede a hacer una redistribución con los 2 nodos y el nodo padre. Generando 3 nodos ¾ llenos.

**Izquierda y derecha / derecha e izquierda:** (solo para B\*)  
 Busca redistribuir en el nodo izquierda, en caso de estar lleno intenta con el derecho.  
 Si ambos están llenos procede a juntar los 3 nodos llenos con los padres, generando 4 nodos ¾ llenos.

## **Arboles B\***

Son Arboles B más eficientes.

Algorítmicamente tiene 2 métodos diferentes de inserción y división.

La característica principal es que los nodos tienen como mínimo 2/3 ocupados.

De esta forma una misma cantidad de datos va a entrar en menos nodos, conllevando a una menor altura. Siendo de esta forma más eficiente respecto a las operaciones de búsqueda.

Algoritmos más complejos que árbol B y más ineficientes en inserción.

Cantidad mínima de descendientes [(2M-1) / 3]

Una hoja contiene al menos [(2M-1)/3] llaves y máximo M-1.

**Redistribuir:**  
La operación de distribución en la inserción en un árbol B\* se basa en que cuando un nodo lleno recibe otro dato (y no puede añadirlo porque está lleno), procede a pasárselo a su nodo hermano adyacente, si este también está lleno se produce una división.

**División:**  
Para la división se agrupa los datos de ambos nodos adyacentes y el nodo padre, se genera un nuevo nodo y se genera una nueva distribución. Generando ahora 3 nodos mínimos (2/3 llenos).

## **Arboles B+**

Son los árboles más eficientes, más usados.

Consiste en un conjunto de registros ordenados por clave de forma secuencial, que junto con un conjunto de índices que permite el acceso rápido a registro.

Todos los datos reales se encuentran en las hojas, en los nodos internos se encuentran copias de datos o ex-datos que funcionan como índice.

La parte de índices de los árboles B+ se puede implementar como un árbol B o B\* (utilizamos B por simplicidad de código).

Los nodos terminales están todos linkeados en un estilo lista, permitiendo una lectura completa del archivo rápida.

### **Arboles B+ de prefijos simples:**

Se utiliza únicamente con claves alfanuméricas, puede usarse claves primarias y secundarias.

Se optimizan los separadores, usando separadores más cortos.

Utilizan el mínimo sub String que sea suficiente para separar.

Por Ej.: si tenemos 2 nodos 1=”Gomez”, 2=”Gonzales”. El separador podría ser “Gon” .

Este hecho de hacer que los separadores ocupen menos permite trabajar con registros de longitud variable.

Al achicar los separadores nos permiten hacer los nodos hojas de mayor tamaño, lo que permite tener menos hojas y por consecuencia menor altura del árbol. Conllevando a una mayor eficiencia de acceso.

Pero también necesitan de más lógica.

# Conclusión arboles:

Arbol B es mas eficiente para busqueda aleatoria.

Arbol B+ siempre que busquemos un archivo vamos a terminar en alguna hoja.

Arbol B+ mejor para cuando se requiere recorrer todo el archivo.

Arbol B+ mayor costo de manutencion, codigo.

Accesos:

arboles binarios n

árbol de búsqueda log2 n

arboles B/B+/B\* 3o4 accesos

|  |  |  |
| --- | --- | --- |
|  | Árbol B | Árbol B+ |
| Ubicación de datos | Nodos (cualquiera) | Nodo Terminal |
| Tiempo de búsqueda | = | = |
| Procesamiento secuencial | Lento (complejo) | Rápido (con punteros) |
| Inserción eliminación | Ya discutida | Puede requerir + tiempo |

# **HASHING**

Es una técnica que permite generar una dirección única para cada clave. Dicha dirección nos da donde se va a guardar / encontrar un dato, según su clave primaria.

Es un sistema de acceso directo el cual va a tener un orden de acceso de 1, o 2 en algún caso especial.

Se organiza por clave primaria, se le aplica una función de hashing la cual da una dirección, en dicha dirección se procederá a almacenar el registro.

Desventajas:   
 -No se pueden usar registros de longitud variable. Registros longitud fija

-No hay orden físico de los datos, no se puede hacer un recorrido secuencial a bajo costo.

-Las llaves no pueden ser duplicadas. Llaves univocas

Soluciones: Hashing más eficientes, bajar densidad de empaquetamiento o aumentar tamaño nodos.

## **Definiciones:**

Sinónimos:  
Dos datos son sinónimos cuando aplicada la función hash estos tienen la misma direccion.

Colisión:  
Ocurre cuando 2 llaves diferentes son sinónimos y se intentan almacenar en un mismo nodo.

Overflow:   
Ocurre cuando dado una colisión, dichos registros no entren en el nodo (depende del tamaño del nodo).

Intruso:   
Cuando dependiendo la función de tratamiento de overflow, un registro ocupa una dirección que no es la que determino la función de hash.

## **Propiedades:**

### **Función hash:**

Es un algoritmo matemático de dispersión que dado una clave primaria devuelve una dirección, la dirección generada aparenta no tener relación con la clave.

Calificación de funciones de hashing:  
 Uniforme / perfectos:   
 Cada clave tiene una única dirección (no se forman colisiones). (Imposible en la actualidad).

Peor: Todas las claves van a la misma dirección de memoria.

Aceptable: Que haya una distribución uniforme (Que haya algunas colisiones, pero pocas).

Funciones de Hashing:  
Centros cuadrados: Se multiplica la llave por sí misma y se toman los dígitos centrales del resultado. Ese resultado posteriormente se lo ajusta al tamaño de memoria  
 (es decir otra cuenta más, previamente establecida).

División: Se divide la clave por un número similar a los espacios de memoria  
 (Se elige un número cercano pero primo).

Desplazamiento:

### **Tamaño nodos / cubetas:**

Cada nodo puede almacenar más de un registro.  
Cuanto mayor tamaño menor overflow (admite más colisión) pero mayor fragmentación.  
El nodo está hecho para entrar en el buffer, por lo que la búsqueda dentro del nodo no lleva tiempo.

### **Densidad de empaquetamiento (DE):**

La densidad es la relación entre espacio asignado al archivo y espacio realmente ocupado.  
 Cuando DE -> 1, el archivo tiende a estar lleno.  
 A menor DE menor Overflow y mayor desperdicio de espacio.  
 N = Numero de nodos, C = Capacidad de nodo, R = cantidad de registros del archivo.  
 DE = R / (C \* N).

La DE permite estimar la performance del hashing.

Estimación de overflow: (HASH DINAMICO)  
 Es la probabilidad de que un nodo (x) reciba cantidad de registros (i).  
 P(I) = (R!/(I!\*(R-I)! ))\*(1/N)^I \* (1-(1/N))^(R-1)  
 P(I) = ((R/N)^I \* E^-(R/N))/I!

Distribución de poisson.

Ej: N=10000 R=10000 DE=1 100%  
 P(0)=0.36 \*10000 3679 registros

Si se reduce la densidad de empaquetamiento se reduce bastante el overflow, pero también habrá mucha fragmentación interna.

Se llega a la conclusión entonces que dado un tamaño razonable de cada nodo (tranquilamente 100), una buena función de hash y el tamaño del archivo adecuado, se llega a tener una probabilidad de overflow de entre 0% vacío a un 4% lleno.

## **Tratamiento de overflow:**

El overflow ocurre cuando se intenta ingresar un dato a un nodo lleno. Hay distintas soluciones.

### **Saturación progresiva:**

Cuando la dirección asignada está llena, empieza a avanzar desde ahí buscando un nodo vacío, donde haya espacio se asigna.

Cuando se está buscando dicho nodo se va a su dirección teórica asignada, en caso de que no este se empieza a avanzar buscándolo, hasta que se encuentre el buscado o un nodo que estuvo siempre vacío (por lo tanto no existe el dato buscado).  
La asignación / búsqueda es circular, si se llega al último nodo se sigue buscando en el primero.  
Si se realiza una baja se debe colocar una marca de borrado para que se sepa que no estuvo vacío.  
Posee la ventaja de que es muy simple su implementación, pero el problema es que puede generar zonas de overflow que hacen que se llenen muchos nodos consecutivos y conlleve a más overflow.  
Por otro lado al estar todos los registros sinónimos cerca hay más posibilidades que cuando se copie el archivo a memoria, también se copie el sinónimo con overflow a memoria.

### **Saturación progresiva encadenada:**

Las altas se realizan igual, pero se va armando una lista con los datos sinónimos.  
El acceso es más eficiente, ya que se visitan únicamente los sinónimos.  
Mismas contras.

### **Dispersión doble:**

Utiliza 2 funciones de dispersión. La primera como uso normal y la segunda como uso exclusivo para casos de overflow.  
Es decir cuando hay OVERFLOW se aplica otra nueva función de dispersión al dato, se suma al primer hash el resultado de la segunda dispersión.  
En caso de que la posición resultante este ocupada se vuelve a sumar el resultado de segunda dispersión, así hasta que dé con una dirección con espacio disponible.

Deberá asignar marca de borrado.  
Soluciona el problema de que se haga overflow en una parte consecutiva.  
La consecuencia es que muy probablemente cuando se copie el cilindro a disco, no se copie la parte donde está el registro buscado.

### **Área de desborde/dispersión separada:**

En los métodos anteriores los registros con overflow se colocan en los mismos nodos.  
Este método tiene un área separada reservada para overflow.  
Si entra un nuevo archivo sinónimo y se genera overflow, se manda dicho archivo al otro area (área de desbordes) y se guarda su dirección y se la asigna al sinonimo previo (estilo lista).  
En el área de desbordes se van colocando los sinónimos en orden que entran, sin importar su dirección teórica.  
El tamaño del área de desbordes se calcula de acuerdo a la estimación de overflow vista previamente.

Desventaja de que los registros con overflow requerirán otro acceso a disco.

## **Tipos de dispersión**

### **Estática:**

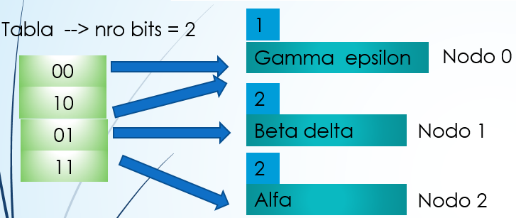
Requiere de un número de direcciones fijas.  
El hash estático es bueno hasta que el archivo se llena, si esto sucede se deberá crear un nuevo hash y migrar los datos. Se suele crear un nuevo hash cuando el archivo llega al 75%, ya que a partir de ahí hay más probabilidades de overflow.

### **Dinámica:**

Existen distintos tipos de hash. Hash dinámico, virtual o extensible.  
En todos los caso se busca mantener siempre una misma función de hash, pero teniendo la capacidad de ir aumentando su tamaño según sea necesario.  
La función Hash genera X cantidad de bits (32, 64 o más).  
Estos hash van a requerir una estructura auxiliar, llamada tabla que va a funcionar como índice.

El hash va a utilizar *i* cantidad de bits, de acuerdo a la cantidad de direcciones que requiera.

Es decir va a poder representar 2^i direcciones. El i indica cuantos bits se toman de la función de hash, empezando desde el final.

La tabla guarda el valor de número de bits, y las direcciones de los nodos que tenga. Puede que 2 posiciones de la tabla apunten a un mismo nodo (Ya que este puede tener un i menor).  
Cada nodo va a guardar su valor de i, esto indica que todos los registros que tengan i últimos bits iguales van a estar ahí.

Cuando se produce overflow:  
1- Se aumenta en 1 a i.  
2- Se crea un nuevo nodo.  
3- Se compara el i del nodo con el n° de bits de la tabla. Si es menor paso 6  
4- Se duplica el tamaño de la tabla.  
5- Se aumenta en 1 el número de bits de la tabla.  
6- Se reacomodan las direcciones, es decir a la tabla le asigna dirección nuevos nodos  
7- Se reacomodan los registros.

El hash dinámico requiere más algoritmia, pero tampoco muy significativo. Para overflow se requiere 2 accesos.   
Pero nos asegura encontrar los datos en 1 acceso, se considera que la tabla va a estar en memoria.



Resumen final:

Los sistemas de bases de datos de hoy en día están compuestos por:   
Arboles binarios balanceados para buscar según clave secundaria o univoca, para posteriormente una vez encontrada la clave primaria del registro buscado, se busca el registro completo en el archivo de datos Hashing.

Accesos:

búsqueda secuencial n/2

búsqueda binaria log2 n

arbole binario n

árbol de búsqueda log2 n

arboles B/B+/B\* 3o4 accesos

hash 1o2 accesos

1. Data Base Management System: Es el software que se encarga de crear y administrar las bases de datos. [↑](#footnote-ref-1)
2. PD: Se debe copiar las claves para ordenación y la dirección del resto de la información de dicha clave. [↑](#footnote-ref-2)
3. Estructuras de archivo, Folk. Habla de las técnicas de merge avanzadas [↑](#footnote-ref-3)
4. Las claves univocas son las candidatas a ser llaves primarias, una de ella es elegida para ser llave primaria [↑](#footnote-ref-4)