

GESTION DE DATOS

TEORIA PARA EL FINAL

CONTENIDO

UNIDAD I

Grafos

1. Conceptos y Definiciones
2. Caminos, pasos y ciclos
3. Caracterización de Grafos
 1. Grafos Simples
 2. Grafos Conexos
 3. Grafos Completos
 4. Grafos Bipartitos
 5. Grafos ponderados
4. Clasificación de Grafos
5. Representación Computacional de Grafos
6. Algoritmos de búsqueda en grafos

UNIDAD II

Arboles

1. Formalización
2. Definiciones
3. Propiedades
4. Árbol binario
5. Árbol binario perfecto
6. Árbol binario completo
7. Árbol binario balanceado
8. Recorrido de árboles
9. Representación implícita de árboles binarios
10. Árboles de expresión
11. Árboles de búsqueda
12. Árbol binario de búsqueda
13. Árboles M-arios
14. Árboles de búsqueda M-arios.
15. Árboles-B
16. Operaciones básicas sobre un árbol-B
17. Arbol B+
18. Arbol B*
19. Resumen de Arboles

Metodos de Clasificación

20. Estabilidad
21. In Situ
22. Clasificación interna y externa
23. Teoría de la complejidad u orden de crecimiento de algoritmos
24. Clases de complejidad
25. Intratabilidad
26. Metodos Elementales
 1. Bubble sort
 2. Selection Sort
 3. Insertion Sort
27. Metodos complejos
 1. Shell Sort

2. Merge Sort
3. Heap Sort - Comparación con otros metodos
4. QuickSort

28. Resumen de metodos de clasificación

Algoritmo de Huffman

Hashing

29. Funciones de hash
30. Clustering (relacionado a colisiones)
31. Tratamiento de las colisiones
32. Hash Dinámico

Arbol B VS Hashing

UNIDAD III

Arquitectura de aplicaciones

Objetos de una BD

1. Tablas
2. Tablas temporales
3. Tablas anidadas (Oracle)
4. Tablas Organizadas por Índice (IOT - Index-Organized Tables) (Oracle)
5. Clusters (Agrupamientos) (Oracle)
6. Constraints (Restricciones)
7. Secuencias
8. Sinónimos (NickName / Synonym)
9. Database Links (Enlaces de Base de Datos) (Oracle)
10. Directories (Directorios) (Oracle)
11. Views (Vistas)
12. Indices
13. Snapshots / Summary Table / Materialized Views
14. Funciones Propias de Motor (Built in Functions)
15. Funciones de Usuario
16. Stored Procedures
17. Triggers
18. Esquema (Oracle)
19. DTS (SqlServer)
20. CURSORES

UNIDAD IV

Conceptos de Bases de Datos

1. Sistema de Bases de Datos
2. Componentes de un sistema de base de datos:
3. Aplicaciones tradicionales vs. Enfoque de bases de datos
4. Independencia de los datos
5. Arquitectura de un sistema de bases de datos (ANSI/SPARC)
6. Funciones del motor de base de datos y del DBMS en su conjunto

Modelo Relacional

Transacciones y Niveles de aislamiento

7. Definición
8. Propiedades ACID
9. Sentencias para una transacción

10. Transacciones anidadas

11. Niveles de aislamiento

Bases de datos orientadas a objetos

12. ¿Qué es una BDOO?

13. Arquitectura de Una BDOO

14. Tres Enfoques de Construcción de Bases de Datos OO

15. Ventajas en BDOO's

16. Posibles Desventajas

17. Características mandatorias ó reglas de oro

18. Manifiesto de sistema de gestión de BDOO

19. SISTEMA DE GESTION DE BDOO (SGBDOO)

20. Características de los SGBDOO

21. Primer intento de Estandarización: ODMG-93.

DATAWAREHOUSE

22. DATASmart

23. OLAP

24. MODELO STAR

25. BD MULTIDIMENSIONAL

26. Data Mining

Anexo Preguntas o Conceptos de Final

RESUMEN GESTION DE DATOS

Unidad I

Grafos

Conceptos y Definiciones

Un grafo es una pareja $G = (V, A)$, donde V es un conjunto de puntos, llamados vértices, y A es un conjunto de pares de vértices, llamadas aristas. Para simplificar, notaremos la arista $\{a, b\}$ como ab . Los grafos son colecciones de objetos llamados vértices o nodos, conectados por líneas denominadas aristas o arcos. Un grafo es utilizado, en la matemática y en las ciencias de la computación, para representar relaciones entre diferentes elementos. Esas relaciones pueden mantener o no jerarquía. En caso que se presente la necesidad de soportar una relación jerárquica, el grafo debe incluir en sus arcos un sentido de dirección de la relación.

En la concepción matemática, la importancia de un grafo radica en establecer relaciones entre los vértices y las aristas. En las ciencias de la computación es muy relevante la figura de los vértices, es por ello que la terminología utilizada para denotar los vértices es nodos, esto es porque se los considera como un conjunto de datos que pueden ser de distinto tipo.

Se considera la característica de "grado" (positivo o negativo) de un vértice v y se indica como (v) , a la cantidad de aristas que llegan o salen de él; para el caso de grafos no orientados, el grado de un vértice es simplemente la cantidad de aristas que tocan este vértice.

Caminos, pasos y ciclos

Un **camino** entre dos nodos a y b , se produce cuando existe una vinculación directa o indirecta entre ambos, esto es cuando puedo vincular mediante uno o mas arcos a dichos nodos entre sí. Dado que en este concepto no está vinculado con la dirección de los arcos, puede existir camino tanto en grafos dirigidos como en grafos no dirigidos.

Un **paso** entre dos nodos a y b , se produce cuando existe un camino entre ambos, pero con un sentido preestablecido, esto es que partiendo del nodo a y siguiendo el sentido de los arcos se llega al nodo b . Como en este caso es relevante el sentido, solo se evalúan pasos en los grafos dirigidos.

Un **ciclo** es un camino, es decir una sucesión de aristas adyacentes, donde no se recorre dos veces la misma arista, y donde se regresa al punto inicial. Un ciclo **hamiltoniano** tiene además que recorrer todos los vértices exactamente una vez (excepto el vértice del que parte y al cual llega). Se habla también de camino hamiltoniano si no se impone regresar al punto de partida.

Hoy en día, no se conocen métodos generales para hallar un ciclo hamiltoniano en tiempo polinómico, siendo la búsqueda por fuerza bruta de todos los posibles caminos u otros métodos excesivamente costosos. Existen, sin embargo, métodos para descartar la existencia de ciclos o caminos hamiltonianos en grafos pequeños.

La **longitud de paso** es la cantidad de arcos involucrados.

Caracterización de Grafos

Grafos Simples

Un grafo es simple si a lo sumo sólo 1 arista une dos vértices cualesquiera. Esto es equivalente a decir que una arista cualquiera es la única que une dos vértices específicos. Un grafo que no es simple se denomina complejo.

Grafos Conexos

Un grafo es conexo si cada par de vértices está conectado por un camino. Un grafo es fuertemente conexo si cada par de vértices está conectado por al menos dos caminos disjuntos. Es posible determinar si un grafo es conexo usando un algoritmo Búsqueda en anchura (BFS) o Búsqueda en profundidad (DFS).

Grafos Completos

Un grafo simple es completo si existen aristas uniendo todos los pares posibles de vértices.

Un K_n , es decir, grafo completo de n vértices $\frac{n(n-1)}{2}$ tiene exactamente aristas.

Grafos Bipartitos

Un grafo G es bipartito si puede expresarse como $G = \{V_1 \cup V_2, A\}$ (es decir, la unión de dos grupos de vértices), bajo las siguientes condiciones:

- V_1 y V_2 son disjuntos y no vacíos.
- Cada arista de A une un vértice de V_1 con uno de V_2 .
- No existen aristas uniendo dos elementos de V_1 ; análogamente para V_2

Grafos ponderados

En muchos casos, es preciso atribuir a cada arista un número específico, llamado valuación, ponderación o coste según el contexto, y se obtiene así un grafo valuado. Formalmente, es un grafo con una función $v: A \rightarrow \mathbb{R}^+$.

Clasificación de Grafos

Según la presencia de dirección:

- Grafos Dirigidos: son aquellos en los cuales los arcos que vinculan a los vértices tienen una dirección definida, la cual marca una jerarquía en la relación modelizada.
- Grafos No dirigidos: son aquellos donde los arcos no tienen una dirección definida que marque propiedad en la relación modelizada.

Según las restricciones de sus relaciones:

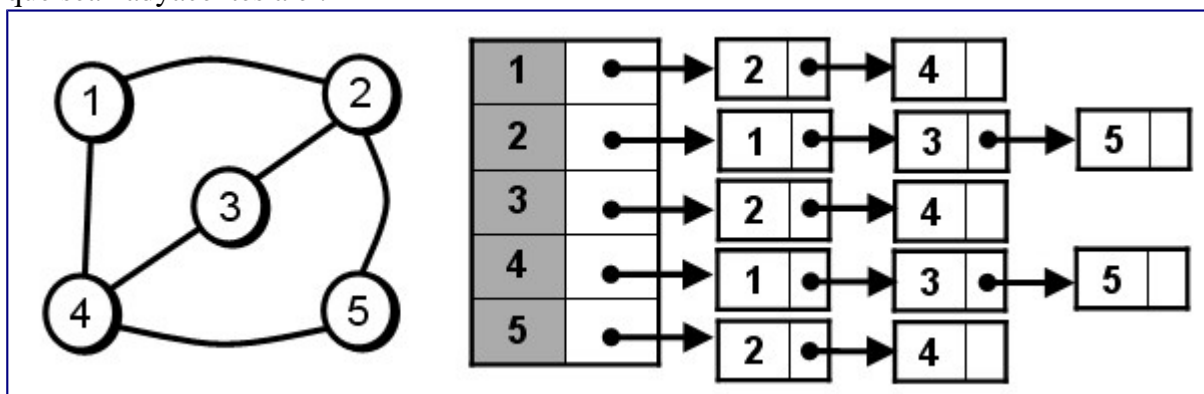
- Grafos Restritos: son aquellos en los cuales la relación modelizada no debe cumplir las propiedades de reflexividad, simetría y transitividad.
- Grafos Irrestritos: son aquellos que pueden modelizar cualquier relación independientemente de las propiedades que cumpla o no.

Representación Computacional de Grafos

Dinámica

Una representación es dinámica, cuando el espacio consumido para representar computacionalmente al grafo, concuerda exactamente con la cantidad de nodos y vértices a representar, esto es que no se consideran todas las posibilidades de relación posibles, sino que solo se representa lo que ocurre en este momento.

Lista de adyacencias: se asocia a cada nodo del grafo una lista que contenga todos aquellos nodos que sean adyacentes a él.



En este caso el espacio ocupado es $O(V + A)$, muy distinto del necesario en la matriz de adyacencia, que era de $O(V^2)$. La representación por listas de adyacencia, por tanto, será más adecuada para grafos dispersos.

Hay que tener en cuenta un aspecto importante y es que la implementación con listas enlazadas determina fuertemente el tratamiento del grafo posterior. Los nodos se van añadiendo a las listas según se leen las aristas, por lo que nos encontramos que un mismo grafo con un orden distinto de las aristas en la entrada producirá listas de adyacencia diferentes y por ello el orden en que los

nodos se procesen variará. Una consecuencia de esto es que si un problema tiene varias soluciones la primera que se encuentre dependerá de la entrada dada.

Estructura de Graal

Es igual que las listas de adyacencia nada mas que en vez de guardar el nodo en la lista, se guarda un puntero al nodo.

Representación de Pfaltz

Las estructuras de datos que utiliza Pfaltz son básicamente dos listas, una de nodos, y otra de arcos, con el diseño de celda que se muestra a continuación:

Estructura Tipo Nodo

| ID_NODO | |
|---------|--------|
| Fp 1..n | |
| Ledge | Redge |
| | Next_N |

- ID_NODO : Identificador único del nodo.
- Fp 1..n Funciones de asignación del nodo.
- LEDGE Puntero con la dirección de la primer celda de tipo “ARCO” de una lista de arcos que llegan al nodo.
- REDGE Puntero con la dirección de la primer celda de tipo “ARCO” de una arista de arcos que parten del nodo.
- NEXT_N Puntero que apunta a la próxima celda de tipo “NODO” de la lista de nodos ingresada.

Estructura Tipo Arista

| ID_ARCO | |
|---------|--------|
| Ge 1..n | |
| LPOINT | RPOINT |
| LLINK | RLINK |
| | NEXT_A |

- ID_ARCO Identificador único del arco.
- Ge 1..n Funciones de asignación del arco.
- LPOINT Puntero con la dirección de la celda de tipo “NODO” de la cuál parte el arco.
- RPOINT Puntero con la dirección de la celda de tipo “NODO” a la cuál llega el arco.
- LLINK Puntero con la dirección de la próxima celda de tipo “ARCO” de una lista de arcos con igual RPOINT (o sea que llega al mismo nodo).
- RLINK Puntero con la dirección de la próxima celda de tipo “ARCO” de una lista de arcos con igual LPOINT (o sea que parte del mismo nodo).
- NEXT_A Puntero que apunta a la próxima celda de tipo “ARCO” de la lista de arcos ingresada

Pfaltz trabaja con una representación dinámica de nodos y arcos enlazados mediante punteros, ocupando espacio en memoria a medida que realmente se necesita (cuando se crea un nuevo nodo o un nuevo arco), y liberándolo cuando se efectúa una baja.

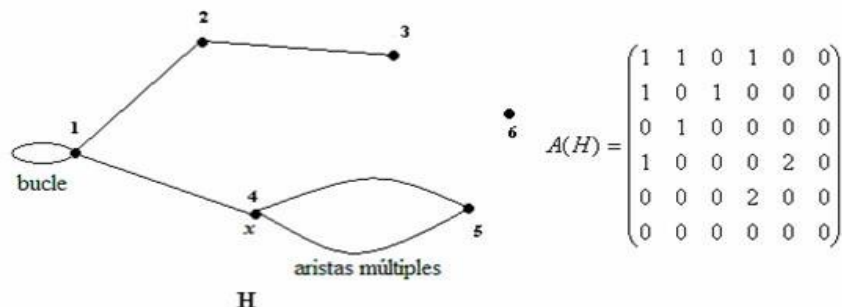
Para un problema medianamente complejo donde se pueden llegar a tener cientos de nodos, la representación de Pfaltz es más eficiente que el uso de matrices. Una matriz sería extremadamente grande teniendo en cuenta que siempre se reserva espacio para las relaciones, independientemente de la cantidad de arcos realmente existentes.

Estaticas

Una representación computacional se denomina estática, cuando el espacio consumido para representar computacionalmente al grafo es invariable y fijo respecto a la cantidad de nodos y vértices a representar, esto es que son consideradas todas las ocurrencias de relaciones que puedan producirse entre todos los nodos, reservando el espacio para dicha ocurrencia potencial.

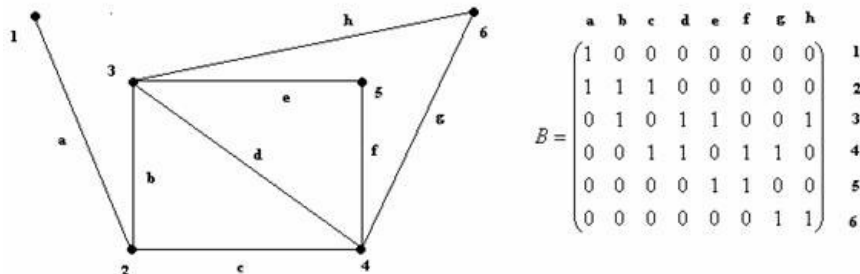
De esta forma la representación se mantiene fija y estática en el tiempo independientemente de las altas y bajas de arcos o vértices que se produzcan en el grafo representado computacionalmente.

Matriz de adyacencias: se asocia cada fila y cada columna a cada nodo del grafo, siendo los elementos de la matriz la relación entre los mismos, tomando como valores la cantidad de aristas que los unen y 0 en caso contrario.



La matriz de adyacencia de un grafo no dirigido es simétrica. En cambio la matriz de adyacencia de un dígrafo no es simétrica. Es una matriz binaria. El número de unos que aparecen en una fila es igual al grado de salida del correspondiente vértice y el número de unos que aparecen en una determinada columna es igual al grado de entrada del correspondiente vértice.

Matriz de incidencias: se asocia cada fila con un nodo y cada columna con una arista del grafo, siendo los elementos de la matriz la relación un 1 si dicho nodo es incidente con dicha arista y un 0 en caso contrario.



La matriz de incidencia sólo contiene ceros y unos (matriz binaria). Como cada arista incide exactamente en dos vértices, cada columna tiene exactamente dos unos. El número de unos que aparece en cada fila es igual al grado del vértice correspondiente. Una fila compuesta sólo por ceros corresponde a un vértice aislado.

Algoritmos de búsqueda en grafos (problema del caballo)

Búsqueda en anchura.

Es un algoritmo para recorrer o buscar elementos en un grafo (usado frecuentemente sobre árboles). Intuitivamente, se comienza en la raíz (eligiendo algún nodo como elemento raíz en el caso de un grafo) y se exploran todos los vecinos de este nodo. A continuación para cada uno de los vecinos se exploran sus respectivos vecinos adyacentes, y así hasta que se recorra todo el árbol.

Formalmente, BFS es un algoritmo de búsqueda sin información, que expande y examina todos los nodos de un árbol sistemáticamente para buscar una solución. El algoritmo no usa ninguna estrategia heurística.

Procedimiento

- Dado un vértice fuente s , Breadth-first search sistemáticamente explora los vértices de G para “descubrir” todos los vértices alcanzables desde s .
- Calcula la distancia (menor número de vértices) desde s a todos los vértices alcanzables.
- Después produce un árbol BF con raíz en s y que contiene a todos los vértices alcanzables.
- El camino desde s a cada vértice en este recorrido contiene el mínimo número de vértices. Es el camino más corto medido en número de vértices.
- Su nombre se debe a que expande uniformemente la frontera entre lo descubierto y lo no descubierto. Llega a los nodos de distancia k , sólo tras haber llegado a todos los nodos a distancia $k-1$.

Falta el código de ej que dio Reinoso en clase.

El tiempo de ejecución es $O(|V|+|E|)$. Nótese que cada nodo es puesto a la cola una vez y su lista de adyacencia es recorrida una vez también.

Búsqueda en profundidad

Es un algoritmo que permite recorrer todos los nodos de un grafo o árbol de manera ordenada, pero no uniforme. Su funcionamiento consiste en ir expandiendo todos y cada uno de los nodos que va localizando, de forma recurrente, en un camino concreto. Cuando ya no quedan más nodos que visitar en dicho camino, regresa (Backtracking), de modo que repite el mismo proceso con cada uno de los hermanos del nodo ya procesado.

El tiempo de ejecución es $O(|V|+|E|)$

Falta el código de ej que dio Reinoso en clase.

Apuntes incluidos en este resumen:

- Teoría de grafos del Ing Reinoso
- Estructura de Pfaltz del Ing Zaffaroni
- Wikipedia

Apuntes no incluidos:

- Teoría de grafos del Ing Zaffaroni
- Estructuras lineales (pilas, listas, colas, etc) del Ing Zaffaroni
- Álgebra relacional del Ing Zaffaroni

Unidad II

Arboles

Un árbol es una colección no vacía de nodos y aristas que cumplen ciertos requisitos. Se lo utiliza principalmente para representar jerarquía y ser utilizado en todo lo que conlleve establecer un orden en un conjunto de valores.

Formalización

Forma no recursiva

Se distingue un nodo como raíz. A cada nodo b , exceptuando la raíz, le llega una arista desde exactamente un nodo a , el cual se le llama padre de b . Decimos que b es uno de los hijos de a . Hay un único camino desde la raíz hasta cada nodo.

Forma recursiva

Un árbol T es un conjunto finito, no vacío de nodos $T = \{r\} \cup T_1 \cup T_2 \cup \dots \cup T_n$ con las siguientes propiedades:

- Un nodo del conjunto es designado como raíz del árbol (r).
- Los nodos restantes son particionados en $N \geq 0$ subconjuntos, T_1, T_2, \dots, T_n en el que cada uno es un árbol.

Definiciones

- Los nodos que no tienen hijos reciben el nombre de hojas o nodos terminales.
- Frecuentemente las hojas son nodos diferentes a los que no son hojas, por ejemplo, pueden no tener nombre o información asociada. Entonces en tales situaciones se denomina nodos externos a las hojas y nodos internos a los nodos que no son hojas.
- El grado de un nodo es el número de subárboles asociados a ese nodo.
- La profundidad o nivel de un nodo en un árbol es la longitud del camino que va desde la raíz hasta ese nodo. Por lo tanto la profundidad de la raíz es siempre 0 y la de cualquier nodo es la de su padre más uno.
- La altura de un nodo es la longitud del camino que va desde el nodo hasta la hoja más profunda bajo él. La altura de un árbol se define como la altura de su raíz.
- Si hay un camino del nodo u al nodo v , entonces decimos que u es un ascendiente de v y que v es un descendiente de u .
- El tamaño de un nodo es igual al número de descendientes que tiene (incluyendo dicho nodo). El tamaño de un árbol se define como el tamaño de su raíz.
- Un árbol ordenado es aquel árbol con raíz en el que se ha sido especificado el orden de los hijos en cada nodo.

Propiedades

- Dados dos nodos cualesquiera de un árbol, existe exactamente un camino que los conecta. Una consecuencia importante de esta propiedad es que cualquier nodo puede ser raíz, ya que, dado cualquier nodo de un árbol, existe exactamente un camino que lo conecta con cualquier otro nodo del árbol.
- Un árbol con N nodos debe tener $N - 1$ aristas porque a cada nodo, excepto la raíz, le llega una arista.
- La altura de cualquier nodo es uno más que la mayor altura de un hijo suyo.
- Un árbol N -ario con $n \geq 0$ nodos internos, contiene $(N - 1)n + 1$ nodos externos.
- Un árbol N -ario de altura $h \geq 0$ tiene como máximo N^h hojas.

Árbol binario

Un árbol binario es un árbol en el que ningún nodo puede tener más de dos hijos, se los distingue como izquierdo y derecho.

En general, la altura promedio de un árbol binario es $O(N)$, pero puede llegar a ser $N - 1$.

Árbol binario perfecto

Un árbol binario perfecto de altura $h \geq 0$ es un árbol binario $T = \{r, TL, TR\}$ con las siguientes propiedades:

- Si $h = 0$, entonces $TL = \text{vacio}$ y $TR = \text{vacio}$
- Por el contrario, si $h > 0$, ambos TL y TR son árboles binarios perfectos de altura $h - 1$.

Un árbol binario perfecto de altura h tiene exactamente $2^{h+1} - 1$ nodos internos.

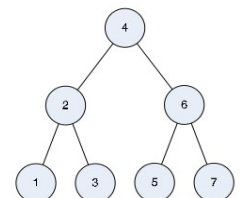
Por lo tanto la cantidad de nodos permitido en este tipo de árboles es:

$$0, 1, 3, 7, 15, 31, \dots, (2^{h+1}) - 1, \dots$$

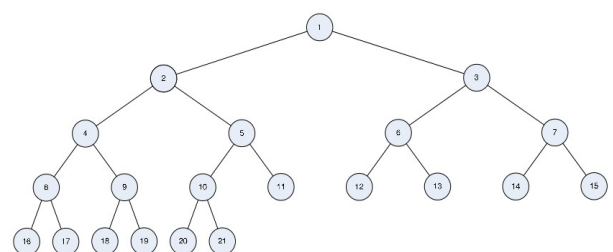
El peor caso para una búsqueda en un árbol binario perfecto es $O(\log n)$.

Árbol binario completo

Un árbol binario en el que todos los niveles están llenos,



Árbol binario perfecto



Árbol binario completo

excepto posiblemente el ultimo nivel, el cual es completado de izquierda a derecha. Un árbol binario completo de altura $h \geq 0$ contiene al menos 2^h nodos y a lo sumo $(2^{(h+1)}) - 1$ nodos.

Árbol binario balanceado

Un árbol binario esta *completamente balanceado* si esta vacío, o ambos subárboles están completamente balanceados y tienen la misma altura. Por lo tanto cualquier camino desde la raíz a una hoja tiene la misma longitud. Los únicos árboles que están perfectamente balanceados son los árboles binarios perfectos.

Un árbol binario no vacío $T = \{r, TL, TR\}$ está *AVL balanceado* si $H_L - H_R \leq 1$, donde H_L es la altura de TL y H_R es la altura de TR. En otras palabras, en todos los nodos, la altura de la rama izquierda no difiere en más de una unidad de la altura de la rama derecha. Debido a esta propiedad estructural, el orden de complejidad de la búsqueda en estos árboles se mantiene en $O(\log n)$

Recorrido de árboles

Esencialmente hay dos métodos distintos para visitar todos los nodos de un árbol, recorrido primero en profundidad y recorrido primero en anchura (u orden de nivel). Entre los de primero en profundidad se encuentran, preorden, postorden, en orden.

Preorden

El recorrido en preorden puede ser definido recursivamente de la siguiente manera:

1. Visitar primero la raíz.
2. Hacer un recorrido preorden a cada uno de los subárboles de la raíz, uno por uno en un orden determinado.

En el caso de de un árbol binario, el algoritmo resulta:

1. Visitar primero la raíz
2. Recorrer el subárbol izquierdo
3. Recorrer el subárbol derecho.

Inorden

El recorrido inorden, también llamado simétrico, solo puede ser usado en árboles binarios. En este recorrido, se visita la raíz entre medio de las visitas entre el subárbol izquierdo y derecho.

1. Recorrer el subárbol izquierdo
2. Visitar la raíz
3. Recorrer el subárbol derecho.

Postorden

El último de los métodos de recorrido de árboles de primero en profundidad es el postorden. A diferencia del preorden donde se visita primero a la raíz, en el recorrido en postorden se visita la raíz a lo último.

1. Hacer un recorrido preorden a cada uno de los subárboles de la raíz, uno por uno en un orden determinado.
2. Visitar por ultimo la raíz.

En el caso de los árboles binarios:

1. Recorrer el subárbol izquierdo
2. Recorrer el subárbol derecho.
3. Visitar por último la raíz.

Primero en anchura

En primero en anchura se visitan los nodos en el orden de su profundidad de en el árbol. Primero se visita todos los nodos en la profundidad cero (la raíz), y después todos los de profundidad igual a uno, etc. Para poder realizar una búsqueda de primero en anchura es necesario valerse de una

estructura de datos auxiliar, la cola.

1. Poner en cola la raíz.
2. Mientras que la cola no esté vacía
3. Quitar primero de la cola y asignarlo a una variable auxiliar, Nodo.
4. Imprimir el contenido de Nodo.
5. Si Nodo tiene hijo izquierdo, poner al hijo izquierdo en la cola.
6. Si Nodo tiene hijo derecho, poner al hijo derecho en la cola.

Representación implícita de árboles binarios

Representar un árbol binario mediante un array.

En este caso los nodos están almacenados en el array en vez de estar vinculados por medio de referencias. La posición del nodo en el array corresponde a su posición en el árbol. El nodo en la posición 0 es la raíz, el nodo en la posición 1 es el hijo izquierdo, y el nodo de la posición 2 es el hijo derecho. Se sigue con este procedimiento de izquierda a derecha en cada nivel del árbol. Los elementos que representan posiciones del árbol sin nodos son rellenados con 0 o con null.

Si el índice de un nodo es I entonces:

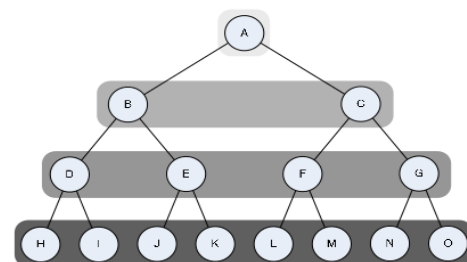
El hijo izquierdo del nodo es: $2I + 1$

El hijo derecho del nodo es: $2I + 2$

El padre de I es: $\lfloor (i - 1) / 2 \rfloor$

Los nodos sin llenar o nodos borrados dejan huecos en el array, desperdiciando memoria. Sin embargo si el borrado de nodos no esta permitido, puede ser una buena alternativa, y además es útil, si es muy costoso obtener memoria para cada nodo de forma dinámica.

| Array | |
|-------|---|
| 0 | A |
| 1 | B |
| 2 | C |
| 3 | D |
| 4 | E |
| 5 | F |
| 6 | G |
| 7 | H |
| 8 | I |
| 9 | J |
| 10 | K |
| 11 | L |
| 12 | M |
| 13 | N |
| 14 | O |



Árboles de expresión

Tomamos como ej la expresión $a / b + (c - d) e$

Los nodos terminales (hojas) de un árbol de expresión son las variables o constantes en la expresión ($a, b, c, d, y e$). Los no terminales son los operadores ($+, -, \times y \div$). Los paréntesis de la ecuación no aparecen en el árbol, ya que justamente las posiciones de los operadores en el árbol son los que le asignan prioridades. Para imprimir se usa el recorrido inorden y se utiliza el siguiente procedimiento:

Si se encuentra un nodo terminal, se lo imprime. De lo contrario, se hace lo siguiente:

1. Imprimir un paréntesis izquierdo
2. Recorrer el subárbol izquierdo
3. Imprimir la raíz
4. Recorrer el subárbol derecho.
5. Imprimir el paréntesis derecho.

Árboles de búsqueda

Soporta las operaciones de búsqueda, inserción y eliminación de forma eficiente. Las claves no aparecen en los nodos de forma arbitraria, sino que hay un criterio de orden que determina donde una determinada clave puede ubicarse en el árbol en relación con las otras claves en ese árbol.

Árbol binario de búsqueda

Una extensión del algoritmo de búsqueda binario que permite tanto inserciones como eliminaciones. El tiempo de ejecución de muchas de sus operaciones es de $O(\log N)$ en el caso medio, pero en el peor de los casos puede llegar a $O(N)$.

Puede definirse como un árbol que satisface la propiedad de búsqueda ordenada. Esto significa que para cada nodo X del árbol, los valores de todas las claves de su subárbol izquierdo son menores

que la clave de X y los valores de todas las claves de su subárbol derecho son mayores que la clave de X .

Observaciones:

- Si a este tipo de árbol se lo recorre en inorden se muestran los elementos ordenados de forma ascendente
- no permite en principio la existencia de elementos duplicados, pero es posible modificarla para si se permitan. Ante la existencia de dos claves duplicadas, una de las soluciones es tener una sola clave, y guardar un contador con el número de repeticiones.

Operaciones:

- Busqueda: Para buscar un elemento se empieza por la raíz y desplaza repetidamente por las ramas izquierda o derecha, dependiendo del resultado de las comparaciones.
- Busqueda del minimo/maximo: el minimo se obtiene recorriendo repetidamente los nodos izquierdos hasta llegar a un nodo que no tenga hijo izquierdo. Para buscar el máximo se recorre de forma similar pero haciéndolo por la derecha.
- Eliminación e inserción de nodos es una tarea que requiere de cierta complejidad. En el caso de querer eliminar un nodo se debe evitar que al hacerlo el árbol no quede desconectado.

El costo de las operaciones del árbol binario de búsqueda es proporcional al número de nodos consultados durante la operación. El costo de acceso a cada nodo es 1 más su profundidad.

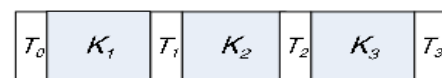
Como conclusión, si el árbol esta bien equilibrado, el costo de los accesos es logarítmico. árbol no equilibrado. Hay N nodos en el camino para recorrer hasta el nodo de mayor profundidad, con lo que el costo de la búsqueda en el peor de los casos es $O(N)$. También es posible demostrar que en el caso medio, la mayoría de las operaciones requieren de un tiempo cercano a $O(\log N)$.

Árboles M-arios

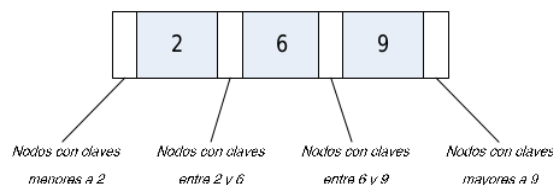
Un árbol M-ario es consiste de n subárboles T_0, T_1, K, T_{n-1} y n - 1 claves K_1, K_2, K, K_{n-1} ,

$T = \{T_0, K_1, T_1, K_2, T_2, K, K_{n-1}, T_{n-1}\}$ donde $2 \leq n \leq M$, de tal manera que las claves y nodos satisfacen las siguientes propiedades de ordenación de datos:

1. Las claves en cada nodo son distintas y están ordenadas, por ejemplo $K_i < K_{i+1}$ para $1 \leq i \leq n - 1$
2. Todas las claves contenidas en el subárbol T_{i-1} son menores que K_i . Al árbol T_{i-1} se lo llama subárbol izquierdo respecto de la clave K_i .
3. Todas las claves contenidas en el subárbol T_i son mayores que K_i . Al árbol T_{i+1} se lo llama subárbol derecho respecto de la clave K_i .



Nodo de un árbol M-ario de búsqueda para M=4



Árboles de búsqueda M-arios.

Si se tiene un conjunto de datos muy grande que no podemos colocarlo en memoria principal, nos veríamos obligados implementar el árbol de búsqueda en un almacenamiento secundario, como el disco. Las características de un disco a diferencia de la memoria principal hacen que sea necesario utilizar valores de M más grandes para poder implementar estos árboles de búsqueda de forma eficiente. Al elegir un valor de M grande, podemos arreglar para que un nodo de un árbol M-ario pueda ocupar un bloque de disco completo. Si cada nodo interno en el árbol M-ario tiene exactamente M hijos, puedes usar el siguiente teorema: $h \geq \lceil \log_{base(M)} ((M - 1)n + 1) \rceil - 1$ Donde n es el número de nodos internos de un árbol de búsqueda. Un nodo en un árbol M-ario de búsqueda que tiene M hijos contiene exactamente M - 1 claves. Por lo tanto, en total hay $K = (M - 1)n$ claves, entonces resulta $h \geq \lceil \log_{base(M)} (K - 1) \rceil - 1$.

Árboles-B

Los Árboles-B son Árboles M-arios balanceados diseñados para funcionar bien en discos magnéticos o en cualquier otro tipo de almacenamiento secundario de acceso directo. El objetivo principal es minimizar las operaciones de entrada y salida hacia el disco. Al imponer la condición de balance, el árbol es restringido de manera tal que se garantice que la búsqueda, la inserción y la eliminación de sean todos de tiempo $O(\log N)$.

La cantidad de datos manejados puede ser tan grande que todos los datos no caben en memoria principal al mismo tiempo, entonces los algoritmos de árboles-B copian los bloques o páginas de disco a memoria principal a medida que se los necesita y se vuelve a escribir solo las páginas que han sido modificadas. Dado que estos algoritmos solo necesitan una cantidad constante de páginas presentes en memoria principal a la vez, el tamaño de la memoria principal no limita el tamaño del árbol-B que puede ser manejado.

Definición

Un árbol-B de orden M es o bien un árbol vacío, o es un árbol M-ario de búsqueda T con las siguientes propiedades:

1. La raíz de T tiene al menos dos subárboles y a lo sumo M subárboles.
2. Todos los nodos internos de T (menos la raíz) tienen entre $\lceil M/2 \rceil$ y M subárboles.
3. Todos los nodos externos de T están al mismo nivel.

Los nodos deben estar, al menos, medio llenos. Esto garantiza que el árbol no degenerara en un simple árbol binario o ternario.

Arbol B+ : solo se almacenan los datos en las hojas del árbol (Ej NTFS, RaiserFS, XFS, índices de BDR)

Operaciones básicas sobre un árbol-B

Búsqueda:

Si necesitamos buscar un ítem x en un árbol-B, debemos comenzar por la raíz. Si el árbol está vacío, la búsqueda falla. De lo contrario, las claves en la nodo raíz son examinadas para determinar si el elemento que se esta buscando esta presente. Si está, la búsqueda termina exitosamente. Si no está, hay tres posibilidades a considerar:

- Si el elemento x a buscar es menor que K_1 , entonces se continúa buscando en el subárbol T_0 .
- Si x es más grande que K_{n-1} , se continúa buscando en el subárbol T_{n-1} .
- Si existe un i tal que $1 \leq i \leq n-1$ para el cual $K_i \leq x \leq K_{i+1}$ entonces se continúa buscando en el árbol T_i .

El tiempo de ejecución de una búsqueda exitosa depende por la profundidad en que se encuentre el elemento a buscar dentro del árbol. El tiempo de ejecución en el peor de los casos esta determinada por la altura del árbol-B.

Inserción

Para insertar un elemento x, comenzamos en la raíz y realizamos una búsqueda para el. Asumiendo que el elemento no esta previamente en el árbol, la búsqueda sin éxito terminará en un nodo hoja.

Este es el punto en el árbol donde el x va a ser insertada.

Se pueden dar 2 situaciones:

- Si el nodo hoja tiene menos de $M-1$ claves en el, simplemente insertamos el elemento en el nodo hoja y damos por terminado.
Por cada clave que insertamos en el nodo, se requiere un nuevo subárbol. Si el nodo es una hoja, el subárbol a insertar está vacío. Por lo tanto cuando insertamos un elemento x, en verdad, estamos insertando el par de elementos (x, ϕ) .
- La hoja esta completa, es decir que queremos insertar el par (x, ϕ) , en un nodo T que ya tiene $M-1$ claves. Esto resultaría en un nodo de árbol-B de orden M no válido por que tiene $M+1$ subárboles y M claves. La solución es dividir el nodo T a la mitad (split),

creando dos nodos, $T'L$ y $T'R$, cada uno conteniendo la mitad de nodos que el original, y una clave restante, llamada $k \lceil M/2 \rceil$. Ahora hay dos casos para considerar:

- Si T no es la raíz, se hace lo siguiente: Primero, $T'L$ reemplaza a T en el padre de T . Luego, tomamos el par $(k \lceil M/2 \rceil, T'R)$ y lo insertamos recursivamente en el padre de T . Si el nodo padre se llena, entonces se divide también y los dos nuevos nodos son insertados en el abuelo. Este proceso continua todo su camino hacia arriba del árbol hasta la raíz.
- Cuando se llena la raíz también se divide. Sin embargo, dado que no hay padre al que insertar los dos nuevos hijos, una nueva raíz es insertada arriba de la vieja raíz. La nueva raíz va a contener exactamente dos subárboles y una sola clave.

Una cosa interesante para destacar, es que la altura del árbol-B solo se incrementa cuando el nodo raíz se divide. Lo que es más, cuando el nodo raíz se divide, la dos mitades son adjuntadas a la nueva raíz, por lo tanto todos los nodos externos permanecen a la misma profundidad, cumpliendo con la definición de árbol-B.

Eliminación

Hay dos estrategias populares de eliminación en un árbol B.

1. Buscar, eliminar el elemento y reestructurar el árbol para recuperar sus invariantes
2. Hacer una sola pasada por el árbol, pero antes de visitar un nodo, reestructurar el árbol para que una vez que se encontró la clave que desea eliminar, se pueda eliminar sin que se dispare la necesidad de una reestructuración más

Hay dos casos especiales a tener en cuenta cuando se elimina un elemento:

1. El elemento en un nodo interno es un separador para sus nodos hijos
2. Eliminación de un elemento puede poner su nodo con el número mínimo de elementos e hijos

Eliminación de un nodo de hoja

1. Busque el valor a eliminar.
2. Si el valor está en un nodo hoja, basta con eliminarlo del nodo.
3. Si ocurre desbordamiento, reequilibrar el árbol

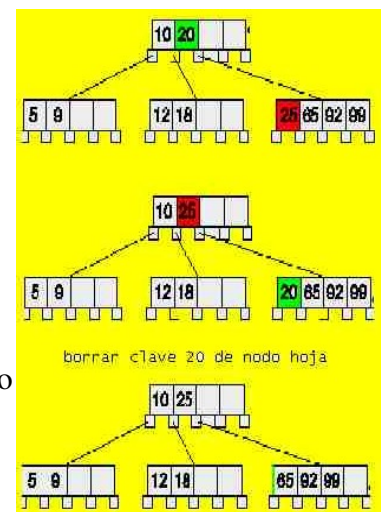
Eliminación de un nodo interno

Cada elemento de un nodo interno actúa como un valor de separación de dos sub-arboles, por lo tanto, tenemos que encontrar un reemplazo para la separación. Tenga en cuenta que el elemento más grande en el subárbol izquierdo es todavía menor que el separador. Del mismo modo, el elemento más pequeño en el subárbol derecho es todavía mayor que el separador. Ambos de estos elementos se encuentran en los nodos hoja, y, uno de los dos puede ser el nuevo separador de los dos subárboles. Descrito algorítmicamente a continuación:

1. Elegir un nuevo separador (ya sea el elemento más grande en el subárbol izquierdo o el elemento más pequeño en el subárbol derecho), retírela del nodo hoja que se encuentra, y vuelva a colocar el elemento a borrar con el nuevo separador.
2. El paso anterior elimina un elemento (el nuevo separador) de un nodo hoja. Si ese nodo hoja es ahora deficiente (tiene menos que el número requerido de nodos), a continuación, reequilibrar el árbol a partir del nodo de hoja.

Reequilibrio después de la eliminación

Comienza a partir de una hoja y procede hacia la raíz hasta que se equilibra el árbol. Si al eliminar un elemento de un nodo lo lleva abajo del mínimo permitido de elementos, entonces algunos elementos deben ser redistribuidos para que todos los nodos queden arriba del mínimo permitido. Por lo general, la redistribución implica mover un elemento a partir de un nodo hermano que tiene



más que el número mínimo de nodos. Esa operación de redistribución se llama una rotación. Si no hay ningún hermano puede prescindir de un nodo, el nodo deficiente debe ser combinado con un hermano. La fusión hace que el padre pierda un elemento de separación, por lo que el padre puede ser deficiente y la necesidad de reequilibrio. La fusión y el reequilibrio puede seguir todo el camino hasta la raíz. Dado que el número de elementos mínimo no se aplica a la raíz, haciendo la raíz es el único nodo deficiente no es un problema. El algoritmo para reequilibrar el árbol es el siguiente:

1. Si existe hermano derecho del nodo deficiente y tiene más de un número mínimo de elementos, a continuación, rote a la izquierda
 1. Copiar separador de los padres al final del nodo deficiente (el separador se mueve hacia abajo, el nodo deficiente ahora tiene el número mínimo de elementos)
 2. Reemplazar separador de los padres con el primer elemento de la derecha hermano (hermano derecho pierde un nodo, pero todavía tiene al menos el número mínimo de elementos)
 3. El árbol está equilibrado
2. De lo contrario, si existe hermano izquierdo del nodo deficiente y tiene más de un número mínimo de elementos, a continuación, rote a la derecha
 1. Copiar separador de los padres para el inicio del nodo deficiente (el separador se mueve hacia abajo; nodo deficiente ahora tiene el número mínimo de elementos)
 2. Reemplazar separador de los padres con el último elemento de la izquierda hermano (hermano izquierdo pierde un nodo, pero todavía tiene al menos el número mínimo de elementos)
 3. El árbol está equilibrado
3. De lo contrario, si ambos hermanos inmediatos tienen sólo el número mínimo de elementos, a continuación, combinar con un hermano
 1. Copiar el separador hasta el final del nodo izquierdo (el nodo izquierdo puede ser el nodo deficiente o puede ser el hermano con el número mínimo de elementos)
 2. Mueva todos los elementos del nodo derecho en el nodo izquierdo (el nodo izquierdo ya tiene el número máximo de elementos)
 3. Ya no es necesario el nodo derecho y puede ser colocado en una lista libre
 4. Retire el separador de los padres (el padre pierde un elemento)
 1. Si el padre es la raíz y ahora no tiene elementos, entonces libere la vieja raíz y haga que el nodo combinado sea la nueva raíz (árbol se hace menos profunda)
 2. De lo contrario, si el padre tiene menos que el número requerido de elementos, a continuación, reequilibrar el padre

Nota: Las operaciones de reequilibrio son diferentes para los árboles B + (por ejemplo, la rotación es diferente porque los padres tiene copia de la clave) y B *-tree (por ejemplo, tres hermanos se fusionan en dos hermanos).

Arbol B+

Representa una colección de datos ordenados de manera que se permite una inserción y borrado eficientes de elementos. Es un índice, multinivel, dinámico, con un límite máximo y mínimo en el número de claves por nodo. Es una variación de un árbol B.

Toda la información se guarda en las hojas. Los nodos internos sólo contienen claves y punteros. Todas las hojas se encuentran en el mismo nivel, que corresponde al más bajo. Los nodos hoja se encuentran unidos entre sí como una lista enlazada para permitir búsqueda secuencial.

Características

1. El número máximo de claves en un registro es llamado el orden del árbol B+.

2. El mínimo número de claves por registro es la mitad del máximo número de claves. Por ejemplo, si el orden de un árbol B+ es n , cada nodo (exceptuando la raíz) debe tener entre $n/2$ y n claves.
3. El número de claves que pueden ser indexadas usando un árbol B+ está en función del orden del árbol y su altura.

Altura: El mejor y el peor caso

Dado un M , el cual corresponde al número máximo de hijos que un nodo puede contener se define por:

- La altura h de un árbol B+ (El peor caso): $\log_M n$
- La altura h de un árbol B+ (Mejor caso): $\log(\frac{M}{2}) n$

Este caso se debe a que si guardamos menos hijos en los nodos, se necesitarán más niveles para almacenar todo.

Cantidad de claves

Para un árbol B+ de orden n , con una altura h :

- El número máximo de claves es: n^h
- El número mínimo de claves es: $2(n/2)^{h-1}$

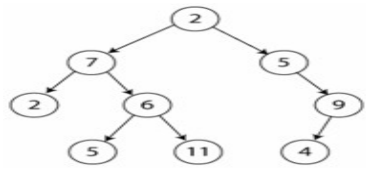
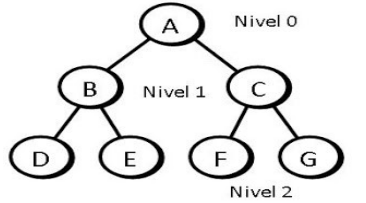
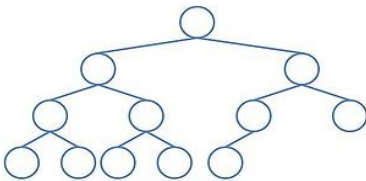
Árbol B*

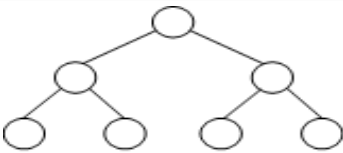
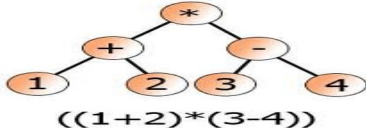
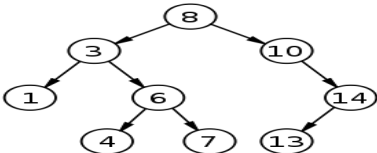
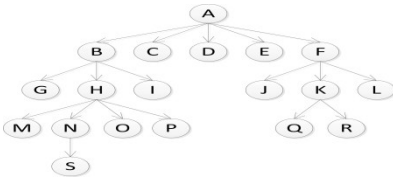
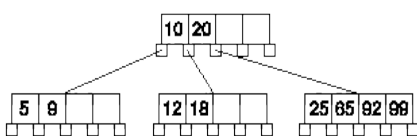
Es una variante de Árbol-B utilizado en los sistemas de ficheros HFS y Reiser4, que requiere que los nodos no raíz estén por lo menos a 2/3 de ocupación en lugar de 1/2. Para mantener esto los nodos, en lugar de generar inmediatamente un nodo cuando se llenan, comparten sus claves con el nodo adyacente. Cuando ambos están llenos, entonces los dos nodos se transforman en tres.

También requiere que la clave más a la izquierda no sea usada nunca.

No se debe confundir un árbol-B* con un árbol-B+, en el que los nodos hoja del árbol están conectados entre sí a través de una lista enlazada, aumentando el coste de inserción para mejorar la eficiencia en la búsqueda.

En Resumen

| Nombre | Definición | Propiedades | Imagen |
|------------------------|---|--|---|
| Árbol Binario | Un árbol binario es un árbol en el que ningún nodo puede tener más de dos hijos, a los cuales se los distingue como izquierdo y derecho. | Altura promedio: $O(\sqrt{N})$ |  |
| Árbol Binario Perfecto | Árbol binario con las siguientes propiedades: 1) Si $h=0$, entonces $T_L=\emptyset$ y $T_R=\emptyset$. 2) Por el contrario, si $h>0$, ambos T_L y T_R son árboles binarios perfectos de altura $h-1$. | Una de las propiedades más importantes de estos árboles es que un árbol binario perfecto de altura h tiene exactamente $2^{(h+1)} - 1$ nodos internos. Peor caso de búsqueda: $O(\log n)$. |  |
| Árbol Binario Completo | Árbol binario en el que todos los niveles están llenos, excepto posiblemente el último nivel, el cual es completado de izquierda a derecha. | Un árbol binario completo de altura $h \geq 0$ contiene al menos 2^h nodos y a lo sumo $2^{(h+1)} - 1$ nodos. |  |

| | | | |
|-----------------------------|---|---|---|
| Árbol Binario Balanceado | Un árbol binario está completamente balanceado si está vacío, o ambos subárboles están completamente balanceados y tienen la misma altura. Por lo tanto cualquier camino desde la raíz a una hoja tiene la misma longitud. | Los únicos árboles que están perfectamente balanceados son los árboles binarios perfectos. En todos los nodos, la altura de la rama izquierda no difiere en más de una unidad de la altura de la rama derecha. Complejidad de búsqueda: $O(\log n)$. |  |
| Árboles de expresión | Árboles binarios utilizados para representar expresiones algebraicas. | Los nodos terminales o hojas son las variables o constantes en la expresión. Los no terminales son los operadores. |  |
| Árbol Binario de Búsqueda | Árbol que soporta las operaciones de búsqueda, inserción y eliminación de forma eficiente. Árbol que satisface la propiedad de búsqueda ordenada. | Las claves siguen un criterio de orden que determina donde una determinada clave puede ubicarse en el árbol en relación con las otras claves en ese árbol. Tiempo de búsqueda medio: $O(\log n)$ Tiempo de búsqueda peor: $O(N)$ |  |
| Árboles de búsqueda M-arios | Consiste de n subárboles y n-1 claves. Un nodo que tiene M hijos contiene exactamente M-1 claves. Hay $K=(M-1)n$ claves. | Las claves en cada nodo son distintas y están ordenadas. $h \geq \lceil \log_M(K-1) \rceil - 1$. Si elegimos M grande, un nodo puede ocupar un bloque de disco completo. |  |
| Árboles B | Árbol vacío o árbol M-ario que cumple: 1) La raíz de T tiene al menos dos subárboles y a lo sumo M subárboles. 2) Todos los nodos internos de T (menos la raíz) tienen entre $\lceil M/2 \rceil$ y M subárboles. 3) Todos los nodos externos de T están al mismo nivel. | Obj. principal: minimizar las operaciones de entrada y salida hacia el disco. Al imponer la condición de balance, el árbol es restringido de manera tal que se garantice que la búsqueda, la inserción y la eliminación de nodos sean todos de tiempo $O(\log N)$. Principales operaciones: búsqueda, inserción y eliminación. |  |

Metodos de Clasificación

Definamos formalmente el problema de la clasificación.

Entrada: Una secuencia de números $\{a_1, a_2, \dots, a_n\}$ $n \geq 0$

Salida: Una reclasificación $\{a'_1, a'_2, \dots, a'_n\}$ de la secuencia inicial tal que $a'_1 \leq a'_2 \leq \dots \leq a'_n$

Por ejemplo una secuencia de entrada como $\{31, 41, 59, 26, 41, 58\}$ retorna la secuencia $\{26, 31, 41, 41, 58, 59\}$

La clasificación de los datos requiere al menos de dos operaciones fundamentales, la comparación de valores, es decir, el tipo de dato debe permitir determinar si un valor es menor, mayor o igual a otro y mover los valores a su posición ordenada, se debe poder reubicar los elementos.

Estabilidad

Un ordenamiento se considera estable si mantiene el orden relativo que tenían originalmente los elementos con claves iguales. Si se tiene dos registros A y B con la misma clave en la cual A aparece primero que B, entonces el método se considera estable cuando A aparece primero que B en

el archivo ordenado.

La mayoría de los métodos de clasificación simples son estables, pero gran parte de los métodos complejos no lo son. Se puede transformar uno inestable en estable a costa de mayor espacio en memoria o mayor tiempo de ejecución.

Una de las ventajas de los métodos estables es que permiten que un array se ordene usando claves múltiples, por ejemplo, por orden alfabético del apellido, luego el nombre, luego el documento, etc.

In Situ

Los métodos in situ son los que transforman una estructura de datos usando una cantidad extra de memoria, siendo ésta pequeña y constante. Generalmente la entrada es sobrescrita por la salida a medida que se ejecuta el algoritmo. Por el contrario los algoritmos que no son in situ requieren gran cantidad de memoria extra para transformar una entrada.

Fundamental para optimización debido a que utilizar la misma estructura disminuye los tiempos de ejecución, ya que no se debe utilizar tiempo en crear nuevas estructuras, ni copiar elementos de un lugar a otro.

Clasificación interna y externa

Si el archivo a ordenar cabe en memoria principal, entonces el método de clasificación es llamado método interno, por otro lado si ordenamos archivos desde un disco u otro dispositivo, se llama método de clasificación externo. La diferencia radica en que el método interno puede acceder a los elementos fácilmente y de forma aleatoria, mientras que el externo debe acceder a los elementos de forma secuencial o al menos en grandes bloques de datos.

Teoría de la complejidad u orden de crecimiento de algoritmos

Estudia, de manera teórica, los recursos requeridos durante el cálculo para resolver un problema.

Los recursos comúnmente estudiados son el tiempo (mediante una aproximación al número de pasos de ejecución de un algoritmo para resolver un problema) y el espacio (mediante una aproximación a la cantidad de memoria utilizada para resolver un problema). Difiere de la teoría de la computabilidad en que esta se ocupa de la factibilidad de expresar problemas como algoritmos efectivos sin tomar en cuenta los recursos necesarios para ello.

Los problemas que tienen una solución con orden de complejidad lineal son los problemas que se resuelven en un tiempo que se relaciona linealmente con su tamaño.

Hoy en día las máquinas resuelven problemas mediante algoritmos que tienen como máximo una complejidad o coste computacional polinómico, es decir, la relación entre el tamaño del problema y su tiempo de ejecución es polinómica. Éstos son problemas agrupados en el conjunto P. Los problemas con coste factorial o combinatorio están agrupados en NP. Estos problemas no tienen una solución práctica, es decir, una máquina no puede resolverlos en un tiempo razonable.

La complejidad en tiempo de un problema es el número de pasos que toma resolver una instancia de un problema, a partir del tamaño de la entrada utilizando el algoritmo más eficiente a disposición.

Intuitivamente, si se toma una instancia con entrada de longitud n que puede resolverse en n^2 pasos, se dice que ese problema tiene una complejidad en tiempo de n^2 . Por supuesto, el número exacto de pasos depende de la máquina en la que se implementa, del lenguaje utilizado y de otros factores. Para no tener que hablar del costo exacto de un cálculo se utiliza la notación O . Cuando un problema tiene costo en tiempo $O(n^2)$ en una configuración de computador y lenguaje dado este costo será el mismo en todos los computadores, de manera que esta notación generaliza la noción de coste independientemente del equipo utilizado.

Clases de complejidad

Los problemas de decisión se clasifican en conjuntos de complejidad comparable llamados clases de complejidad.

La clase de complejidad P es el conjunto de los problemas de decisión que pueden ser resueltos en una máquina determinista en tiempo polinómico, lo que corresponde intuitivamente a problemas

que pueden ser resueltos aún en el peor de sus casos.

La clase de complejidad NP es el conjunto de los problemas de decisión que pueden ser resueltos por una máquina no determinista en tiempo polinómico. Todos los problemas de esta clase tienen la propiedad de que su solución puede ser verificada efectivamente.

Intratabilidad

Los problemas que pueden ser resueltos en teoría, pero no en práctica, se llaman intratables. Qué se puede y qué no en la práctica es un tema debatible, pero en general sólo los problemas que tienen soluciones de tiempos polinomiales son solubles para más que unos cuantos valores. Entre los problemas intratables se incluyen los de EXPTIME- completo. Si NP no es igual a P, entonces todos los problemas de NP-completo son también intratables.

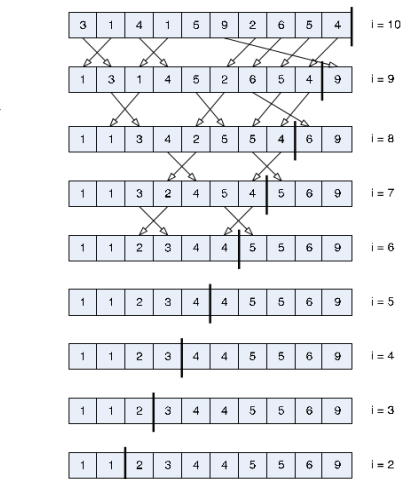
Metodos

Metodos Elementales

Bubble sort

Uno de los más lentos y poco recomendables. Consiste en hacer $N - 1$ pasadas sobre los datos, donde en cada paso, los elementos adyacentes son comparados e intercambiados si es necesario. Durante la primera pasada el elemento más grande va “burbujeando” a la última posición del array. En general, luego de K pasadas por el array, los últimos K elementos del array se consideran bien ordenados por lo que no se los tendrá en cuenta.

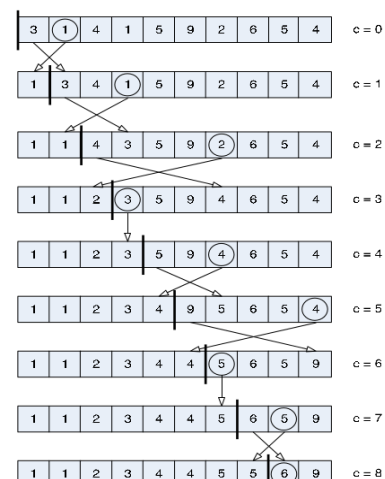
El tiempo de ejecución en el peor de los casos es de $O(n^2)$ y en el mejor es de $O(n)$.



En la figura se puede ver el alcance del bucle externo delimitado con una línea gruesa. Del lado derecho de esa línea los elementos ya están en su posición final.

Selection Sort

Es más rápido que Bubble Sort, y no es mucho más complejo. Comienza buscando el elemento más pequeño del array y se lo intercambia con el que está en la primera posición, luego se busca el segundo elemento más pequeño y se lo coloca en la segunda posición. Se continua con este proceso hasta que todo el array este ordenado.

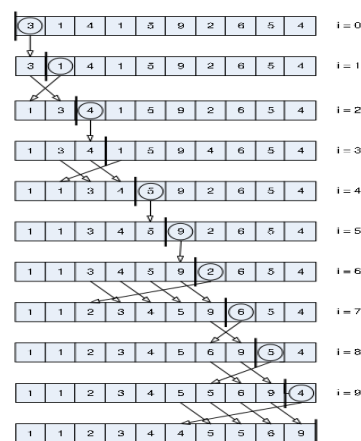


En la figura se puede ver que en el primer paso del algoritmo, luego de buscar en todo el array, se encuentra que el número 1 es el menor de los elementos del array y ya que es el elemento más pequeño le corresponde la primera posición.

Insertion Sort

Es eficiente para ordenar arrays que tienen pocos elementos y están semiordenados y, en la mayoría de los casos, es más rápido que Selection Sort y Bubble Sort. Se basa en la idea del ordenamiento parcial, en el cual hay un marcador que apunta a una posición donde a su izquierda se considera que están los elementos parcialmente ordenados.

El algoritmo comienza eligiendo el elemento marcado para poder insertarlo en su lugar apropiado en el grupo parcialmente ordenado, para eso sacamos temporalmente al elemento marcado y movemos los restantes elementos hacia la derecha. Nos detenemos cuando el elemento a ser cambiado es más pequeño que el elemento marcado, entonces ahí se intercambian el elemento que está en esa posición con la del elemento marcado. El tiempo de ejecución es de $O(n^2)$ y es alcanzable si el array viene ordenado en orden inverso. Si el array está ordenado, el



bucle interno no se ejecuta, tomando un tiempo $O(n)$.

Binary Insertion Sort: tiene en cuenta que la clasificación por inserción usa una búsqueda lineal para encontrar la posición del elemento marcado. Propone usar una búsqueda binaria dentro del grupo de elementos de la izquierda, el cual ya se encuentra ordenado. Con esta modificación requiere solo $O(\log n)$ comparaciones para hallar su posición.

Metodos complejos

Shell Sort

Primero compara los elementos que más lejanos y luego va comparando elementos más cercanos para finalmente realizar un Insertion Sort. Para lograr esto se utiliza una secuencia H_1, H_2, \dots, H_t denominada secuencia de incrementos. Cualquier secuencia es válida siempre que $H_1 = 1$.

Después de ordenar usando la secuencia h_k , se puede afirmar que todos los elementos separados por una distancia de h_k estarán ordenados y por lo tanto se dice que esta h -ordenado.

Una forma de implementar esto es aplicar Insertion Sort dentro de cada subvector h_k independiente. Cuando el $h_k = 1$ el método de ordenamiento es igual a un Insertion Sort. Esto se puede lograr modificándolo, para que incremente o decremente de a h unidades.

Uno puede visualizar el algoritmo Shell sort de la siguiente manera: coloque la lista en una tabla y ordene las columnas (usando un ordenamiento por inserción). Repita este proceso, cada vez con un número menor de columnas más largas. Al final, la tabla tiene sólo una columna. Mientras que transformar la lista en una tabla hace más fácil visualizarlo, el algoritmo propiamente hace su ordenamiento en contexto (incrementando el índice por el tamaño de paso, esto es usando $i += \text{tamaño_de_paso}$ en vez de $i++$).

Por ej, considere una lista de números como [13 14 94 33 82 25 59 94 65 23 45 27 73 25 39 10]. Si comenzamos con un tamaño de paso de 5, podríamos visualizar esto dividiendo la lista de números en una tabla con 5 columnas.

Esto quedaría así:

13 14 94 33 82
25 59 94 65 23
45 27 73 25 39
10

Entonces ordenamos cada columna, lo que nos da

10 14 73 25 23
13 27 94 33 39
25 59 94 65 82
45

Cuando lo leemos de nuevo como una única lista de números, obtenemos

[10 14 73 25 23 13 27 94 33 39 25 59 94 65 82 45]. Aquí, el 10 que estaba en el extremo final, se ha movido hasta el extremo inicial. Esta lista es entonces de nuevo ordenada usando un ordenamiento con un espacio de 3 posiciones, y después un ordenamiento con un espacio de 1 posición (ordenamiento por inserción simple).

Una primera aproximación sugerida originalmente por Shell fue la de usar un incremento un medio menor que el anterior ($N/2$). Si bien fue una mejora al Insertion Sort, esta secuencia no terminó siendo la mejor, ya que en algunos casos se terminaba teniendo un tiempo de ejecución de $O(n^2)$.

Otra aproximación, basada puramente en la experimentación, consiste en dividir cada intervalo por la constante 2,2 en vez de por 2, lo cual consigue un tiempo de ejecución promedio debajo de $O(n^{5/4})$. Una de las secuencias que arroja mejores resultados es la secuencia de Knuth, se obtiene con la siguiente expresión recursiva: $H = 3H + 1$. Cuando el valor inicial de $H = 1$

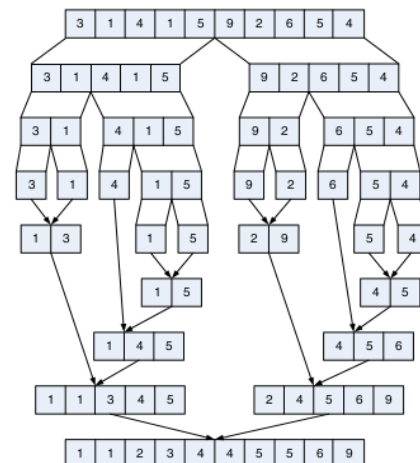
Primero se determina cual va a ser el h inicial. Se usa la formula $H = 3H + 1$. El proceso termina cuando se encuentra un h más grande que el tamaño del array. De ahí en más se usa la inversa de la formula indicada anteriormente $H = (h - 1) / 3$. Cada vez en que iteremos el bucle exterior se disminuye el intervalo usando la formula inversa, y nos detenemos cuando el array fue 1-Ordenado. La eficiencia se considera que este varia entre $O(n^{3/2})$ y $O(n^{7/6})$.

Merge Sort

Es un algoritmo recursivo que utiliza la técnica de divide y vencerás para obtener un tiempo de ejecución $O(n \log n)$ sin importar cual sea la entrada. Se basa en la fusión de dos o más secuencias ordenadas en una única secuencia ordenada. Una de las desventajas de este algoritmo es que requiere de memoria extra proporcional a la cantidad de elementos del array. Es un algoritmo a considerar si estamos buscando velocidad, estabilidad, donde no se tolera un ‘peor de los casos’ y además disponemos de memoria extra. Algo que hace más atractivo a Merge Sort es que suele acceder de forma secuencial a los elementos y es de gran utilidad para ordenar en ambientes donde solo se dispone de acceso secuencial a los registros.

El algoritmo tiene como caso base una secuencia con exactamente un elemento en ella. Y ya que esa secuencia esta ordenada, no hay nada que hacer. Por lo tanto para ordenar una secuencia $n > 1$ elementos se deben seguir los siguientes pasos:

1. Dividir la secuencia en dos subsecuencias más pequeñas.
2. Ordenar recursivamente las dos subsecuencias
3. Fusionar las subsecuencias ordenadas para obtener el resultado final.



Heap Sort

Se basa en una estructura de datos llamada montículo binario (ver apunte de árboles), que es una de las formas de implementar una cola de prioridad. Más precisamente un montículo binario es un árbol binario completo representado mediante un array, en el cual cada nodo satisface la condición de montículo. Para que se cumpla la condición de montículo la clave de cada nodo debe ser mayor (o igual) a las claves de sus hijos, si es que tiene. Esto implica que la clave más grande está en la raíz.

Este algoritmo tiene un tiempo de ejecución que nunca supera $O(n \log n)$ y no requiere espacio de memoria adicional (in situ).

Consiste de dos fases.

1. El array desordenado es transformado en un montículo desde abajo hacia arriba, es decir que empezamos desde las hojas y continuamos hacia la raíz. Cada nodo es la raíz de un submontículo que cumple con la condición de montículo, excepto posiblemente en su raíz. Para eso intercambiamos la raíz con el hijo más grande, este proceso continúa hacia abajo, hasta que llegamos a una posición en donde la condición de montículo se cumple, o hasta que llegamos a una hoja. Este proceso de “filtrado descendente” comienza en $(N/2) - 1$, ya que es esa la posición donde hay un nodo con al menos un hijo.
2. El vector ordenado se construye seleccionando el elemento más grande, quitándolo del montículo y colocándolo en la secuencia ordenada. El elemento más grande del montículo se encuentra en la raíz, y siempre está en la posición 0 del array. Se intercambia repetidamente el elemento más grande del montículo por el próximo elemento de la posición en la secuencia ordenada. Después de cada intercambio, hay un nuevo valor en la raíz del montículo y el nuevo valor es filtrado hacia abajo hasta su posición correcta en el montículo, para que vuelva a cumplir con la condición de montículo.

Comparación con otros metodos

QuickSort suele ser un poco más rápido, debido a algunos factores, pero el peor de los casos el tiempo de quicksort ejecución es $O(n^2)$, lo cual es inaceptable para grandes conjuntos de datos y puede ser provocada deliberadamente dado suficiente conocimiento de la aplicación, creando un riesgo de seguridad.

Por lo tanto, debido a que el límite superior en el tiempo de funcionamiento de heapsort es $O(n \log n)$ y que su límite superior en almacenamiento auxiliar es constante, los sistemas integrados con restricciones de tiempo real o sistemas relacionados con la seguridad a menudo utilizan heapsort.

Heapsort también compite con Merge sort, que tiene los mismos límites de tiempo. Merge sort requiere $\Omega(n)$ espacio auxiliar, pero heapsort requiere solamente una cantidad constante. Heapsort normalmente se ejecuta más rápido en la práctica en máquinas con pequeños o lentos cachés de datos. Por otro lado, Merge sort tiene varias ventajas sobre heapsort:

1. Merge sort sobre arrays tiene mucho mejor rendimiento de la caché de datos, a menudo superando heapsort en las modernas computadoras de escritorio, ya que merge sort accede con frecuencia posiciones de memoria contiguas (buena localidad de referencia), las referencias heapsort se extienden por todo el heap
2. Heapsort no es estable; merge sort es estable.
3. Merge sort paraleliza bien y puede alcanzar la linealidad con una implementación trivial; heapsort no es un candidato obvio para un algoritmo paralelo.
4. Merge sort se puede adaptar para funcionar con listas enlazadas con $O(1)$ espacio adicional. Heapsort puede ser adaptado para operar en las listas doblemente enlazadas con sólo $O(1)$ sobrecarga de espacio adicional.
5. Merge sort pertenece a la clasificación externa; heapsort no lo es. La localidad de las referencias son el problema.

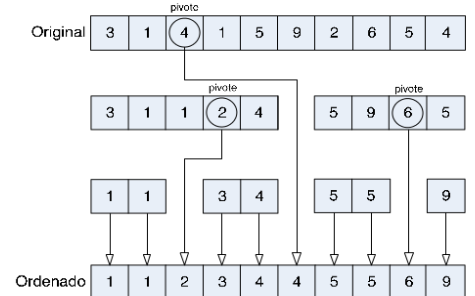
QuickSort

Es el algoritmo que mejor responde en la mayoría de los casos, con un tiempo promedio de $O(n \log n)$ y $O(n^2)$ en el peor de los casos. Es recursivo pero se usan versiones iterativas para mejorar su rendimiento, también es in-situ, ya que usa solo una pila auxiliar. Está basado en la idea de divide y vencerás, en el cual un problema se soluciona dividiéndolo en dos o más subproblemas, resolviendo recursivamente cada uno de ellos para luego juntar sus soluciones para obtener la solución del original. Consiste en los siguientes cuatro pasos:

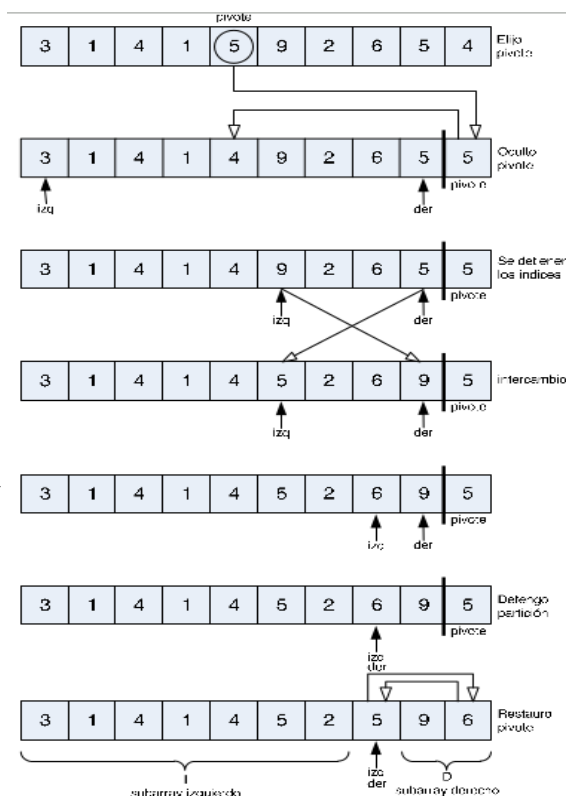
1. Si el número de elementos de S es 0 o 1, terminar.
2. Elegir un elemento v del conjunto S , llamado pivote.
3. Particionar S en dos grupos, donde $I = \{x \in S - \{v\} \mid x \leq v\}$ y $D = \{x \in S - \{v\} \mid x \geq v\}$
4. Devolver el resultado de QuickSort(I), seguido de v y seguido del resultado de QuickSort(D)

Estrategia de partición

Una de las más usadas por ser in-situ y por su eficiencia es la siguiente. Se elige un pivote y se lo intercambia con el último elemento. Se tienen dos índices, *izq* que se mueve hacia la derecha y *der* que se mueve hacia la izquierda. El índice *izq* explora el array de derecha a izquierda y se detiene cuando encuentra un elemento mayor o igual al pivote. El índice *der* hace lo mismo en sentido contrario y se detiene al encontrar un elemento menor o igual al pivote. Los dos elementos en los que se detiene el proceso están mal ubicados y por lo tanto se los intercambia. Continuando de esta forma se tiene la seguridad que los elementos que están a la izquierda del índice *izq* son menores que el pivote y que los situados a la derecha de *der* son mayores.



En el ejemplo se elige un pivote cualquiera, en este caso el 4, luego se agrupan los elementos que son menores que el pivote en el subgrupo izquierdo, y los mayores en el derecho. Luego se aplica QuickSort a los subgrupos resultantes. Es importante remarcar que la posición donde va el pivote luego de realizar la partición, es su posición definitiva en el array ordenado.



Cuando los índices se encuentran el proceso se detiene, y se intercambia el ultimo elemento con el elemento al que apunta el índice izq. Ahora el array esta listo para llamar a QuickSort con los dos subconjuntos resultantes, es decir que se llamará a QuickSort(I) y a QuickSort(D).

Vale la pena observar que el algoritmo no necesita memoria auxiliar más allá de los índices, y que cada elemento es comparado con el pivote exactamente una vez.

Mejor de los casos

El mejor de los casos se obtiene cuando el proceso de particionamiento determina dos subsecuencias con cantidad de elementos iguales, es decir de $N / 2$. Este caso produce un tiempo de ejecución cercano a $O(n \log n)$.

Peor de los casos

El peor de los casos se da cuando el proceso de particionamiento produce una subsecuencia con $N-1$ elementos y otra con un solo elemento. Si esto ocurre en cada paso del algoritmo, dado que el particionamiento cuesta un tiempo N , el tiempo de ejecución es de $O(n^2)$.

Seleccionando el pivote

Una mala elección sería elegir el primer elemento como pivote, que si bien sería aceptable en una entrada aleatoria nos llevaría al peor caso si la entrada esta ya ordenada o en orden inverso.

Una mejor aproximación sería elegir el elemento central como pivote, es decir en la posición $(izq + der) / 2$ del array. Otra elección que es mucho mejor es la de elegir 3 valores, por ej el primero, el del medio y el ultimo, y agarrar el menor de esos 3 como pivote.

En Resumen

| <i>Nombre</i> | <i>Mejor Caso</i> | <i>Caso medio</i> | <i>Peor Caso</i> | <i>Estable</i> | <i>Comentarios</i> |
|-----------------------|-------------------|-------------------|------------------|----------------|---|
| <i>Bubble sort</i> | $O(n)$ | $O(n^2)$ | $O(n^2)$ | Si | El más lento de todos. Uso pedagógico. Consiste en hacer $N - 1$ pasadas sobre los datos, donde en cada paso, los elementos adyacentes son comparados e intercambiados si es necesario. |
| <i>Selection sort</i> | $O(n^2)$ | $O(n^2)$ | $O(n^2)$ | Si | Apto si queremos consumir siempre la misma cantidad de tiempo. Comienza buscando el elemento más pequeño del array y se lo intercambia con el que está en la primera posición. Se continúa con este proceso hasta que todo el array esté ordenado. Muy bueno para archivos con registros muy grandes y claves muy pequeñas, puesto que la mayoría de los elementos se mueven a lo sumo una vez. |
| <i>Insertion sort</i> | $O(n)$ | $O(n)$ | $O(n^2)$ | Si | Conveniente cuando el array está casi ordenado y tiene pocos elementos. Se basa en la idea de ordenamiento parcial, en el cual hay un marcador que apunta a una posición donde a su izquierda se considera que están los elementos parcialmente ordenados, es decir ordenados entre ellos pero no en sus posiciones finales. El algoritmo comienza eligiendo el elemento marcado para poder insertarlo en su lugar apropiado en el grupo parcialmente ordenado, para eso sacamos temporalmente al elemento marcado y movemos los restantes elementos hacia la derecha. Nos detenemos cuando el elemento a ser cambiado es más pequeño que el elemento marcado, entonces ahí se intercambian el elemento que está en esa posición con la del elemento marcado. |
| <i>Shell sort</i> | $O(n^{5/4})$ | $O(n^{3/2})$ | $O(n^2)$ | No | Dependiente de la secuencia de incrementos. Evita la gran cantidad de movimientos. Primero compara los elementos más lejanos y luego va comparando elementos más cercanos para finalmente realizar un Insertion sort. |

| | | | | | |
|-------------------|---------------|---------------|---------------|----|---|
| <i>Merge sort</i> | $O(n \log n)$ | $O(n \log n)$ | $O(n \log n)$ | Si | Adecuado para trabajos en paralelo. Requiere memoria extra proporcional a la cantidad de elementos del array (contra). Dividir la secuencia en dos subsecuencias más pequeñas, ordenarlas recursivamente, fusionar las subsecuencias ordenadas para obtener el resultado final. |
| <i>Heap sort</i> | $O(n \log n)$ | $O(n \log n)$ | $O(n \log n)$ | No | El método acotado en el tiempo muy utilizado para grandes volúmenes de datos. No requiere memoria adicional. Primero, el array desordenado es transformado en un montículo desde abajo hacia arriba. Cond. de montículo: la clave de cada nodo debe ser mayor a las claves de sus hijos, la clave más grande es la raíz. Segundo, una vez construido el montículo, se construye el vector ordenado seleccionando el elemento más grande, quitándolo del montículo y colocándolo en la secuencia ordenada. |
| <i>Quick sort</i> | $O(n \log n)$ | $O(n \log n)$ | $O(n^2)$ | No | El más rápido en la práctica. Implementado en gran cantidad de sistemas. Es in situ ya que solo usa una pila auxiliar. Número de elementos es 0 o 1, termina. Elegir un pivote. Particionar en dos grupos: mayores que el pivote y menores. Repetir hasta tener ordenado. |

Algoritmo de Huffman

Concepto

Define un código variable para cada caracter sin la necesidad de utilizar un prefijo. De esta manera es posible que, aquellos caracteres que son más frecuentes dentro de un alfabeto, se codifiquen con la menor cantidad de bits posibles mientras que aquellos que no sean tan frecuentes podrán tener una codificación un poco más amplia, pero siempre tendiendo a ser la menor posible.

Para ello, Huffman hace uso de una tabla de frecuencias donde se guardan los caracteres empleados en el texto y la cantidad de veces que son utilizados y de un árbol binario como estructura de datos. El código generado por el algoritmo, está basado en el código Morse y define una longitud variable de codificación basada en valores estadísticos.

Formato de las estructuras

Tabla de Frecuencias

| Char | Frec | Codigo | Bits | Dir tree | Total (frec*bits) |
|-------|------|--------|------|----------|-------------------|
| X | 5 | 001 | 3 | | 15 |
| Total | | | | | 15 |

Estructura del Nodo

| | |
|---------------|----------|
| PADRE | |
| Frecuencia | |
| Identificador | |
| Hijo Izq | Hijo der |

El algoritmo se basa en :

1. Hallar los dos caracteres con la frecuencia más baja de la tabla de frecuencias. Si un caracter tiene frecuencia cero se ignora. Si dos o más caracteres tienen igual frecuencia se eligen dos cualesquiera.
2. Combinar los dos caracteres en un árbol binario. Cada hoja de este árbol contiene uno de los caracteres que se quiere codificar en el campo identificador, la frecuencia obtenida de la tabla de frecuencias en el campo frecuencia, nil en los campos hijo izq. e hijo der. y la dirección de la raíz del árbol en el campo padre.

La raíz del árbol (o el padre de las hojas) contiene blancos o nulos en el campo identificador, la sumatoria de las frecuencias de sus hijos en el campo frecuencia y la dirección de cada hijo en los campos hijo der. e hijo izq. respectivamente.

3. Ejecutar nuevamente el punto 2). Esta vez entra en la comparación el valor de la frecuencia del árbol recientemente creado en lugar de las frecuencias de sus caracteres
Este proceso se ejecuta hasta que quede sólo un árbol para todos los caracteres de la tabla de frecuencias y que de él pueda ser leído un char directamente.

Observaciones :

- La optimización del algoritmo implica ocupar el menor espacio posible, por lo cual hay que tratar de disminuir la profundidad de los nodos maximales, es decir, la longitud de paso máximo.
- El problema del prefijo, que se presenta en las codificaciones de longitud variable queda solucionado en el árbol de Huffman al ser el nodo maximal el que contiene el caracter. De esa manera, al detectar un nodo maximal, es decir Hijo Der. = Hijo Izq. = nil puedo asegurar que en ese momento de la lectura de bits se produce el fin de la codificación de un caracter y que con el siguiente bit comenzará el próximo caracter.

Codificación y Decodificación

Una vez que se obtuvo la tabla de frecuencias y de esta, el árbol binario estamos en condiciones de **decodificar** un tren de bits. Por ej 000 010 111 10010

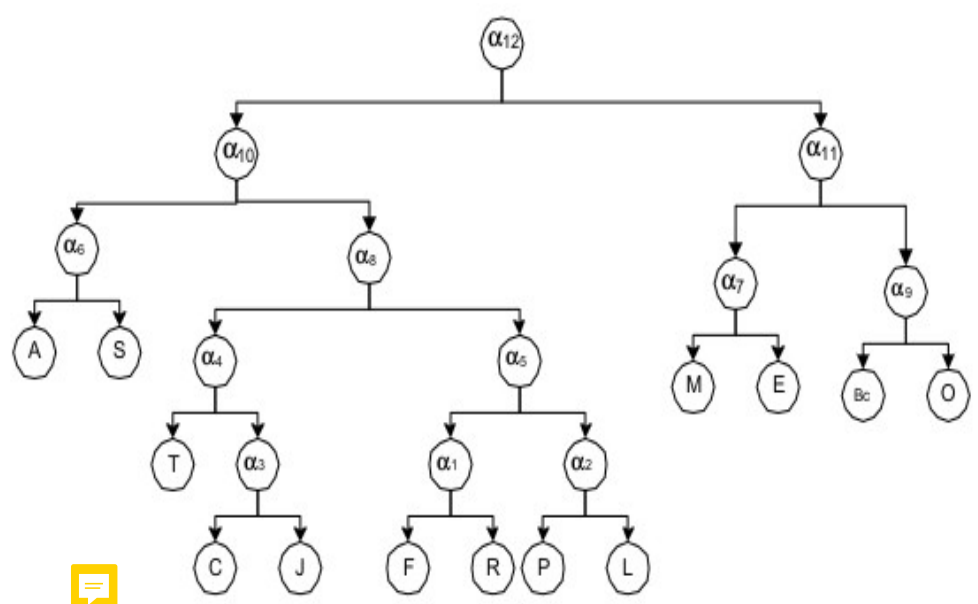
Por cada bit leído se deberá recorrer el árbol binario comenzando por la raíz y visitando los subárboles izquierdo o derecho según corresponda, de acuerdo a si el bit leído es un 0 ó un 1 respectivamente.

Para **codificar** un caracter, será necesario guardar la dirección que le fue asignada al nodo al haber sido dado de alta en el árbol. Una solución sería la de agregar otra columna en la tabla de frecuencias que contenga las direcciones físicas que ocupan los nodos que contienen los caracteres. Obtenida la dirección del nodo que contiene el caracter que se quiere codificar, se recorre el árbol en forma ascendente mediante la dirección guardada en el campo que apunta al padre del nodo. Al bajar de nivel, en dirección a la raíz del árbol, se recupera un 0 ó un 1 en función de si el hijo es izquierdo (se codifica un 0) o si es derecho (un 1). Cada bit obtenido, es ingresado en una **pila**. Este proceso se ejecuta hasta llegar a la raíz del árbol (campo PADRE = nil). En este momento se tiene la codificación del caracter invertida en la pila por lo que al vaciar la pila, se obtendrá la codificación binaria del caracter.

Ejemplo

“Tomemos como ejemplo esta frase”

| Status | Char | Freq | Code | Dir_tree |
|--------|---------------|------|------|-----------------------|
| x | Blanco | 5 | | dir (Bco) |
| x | O | 5 | | dir (O) |
| x | M | 4 | | dir (M) |
| x | E | 4 | | dir (E) |
| x | A | 3 | | dir (A) |
| x | S | 3 | | dir (S) |
| x | T | 2 | | dir (T) |
| x | C | 1 | | dir (C) |
| x | J | 1 | | dir (J) |
| x | P | 1 | | dir (P) |
| x | L | 1 | | dir (L) |
| x | F | 1 | | dir (F) |
| x | R | 1 | | dir (R) |
| x | α_1 | 2 | | dir (α_1) |
| x | α_2 | 2 | | dir (α_2) |
| x | α_3 | 2 | | dir (α_3) |
| x | α_4 | 4 | | dir (α_4) |
| x | α_5 | 4 | | dir (α_5) |
| x | α_6 | 6 | | dir (α_6) |
| x | α_7 | 8 | | dir (α_7) |
| x | α_8 | 8 | | dir (α_8) |
| x | α_9 | 10 | | dir (α_9) |
| x | α_{10} | 14 | | dir (α_{10}) |
| x | α_{11} | 18 | | dir (α_{11}) |
| | α_{12} | 32 | | dir (α_{12}) |



Hashing

En las técnicas de búsqueda basadas en estructuras arbóreas, siempre es necesario comparar una cierta cantidad de claves hasta encontrar la buscada. Lo óptimo, sería minimizar la cantidad de comparaciones, evitando aquellas que se consideran innecesarias o redundantes con lo cuál se disminuirían los tiempos de recuperación. De esta manera es posible pensar en establecer una relación directa entre la dirección de los datos y el valor de la clave en lugar de obtener la dirección en función de la posición relativa de la clave respecto de las restantes.

Supongamos que el valor de la clave key es un número entero de tres dígitos. Puede definirse entonces una tabla, `table()`, de 999 elementos, donde cada uno guardará la dirección que le corresponda al valor absoluto de la clave tomado como subíndice.

Si bien este algoritmo es óptimo en términos de tiempos de recuperación, por no realizar ninguna comparación, es impracticable pues es imposible subordinar el valor de una clave a la cantidad máxima de elementos de una tabla.

Lo ideal sería hallar algún método que permita hallar la dirección de un registro en función de su clave, con la menor cantidad de comparaciones y ocupando el menor espacio posible.

Este método se denomina función de hashing, la cual identificaremos con `hash()`, y puede definirse entonces como la función que aplicada a una clave devuelve el subíndice de la tabla (donde se encuentra la dirección con los datos del registro buscado).

Para minimizar el desperdicio, lo óptimo sería conocer previamente la cantidad máxima de registros que puede tener el archivo, `maxrec`, y dimensionar la tabla con un número máximo de elementos, `maxtab`, igual al primer número primo que sea mayor en valor absoluto a `maxrec`. Ejemplo 4 : si `maxrec = 300 => maxtab = 307`

El principal problema inherente a este método se produce cuando el valor que devuelve la función de hashing `hash()` es el mismo para dos o más claves iguales, lo que se define como colisión. Más precisamente:

- Colisión : Sean `k1` y `k2` claves pertenecientes a los registros `r1` y `r2` respectivamente, con $k1 \neq k2$. Y sea `h(k)` la función de hash. Si $h(k1)=h(k2)$ entonces decimos que se produce una colisión.

La existencia de lo que llamamos colisiones es crucial debido a que, dado que la cantidad de claves posibles siempre es mayor que la cantidad de posiciones en la tabla, entonces siempre se producirán colisiones. Si bien pueden plantearse métodos que intenten reducir la probabilidad de colisión, la misma nunca llegará a ser cero. Lo que es un concepto en sí matemático, probabilístico ($\forall \text{hash}(\text{key}): P(\text{colisión}) > 0$), tiene una implicancia fundamental desde el punto de vista computacional, algorítmico: que inevitablemente se producirá una colisión al cabo de un tiempo finito.

Generalizando podríamos decir que: una función de hash es eficiente cuando minimiza el número de colisiones y ocupa los elementos de la tabla de manera uniforme. Si bien cuanto mayor sea el número de elementos de la tabla, menor será la probabilidad de colisión, el dimensionamiento desmesurado de la tabla ocasionaría un desperdicio de espacio no deseado y otra vez se deberá analizar cada situación en particular, en términos de tiempo de procesamiento y espacio utilizado.

Funciones de hash

El principal objetivo de una función de hash es minimizar la probabilidad de colisión sin ocupar memoria innecesariamente. Existen distintos métodos pero en la práctica nunca se utiliza uno en particular sino una combinación de varios de acuerdo a la aplicación que se esté analizando.

Método de la división o del módulo

Consiste en obtener un valor entre 1 y `maxtab` resultante del resto de la división del valor numérico de la clave y `maxtab`, es decir que se utiliza la función módulo como función de hash. Ejemplo 6 :

$\text{hash}(\text{key}) = \text{key} \bmod 947 \Rightarrow \text{hash}(2866) = 25$

Método del cuadrado medio.

Consiste en multiplicar el valor numérico de la clave por sí mismo y obtener un valor entre 1 y maxtab de los dígitos medios del valor obtenido. Ejemplo 7 : si maxtab = 947 y se quiere hallar el valor de la clave 2866

$\text{hash}(\text{key}) = \text{números medios de } (\text{key}^2) \Rightarrow$

$\text{hash}(2866) = 8213956 \Rightarrow \text{hash}(2866) = 139$

Si maxtab = 1013 podría tomarse indistintamente

$\text{hash}(2866) = 2139$ o $\text{hash}(2866) = 1395$

Método de dobles.

Consiste en dividir la cantidad de dígitos del valor numérico de la clave en dos o más partes iguales y realizar la operación or exclusivo con el valor binario de cada una de esas partes. Ejemplo 8 : si maxtab = 947 y se quiere hallar el valor de la clave 2866

Se divide la clave en dos partes: $a_1 = 28$ y $a_2 = 66$.

El valor binario de a_1 es = 11100

El valor binario de a_2 es = 1000010

La operación or exclusivo aplicada sobre los valores binarios de a_1 y a_2 dará un valor de subíndice en binario = 1011110 que pasado al sistema decimal es igual a : $\text{hash}(2866) = 94$

Clustering (relacionado a colisiones)

Cuando un elemento tiene una probabilidad de ser ocupado mucho mayor que otro.

Tratamiento de las colisiones

Rehashing o direccionamiento abierto (Solucion estatica).

Se define rehashing como la utilización de una segunda función de hash hasta que se encuentre una posición libre.

Varias soluciones:

1. **Trivial:** es la llamada solución lineal; si la posición de la tabla de direcciones, $\text{table}(\text{dir})$, en el valor de subíndice obtenido al aplicar la función de hash a una clave, dir , está ocupado, se ocupa la siguiente posición libre de la tabla. $\text{rhash}(i) = (i+1) \% \text{maxtab}$
El proceso consiste entonces en dada una clave key , se le aplica la función de hashing $\text{hash}()$ y se obtiene un valor de subíndice dir , si el elemento de la tabla $\text{table}(\text{dir})$ está ocupado por otra clave, se aplicará una función de reasignación $\text{rhash}()$ al valor dir para obtener un nuevo valor de subíndice. Este proceso es recursivo hasta hallar un elemento libre en la tabla.
2. **Generalizacion de la solucion lineal:** $\text{rhash}(\text{dir}) = (\text{dir} + c) \bmod \text{maxtab}$ donde c y maxtab son números primos.
3. **Doble hashing (para evitar el clustering):** Este método utiliza en la función de reasignación otra función que toma como argumento el valor de la clave. Es decir : $\text{hash}(\text{key}) = \text{dir}$. Si $\text{table}(\text{dir})$ es distinto de vacío, es decir está ocupada por otra clave, $\text{rhash}(\text{dir}) = (\text{dir} + f(\text{key}) \bmod (\text{maxtab}))$ donde $f(\text{key})$ es una función definida por el programador que toma como argumento el valor numérico de la clave.
4. **Hacer depender la función de reasignación del número de veces que es aplicada.** De esta manera, $\text{rhash}(\text{dir}, \text{arg1})$ tendrá dos argumentos. $\text{rhash}(\text{dir}, i) = (i+1) * \text{dir} \bmod \text{maxtab}$ donde i es una variable inicializada en 1

El principal problema del método de direccionamiento abierto, es que se basa en el uso de una estructura estática, es decir una tabla, la cuál debe ser dimensionada previamente y que en ocasiones puede resultar chica, en cuyo caso se deberá crear una tabla más grande y reorganizar todos los punteros en función del nuevo valor que tome maxtab. Otro problema es la dificultad que se presenta al dar de baja un elemento de la tabla. Este tipo de inconvenientes podría solucionarse con el uso de marcas de estado pero el proceso de recuperación sería cada vez más complejo y se

perdería de vista los objetivos principales que se plantearon en un comienzo respecto de disminuir la cantidad de accesos y disminuir el espacio utilizado por las estructuras de datos auxiliares.

Observaciones:

Debe prestarse especial atención en el diseño del proceso de reasignación pues puede darse el caso de que un algoritmo busque eternamente un elemento disponible en la tabla o que informe que no hay más espacio disponible cuando en realidad sí hay elementos vacíos. Un ejemplo claro de una función de reasignación que pueda llegar a conclusiones equivocadas es la función $rhash(dir) = dir + 2$. En este caso, llevado al extremo, puede darse que todos los elementos de la tabla con subíndice par estén ocupados, que los impares estén vacíos y la función indique que no existe más espacio disponible.

Chaining o encadenamiento. (soluciones dinámicas)

Consiste en enlazar entre sí los distintos pares de clave/puntero que colisionen entre sí, de forma tal de agruparlos. El encadenamiento puede darse en forma interna dentro de la misma tabla de hash, o también utilizando algún tipo de estructura auxiliar, lo que se conoce como “separate chaining”.

Hashing into buckets (separate chaining)

Permite resolver el problema de las colisiones a través de listas linkeadas que agrupan los registros cuyas claves randomizadas generan la misma dirección. Básicamente el proceso tiene como estructuras de datos una tabla principal, conocida como área base, y una lista linkeada, área de overflow, cuyo registro tiene tres campos. En uno se guarda la clave, en el otro se guarda la dirección en donde residen los datos y en el tercer campo se guarda la dirección del próximo registro con igual valor inicial de subíndice dir.

Procedimiento:

Cuando una función de hashing aplicada a una clave devuelve un elemento de tabla que fue ocupado previamente por otra clave, el proceso pide entonces una dirección de memoria para guardar los datos de la segunda clave y establece la relación mediante un link entre la primer clave, cuyos datos se encuentran en la tabla en el área base, con la segunda clave, situada en la lista linkeada en el área de overflow. Si se quisiera dar de baja el nodo, simplemente habría que dar una baja en la lista.

La principal desventaja de este procedimiento es que se utiliza espacio para los punteros, pero comparado con el método de direccionamiento abierto, es posible construir una tabla principal más pequeña con el consecuente ahorro de espacio y sin el problema que se presenta al tener todas las entradas de la tabla ocupadas. En este caso, deberá medirse la eficiencia del procedimiento a través de la cantidad de accesos que deberán realizarse, es decir la cantidad de nodos a recorrer en la lista, para hallar una clave dada. Si el número de nodos es muy grande, se perderá mucho tiempo en recuperar un dato, con lo cuál habría que rediseñar la función de hashing inicial o el tamaño de la tabla principal.

Hash Dinámico

Las tablas hash se presentaron como una alternativa hacia las estructuras tipo árbol ya que permitían el almacenamiento de grandes volúmenes de información y algoritmos eficientes para la administración sobre estas estructuras (inserción, eliminación y búsqueda).

Sin embargo, presentan 2 grandes problemas:

1. No existen funciones hash perfectas que permitan asegurar que por cada transformación de un elemento habrá una única correspondencia en la clave que contiene este elemento.
2. Son estructuras estáticas que no pueden crecer ya que necesitan un tamaño fijo para el funcionamiento de la estructura.

Para solucionar el segundo problema se implementa la utilización de métodos totales y métodos parciales. Convirtiendo la tabla hash en una estructura dinámica capaz de almacenar un flujo de información y no una cantidad fija de datos.

Métodos Totales

Método de las expansiones totales

El método de las expansiones totales consiste en realizar una duplicación del tamaño del arreglo establecido para realizar la tabla hash, esta expansión se ejecuta cuando se supera la densidad de ocupación. Así si se tiene una tabla hash de tamaño N, al realizar la expansión total se obtendrá una tabla hash de 2N, al realizar una segunda expansión se obtendrá una tabla hash de 4N, al realizar una tercera expansión se obtendrá una tabla hash de 8N y en general el tamaño de la tabla para una i-ésima expansión se define como aparece a continuación:

$$T = 2^i * N$$

Dónde: N: Tamaño de la Tabla. i: Número de expansiones que se quieren realizar. T: Nuevo tamaño de la Tabla.

La densidad de ocupación se define como el cociente entre el número de registros ocupados y el número de registros disponibles; así se tiene que:

$$\rho o = (ro / rd) * 100$$

Dónde: ro: Registros Ocupados. rd: Registros Disponibles. po: Densidad de Ocupación.

Cada vez que se pretende insertar un elemento es necesario calcular la densidad de ocupación, si se supera esta densidad se procede a implementar la expansión. Al realizar cada una de las expansiones es necesario volver a implementar la función hash para cada uno de los registros almacenados en la tabla y volver a insertarlos de nuevo en la tabla.

Método de las reducciones totales

Este método surge como una consecuencia del método de expansiones totales presentado anteriormente. En este método la densidad de ocupación disminuye de tal manera que acepta una reducción del tamaño de la tabla hash a la mitad. Así si se tiene una tabla hash de N, la primera reducción dará como resultado la N/2, la segunda reducción dará como resultado N/4, la tercera reducción dará N/8 y la i-ésima reducción dará como resultado:

$$T = N / 2^i$$

Dónde: N: Tamaño de la Tabla. i: Número de expansiones que se quieren realizar. T: Nuevo tamaño de la Tabla.

Para realizar una reducción la densidad de ocupación se debe disminuir a un valor menor al rango establecido y los registros se deben eliminar de tal manera que los registros resultantes se puedan ingresar en una tabla hash que posea la mitad del tamaño de la tabla original. Cada vez que se implementa una reducción es necesario volver a utilizar la función hash con cada uno de los registros almacenados.

Métodos Parciales

Método de las expansiones parciales

El método de las expansiones parciales consiste en incrementar en un 50% el tamaño del arreglo establecido para realizar la tabla hash, esta expansión se ejecuta cuando se supera la densidad de ocupación. Así si se tiene una tabla hash de tamaño N, al realizar la expansión parcial se obtendrá una tabla hash de 1.5 N, al realizar una segunda expansión se obtendrá una tabla hash de 2.25 N, al realizar una tercera expansión se obtendrá una tabla hash de 3.375 N y en general el tamaño de la tabla para una i-ésima expansión se define como:

$$T = \downarrow ((1.5)^i * N)$$

Dónde: N: Tamaño de la Tabla. i: Número de expansiones que se quieren realizar. T: Nuevo tamaño de la Tabla.

Cada vez que se pretende insertar un elemento es necesario calcular la densidad de ocupación, si se supera esta densidad se procede a implementar la expansión. Al realizar cada una de las expansiones es necesario volver a implementar la función hash para cada uno de los registros

almacenados en la tabla hash y volver a insertarlos de nuevo en la tabla.

Método de las reducciones parciales

Este método surge como una consecuencia del método de expansiones parciales. En este método la densidad de ocupación disminuye de tal manera que acepta una reducción del tamaño de la tabla hash al 50%. Así si se tiene una tabla hash de N, la primera reducción dará como resultado la 0.5 N, la segunda reducción dará como resultado 0.25 N, la tercera reducción dará 0.125 N y la i-ésima reducción dará como resultado:

$$T = \uparrow ((0.5)^i * N)$$

Dónde: N: Tamaño de la Tabla. i: Número de reducciones que se quieren realizar. T: Nuevo tamaño de la Tabla.

Para realizar una reducción la densidad de ocupación debe disminuir a un valor menor al rango establecido y los registros se deben eliminar de tal manera que los registros resultantes se puedan ingresar en una tabla hash que posea la mitad del tamaño de la tabla original. Cada vez que se implementa una reducción es necesario volver a utilizar la función hash con cada uno de los registros almacenados.

B Tree VS Hash Table

B Tree

- Supports equality and range searches, multiple attribute keys and partial key searches
- Either a separate index or the basis for a storage structure
- Responds to dynamic changes in the table

Hash Table

- Does not support range search
 - Since adjacent elements in range might hash to different buckets, there is no way to scan buckets to locate all search key values v between v1 and v2
- Although it supports multiple attribute keys, it does not support partial key search
 - Entire value of v must be provided to h
- Dynamically growing files produce overflow chains, which negate the efficiency of the algorithm
- The optimizer cannot use a hash index to speed up ORDER BY operations. (This type of index cannot be used to search for the next entry in order.)

Apuntes incluidos:

- Arboles del Ing Reinoso
- Metodos de Clasificación del Ing Reinoso
- Huffman del Ing Zaffaroni
- Huffman caso practico del Ing Zaffaroni
- Metodos de acceso del Ing Zaffaroni
- Wikipedia

Apuntes no Incluidos

- Sort Topologico – Ing Zaffaroni
- Heapsort – Ing Zaffaroni
- Ordenamiento y busqueda – Ing Zaffaroni
- Arboles – Ing Zaffaroni

Unidad III

Arquitectura de aplicaciones

| Tipo de Arquitectura | Definición | Ventajas | Desventajas |
|----------------------|--|--|---|
| Monocapa | Sistemas monousuarios, no existe concurrencia de acceso a la información. Denominados Desktop, consisten en una aplicación en la PC del usuario, compuesta por un instalador y un ejecutable que permite acceder a la misma. No posee conectividad. Interactúa únicamente con el Sistema Operativo. Lógica de negocio centralizada en el código de la aplicación instalada. Persistencia: los datos se guardan en archivos propios de la aplicación, se almacenan en cada máquina. | <ul style="list-style-type: none"> -Simple de construir. App conformada por un único componente. -Facilidad en el rastreo de bugs: los problemas se encuentran en el único componente que conforma la app. -Acceso directo a datos. -Fácil puesta en marcha: instalación. | <ul style="list-style-type: none"> -Integración de datos entre apps de distintas PCs es complicada, es necesario realizar una consolidación de los archivos persistidos en cada PC. -Soluciones a problemas de integración: 1)Consolidación de datos a mano. 2)Utilizar procesos batch automáticos, corriendo fuera de hora, que integren todos los datos y verifiquen su integridad. -Acoplados al SO para el cual fueron desarrollados. |
| Cliente – Servidor | Compuestas por dos módulos: 1)App Cliente: componente Desktop, instalado en la PC de cada usuario. 2)App Servidor: servidor centralizado, con una BD que hace de repositorio de la app. BD es un programa corriendo en el servidor con una IP y puerto de escucha. App cliente se conecta a la BD directamente mediante un conector o driver de BD. La lógica de negocio puede estar distribuida en dos partes: 1)Código de la app Desktop del cliente. 2)BD manifestada como tablas, stored procedures, views, triggers, constraints. | <ul style="list-style-type: none"> -Al tener BD: concurrencia de consultas, carga de datos simultánea, consulta filtrada y performante sobre gran volumen de datos, integridad de los datos. -Eliminados los problemas de integración de datos. Trabajar varios usuarios a la vez y ver cambios producidos de forma inmediata. -Se delega complejidad del sistema en BD. -Usuarios situados en diferentes lugares físicos y acceder al mismo servidor de BD. | <ul style="list-style-type: none"> -PC cliente debe tener acceso a la IP del servidor, implicando que todas las PC cliente deben encontrarse en la misma red que el servidor. Soluciones: 1)todas las PC conectadas a la misma red LAN. 2)Acceso a través de Internet (VPN). -Comunicación es binaria por un puerto no Standard. Proxy suele bloquear esta comunicación por: 1)Bloqueo por puertos: BD no suelen tener sus servicios de escucha en los puertos que normalmente se encuentran abiertos. 2)Bloqueo por protocolo: la información que viaja en esta comunicación no pertenece al protocolo HTTP, por lo tanto también puede ser identificada y bloqueada. -Requiere instalación: app Cliente suele requerir instalación sobre la PC del usuario. Puede generar inconvenientes si el usuario no posee los permisos necesarios. -App cliente acoplada al SO. -Aumenta la complejidad de actualización de la app. Cada vez que surjan cambios será necesario instalar una nueva versión de la app en cada PC. Soluciones: 1)Crear un módulo de detección de actualizaciones en forma automática(SP); no debería permitir que el usuario utilice la app por mucho tiempo con una versión discontinuada. 2)Crear un sitio del cual se pueda descargar la última versión de la app, con el riesgo de que los usuarios no instalen la nueva. 3)Enviar personas a cada PC para instalar y |

| | | | |
|-------------------------------|---|---|--|
| | | | <p>configurar la nueva versión.</p> <ul style="list-style-type: none"> -Aumento de complejidad en rastreo de bugs: error en la BD o en la app cliente. - App dividida en dos partes, complica programación. -App cliente puede ocupar un tamaño significativo, complicando aún más la actualización. -Problemas de seguridad: cliente podría alterar el código de la app para evitar validaciones de dominio. Una mala administración de permisos de usuario podría dejar expuesta toda la info almacenada en la BD. |
| Multicapa | <p>Compuestas por tres módulos: 1-App cliente: puede ser de dos formas: 1)Desktop: un cliente instalable en la PC. 2)Web: un cliente accedido a través de un Browser de Internet. 2-Servidor de apps: app con código que se ejecuta en el server. 3-BD: En este modelo es el servidor de apps la que se conecta a la BD. Conexión a través de un driver de BD. La app cliente no accede directamente a los datos sino que lo hace a través de la app servidor.</p> | | |
| Multicapa con cliente Desktop | <p>El cliente se considera liviano porque el usuario no posee la app entera en su poder, sino solo una cáscara del sistema. Muchas reglas de negocio pueden ser ahora implementadas en la app servidor. Nueva comunicación entre app cliente y servidor:</p> <ul style="list-style-type: none"> -Comunicación tradicional a bajo nivel, mediante la utilización de sockets. -Comunicación mediante RPC(Remote Procedure Call) o RMI(Remote Method Invocation): proveen mecanismos automáticos para la ejecución de código remoto, junto con la serialización de los parámetros de entrada y salida. -Uso de WebServices: app Cliente puede utilizar el protocolo HTTP para intercambiar mensajes con la app servidor, mediante archivos en formatos XML. | <ul style="list-style-type: none"> -App Cliente con interfaz gráfica más robusta, interactiva y performante que en Web. -Lógica de la pantalla no tan acoplada a la fuente de origen de datos. -Elimina problemas de seguridad: la app cliente no posee acceso directo a la BD y las validaciones de negocio mas importantes pueden ser programadas en la app servidor. - App cliente más liviano al no contar con toda la lógica de negocio y por ello más fácil la actualización de versiones. | <ul style="list-style-type: none"> -Sigue siendo necesario desarrollar un módulo de actualización para la instalación de futuras versiones. -Lógica de app dividida en 3 partes: complica la detección y arreglo de fallas. -Desarrollo más complicado: es necesario determinar que librerías de código utilizará el cliente y cuáles el servidor. -Interacción de la capa Servidor en los pedidos a la BD puede representar un cuello de botella en la performance de la app. -Si no se utiliza WebServices: mismos problemas de conectividad que la arquitectura Cliente-Servidor. |
| Multicapa con cliente Web | <p>Se compone de 3 partes:</p> <ul style="list-style-type: none"> -App cliente ejecutando en un Browser de Internet. Código de la app podrá estar compuesto por diversos elementos, dependiendo de la plataforma de desarrollo elegida. -App server: debe poder lidiar con el protocolo HTTP. -BD. <p>La app cliente es descargada y ejecutada dentro de un Browser de Internet, no requiere instalación. Browser candidato ideal para solucionar problemas de conectividad, acceso y acoplamiento. Principales características: existen versiones de Browser para todos los SO; el protocolo HTTP es de acceso libre. La app cliente se comunica con la app servidor por HTTP utilizando el puerto 80. La app servidor se comunica con la BD a través de un driver de BD.</p> | <ul style="list-style-type: none"> -Eliminan el problema de acceso, cualquier usuario con acceso a Internet puede acceder a la app. -Eliminan los problemas de instalación de la app y al no requerir instalación, el usuario siempre accede a la última versión de la app. -Se eliminan los problemas de conectividad pues se accede por internet. -Al no instalarse se eliminan los problemas de acoplamiento al SO. -Son ágiles para un usuario casual, pues no requieren instalación y pueden ser accedidas desde cualquier PC. -Elimina problemas de seguridad: el usuario solo posee una parte de la app. | <ul style="list-style-type: none"> -Es necesario tener el Browser perfectamente configurado: permitir la apertura de ventanas emergentes, permitir la ejecución de JavaScript, permitir la ejecución de Plugins, contar con los Plugins en la versión correcta, tener habilitado un nivel de seguridad que permite acceder a la app. -EL protocolo HTTP añade un gran overhead al tamaño de los mensajes transmitidos (gran volumen de datos). -Performance de ejecución de código JavaScript en Browser es inferior a la de otros lenguajes existentes como Java, .NET, PHP. -Calidad de interfaz gráfica de usuario inferior a la de app Desktop. -Desacoplamiento del SO, pero dependientes del Browser. |

| | | | |
|--|--|--|---|
| | | | <p>Problemas más frecuentes: no todos los browsers renderizan HTML de la misma forma, con lo cual una misma app puede visualizarse de forma distinta en browsers diferentes; JavaScript interpretado de forma distinta y con distintos niveles de performance en cada Browser; hojas de estilo (CSS) poseen propiedades que funcionan en algunos Browsers y otros no, y muchas propiedades que funcionan en todos los Browsers son renderizadas de forma distinta.</p> <p>-Cliente descarga el 100% de la app cada vez que accede a la misma, lo que puede ser perjudicial en apps grandes.</p> <p>-Sin acceso a Internet, no hay acceso a la app.</p> <p>-Mayor tiempo de desarrollo que para una app Desktop.</p> <p>-Cantidad de lenguajes diferentes que conviven en una misma app complica el mantenimiento de la misma.</p> |
|--|--|--|---|

Objetos de una BD

Tablas

Es la unidad básica de almacenamiento de datos. Los datos están almacenados en filas y columnas. Son de existencia permanente y poseen un nombre identificador. Cada columna tiene entre otros datos un nombre, un tipo de datos y un ancho (este puede estar predeterminado por el tipo de dato)

Motor SQL Server

```
CREATE TABLE ordenes (
    N_orden int NULL ,
    N_cliente int NULL ,
    F_orden datetime NULL ,
    C_estado smallint NULL ,
    F_alta_audit timestamp NULL ,
    D_usuario varchar (20) NULL )
```

Tablas temporales

Es posible crear tablas temporales que duren mientras se ejecute la aplicación o lo que el usuario desee dentro de la misma sesión. No es posible alterar tablas temporarias. Si DROPearlas y crear los índices que necesite la aplicación. Las tablas temporales pueden ser creadas y usadas mientras dure la sesión. La tabla no grabará ningún log transaccional si así se configura. No se actualizan las tablas de catálogo.

En algunos motores de BD las Tablas temporales generadas pueden ser Globales para las cuáles los datos en una tabla temporal serán automáticamente borrados en el caso de la terminación de la sesión. Cada sesión sólo puede ver y modificar sus propios datos. Sobre ellos, no se generan bloqueos de DML.

Motor SqlServer

Se pueden crear tablas temporales locales y globales. Las tablas temporales locales son visibles sólo

en la sesión actual; las tablas temporales globales son visibles para todas las sesiones.

Coloque un prefijo de signo numérico simple (#table_name) en los nombres de las tablas temporales locales y un prefijo de un signo numérico doble (##table_name) en los nombres de las tablas temporales globales.

Creación de tabla temporal en base a la tabla ordenes. La tabla Ordenes_pendientes se borrará automáticamente cuando finalice la sesión.

```
CREATE TABLE #ordenes_pendientes (
```

```
    N_orden INTEGER,  
    N_cliente INTEGER,  
    F_orden DATE,  
    I_Total DECIMAL(15 , 2),  
    C_estado SMALLINT,  
    F_alta_audit TIMESTAMP,  
    D_usuario VARCHAR(20) )  
WITH NO LOG;
```

```
INSERT INTO #ordenes_Pendientes
```

```
    SELECT * FROM ordenes WHERE c_estado = 1
```

Creación y carga de una tabla temporal desde un SELECT.

```
SELECT *
```

```
INTO #ordenes_Pendientes
```

```
FROM ordenes
```

```
WHERE c_estado = 1
```

Tablas anidadas (Oracle)

Es posible crear una tabla con una columna cuyo tipo de dato sea otra tabla. De esta forma, las tablas pueden anidarse dentro de otras tablas como valores en una columna.

Tablas Organizadas por Índice (IOT - Index-Organized Tables) (Oracle)

Una IOT tiene una organización del almacenamiento que es una variante del Árbol-B. A diferencia de una tabla común, en la que los datos se guardan como un conjunto desordenado, en una IOT se guardan en una estructura de índice de Árbol-B, ordenado a la manera de una clave primaria. Sin embargo, además de almacenar los valores de las columnas de clave primaria de cada tabla, cada entrada en el índice almacena además los valores de las columnas no clave.

De esta forma, en vez de mantener dos estructuras separadas de almacenamiento, una para la tabla y una para el índice de Árbol-B, el sistema mantiene sólo un índice Árbol-B, porque además de almacenar el rowid de las filas, también se guardan las columnas no clave.

Clusters (Agrupamientos) (Oracle)

Un cluster es un grupo de tablas que comparten los mismos bloques de datos porque tienen columnas comunes compartidas y que a menudo se usan juntas. Cuando se agrupan tablas, el motor guarda físicamente todas las filas de cada una de ambas tablas en los mismos bloques de datos.

Esto brinda algunos beneficios:

- Reducción de E/S a disco para los joins de las tablas agrupadas
- Mejora en tiempos de acceso para los joins de las tablas agrupadas
- En un cluster, el valor de clave de cluster es el valor de las columnas clave del cluster para una fila en particular. Cada valor de clave del cluster se guarda sólo una vez en el cluster y en el índice del cluster, independientemente de cuántas filas de diferentes tablas contengan

dicho valor. Por lo tanto, se requiere menos espacio para almacenar tablas relacionadas e índices de datos en un cluster.

Constraints (Restricciones)

Integridad de entidades:

Es usada para asegurar que los datos pertenecientes a una misma tabla tienen una única manera de identificarse, es decir que cada fila de cada tabla tenga una primary key capaz de identificar unívocamente una fila y esa no puede ser nula.

Interviene:

- **PRIMARY KEY CONSTRAINT:** Puede estar compuesta por una o más columnas, y deberá representar unívocamente a cada fila de la tabla. No debe permitir valores nulos.

Integridad Referencial:

Es usada para asegurar la coherencia entre los datos. Ej.: Si se ingresa un comprobante de un cliente, este debe existir en la tabla de clientes.

Intervienen:

- **PRIMARY KEY CONSTRAINT**
- **FOREIGN KEY CONSTRAINT:** Puede estar compuesta por una o más columnas, y estará referenciando a la PRIMARY KEY de otra tabla.

Los constraints referenciales permiten a los usuarios especificar claves primarias y foráneas para asegurar una relación PADRE-HIJO (MAESTRO-DETALLE).

Reglas que son aseguradas con el uso de constraints referenciales:

1. Si un usuario BORRA (DELETE) una PRIMARY KEY y dicha clave está correspondida por FOREIGNS KEY en otras tablas, el DETELE fallará. Esta regla podría pasarse mediante la aplicación de cláusulas de ON DELETE CASCADE (Borrado en Cascada).
2. Si un usuario MODIFICA (UPDATE) una PRIMARY KEY y la clave original está correspondida por FOREIGNS KEY en otras tablas, el UPDATE fallará
3. No hay restricciones asociadas al borrado de FOREIGNS KEYS.
4. Si un usuario MODIFICA (UPDATE) una FOREIGN KEY y no hay una PRIMARY KEY en la tabla de referencia que corresponda a esa nueva clave NO NULA, el UPDATE fallará.
5. Todos los valores en una PRIMARY KEY deben ser únicos. Al tratar de insertar una clave duplicada en una PRIMARY KEY dará un error.
6. Cuando un usuario INSERTA (INSERT) una fila en una tabla hijo, si todas las FOREIGN KEYS son NO NULAS, todas deberán estar correspondidas por una PRIMARY KEY, en caso contrario, el INSERT fallará.

TIPOS DE CONSTRAINTS REFERENCIALES

1. **CICLIC REFERENTIAL CONTSTRAINT:** Asegura una relación de PADRE-HIJO entre tablas. Es el más común. Ej. CLIENTE -> FACTURAS
PK Clientes.c_cliente
FK Facturas.c_cliente que referencia a la PK de la tabla Clientes.
2. **SELF REFERENCING CONSTRAINT:** Asegura una relación de PADRE-HIJO entre la misma tabla. Ej. EMPLEADOS -> EMPLEADOS
PK Empleados.c_empleado
FK Empleados.c_jefe que referencia a la PK de la tabla Empleados, este campo además admite NULOS. Esto significa que para todo empleado que el campo jefe sea distinto de NULL, dicho código debe existir como empleado.
3. **MULTIPLE PATH CONSTRAINT:** Se refiere a una PRIMARY KEY que tiene múltiples FOREIGN KEYS. Este caso también es muy común.
Ej. CLIENTES -> FACTURAS

CLIENTES -> RECLAMOS

PK Clientes.c_cliente

FK Facturas.c_cliente que referencia a la PK de la tabla Clientes.

FK Reclamos.c_cliente que referencia a la PK de la tabla Clientes.

Integridad Semántica

Es la que nos asegura que los datos que vamos a almacenar tengan una apropiada configuración.

- **DATA TYPE:** Este define el tipo de valor que se puede almacenar en una columna.
- **DEFAULT CONSTRAINT:** Es el valor insertado en una columna cuando al insertar un registro ningún valor fue especificado para dicha columna. El valor default por default es el NULL.
 - Se aplica a columnas no listadas en una sentencia INSERT.
 - El valor por default puede ser un valor literal o una función SQL (USER, TODAY, etc.)
 - Aplicado sólo durante un INSERT (NO UPDATE).
- **CHECK CONSTRAINT:** Especifica condiciones para la inserción o modificación en una columna. Cada fila insertada en una tabla debe cumplir con dichas condiciones. Actúa tanto en el INSERT, como en el UPDATE.
 - Es una expresión que devuelve un valor booleano de TRUE o FALSE.
 - Son aplicados para cada fila que es INSERTADA o MODIFICADA.
 - Todas las columnas a las que referencia deben ser de la misma tabla (la corriente).
 - No puede contener subconsultas, secuencias, funciones (de fecha, usuario) ni pseudocolumnas.
 - Todas las filas existentes en una tabla deben pasar un nuevo constraint creado para dicha tabla. En el caso de que alguna de las filas no cumpla, no se podrá crear dicho constraint.
- **UNIQUE CONSTRAINT:** Especifica sobre una o más columnas que la inserción o actualización de una fila contiene un valor único en esa columna o conjunto de columnas.
- **NOT NULL CONSTRAINT:** Asegura que una columna contenga un valor durante una operación de INSERT o UPDATE. Se considera el NULL como la ausencia de valor.

Motor SQL Server

Creación de una tabla con Primary Key y otra con una Primary key compuesta y con una Foreign key.

```
CREATE TABLE ordenes (  
    N_orden int PRIMARY KEY ,  
    N_cliente int NULL ,  
    F_orden datetime NULL ,  
    C_estado smallint NULL ,  
    F_alta_audit datetime NULL ,  
    D_usuario varchar (20) NULL )  
  
CREATE TABLE items_ordenes (  
    N_orden int REFERENCES ordenes ,  
    N_item int NOT NULL ,  
    C_producto int NULL ,  
    Q_cantidad int NULL ,  
    I_precunit numeric(9, 3) NULL,  
    PRIMARY KEY (N_orden, N_item) )
```

Creación de una tabla con Primary Key y una Foreign Key consigo misma.

```
CREATE TABLE empleados (
    N_empleado int PRIMARY KEY ,
    D_Apellido varchar (60) NULL ,
    D_nombres varchar (60) NULL ,
    N_cuil numeric(11, 0) NULL ,
    N_jefe int NULL REFERENCES empleados )
```

El motor no permitirá ingresar un empleado cuyo nro. de jefe no exista como número de empleado. Lo que si permitirá es ingresar un nro. de jefe NULO.

Creación de una tabla con una restricción de NOT NULL y otra de DEFAULT.

```
CREATE TABLE ordenes (
    N_orden int NOT NULL ,
    N_cliente int NULL ,
    F_orden datetime NULL ,
    C_estado smallint NULL DEFAULT 1,
    F_alta_audit datetime NULL ,
    D_usuario varchar (20) NULL )
```

El motor de BD no permitirá ingresar un registro a la tabla ordenes con un nro. de orden NULA y en sólo en el caso de un Insert si campo c_estado viene NULO le asignará por default el valor 1.

Creación de una tabla con una restricción de UNIQUE y otra de CHECK

```
CREATE TABLE empleados (
    N_empleado int IDENTITY (1, 1) NOT NULL ,
    D_Apellido varchar (60) NULL ,
    D_nombres varchar (60) NULL ,
    N_cuil numeric(11, 0) NULL UNIQUE,
    F_nacimiento datetime NULL ,
    F_ingreso datetime NULL ,
    N_jefe int NULL,
    CHECK (F_nacimiento < F_ingreso) )
```

Al ingresar un registro el sistema va a validar que no haya en la tabla de empleados otro empleado con el mismo nro. de cuil, y tanto para la inserción como para las modificaciones futuras se va a asegurar que para cada registro de la tabla siempre se cumpla el valor del campo f_ingreso sea mayor el valor del campo f_nacimiento.

Secuencias

Los generadores de secuencias proveen una serie de números secuenciales, especialmente usados en entornos multiusuarios para generar una números secuenciales y únicos sin el overhead de I/O a disco o el lockeo transaccional.

Una secuencia debe tener un nombre, debe ser ascendente o descendente, debe tener definido el intervalo entre números, tiene definidos métodos para obtener el próximo número ó el actual (entre otros).

Motor SQL Server

El motor SQL Server no posee secuencias, en su defecto se define un campo como IDENTITY que permite realizar lo mismo. Al insertar una fila en dicha tabla, el motor va a buscar el próximo nro. del más alto existente en la tabla.

```
CREATE TABLE ordenes (
    N_orden int IDENTITY (1, 1) NOT NULL ,
    N_cliente int NULL ,
    F_orden datetime NULL ,
```

I_Total decimal(15 , 2),
C_estado smallint NULL ,
F_alta_audit datetime NULL ,
D_usuario varchar (20) NULL)

```
INSERT INTO ordenes  
(N_cliente, F_orden, I_total, C_estado, F_alta_audit, D_usuario)  
VALUES (17, '15/05/2006', 100, 1, GETDATE(), USER)
```

Las funciones GETDATE() y USER son funciones propias de sql Server y recuperan el año, mes, día, hora, minutos, segundos y fracción, y el usuario que tiene abierto la sesión, respectivamente. El valor para el campo identity no debe completarse.

Sinónimos (NickName / Synonym)

Es un alias definido sobre una tabla, vista ó snapshot. Dependiendo el motor de BD puede ser también definido sobre un procedure, una secuencia, función o package. Se usan a menudo por seguridad o por conveniencia (por ej. en entornos distribuidos). Permiten enmascarar el nombre y dueño de un determinado objeto. Proveen de una ubicación transparente para objetos remotos en una BD distribuida. Simplifican las sentencias SQL para usuarios de la BD.

Database Links (Enlaces de Base de Datos) (Oracle)

Un database link es un puntero que define una ruta de comunicacion unidireccional desde un servidor de base de datos hasta otro. Para acceder al enlace, se debe estar conectado a la base de datos local que contiene la entrada en el diccionario de datos que define dicho puntero.

Hay dos tipos de enlace según la forma en que ocurre la conexión a la base remota:

- Enlace de usuario conectado: el usuario debe tener una cuenta en la base remota con el mismo nombre de usuario de la base local.
- Enlace de usuario fijo: el usuario se conecta usando el nombre de usuario y password referenciados en el enlace.

Directories (Directorios) (Oracle)

Un directorio especifica un alias para un directorio en el file system del servidor, donde se ubican archivos binario externos (BFILES) y datos de tablas externas.

Se pueden usar nombres de directorios al referirse a ellos desde el código, en vez de hardcodear el nombre de ruta de sistema operativo, suministrando por lo tanto una mayor flexibilidad en la administración de archivos.

Los directorios no son propiedad de ningún esquema individual.

Views (Vistas)

También llamada una tabla virtual o ventana dinámica, es un conjunto de columnas, ya sea reales o virtuales, de una misma tabla o no, con algún filtro determinado o no.

De esta forma, es una presentación adaptada de los datos contenidos en una o más tablas, o en otras vistas. Una vista toma la salida resultante de una consulta y la trata como una tabla. Se pueden usar vistas en la mayoría de las situaciones en las que se pueden usar tablas. A diferencia de una tabla, una vista no aloca espacio de almacenamiento, ni contiene datos almacenados, sino que está definida por una consulta que extrae u obtiene datos desde las tablas a las que la vista hace referencia.

Se pueden utilizar para:

1. Suministrar un nivel adicional de seguridad restringiendo el acceso a un conjunto predeterminado de filas o columnas de una tabla.

2. Ocultar la complejidad de los datos.
3. Simplificar sentencias al usuario.
4. Presentar los datos desde una perspectiva diferente a la de la tabla base.
5. Aislar a las aplicaciones de los cambios en la tabla base.

RESTRICCIONES:

1. No se pueden crear índices en las Views
2. Una view depende de las tablas a las que se haga referencia en ella, si se elimina una tabla todas las views que dependen de ella se borrarán o se pasará a estado INVALIDO, dependiendo del motor. Lo mismo para el caso de borrar una view de la cual depende otra view.
3. Algunas views tienen restringido los: Inserts, Deletes, Updates.
 - Aquellas que tengan joins
 - Una función agregada
 - Tener en cuenta ciertas restricciones para el caso de Actualizaciones:
 - Si en la tabla existieran campos que no permiten nulos y en la view no aparecen, los inserts fallarían
 - Si en la view no aparece la primary key los inserts podrían fallar
4. No se pueden definir Triggers sobre una Vista.
5. Se puede borrar filas desde una view que tenga una columna virtual.
6. Con la opción WITH CHECK OPTION, se puede actualizar siempre y cuando el chequeo de la opción en el where sea verdadero.
7. Al crear la view el usuario debe tener permiso de select sobre las columnas de las tablas involucradas.
8. No es posible adicionar a una View las cláusulas de: ORDER BY y UNION.

Motor SQL Server

Creación de Vista a partir de un Select de dos tablas.

```
CREATE VIEW V_Ordenes_Pendientes
```

```
AS
```

```
SELECT ordenes.N_orden AS Orden_nro, ordenes.F_orden AS Fecha_orden, clientes.d_apellido +  
' ' + clientes.d_nombre AS Cliente, ordenes.i_total AS Importe_total  
FROM ordenes INNER JOIN clientes ON ordenes.N_cliente = clientes.n_cliente  
WHERE (ordenes.C_estado = 1)
```

```
SELECT * FROM v_ordenes_pendientes
```



Obtendríamos por ejemplo los siguientes resultados:

| | | | |
|-------------|----------------|--------------------|---------------|
| Orden_nro | 15/05/2006 | 15/05/2006 | Importe_total |
| Fecha_orden | Cliente | | |
| 122 | Ledesma, Mario | Contempomi, Felipe | 1234.56 |
| 123 | 123 | | 2345,78 |


Indices

Son estructuras opcionales asociadas a una tabla. La función de los índices es la de permitir un acceso más rápido a los datos de una tabla, se pueden crear distintos tipos de índices sobre uno o más campos. Los índices son lógicamente y físicamente independientes de los datos en la tabla asociada. Se puede crear o borrar un índice en cualquier momento sin afectar a las tablas base o a otros índices.

TIPOS DE INDICES

1. Btree Index : Estructura de índice estándar.
2. Btree Cluster Index : La tabla se ordena igual que el índice. 
3. Reverse Key Index (Oracle) : Invierte los bytes de la clave a indexar. Esto sirve para los índices cuyas claves son una serie constante con por ej. Crecimiento ascendente. para que las inserciones se distribuyan por todas las hojas del árbol de índice.
4. Bitmap Index (Oracle) : Son utilizados para pocas claves con muchas repeticiones . Cada bit en el Bitmap corresponde a una fila en particular. Si el bit esta en on significa que la fila con el correspondiente rowid tiene el valor de la clave. 

CARACTERÍSTICAS DIFERENCIADORAS PARA LOS ÍNDICES

- 
1. Unique : Índice de clave única.
 2. Cluster : Este tipo de índice provoca al momento de su creación que físicamente los datos de la tabla sean ordenados por el mismo. (Informix / SQLServer / DB2) . Variante en ORACLE: un Cluster es un objeto de la base que contiene los datos de una o más tablas, con una o más columnas en común. Oracle almacena juntas todas las filas (de todas las tablas) que comparten la misma clave de cluster. Un índice cluster, crea un índice en el cluster que se especifica.
 3. Duplicado : Permite múltiples entradas para una misma clave.
 4. Compuesto La clave se compone de varias columnas.
Las principales funciones de un índice compuesto son:
 1. Facilitar múltiples joins entre columnas
 2. Incrementar la unicidad del valor de los índices

Ejemplo de índice compuesto:

customer_num, lname, fname

Este índice se usará en o para los siguientes casos:

- Joins sobre customer_num, o customer_num y lname o customer_num, lname y fname, es decir, sentencias donde la cláusula WHERE haga referencia a todo o a una porción inicial del índice (las columnas más selectivas o más accedidas, irán al principio del índice).
 - Filtros sobre customer_num, o customer_num y lname o customer_num, lname y fname.
 - ORDERS BY sobre o customer_num y lname o customer_num, lname y fname.
 - Joins sobre customer_num y Filtros sobre lname y fname.
 - Joins sobre customer_num y lname y Filtros sobre fname.
5. Function based Index (Oracle): Calcula el valor de la función o la expresión, y lo guarda en el índice. Puede crearse tanto como un índice B-Tree como Bitmap.

BENEFICIOS DE LA UTILIZACIÓN DE INDICES:

1. Se le provee al sistema mejor performance al equipo ya que no debe hacer lecturas secuenciales sino accede a través de los índices, sólo en los casos que las columnas del Select no formen parte del índice.
2. Mejor performance en el ordenamiento de filas
3. Asegura únicos valores para las filas almacenadas
4. Cuando las columnas que intervienen en un JOIN tienen índices se le da mejor performance si el sistema logra recuperar los datos a través de ellas

COSTO DE LA UTILIZACIÓN DE INDICES

1. El primer costo asociado es el espacio que ocupa en disco, que en algunos casos suele ser mayor al que ocupan los datos.

2. El segundo costo es el de procesamiento, hay que tener en cuenta que cada vez que una fila es insertada o modificada o borrada, el índice debe estar bloqueado, con lo cual el sistema deberá recorrer y actualizar los distintos índices.

¿CUANDO DEBERIAMOS INDEXAR ?

- Indexar columnas que intervienen en Joins
- Indexar las columnas donde se realizan filtros
- Indexar columnas que son frecuentemente usadas en orders by
- Evitar duplicación de índices : Sobre todo en columnas con pocos valores diferentes Ej: Sexo, Estado Civil, Etc.
- Limitar la cantidad de índices en tablas que son actualizadas frecuentemente : Porque sobre estas tablas se estarán ejecutando Selects extras
- Verificar que el tamaño de índice debería ser pequeño comparado con la fila : Tratar sobre todo en crear índices sobre columnas cuya longitud de atributo sea pequeña. No crear índices sobre tablas con poca cantidad de filas, no olvidar que siempre se recupera de a páginas. De esta manera evitaríamos que el sistema lea el árbol de índices
- Tratar de usar índices compuestos para incrementar los valores únicos : Tener en cuenta que si una o más columnas intervienen en un índice compuesto el optimizador podría decidir acceder a través de ese índice aunque sea solo para la búsqueda de los datos de una columna, esto se denomina “partial key search”
- Usando cluster index se agiliza la recuperación de filas : Uno de los principales objetivos de la Optimización de bases de datos es reducir la entrada/salida de disco. Reorganizando aquellas tablas que lo necesiten, se obtendría como resultado que las filas serían almacenadas en bloques contiguos, con lo cual facilitaría el acceso y reduciría la cantidad de accesos ya que recuperaría en menos páginas los mismos datos.

CONSTRUCCIÓN DE ÍNDICES EN PARALELO

Los motores de BD usan en general métodos de construcción de índices en paralelo. El arbol B+ es construido por 2 o más procesos paralelos. Para esto el motor realiza una muestra de la filas a Indexar (aproximadamente 1000) y luego decide como separar en grupos. Luego scanea las filas y las ordena usando el mecanismo de sort en paralelo. Las claves ordenadas son colocadas en los grupos apropiados para luego ir armando en paralelo un subárbol por cada grupo. Al finalizar los subárboles se unen en un único Arbol B+.

Motor SQL Server

- Sql Server utiliza una estructura de Arbol B+.
- Máxima cantidad de campos para la clave de un índice compuesto: 16.

Creación de un índice único y simple:

```
CREATE UNIQUE INDEX ix1_ordenes ON ordenes (n_orden);
```

Creación de Indice duplicado y compuesto.

```
CREATE INDEX ix2_ordenes ON ordenes (n_cliente, f_orden);
```

Creación de Indice cluster.

```
CREATE CLUSTERED INDEX ix3_ordenes ON ordenes(N_orden) ON [PRIMARY];
```

Manejo del Load Factor

FILLFACTOR– Porcentaje de cada página del índice a ser dejado como espacio libre en su creación. Por ej. Si el FILLFACTOR=20, en la creación del índice se ocupará hasta el 80% de cada nodo.



```
CREATE UNIQUE INDEX ix1_ordenes ON ordenes(N_orden)
WITH FILLFACTOR = 20;
```

Snapshots / Summary Table / Materialized Views

Los snapshots, también llamados vistas materializadas o tablas sumariadas, son objetos del esquema de una BD que pueden ser usados para sumarizar, precomputar, distribuir o replicar datos. Se utilizan sobre todo en DataWarehouse, sistemas para soporte de toma de decisión, y para computación móvil y/o distribuida. Consumen espacio de almacenamiento en disco. Deben ser recalculadas o refrescadas cuando los datos de las tablas master cambian. Pueden ser refrescadas en forma manual, o a intervalos de tiempo definidos dependiendo el motor de BD.

Funciones Propias de Motor (Built in Functions)

Estos objetos son funciones ya desarrolladas con el Motor de BD, las cuales pueden clasificarse de la siguiente manera:

Funciones Agregadas

Se aplican a un conjunto de valores derivados de una expresión, actúan específicamente en agrupamientos. SUM, COUNT, AVG, MAX, MIN, etc.

Funciones Escalares

Se aplican a un dato específico de cada fila de una consulta, o en una comparación en la sección WHERE.

- Funciones Algebraicas and Trigonometricas
- Funciones Estadísticas
- Funciones de Fecha
- Funciones de Strings
- Otras

Funciones de Tablas (algunos motores)

Retornan el equivalente a una tabla y pueden ser usadas solamente en la sección FROM de una sentencia. ORACLE: Se denominan “Vista en línea (inline view)”

Funciones de Usuario

Una función es un objeto de la base de datos que puede recibir uno o más parámetros de input y devolver sólo un parámetro de output.

Motor SQL Server

```
CREATE FUNCTION nombre_dpto (@p_c_empleado INT) RETURNS varchar(30)
AS
BEGIN
RETURN (SELECT d_dpto FROM departamentos d, empleados e
        WHERE d.c_dpto = e.c_dpto AND e.c_empleado = p_c_empleado)
END
```

Invocación de función en un Select en la sección WHERE

```
SELECT * FROM departamentos d WHERE d.d_dpto = nombre_dpto(6010)
```

Invocación de función en un Select en la sección Lista de Columnas

```
SELECT e.c_empleado, e.d_nombre, e.d_apellido, nombre_dpto(e.c_dpto) depto_nombre
FROM empleados e WHERE e.c_empleado = 6010
```

Stored Procedures

Es un procedimiento almacenado como un objeto en la Base de Datos.

Características:

1. Incluyen sentencias de SQL y sentencias de lenguaje propias. Lenguaje SPL (Informix), PL/SQL (Oracle), TRANSAC/SQL (SQL Server).
2. Son almacenados en la base de Datos
 - Solo las sentencias del lenguaje pr son permitidas además de las de SQL
 - Algunos motores permiten además Stored Procedures en JAVA.
3. Se guarda la sentencia SQL ya parseada y optimizada
 - Antes de ser almacenada en la base de datos las sentencias SQL son parseadas y optimizadas. Cuando el stored procedure es ejecutado puede que no sea necesario su optimización, en caso contrario se optimiza la sentencia antes de ejecutarse.

VENTAJAS DE LOS STORED PROCEDURES:

1. Pueden reducir la complejidad en la programación. Creando SP con las funciones + usadas.
2. Pueden ganar perfomance en algunos casos
3. Otorgan un nivel de seguridad extra
4. Pueden definirse ciertas reglas de negocio independientemente de las aplicaciones.
5. Diferentes aplicaciones acceden al mismo código ya compilado y optimizado.
6. En un ambiente cliente servidor, no sería necesario tener distribuido el código de la aplicación
7. En proyectos donde el código puede ser ejecutados desde diferentes interfaces, Ud. mantiene un solo tipo de código.
8. Menor tráfico en el PIPE / SOCKET, no en la cantidad de bytes que viajan sino en los ciclos que debo ejecutar una instrucción.

Ej.: Si este proceso se ejecuta desde una workstation en red, por la red viajarán 1000 instrucciones Update/Delete, x cada una el motor tendrá que Parsearla y Optimizarla. El costo es muy alto.

FOR

```
SELECT ....
IF status
    UPDATE
ELSE
    DELETE
END IF
```

END FOR

Sería mejor hacer desde la aplicación cliente servidor , ejecutar un stored procedure:

```
EXECUTE PROCEDURE PEPE(var,var) 1 sola vez
```

En este caso por la red viajará sólo 1 instrucción (execute procedure...), el motor no parseará las instrucciones porque ya están parseadas dentro del Procedure, y las optimizará en caso de que sea necesario.

Motor SQL Server

Creación de un Stored Procedure que inserta una orden de pedido a partir de los datos existentes en varias tablas temporales manejando una transacción y creación de un procedure de grabación de log de errores.

```
CREATE PROCEDURE dbo.sp_inserta_orden @v_n_orden INT
AS SET NOCOUNT ON
```

```
    DECLARE @nError int, @error_info char(30)
    SET @nError = 0
    BEGIN TRANSACTION
    INSERT INTO ordenes SELECT * FROM ordenes_tmp WHERE n_orden = @v_n_orden
```

```

INSERT INTO item_ordenes SELECT * FROM items_tmp WHERE n_orden =
@v_n_orden
set @nError = @@error
IF( @nError <> 0 )
BEGIN
    ROLLBACK
    SELECT @nError AS nError, @error_info AS Descripcion
    exec sp_error_log @nError, @error_info, @v_n_orden, 'sp_inserta_orden'
END
ELSE
BEGIN
    COMMIT
    SELECT 0 AS nError , 'Se insertó bien' AS Descripcion
END
GO
CREATE PROCEDURE dbo.sp_error_log @sql_err int, @error_info char(70), @v_nro_orden
INT, @proc_name char(18)
AS SET NOCOUNT ON
    INSERT INTO error_logs VALUES (@proc_name, @v_nro_orden, @sql_err, @error_info,
    USER, getdate())
GO

```

Ejecución del Stored Procedure desde SQL de forma Online

```
EXEC sp_inserta_orden 1234
```

Ejecutará el procedure insertando la orden Nro. 1234. En el ejemplo se asume que las ordenes a pasar como parámetro existen y fueron validadas por otro programa.

Triggers

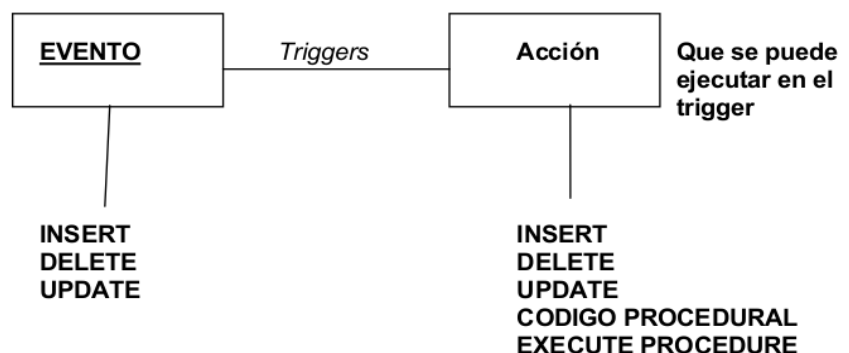
Es un mecanismo que ejecuta una sentencia de SQL automáticamente cuando cierto evento ocurre.

Eventos:

1. Insert
2. Update
3. Delete

¿Que se puede ejecutar en el trigger?:

1. Insert
2. Update
3. Delete
4. Código Procedural
5. Ejecutar un SP

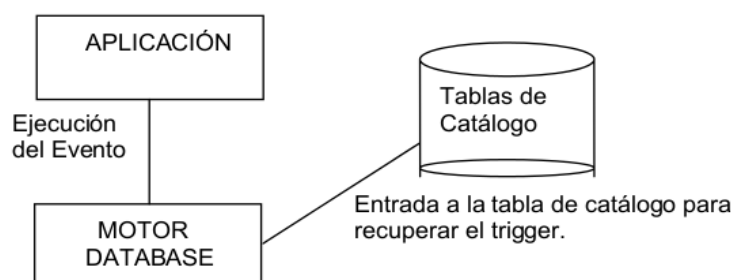


Cuando el evento ocurre sobre una tabla, se dispara la acción.

Esta ejecución es independiente de la aplicación que la invoca

Solo en el evento UPDATE puedo hacer referencia a una columna

Cuando un trigger es ejecutado:



La acción del trigger es recuperada, optimizado y ejecutado.

¿PORQUE USAR TRIGGERS?

1. Se pueden aplicar las reglas de negocio : Por Ej.: Si el inventario de una columna pasa x valor, entonces insertar un pedido en la tabla de compras
2. Valores de columnas derivadas : En algunos casos es necesario que ante tal acción se deriven datos en base a otros.
3. Replicación automática de tablas
4. Logs. De Auditoría
5. Delete en Cascada
6. Autorización de Seguridad

Eventos:

INSERT ON tab_name

DELETE FROM tab_name

UPDATE tab_name

UPDATE of col_name ON tab_name

CONEXION (Oracle)

DESCONEXION (Oracle)

Observaciones:

- Se permiten múltiples triggers sobre una tabla, pero sólo 1 por tipo. Para el caso de UPDATE las columnas deben ser mutuamente exclusivas.
- La tabla especificada por el trigger debe estar en modo local, no acepta tablas en servers remotos.

Acciones de triggers

Pueden ser sentencias SQL, 1 o varias, o Stored Procedures. En caso de Oracle se puede ejecutar directamente código PL/SQL.

Se pueden ejecutar en distintos momentos:

- before (execute procedure xyz()) -> Se ejecuta antes de que el evento de trigger ocurra
- for each row (execute procedure xyz()) -> Se ejecuta para cada una de las filas del evento
- after (execute procedure xyz()) -> Se ejecuta despues de que el evento de trigger ocurra

USOS EN LOGIN Y RECOVERY

- Para BD sin logs No ocurre el Rollback.
- En base de datos con log, el evento y la acción del trigger se les hace un roll back automático.

Motor SQL Server

CREATE TRIGGER employee_insupd ON employee FOR INSERT, UPDATE

AS

/* Get the range of level for this job type from the jobs table. */

```
DECLARE @min_lvl tinyint, @max_lvl tinyint, @emp_lvl tinyint, @job_id smallint
SELECT @min_lvl = min_lvl, @max_lvl = max_lvl, @emp_lvl = i.job_lvl, @job_id =
i.job_id
FROM employee e INNER JOIN inserted i ON e.emp_id = i.emp_id JOIN jobs j ON
j.job_id = i.job_id
IF (@job_id = 1) and (@emp_lvl <> 10)
```

```

BEGIN
    RAISERROR ('Job id 1 expects the default level of 10.', 16, 1)
    ROLLBACK TRANSACTION
END
ELSE IF NOT (@emp_lvl BETWEEN @min_lvl AND @max_lvl)
BEGIN
    RAISERROR ('The level for job_id:%d should be between %d and %d.', 16, 1,
        @job_id, @min_lvl, @max_lvl)
    ROLLBACK TRANSACTION
END

```

Packages (Oracle / DB2)

Un Package (paquete) es un objeto que agrupa tipos y subprogramas relacionados lógicamente. Habitualmente tienen dos partes: una especificación y un cuerpo. La especificación es la interfaz para las aplicaciones: declara los tipos, variables, constantes, excepciones, cursores y subprogramas disponibles para su uso. El cuerpo define en forma completa los cursores y subprogramas, y por lo tanto implementa la especificación.

Los paquetes suministran varias ventajas: modularidad, facilidad en el diseño de la aplicación, ocultamiento de información y mejora en el rendimiento.

Esquema (Oracle)

Es el conjunto de objetos de los que es dueño un usuario, y tiene el mismo nombre que el usuario. Cada usuario posee un solo esquema. El esquema lo crea Oracle al crearse el usuario.

DTS (SqlServer)

Los Servicios de transformación de datos (DTS) solo existen en el motor SQL Server y proporcionan un conjunto de herramientas que permiten extraer, transformar y consolidar datos de distintos orígenes (Tablas de base de datos, excel, Archivos de Texto, Access, etc) en uno o varios destinos compatibles con la conectividad DTS. Para crear soluciones de transferencia de datos personalizadas se deben generar gráficamente los paquetes.

Importación y exportación de datos.

DTS puede importar datos de un archivo de texto o de un origen de datos OLE DB (por ejemplo, una base de datos de Microsoft Access 2000) en SQL Server. De forma alternativa, puede exportar datos desde SQL Server a un destino de datos OLE DB (por ejemplo, una hoja de cálculo Microsoft Excel 2000). DTS también permite la carga de datos de alta velocidad desde archivos de texto a tablas de SQL Server.

Transformación de datos.

El Diseñador DTS incluye la tarea Transformar datos, que permite seleccionar datos de una conexión de origen de datos, asignar las columnas de datos a un conjunto de transformaciones y enviar los datos transformados a una conexión de destino. El Diseñador DTS también contiene una tarea de consulta controlada por datos que permite asignar datos a consultas parametrizadas.

Copia de objetos de base de datos.

Con DTS, puede transferir, además de los datos, índices, vistas, inicios de sesión, procedimientos almacenados, desencadenadores, reglas, valores predeterminados, restricciones y tipos de datos definidos por el usuario. Además, puede generar secuencias de comandos para copiar los objetos de base de datos.

ANEXO:

CURSORES

Las operaciones de una base de datos relacional actúan en un conjunto completo de filas. El conjunto de filas que devuelve una instrucción SELECT está compuesto de todas las filas que satisfacen las condiciones de la cláusula WHERE de la instrucción. Este conjunto completo de filas que devuelve la instrucción se conoce como el conjunto de resultados.

Las aplicaciones, especialmente las aplicaciones interactivas en línea, no siempre pueden trabajar de forma efectiva con el conjunto de resultados completo si lo toman como una unidad. Estas aplicaciones necesitan un mecanismo que trabaje con una fila o un pequeño bloque de filas cada vez. Los cursores son una extensión de los conjuntos de resultados que proporcionan dicho mecanismo.

Los cursores amplían el procesamiento de los resultados porque:

- Permiten situarse en filas específicas del conjunto de resultados.
- Recuperan una fila o bloque de filas de la posición actual en el conjunto de resultados.
- Aceptan modificaciones de los datos de las filas en la posición actual del conjunto de resultados
- Aceptan diferentes grados de visibilidad para los cambios que realizan otros usuarios en la información de la base de datos que se presenta en el conjunto de resultados.
- Proporcionan instrucciones de Transact-SQL en secuencias de comandos, procedimientos almacenados y acceso de desencadenadores a los datos de un conjunto de resultados.

Proceso de cursores

1. Asigne un cursor al conjunto de resultados de una instrucción Transact-SQL y defina las características del cursor como, por ejemplo, si sus filas se pueden actualizar.
2. Ejecute la instrucción de Transact-SQL para llenar el cursor.
3. Recupere las filas del cursor que desea ver. La operación de recuperar una fila o un bloque de filas de un cursor recibe el nombre de recopilación. Realizar series de recopilaciones para recuperar filas, ya sea hacia adelante o hacia atrás, recibe el nombre de desplazamiento.
4. Existe la opción de realizar operaciones de modificación (actualización o eliminación) en la fila de la posición actual del cursor.
5. Cierre el cursor.

Ejemplo:

```
-- Declare the variables to store the values returned by FETCH.
DECLARE @au_lname varchar(40), @au_fname varchar(20)
DECLARE authors_cursor CURSOR FOR
SELECT au_lname, au_fname FROM authors WHERE au_lname LIKE "B%"
ORDER BY au_lname, au_fname
OPEN authors_cursor
-- Perform the first fetch and store the values in variables.
FETCH NEXT FROM authors_cursor INTO @au_lname, @au_fname
-- Check @@FETCH_STATUS to see if there are any more rows to fetch.
WHILE @@FETCH_STATUS = 0
BEGIN
-- Concatenate and display the current values in the variables.
PRINT "Author: " + @au_fname + " " + @au_lname
-- This is executed as long as the previous fetch succeeds.
FETCH NEXT FROM authors_cursor INTO @au_lname, @au_fname
END
CLOSE authors_cursor
DEALLOCATE authors_cursor
GO
```


Apuntes tenidos en cuenta:

- Arquitectura de aplicaciones del Ing Reinoso
- Objetos de una BD del Ing Zaffaroni

Unidad IV

Conceptos de Bases de Datos

Sistema de Bases de Datos

Básicamente es un sistema cuya finalidad es almacenar información y permitir a los usuarios recuperar y actualizar esa información a partir de peticiones. Desde una visión muy simplificada podemos decir que una Base de datos nos permite:

- Crear archivos en la base de datos
- Eliminar archivos existentes en la base
- Incorporar nuevos datos a archivos existentes en la base
- Consultar datos de los archivos existentes
- Actualizar datos de los archivos existentes
- Eliminar datos de los archivos existentes

¿Que es una Base de Datos?

Una BD es un conjunto de datos persistentes e interrelacionados que es utilizado por los sistemas de aplicación de una empresa, los mismos se encuentran almacenados en un conjunto independiente y sin redundancias.

Componentes de un sistema de base de datos:

1. **Los datos** : La información en una BD esta integrada y compartida. Integrada ya que es la unificación de distintos archivos de datos y en consecuencia elimina las redundancias. Compartida porque son varios los usuarios que acceden a la información en forma concurrente o no, con diversos objetivos.
2. **La tecnología** : El equipamiento constituido por las unidades de almacenamiento (discos rígidos, disks arrays, etc.), los procesadores y la memoria principal, más el sistema operativo sobre el que corre el sistema, componen el conjunto de recursos utilizados por el Sistema de Base de Datos.
3. **Los programas**: Se refiere a los programas que componen el concepto de DBMS (Data Base Manager System) o motor de base de datos; con el objeto de abstraer a los usuarios de los detalles tecnológicos, brindándoles los servicios que tiene como objetivo. El DBMS ofrece a los usuarios una percepción de alto nivel. El DBMS es el componente más importante del sistema de BD pero no el único. Los demás software son entre otros herramientas de desarrollo, librerías de funciones, frameworks de desarrollo, generadores de reportes, entre otros.
4. **Los usuarios** :
 1. **El programador de aplicaciones**, que accede a los servicios brindados por el DBMS, mediante la inclusión de las solicitudes correspondientes en el código de los programas. Estos programas pueden ser batch u on-line.
 2. **El usuario**, que utiliza los programas on-line que les suministraron los desarrolladores, u otras interfaces donde puedan ingresar sus consultas en lenguaje SQL, por ejemplo. (aunque no necesariamente se hacen solo consultas).
 3. **El DBA** es la persona que tiene como responsabilidad Crear, alterar o borrar las distintas estructuras de una base de datos.

Aplicaciones tradicionales vs. Enfoque de bases de datos

Enfoque orientado a la aplicación

Cada aplicación es "dueña" de sus datos. Supongamos un sistema simple que consta de tres programas o aplicaciones a la vista del usuario. Cada programa manipula a nivel físico los archivos, peticionando al S.O. las operaciones de lectura/escritura sobre los mismos. El desarrollador debe tener en cuenta durante el diseño y construcción factores relativos tales como: concurrencia entre la aplicación 1 y 2, ya que ambas utilizan el Archivo 1.

Enfoque orientado a los datos o enfoque de base de datos.

Una persona (que en realidad representa a la empresa) es "dueña" de los datos. Este es el administrador de datos. Es quién define que datos se van a colocar en la base y que políticas de seguridad se pueden aplicar a los datos. Esta política será aplicada por el DBA, que es el técnico responsable por la operación y mantenimiento de la base.

Los archivos de datos (tablas) residen en la Base de Datos, y los programas o consultas de usuarios son enviadas mediante algún mecanismo de comunicación estándar al DBMS, que es quién realiza realmente la consulta/actualización y devuelve el resultado al programa. El DBMS se encarga entre otras cosas de la seguridad, concurrencia, etc. Las peticiones no se hacen al S.O. sino al DBMS mediante instrucciones SQL.

Ventajas del enfoque de bases de datos

1. Es posible **aplicar una política de seguridad**: ofrece a la organización un sistema de control centralizado de su información. Al tener a su mando la totalidad de los datos, el DBA puede asegurarse de que solo se acceda a los datos según lo establecido.
2. Es posible hacer cumplir normas aplicables a la representación de los datos: tienen que ver con los tipos de datos utilizados, las normativas para nombrar y documentar datos.
3. Es posible disminuir la redundancia: en el otro enfoque, cada aplicación tiene sus propios archivos, por lo que el mismo dato real puede estar replicado varias veces en distintos archivos. Esto genera un costo de espacio adicional. Sin embargo, hay veces que no se desea eliminar toda la redundancia por razones de performance.
4. Es posible **evitar la inconsistencia**. Al estar un dato almacenado en distintos lugares, voy a necesitar mantener actualizadas todas las copias existentes del mismo dato. Si no lo hago así, genero inconsistencias. En un enfoque de base de datos centralizado, es posible (hasta cierto punto), eliminar las inconsistencias haciendo que exista una única entrada para un dato determinado, o que si bien el dato se encuentre redundante por motivos de performance, la actualización se realice en forma consistente.
5. Es posible **compartir los datos**. No solo las aplicaciones existentes comparten la información (debido a las ventajas del control, la no redundancia y la consistencia), sino que también puedo crear nuevas aplicaciones que utilicen esos datos.
6. Es posible **mantener la integridad**. Aunque se eliminen las redundancias, aun la base de datos puede contener información errónea. A esto se refiere el concepto de integridad: por ejemplo, que no se encuentren registradas ventas de artículos que no existen, entradas de alumnos sin legajo, etc. El DBA puede definir reglas para que la base mantenga automáticamente la integridad.
7. Es posible **establecer un criterio para la organización de los datos**, de manera de favorecer al rendimiento de ciertas aplicaciones, al considerarse más importantes que otras.
8. Es posible contar con la **independencia de los datos**. Esta ventaja se constituye de por sí en un objetivo de los sistemas de bases de datos.

Independencia de los datos

Enfoque de aplicaciones tradicionales

Los datos y programas están fuertemente acoplados. Los requerimientos de un programa determinan la forma en la que se van a organizar los datos en disco y el método de acceso más apropiado. Además, hay un conocimiento total de la organización de datos y de método de acceso al momento de programar la aplicación, lo que determinará su lógica interna.

La aplicación es dependiente de los datos. Es imposible alterar la estructura de los datos o la técnica de acceso sin afectar los programas.

Enfoque de bases de datos

- **Independencia lógica.** Cada aplicación requiere una vista diferente de los mismos datos. Por ejemplo la aplicación A que requiere un listado de todos los clientes (código y nombre solamente) ordenados por nombre lo solicitará al DBMS. El DBMS resuelve la consulta mediante la lectura de los campos y los registros apropiados y lo entrega a la aplicación "cliente". El programa "A", ni se enteró de la existencia de otros campos en el archivo, ni siquiera como hizo el motor para resolver la consulta en el orden indicado.
- **Independencia física.** Es posible modificar la estructura de almacenamiento, la distribución física o la técnica de acceso sin afectar las aplicaciones. En el ejemplo del programa A, puedo crear un índice por nombre para que se resuelva el listado más rápidamente. En ningún caso los programas son afectados.

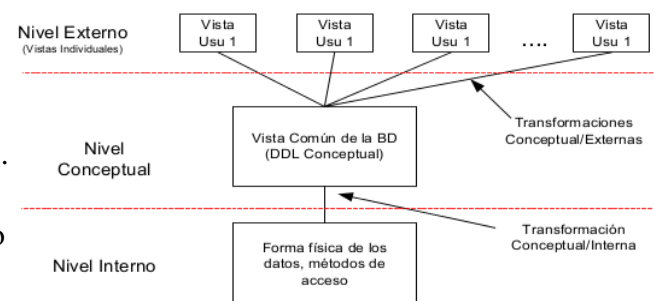
La independencia lograda nunca es absoluta.

Arquitectura de un sistema de bases de datos (ANSI/SPARC)

Se ajusta bastante bien a la mayoría de los sistemas de BD.

Niveles:

- **Interno (Físico):** el más cercano al almacenamiento físico, es el que se ocupa de la forma como se almacenan físicamente los datos.
- **Conceptual (Logico):** es un nivel de mediación entre interno y externo. se ocupa de una sola vista que las agrupe a todas. Es también una representación 'abstracta' de la base de datos, es decir, implica estructuras orientadas hacia el usuario (archivo, registro, campo) y no hacia la máquina (byte, palabra, tamaño de página, tamaño de buffer, etc)
- **Externo (Vistas):** es el más cercano al usuario. Se ocupa de la forma como los usuarios individuales percibirán los datos.



Nivel externo o VISTAS

Los usuarios utilizarán algún tipo de lenguaje para acceder a los datos. En el caso de los programadores puede ser 4GL, PL/SQL, C, COBOL, VB, etc. Los usuarios finales en su lugar pueden usar algún tipo de interfase especial para acceder a los datos. Sea cual fuere la forma, dentro de estos lenguajes debe haber un sublenguaje (por ejemplo embebiendo uno dentro del otro) que se ocupa específicamente de las operaciones con la base de datos.

Operaciones:

- conexión
- definición de datos (DDL)
- manipulación de datos (DML)
- desconexión.

El DBA es quien nos define esas vistas. Estas vistas están compuestas por registros lógicos que no necesariamente tienen algún tipo de correspondencia con cada registro físico de un archivo o tabla en el nivel conceptual. Para que esto ocurra, hay una correspondencia entre el esquema conceptual y el esquema externo.

Una vista externa es el contenido de una Base de datos como lo ve algún usuario en particular (para

este usuario la vista es la Base de Datos).

Nivel conceptual o LOGICO

Representa de una forma 'entendible' de toda la información contenida en una base de datos. En lugar de describir datos personales, describe a los datos de toda la organización. Debe ofrecer un panorama de los datos como realmente son, a diferencia de como los usuarios lo ven debido a las limitaciones existentes (lenguaje, hardware, interface, etc.).

La vista conceptual se define mediante un esquema conceptual. Este esquema conceptual se escribe en DDL. Contiene definiciones del contenido de la base, tipos de datos, restricciones, reglas de integridad, etc.

Nivel interno o FISICO

En este nivel se define como se almacenan los datos en disco, es una representación de bajo nivel de toda la base de datos. Por ejemplo, se especifican las estructuras de los registros, se definen índices y métodos de acceso, en que secuencia física se encuentran los registros, etc.

Transformaciones

Transformación Conceptual / Interna.

La transformación conceptual / interna define la correspondencia entre la vista conceptual y la base de datos almacenada, y especifica la representación en el nivel interno de las filas y columnas del modelo conceptual.

Transformación Externa/Conceptual

Dada una determinada vista externa, esta transformación define la correspondencia entre dicha vista externa y la vista conceptual. Por ej.: se puede cambiar el tipo de dato a obtener, combinar varios campos conceptuales en un único registro externo.

Funciones del motor de base de datos y del DBMS en su conjunto

1. **Diccionario de datos.** Es una base de datos del sistema, que define a los objetos dentro del mismo. Es lo que se llama 'metadatos'. Allí se almacenan las diversas vistas (externas, conceptuales, internas). El diccionario tiene que estar integrado a la base.
2. **Control de la seguridad.** La seguridad implica controlar que los usuarios estén autorizados para hacer lo que intentan. Para ello los DBMS poseen lo que se llama el catálogo. El catálogo mismo está formado por entidades e interrelaciones (por lo que en un esquema relacional van a ser tablas). En ellos se almacenan datos acerca de los objetos de la base (tablas, índices, vistas, usuarios, permisos, etc). El DBA puede alterar estos catálogos de manera de otorgar/revocar los permisos necesarios a los usuarios/grupos de usuarios creados en la base. Otro mecanismo que se emplea para la seguridad son las vistas.
3. **Mecanismos para garantizar la integridad y la consistencia**
 - **Constraints:** son controles de integridad que se les pueden incorporar a la base. Por ejemplo, constraints de PRIMARY KEY, integridad referencial, etc.
 - **Triggers:** un procedimiento que se ejecuta ante un determinado evento sobre un objeto. Antes o después de UPDATE/DELETE/INSERT.
 - **Transacciones:** son una unidad lógica de trabajo. Partiendo de que una transacción lleva la base de datos de un estado correcto a otro estado correcto, el motor posee mecanismos de manera de garantizar que la operación completa se ejecute o falle.
 - **Logical Logs:** Es un registro donde el motor almacena la información de cada operación llevada a cabo con los datos.
4. **Backup y recovery :** Utilitarios para realizar backups y recuperación ante caídas.
 - **Recovery:** Lleva la BD a un estado consistente utilizando los logs transaccionales.
 - **Backup**

- **Backup Full:** backup de toda la base de datos
 - **Backup en caliente:** Con usuarios. Se guardan los datos anteriores al inicio del backup e imágenes con timestamp de los datos modificados luego del inicio del backup.
 - **Backup en frío:** sin usuarios
 - **Backup Log transaccional:** se hace durante el día
 - **Backup Diferencial**
 1. **Backup Incremental:** backup de los cambios que hubo a partir de un backup anterior (no importa si fue full o incremental). Se hace cuando no se puede hacer un backup full por limitaciones de tiempo
 2. **Backup Acumulativo:** backup de los cambios que hubo a partir de un backup full anterior.
5. **Administración de la organización física de los datos en disco y/o memoria.**
Los motores de bases de datos poseen su propia (y compleja) forma de hacer uso de los recursos de disco que el sistema le brinda. Es por ello que entran en juego una serie de consideraciones como ser:
- El tamaño de página.
 - El mecanismo de paginación del disco a memoria.
 - Los tamaños de los buffers para acceder al disco.
 - El tamaño de memoria compartida que va a alocar.
 - La administración de esos recursos de memoria.
- Si se van a colocar los datos dentro de 'archivos' del sistema operativo (haciendo uso de los FILE SYSTEM de usuario) o si se van a utilizar directamente los recursos a nivel máquina (RAW DEVICE). Definición de los 'extents', tamaños que se alocan al requerir espacio para un nuevo objeto o la ampliación de uno ya existente. Distribución física de la base de datos en 'DATAFILES'. Particionamiento o Fragmentación de las tablas según su lógica interna.
6. **Mecanismo de conectividad.** El motor otorga y lleva un control de sesiones de usuarios. Estos usuarios pueden ser usuarios finales o programas. La conexión a la base puede ser a través del mecanismo de conectividad (red) existente (DEC, TCP/IP), o por SHARED MEMORY.
7. **Administración y chequeo de los recursos de la base .** Poseen una serie de utilitarios para monitorear el uso de los recursos de la base, determinar quienes están accediendo a la base en un momento dado, que se encuentran haciendo, etc.
8. **Concurrencia en lecturas y actualizaciones** Los distintos usuarios que acceden en un momento dado a la base, deben percibir a la misma en forma consistente, a través de su 'vista'. Como este acceso es simultáneo, los DBMS permiten la concurrencia mediante mecanismos de recuperación de transacciones y bloqueos. Debe permitir detectar (e eliminar) deadlocks. También puede permitir transacciones en forma distribuida.
9. **Facilidades de auditoría.** A los DBMS se les puede activar la opción de guardar en un log un registro del uso de los recursos de la base para auditar posteriormente.
10. **Logs del sistema.** Mediante este tipo de logs, el DBA puede llegar a determinar cual fue, por ejemplo, el problema que produjo una caída del sistema.
11. **Acomodarse a los cambios,** crecimientos, modificaciones del esquema. El motor permite realizar cambios a las tablas constantemente, casi en el mismo momento en que la tabla está siendo consultada.
12. **Creación de triggers y stored procedures.** Con el objeto de fortalecer aún más el concepto

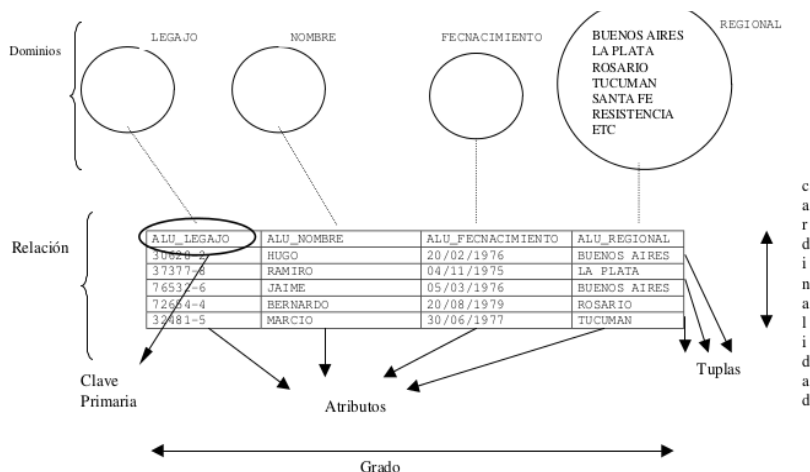
de independencia entre datos y programas de aplicación, los motores nos dan la facilidad de incorporar los llamados STORED PROCEDURES. Básicamente, son un conjunto de instrucciones en un lenguaje que el motor entiende, que pueden ser llamados en forma externa desde las aplicaciones programa o usuarios. De esta manera se pueden cambiar los programas, por ejemplo, por una decisión de mejorar la interfase visual, utilizando las reglas del negocio, que están definidas dentro de la base. Poseen la desventaja de ser propietarios.

Modelo Relacional

Se define en 3 partes:

- estructura ----->
- integridad
- manipulación de datos

| Término relacional | "Equivalente" informal |
|--------------------|--|
| Relación | Tabla |
| Tupla | Fila o registro |
| Cardinalidad | Numero de filas |
| Atributo | Columna o campo |
| Grado | Número de columnas |
| Clave primaria | Identificador único o clave primaria también |
| Dominio | "tipo de dato", conjunto de valores posibles, etc. |



DOMINIOS

Una colección de valores, de los cuales los atributos obtienen sus valores reales. Los dominios son la menor unidad de semántica de información desde el punto de vista del modelo, son atómicos, o sea que no se pueden "descomponer". Por supuesto que al decir "desde el punto de vista modelo", quiere decir que no es válido descomponer, por ejemplo un número de legajo válido en dígitos decimales. En este caso se pierde el significado.

En consecuencia un dominio es un conjunto de valores escalares, todos del mismo tipo.

Observación: como nulo no es un valor, los dominios no contienen nulos.

Importancia de los dominios: Los dominios nos permiten restringir las comparaciones.

El concepto de dominio es mucho más amplio que el de tipo de datos de un DBMS o de un lenguaje de programación.

RELACIONES

- **Definición:** una relación R se define sobre un conjunto de dominios D1,D2, ..., Dn y se compone de una cabecera y un cuerpo.
- **Cabecera:** está formada por un conjunto fijo de atributos, o mejor dicho pares atributo-dominio, tal que a cada atributo le corresponde un dominio.
Cabecera = { (A1:D1), (A2:D2), ..., (An:Dn). }
- **Cuerpo:** un conjunto de tuplas que varía con el tiempo. Cada tupla a su vez está formada por un conjunto de pares atributo-valor:
- **Tuplai=** { (A1:vi1) ... (An:vin) } con $1 \leq i \leq m$, siendo m cardinalidad de la relación, n grado de la relación

Una relación NO ES una tabla. La tabla es una representación de una relación. Es muy ventajoso en términos de uso y comprensión (esta una de las mayores ventajas del modelo relacional) que una relación tenga un equivalente informal tan simple de poner en papel como una "tabla". Sin embargo, una relación no sigue un orden determinado, como por ejemplo lo hace la tabla con sus columnas o sus filas.

El valor n se llama grado (cantidad de atributos de la relación). Los atributos de una relación no varían con el tiempo. Si hacemos un ALTER TABLE en SQL para agregar una columna,

modificamos la tabla, pero en lo que se refiere al modelo relacional, implica la creación de una nueva relación de grado $n+1$ a partir de una de grado n existente.

El valor m , es la cardinalidad de la relación. La cardinalidad es un concepto dinámico, varía con el tiempo. Es por ello básicamente que un dominio y una relación unaria no son la misma cosa.

PROPIEDADES DE LAS RELACIONES

1. No existen en ella tuplas repetidas
2. Las tuplas no están ordenadas
3. Los atributos no están ordenados
4. Todos los valores de los atributos son atómicos.

TIPOS DE RELACIONES

- **Relaciones base:** o relaciones reales, son el ejemplo de las tablas. Dada su importancia en el modelo de datos (debido a su necesidad en las distintas aplicaciones), el diseñador de la base le escogió un nombre y tiene existencia permanente ya que es parte de la base de datos en sí (son los datos).
- **Vistas:** Son relaciones virtuales, que se definen en términos de otras relaciones. Tienen un nombre definido.
- **Instantáneas o Snapshots:** también son relaciones derivadas, que tienen un nombre. La diferencia es que los datos son almacenados (copiados). La instantánea puede definir también si es actualizable y cada cuanto tiempo.
- **Resultados de consultas (queries).** No tienen existencia permanente dentro de la base, pero pueden ser nombradas.
- **Resultados intermedios de consultas** (por ejemplo los "subqueries"). No tienen existencia permanente dentro de la base y no pueden ser nombrados.
- **Relaciones temporales:** es nombrada, como 1,2, o 3, pero se destruye en forma automática en algún momento apropiado.

BASE DE DATOS RELACIONAL

"Es una base de datos percibida por el usuario como una colección de relaciones normalizadas de diversos grados que varía con el tiempo"

El modelo relacional define en general dos reglas de integridad aplicables a cualquier base de datos, que se definen a partir de los conceptos de clave primaria y clave foránea.

Las reglas de integridad formarán parte del modelo representado en la base y como tales deben ser definidas sobre las relaciones base, es decir, aquellas relaciones nombradas de existencia permanente en la base (las tablas).

CLAVE PRIMARIA

La relación tiene varias claves candidatas. De todas ellas, escojo una como clave primaria, y a las demás las llamamos alternativas.

Definición

El atributo K , simple o compuesto de una relación R , es clave candidata de $R \leftrightarrow$ se cumplen, independientemente del tiempo, las siguientes propiedades:

- **Unicidad**
 $\forall t_i, t_j \in R / i \neq j \Rightarrow K_i \neq K_j$
- **Minimalidad**
Si K es compuesta de r elementos, cualquier clave compuesta de un subconjunto de $r-1$ atributos de K , perderá la propiedad de unicidad.

Como consecuencia toda relación tiene una y solo una clave primaria. Esta es la clave que cobra importancia para la definición de las reglas de integridad de las entidades. Las demás son solo una consecuencia del método para la elección de una clave primaria. El concepto de clave primaria no

implica la existencia de un índice o cualquier otra estructura de acceso a los datos.

Las claves primarias son el único modo garantizado por el sistema de direccionamiento de una fila específica (tupla) dentro de una tabla (relación). Siempre que se solicite a un sistema relacional por las tuplas con un valor de clave primaria determinado, producirá a lo sumo una tupla.

REGLA DE INTEGRIDAD DE LAS ENTIDADES

"Ningún componente de la clave primaria de una relación base puede aceptar nulos"

No es solamente el "NULL". Por ejemplo, si el número de legajo es tipo CHAR, una cadena en blanco es un valor nulo para la presente definición.

CLAVE FORÁNEA

Una clave foránea es un atributo (o conjunto de atributos) de la relación R2 cuyos valores deben coincidir con los de la clave primaria de alguna relación R1. El valor de esta clave foránea nos da una referencia a la fila donde se encuentra el valor de clave primaria. La relación donde se encuentra la clave primaria es la "relación referenciada" o "relación objetivo".

Definición

El atributo K, simple o compuesto de una relación R2, es clave foránea de R2 \leftrightarrow Se cumplen, independientemente del tiempo, las siguientes propiedades:

- Cada valor K_i correspondiente a la tupla t_i , es totalmente nulo o totalmente no nulo (o bien todos los atributos que la componen son nulos, o bien son todos no nulos).
 - $\exists R1$, P / P es clave primaria de R1 y se satisface: $\forall K_i \in R2, K_i \text{ no nulo } \exists P_j \in R1 / K_i = P_j$
1. La concordancia de las claves primaria en R1 y foránea en R2 viene dada al estar definidas sobre el mismo dominio.
 2. La clave foránea puede o no ser parte de la clave primaria de R2, no importa para esta definición.
 3. R1 y R2 no son necesariamente distintas. Si $R1 = R2$, se dice que existe una relación autorreferencial.
 4. Las claves foráneas deben en ciertas ocasiones aceptar nulos. Por ejemplo, una relación autorreferencial para el caso de la primer tupla que ingrese al conjunto.

REGLA DE INTEGRIDAD REFERENCIAL

"La base de datos no debe contener valores no nulos de clave foránea para los cuales no exista un valor concordante de clave primaria en la relación referenciada"

MECANISMOS PARA EL MANEJO DE LA INTEGRIDAD REFERENCIAL

La regla de integridad referencial habla de mantener en un estado consistente de la base de datos. En consecuencia, cualquier operación que no deje la base en estado consistente será incorrecta. Para estas operaciones se deberá definir algún tipo de acción para mantener la integridad de los datos. El analista a cargo del diseño de la base debe definir la forma en la que el DBMS manejará la integridad referencial. Los aspectos a definir son:

1. Si la clave foránea acepta nulos.
2. La acción a llevar a cabo si se intenta eliminar un registro con una clave primaria referenciada por una clave foránea de otra relación. Existen tres opciones:
 1. RESTRICT: no se permite la eliminación del registro "padre".
 2. CASCADE: se eliminan también los registros que la referencian.
 3. ANULACION: se le asigna nulo a todas las claves foráneas (la clave foránea debe permitir nulos)
3. La acción a llevar a cabo si se intenta modificar la clave primaria de un registro referenciado.
También existen tres opciones:

1. RESTRICT: no se permite la modificación del registro “padre”.
2. CASCADE: se modifican también las claves foráneas que la referencian.
3. ANULACION: se le asigna nulo a todas las claves foráneas (la clave foránea debe permitir nulos)

Por último, debemos considerar que los mecanismos presentados son los más comunes: mediante la utilización de TRIGGERS, el diseñador de la base puede realizar operaciones más complejas que verifiquen que la base se encuentre en un estado consistente. O por ejemplo, guardar un log con las modificaciones. Inclusive aquellas que no involucran el concepto de integridad referencial.

Transacciones y Niveles de aislamiento

Definición:

Una transacción es un conjunto de operaciones que se ejecutan como una única unidad. Estas transacciones deben cumplir 4 propiedades fundamentales conocidas como ACID (atomicidad, consistencia, aislamiento y durabilidad).

Propiedades ACID

Atomicidad: cualquier cambio que produce una transacción es atómico. Es decir, se ejecutan todas las operaciones o no se ejecuta ninguna. En otras palabras, esta propiedad asegura que una transacción se realiza o no se realiza en forma completa

Consistencia: propiedad que asegura que una transacción no romperá la integridad de una base de datos.

Aislamiento: propiedad que asegura que no se afectarán entre sí las transacciones que se ejecuten de manera concurrente. Cada transacción puede trabajar con un conjunto de datos que no se vera alterado por otra transacción concurrente.

Durabilidad: propiedad que asegura la persistencia de una transacción, es decir, una vez que la transacción quedó aceptada no podrá deshacerse aunque falle el sistema.

Sentencias para una transacción

- **BEGIN TRAN [SACTION] [transaction_name | @tran_name_variable]**
Es la sentencia que se utiliza para iniciar una transacción explícita. En este momento los datos a los que accede la conexión son física y lógicamente coherentes. Incrementa @@TRANCOUNT en 1.
- **COMMIT [TRAN [SACTION] [transaction_name | @tran_name_variable]]**
Marca el final de una transacción correcta, implícita o definida por el usuario. Si @@TRANCOUNT es 1, COMMIT TRANSACTION hace que todas las modificaciones efectuadas sobre los datos desde el inicio de la transacción sean parte permanente de la base de datos, libera los recursos mantenidos por la conexión y reduce @@TRANCOUNT a 0.
- **ROLLBACK [TRAN [SACTION] [transaction_name | @tran_name_variable | savepoint_name | @savepoint_variable]]**
Deshace una transacción explícita o implícita hasta el inicio de la transacción o hasta un punto de almacenamiento dentro de una transacción. Elimina todas las modificaciones de datos realizadas desde el inicio de la transacción o hasta un punto de almacenamiento. También libera los recursos que retiene la transacción.
ROLLBACK TRANSACTION sin un savepoint_name o transaction_name deshace todas las instrucciones hasta el principio de la transacción. Cuando se trata de transacciones anidadas, esta misma instrucción deshace todas las transacciones internas hasta la instrucción BEGIN TRANSACTION más externa. En ambos casos, ROLLBACK TRANSACTION disminuye la función del sistema @@TRANCOUNT a 0, mientras que ROLLBACK TRANSACTION con savepoint_name no disminuye @@TRANCOUNT.
- **SAVE [TRAN | TRANSACTION] { savepoint_name | @savepoint_variable }**

El punto de retorno define una ubicación a la que la transacción puede volver si se cancela parte de la transacción de forma condicional. Si se revierte una transacción hasta un punto de retorno, se debe continuar hasta su finalización con más instrucciones Transact-SQL si es necesario y una instrucción COMMIT TRANSACTION o se debe cancelar completamente al revertir la transacción hasta su inicio.

Transacciones anidadas

MSSQL permite el anidamiento de transacciones, para esto es fundamental tener en cuenta que existe una variable global @@TRANCOUNT que tiene valor 0 si no existe ningún nivel de anidamiento, 1 si hay una transacción anidada, 2 si estamos en el segundo nivel de anidamiento, y así sucesivamente.

La dificultad de trabajar con transacciones anidadas está en el comportamiento que tienen ahora las sentencias 'COMMIT TRAN' y 'ROLLBACK TRAN'.

ROLLBACK TRAN: Dentro de una transacción anidada esta sentencia deshace todas las transacciones internas hasta la instrucción BEGIN TRANSACTION más externa.

COMMIT TRAN: Dentro de una transacción anidada esta sentencia únicamente reduce en 1 el valor de @@TRANCOUNT, pero no "finaliza" ninguna transacción ni "guarda" los cambios. En el caso en el que @@TRANCOUNT=1 (cuando estamos en la última transacción) COMMIT TRAN hace que todas las modificaciones efectuadas sobre los datos desde el inicio de la transacción sean parte permanente de la base de datos, libera los recursos mantenidos por la conexión y reduce @@TRANCOUNT a 0.

Ejemplo 1

CREATE TABLE prueba (Columna int)

go

```
BEGIN TRAN TranExterna -- @@TRANCOUNT ahora es 1
    SELECT 'El anidamiento es', @@TRANCOUNT
    INSERT INTO prueba VALUES (1)
    BEGIN TRAN TranInterna1 -- @@TRANCOUNT ahora es 2.
        SELECT 'El anidamiento es', @@TRANCOUNT
        INSERT INTO prueba VALUES (2)
        SAVE TRAN Insert2
        BEGIN TRAN TranInterna2 -- @@TRANCOUNT ahora es 3.
            SELECT 'El anidamiento es', @@TRANCOUNT
            INSERT INTO prueba VALUES (3)
        ROLLBACK TRAN Insert2 -- se deshace lo hecho al punto guardado.
            SELECT 'El anidamiento es', @@TRANCOUNT
        COMMIT TRAN TranInterna2 -- Reduce @@TRANCOUNT a 2.
        SELECT 'El anidamiento es', @@TRANCOUNT
    COMMIT TRAN TranInterna1 -- Reduce @@TRANCOUNT a 1.
    SELECT 'El anidamiento es', @@TRANCOUNT
COMMIT TRAN TranExterna -- Reduce @@TRANCOUNT a 0.
-- Se lleva a cabo la transacción externa y todo se guarda en la base de datos.
SELECT 'El anidamiento es', @@TRANCOUNT
SELECT * FROM prueba
```

Ejemplo 2

--TRANSACCIONES EN SP (INSERTAR 1, 3
Y NO INSERTAR EL 2)

--FORMA INCORRECTA

```
CREATE PROCEDURE Inserta2 AS
BEGIN TRAN --Uno
```

```

        INSERT INTO Prueba VALUES (2)
ROLLBACK TRAN --Uno
GO
CREATE PROCEDURE Inserta1
AS
BEGIN TRAN --Dos
        INSERT INTO Prueba VALUES ( 1)
EXEC Inserta2
        INSERT INTO Prueba VALUES (3)
COMMIT TRAN --Dos
GO

```

Ejemplo 2

--FORMA CORRECTA

```

CREATE PROCEDURE Inserta2 AS
SAVE TRAN PasoGuardado
        INSERT INTO Prueba VALUES (2)
ROLLBACK TRAN PasoGuardado
GO
CREATE PROCEDURE Inserta1
AS
BEGIN TRAN --Dos
        INSERT INTO Prueba VALUES ( 1)
EXEC Inserta2
        INSERT INTO Prueba VALUES (3)
COMMIT TRAN --Dos
GO

```

Niveles de aislamiento

El nivel de aislamiento de una transacción (transaction isolation level) define el grado en que se aísla una transacción de las modificaciones de recursos o datos realizadas por otras transacciones. El comportamiento por defecto de SQL Server en las operaciones de lectura y de escritura:

- En operaciones de escritura. Siempre se obtiene un bloqueo exclusivo que se mantiene hasta que se completa la transacción.
- En operaciones de lectura. El comportamiento dependerá del nivel de aislamiento de la transacción. Por defecto, SQL Server utiliza el modo de aislamiento basado en bloqueos READ COMMITTED.

Tipos de niveles

- **READ COMMITTED:** permite que entre dos lecturas de un mismo registro en una transacción A, otra transacción B pueda modificar dicho registro, obteniéndose diferentes resultados de la misma lectura. Evita las lecturas sucias (dirty reads), pero por el contrario, permite lecturas no repetibles. Es la opción por defecto en SQL. Con este nivel de aislamiento, una operación de lectura (SELECT) establecerá bloqueos compartidos (shared locks) sobre los datos que está leyendo. Sin embargo, dichos bloqueos compartidos finalizarán junto con la propia operación de lectura, de tal modo que entre dos lecturas cabe la posibilidad de que otra transacción realice una operación de escritura (ej: UPDATE), en cuyo caso, la segunda lectura obtendrá datos distintos a la primera lectura (lecturas no repetibles).
- **READ UNCOMMITTED:** puede recuperar datos modificados pero no confirmados por otras transacciones (lecturas sucias - dirty reads). En este nivel se pueden producir todos los efectos secundarios de simultaneidad (lecturas sucias, lecturas no repetibles y lecturas fantasma - ej: entre dos lecturas de un mismo registro en una transacción A, otra transacción B puede modificar dicho registro), pero no hay bloqueos ni versiones de lectura, por lo que se minimiza la sobrecarga.
- **REPEATABLE READ:** evita que entre dos lecturas de un mismo registro en una transacción A, otra transacción B pueda modificar dicho registro, con el efecto de que en la segunda lectura de la transacción A se obtuviera un dato diferente. De este modo, ambas lecturas serían iguales (lecturas repetidas). Para ello, una operación de lectura (SELECT) establecerá bloqueos compartidos (shared locks) sobre los datos que está leyendo, y los

mantendrá hasta el final de la transacción, garantizando así que no se produce lecturas no repetibles (non repeatable reads). Sin embargo, este modo de aislamiento no evita las lecturas fantasma, es decir, una transacción podría ejecutar una consulta sobre un rango de filas (ej: 100 filas) y de forma simultánea otra transacción podría realizar un inserción de una o varias filas sobre el mismo rango.

- **SERIALIZABLE:** garantiza que una transacción recuperará exactamente los mismos datos cada vez que repita una operación de lectura (es decir, la misma sentencia SELECT con la misma cláusula WHERE devolverá el mismo número de filas, luego no se podrán insertar filas nuevas en el rango cubierto por la WHERE, etc. - se evitarán las lecturas fantasma), aunque para ello aplicará un nivel de bloqueo que puede afectar a los demás usuarios en los sistemas multiusuario (realizará un bloqueo de un rango de índice - conforme a la cláusula WHERE - y si no es posible bloqueará toda la tabla). Evita los problemas de las lecturas sucias (dirty reads), de las lecturas no repetibles (non repeatable reads), y de las lecturas fantasma (phantom reads).

La instrucción set transaction isolation level se utiliza para especificar el nivel de aislamiento de la transacción

```
SET TRANSACTION ISOLATION LEVEL
{ READ COMMITTED | READ UNCOMMITTED
| REPEATABLE READ | SERIALIZABLE
}
```

| Nivel de aislamiento | Lectura sucia | Lectura no repetible | Lectura Fantasma |
|----------------------|---------------|----------------------|------------------|
| READ UNCOMMITTED | Sí | Sí | Sí |
| READ COMMITTED | No | Sí | Sí |
| REPEATABLE READ | No | No | Sí |
| SERIALIZABLE | No | No | No |

Bases de datos orientadas a objetos

¿Qué es una BDOO?

Es una base de datos inteligente. Soporta el paradigma orientado a objetos almacenando datos y métodos, y no sólo datos. Está diseñada para ser eficaz, desde el punto de vista físico, para almacenar objetos complejos.

Evita el acceso a los datos; esto es mediante los métodos almacenados en ella. Es más segura ya que no permite tener acceso a los datos (objetos); esto debido a que para poder entrar se tiene que hacer por los métodos que haya utilizado el programador.

Arquitectura de Una BDOO

Los primeros se diseñaron como una extensión de los lenguajes de programación como Smalltalk ó C++. El LMD (lenguaje para el manipulación de datos; también conocido como DML) y el LDD (lenguaje para la definición de los datos; también conocido como DDL) constrúan un lenguaje OO común.

El diseño de las BDOO actuales debe aprovechar al máximo el CASE e incorporar métodos creados con cualquier técnica poderosa, incluyendo enunciados declarativos, generadores de códigos e inferencias con base en reglas.

Características:

- **Versiones:** La mayoría de los sistemas de bases de datos sólo permiten que exista una representación de un ente de la base de datos dentro de esta. Las versiones permiten que las representaciones alternas existan en forma simultánea.
- **Transacciones compartidas:** Las transacciones compartidas soportan grupos de usuarios en estaciones de trabajo, los cuales desean coordinar sus esfuerzos en tiempo real, los usuarios pueden compartir los resultados intermedios de una base de datos. La transacción compartida permite que varias personas intervengan en una sola transacción

Desarrollo con Bases de Datos OO

Las BDOO se desarrollan al describir en primer lugar los tipos de objetos importantes del dominio

de aquellos tipos de objetos. Estos tipos de objetos determinan las clases que conformarán la definición de la BDOO.

Tres Enfoques de Construcción de Bases de Datos OO

- El Primero: se puede utilizar el código actual altamente complejo de los sistemas de administración de las bases de datos, de modo que una BDOO se implante más rápido sin tener que iniciar de cero. Las técnicas orientadas a objetos se pueden utilizar como medios para el diseño sencillo de sistemas complejos. Los sistemas se construyen a partir de componentes ya probados con un formato definido para las solicitudes de las operaciones del componente.
- El Segundo: considera a la BDOO como una extensión de la tecnología de las bases de datos por relación. De este modo, las herramientas, técnicas, y vasta experiencia de la tecnología por relación se utilizan para construir un nuevo SABD. Se pueden añadir apuntadores a las tablas de relación para ligarlas con objetos binarios de gran tamaño (BLOB). La base de datos también debe proporcionar a las aplicaciones clientes un acceso aleatorio y por partes a grandes objetos, con el fin de que sólo sea necesario recuperar a través de la red la parte solicitada de los datos.
- El Tercero: reflexiona sobre la arquitectura de los sistemas de bases de datos y produce una nueva arquitectura optimizada, que cumple las necesidades de la tecnología OO. La tecnología de relación es un subconjunto de una capacidad más general. Además que las BDOO no de relación son aproximadamente dos veces más rápidas que las bases de datos por relación para almacenar y recuperar la información compleja.

Ventajas en BDOO's

1. Flexibilidad, y soporte para el manejo de tipos de datos complejos

- Mediante la utilización de subclases se heredan todos los atributos, características de una definición original, y se pueden especificar nuevos campos que se requieren así como los métodos para manipular solamente estos campos. Naturalmente se generan los espacios para almacenar la información adicional de los nuevos campos. Esto presenta la ventaja adicional que una BDOO puede ajustarse a usar siempre el espacio de los campos que son necesarios, eliminando espacio desperdiciado en registros con campos que nunca usan.

2. Manipula datos complejos en forma rápida y ágilmente.

- La estructura de la base de datos está dada por referencias (o apuntadores lógicos) entre objetos.

Posibles Desventajas

1. Inmadurez del mercado de BDOO

- Constituye una posible fuente de problemas por lo que debe analizarse con detalle la presencia en el mercado del proveedor para adoptar su producto en una línea de producción sustantiva.

2. Falta de estándares en la industria orientada a objetos

Rendimiento

Las BDOO permiten que los objetos hagan referencia directamente a otro mediante apuntadores suaves. Esto hace que las BDOO pasen más rápido del objeto A al objeto B que las BDR, las cuales deben utilizar comandos JOIN para lograr esto. Incluso el JOIN optimizado es más lento que un recorrido de los objetos. Así, incluso sin alguna afinación especial, una BDOO es en general más rápida en esta mecánica de caza-apuntadores.

Las BDOO hacen que el agrupamiento sea más eficiente. La mayoría de los sistemas de bases de datos permiten que el operador coloque cerca las estructuras relacionadas entre sí, en el espacio de

almacenamiento en disco. Esto reduce en forma radical el tiempo de recuperación de los datos relacionados, puesto que todos los datos se leen con una lectura de disco en vez de varias.

Características mandatorias ó reglas de oro

Un sistema de BDOO debe satisfacer dos criterios:

1. Debe tener un BDMS

- Persistencia
- Manejador de almacenamiento secundario
- Concurrencia
- Recuperación
- Facilidad de Query

2. Debe ser un sistema OO

- Objetos Complejos
- Identidad del objeto
- Encapsulamiento
- Tipos ó Clases
- Sobre paso con combinación retrasada
- Extensibilidad
- Completado Computacional.

Manifiesto de sistema de gestión de BDOO

- Mandatorias: son las que el Sistema debe satisfacer a orden de tener un sistema de BDOO
 - Predominancia combinada con enlace retardado: se puede definir que sea Excel, Autocad, etc. desde la programación.
 - Extensibilidad: proporciona los tipos de datos como: Caracter, booleano, String, etc.
 - Concurrencia: permite que varios usuarios tengan acceso a una BD al mismo tiempo.
 - Recuperación: cuando se hace una transacción pero no se puede realizar y se regresa al mismo estado.
 - Facilidad de “Consultas a Modo”: esto es que se tienen diferentes estándares.
- Opcionales: son las que pueden ser añadidas para hacer el sistema mejor pero que no son Mandatorias
 - Herencia Múltiple: tienen características de padres diferentes y proporcionan mecanismos para saber de 2 o más opciones cual conviene.
 - Verificación de tipos de inferencia.
 - Distribución: que se puede tener parte de una BD en un servidor y otra parte en otro.
 - Sistemas de Representación: forma en como se presentan los esquemas.
 - Uniformidad: todo debe ser igual. Diseño de ventanas, etc.
 - Asociaciones y Cardinalidad de Asociaciones: Cardinalidad: 1:1 (Uno a Uno), 1:M (Uno a Muchos), M:1 (Muchos a Uno), M:M (Muchos a Muchos).
- Abiertas: son los puntos donde el diseñador puede hacer un número de opciones y estas son el paradigma de la programación la representación del sistema ó el tipo de sistema y su uniformidad.
 - Es como si fuera una especialización con cierta marca de software.

Otras Características

Generales:

- **Control de Concurrency**

- Modo Pesimista.- Tomo el dato no dejo que nadie lo tome para que no accedan al mismo dato.
- Modo Optimista.- Es el que dice yo le saco una copia y piensa nadie lo va a modificar.
- Modo Mixto.- Combinación del Pesimista y el Optimista.
- Modo Semioptimista.- Toma las virtudes del Optimista.

Bloqueos

- Bloqueos de Lectura.- Leer una dato y que no quieres que nadie lo modifique mientras los estas usando.
- Bloqueos de Escritura.- Bloquear el Objeto mientras yo estoy escribiendo(nadie más puede escribir).
- Bloqueos Nulos.- Es para Sincronización.(ejem. "papel Out"notificación de una impresora).
- Bloqueos de Notificación.- Es para Sincronización.(ejem."papel Out" notificación de una impresora).

SISTEMA DE GESTION DE BDOO (SGBDOO)

Un Sistema de Gestión de Bases de Datos (SGBD) es un conjunto de datos relacionados entre sí y un grupo de programas para tener acceso a esos datos.

Un Sistema de Gestión de Bases de Datos Orientadas a Objetos (SGBDOO) se puede decir que es un SGBD que almacena objetos, permitiendo concurrencia, recuperación. Para los usuarios tradicionales de bases de datos, esto quiere decir que pueden tratar directamente con objetos, no teniendo que hacer la traducción a registros o tablas.

Características de los SGBDOO

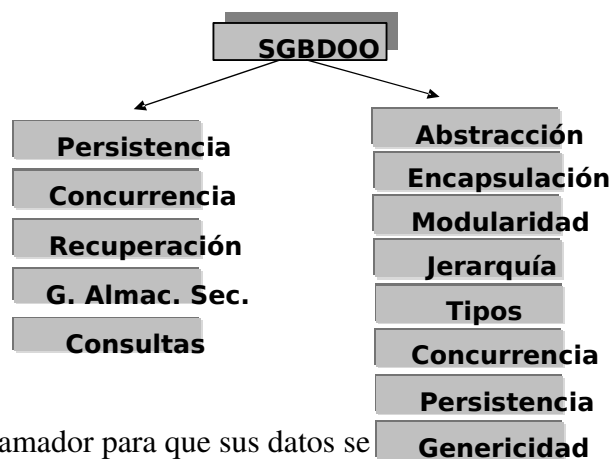
Debe satisfacer dos criterios:

1. Ser un sistema orientado a objetos

- abstracción,
- encapsulación,
- modularidad,
- jerarquía, control de tipos,
- concurrencia,
- persistencia
- genericidad.

2. Ser un sistema de gestión de bases de datos.

- **Persistencia:** Es la capacidad que tiene el programador para que sus datos se conserven al finalizar la ejecución de un proceso, de forma que se puedan reutilizar en otros procesos.
- **Concurrencia:** Se relaciona con la existencia de muchos usuarios interactuando concurrentemente en el sistema. Este debe controlar la interacción entre las transacciones concurrentes para evitar que se destruya la consistencia de la base de datos.
- **Recuperación ante fallos del sistema:** Proporcionar como mínimo el mismo nivel de recuperación que los sistemas de bases de datos actuales. De forma que, tanto en caso de fallo de hardware como de fallo de software, el sistema pueda retroceder hasta un estado coherente de los datos.
- **Gestión del almacenamiento secundario:** es soportada por un conjunto de mecanismos que no son visibles al usuario, tales como gestión de índices, agrupación de datos, selección del camino de acceso, optimización de consultas, etc. Estos mecanismos evitan



que los programadores tengan que escribir programas para mantener índices, asignar el almacenamiento en disco, o trasladar los datos entre el disco y la memoria principal, creándose de esta forma una independencia entre los niveles lógicos y físicos del sistema.

- **Facilidad de consultas:** Permitir al usuario hacer cuestiones sencillas a la base de datos. Este tipo de consultas tienen como misión proporcionar la información solicitada por el usuario de una forma correcta y rápida.

Primer intento de Estandarización: ODMG-93.

El lenguaje de bases de datos es especificado mediante un lenguaje de definición de datos (ODL), un lenguaje de manipulación de datos (OML), y un lenguaje de consulta (OQL), siendo todos ellos portables a otros sistemas con el fin de conseguir la portabilidad de la aplicación completa.

Lenguaje ODL

El lenguaje de definición de datos (ODL) en un SGBDOO es empleado para facilitar la portabilidad de los esquemas de las bases de datos. Este ODL no es un lenguaje de programación completo, define las propiedades y los prototipos de las operaciones de los tipos, pero no los métodos que implementan esas operaciones.

El ODL intenta definir tipos que puedan implementarse en diversos lenguajes de programación; no está por tanto ligado a la sintaxis concreta de un lenguaje de programación particular. De esta forma un esquema especificado en ODL puede ser soportado por cualquier SGBDOO que sea compatible con ODMG-93.

Lenguaje OML

El lenguaje de manipulación es empleado para la elaboración de programas que permitan crear, modificar y borrar datos que constituyen la base de datos. ODMG-93 sugiere que este lenguaje sea la extensión de un lenguaje de programación, de forma que se pueden realizar entre otras las siguientes operaciones sobre la base de datos: Creación, Borrado, Modificación e Identificación de un objeto

Lenguaje OQL

El lenguaje de consulta propuesto por ODMG-93, presenta las siguientes características:

- No es computacionalmente completo. Sin embargo, las consultas pueden invocar métodos, e inversamente los métodos escritos en cualquier lenguaje de programación pueden incluir consultas.
- Tiene una sintaxis abstracta.
- Su semántica formal puede definirse fácilmente.
- Proporciona un acceso declarativo a los objetos.
- Se basa en el modelo de objetos de ODMG-93.
- Tiene una sintaxis concreta al estilo SQL, pero puede cambiarse con facilidad.
- Puede optimizarse fácilmente.
- No proporciona operadores explícitos para la modificación, se basa en las operaciones definidas sobre los objetos para ese fin.
- Proporciona primitivas de alto nivel para tratar con conjuntos de objetos, pero no restringe su utilización con otros constructores de colecciones.

Existen dos posibilidades para asociar un sublenguaje de consulta a un lenguaje de programación: fuerte y débilmente.

- El primer caso consiste en una extensión de la gramática del lenguaje asociado.
- En el segundo caso, las funciones query tienen unos argumentos String que contienen las preguntas.

Objetivos

El desarrollo del SGBDOO tiene como finalidad principal la verificación de las hipótesis que se plantean a continuación:

1. **Desarrollo más sencillo del propio SGBDOO.** Esto parece lógico ya que algunas de las funciones que debería implementar el SGBD (ej. Persistencia) ya están disponibles dentro del propio sistema operativo. Al tratarse de un sistema integral orientado a objetos, se obtienen las ventajas de la orientación a objetos: así por ejemplo, es posible reutilizar el código de persistencia ya existente, o extenderlo añadiendo únicamente la funcionalidad adicional necesaria para el SGBDOO. Esta funcionalidad puede proporcionarse mediante un motor de base de datos adecuado que complemente las características proporcionadas por el sistema operativo.
2. **Mayor integración en el sistema.** Es decir, los objetos de la base de datos son simplemente unos objetos más dentro de los objetos del sistema operativo que proporcionan servicios. Es más, puede pensarse en el SGBDOO como el elemento que cumpla el papel de los sistemas de ficheros en los sistemas operativos tradicionales. El SGBDOO no sería utilizado como un sistema independiente del sistema operativo, si no que el usuario podría utilizarlo como sistema de gestión de los objetos del sistema operativo, haciendo consulta sobre los mismos.
3. **Mayor rendimiento.** Dado que el propio sistema integral ya es orientado a objetos, no existe la necesidad de desarrollar capas superpuestas a un sistema operativo tradicional para salvar el espacio existente entre el paradigma del sistema operativo y el de la base de datos.
4. **Mayor productividad.** La programación de aplicaciones de bases de datos es más productiva ya que no es necesario que el programador cambie constantemente de filosofías: una para trabajar con la base de datos y otra para manejar el sistema operativo. Ambos elementos utilizan ahora el mismo paradigma de orientación a objetos.

DATAWAREHOUSE

PORQUE SEPARAR DW DE BD OPERACIONAL?

Por 2 grandes motivos:

- ✓ **PERFORMANCE:** Las bases de datos operacionales están diseñadas y optimizadas para cargas de trabajo y volumen de transacciones conocidos. Por otro lado, consultas analíticas complejas podrían degradar la performance de las operaciones transaccionales que son críticas para el funcionamiento del negocio. Por otro lado el DW necesita tener los datos organizados de manera desnormalizada, trabajando de manera multidimensional.
- ✓ **FUNCIONALIDAD:** los DW mantienen información históricas que la DB OPERACIONAL pensando en la performance no hacen. Además en los DW se agregan datos desde diversas fuentes (encuestas, otras consultas, archivos de texto, etc.), que la BD operacional no trabaja

Componentes

DTS: Data transfer system, se utiliza para transformar un sistema OLTP en uno OLAP.

Mientras que OLTP está normalizado OLAP se basa en la des normalización y redundancia de la información para la toma de decisiones. se ejecuta periódicamente para generar información.

DATAWAREHOUSE: Es un almacén de información donde la misma se guarda en forma desnormalizada. La información proviene de distintas partes de un sistema e incluso de diferentes sistemas externos. La idea de un data Warehouse es mediante el data minining explotar la información contenida y poder sacar buena información para la toma de decisiones.

Los diferentes modelos de construcción son MOLAP (multidimensional), ROLAP (relacional) y

HOLAP (hibrido).

Permite :

- Integración de bases de datos heterogéneas (relacionales, documentales, Geográficas, archivos, etc.).
- Ejecución de consultas complejas no predefinidas visualizando el resultado en forma de gráfica y en diferentes niveles de agrupamiento y totalización de datos.

Un Data Warehouse es una colección de datos

- ✓ **orientada a sujetos:** Un primer aspecto de un DW es que está orientado a los mayores sujetos de la empresa. El mundo operacional está diseñado alrededor de aplicaciones y funciones, como por ejemplo pagos, ventas, entregas de mercadería, para una institución comercial. Un DW está organizado alrededor de los mayores sujetos, como cliente, vendedor, producto y actividades.
- ✓ **Integrada:** Cuando los datos son movidos del ambiente operacional, son integrado antes de entrar en el warehouse. Por ejemplo, un diseñador puede representar el sexo como "M" y "F", otro puede representarlo como "0" y "1", o "x" e "y", y otro usar las palabras completas "masculino" y "femenino". No importa la fuente de la cual el sexo llegue al DW, debe ser guardado en forma consistente; los datos deben ser integrados.
- ✓ **variante en el tiempo:** Los datos en el warehouse son precisos para un cierto momento, no necesariamente ahora; por eso se dice que los datos en el warehouse son variantes en el tiempo. Los datos de un warehouse, una vez almacenados, no pueden ser modificados (no se permiten updates). En el ambiente operacional, los datos, precisos al momento de acceso, pueden ser actualizados, según sea necesario.
- ✓ **simple de utilizar:** Solo ocurren dos operaciones, la carga inicial, y el acceso a los datos. No hay necesidad de updates (en su sentido general). Hay consecuencias muy importantes de esta diferencia de procesos con un sistema operacional: nivel de diseño, en un warehouse, no hay que controlar anomalías producidas por los updates, ya que no hay updates. Se pueden tomar libertades de diseño físico como optimizar el acceso a los datos, y de normalización física. Muy simple para backup, integridad, etc.
- ✓ **no volátil:** la información es útil sólo cuando es estable. Los datos operacionales cambian sobre una base momento a momento. La perspectiva más grande, esencial para el análisis y la toma de decisiones, requiere una base de datos estable.
- ✓ **soporta el proceso de toma de decisiones (PRINCIPAL FUNCION).**

DATASmart: es un componente específico dentro de un DataWarehouse, está centrado en un área específica de la empresa. Son mucho menos costosos, pero tienen un alcance mucho más limitado, ya que se enfocan en un conjunto concreto de necesidades y de una sola área, no de toda la organización. Es un subconjunto de un DW.

Generalmente se utilizan BD multidimensionales, a diferencia de los DW que usan DB relacionales, y que cuentan con info de toda la empresa.

TABLA DE HECHOS

Es la tabla primaria en cada modelo dimensional, orientada a contener las medidas del negocio.

Cada fact table representa una relación de muchos a muchos y contiene un conjunto de dos o más foreign keys (claves foráneas) que hacen referencias a sus respectivas tablas de dimensión

TABLA DE DIMENSION

La Dimension Table restringe los criterios de selección de los datos de la fact table. Cada dimensión está definida por su clave primaria que sirve como base para la integridad referencial con la Fact table con la cual hace join. Muchas tablas de dimensión contienen atributos textuales (campos) que son la base para restringir y agrupar las consultas sobre el data warehouse.

Dependiendo en cómo se elija organizar el esquema físico, una tabla de dimensión puede almacenar información de uno o más atributos relacionados. (puede estar normalizada o desnormalizada dependiendo del modelo COPO DE NIEVE o STAR).

OLAP: procesamiento analítico en línea (On-Line Analytical Processing). Es una solución utilizada en el campo de la llamada Inteligencia empresarial (o Business Intelligence) cuyo objetivo es agilizar la consulta de grandes cantidades de datos. Para ello utiliza estructuras multidimensionales (o Cubos OLAP) que contienen datos resumidos de grandes Bases de datos o Sistemas Transaccionales (OLTP). Se usa en informes de negocios de ventas, marketing, informes de dirección, minería de datos y áreas similares.

Ofrecen al usuario una vista multidimensional de los datos, y un lenguaje de consulta de alto nivel y veloz, gracias a desnormalización de los datos.

Que mejora?

Mejora debido a la redundancia y desnormalización de la información. Por ende no es necesario hacer joins entre tablas y **recalcular datos en cada acceso**, y la información se devuelve más rápidamente. Es poco performante en cuanto a espacio ocupado en disco. La redundancia hace que el modelo sea muy grande y poco mantenible.

Existen distintas implementación de OLAP:

- **MOLAP (Multidimensional OLAP):** Cuando la tecnología OLAP se monta sobre una base de datos multidimensional se la suele denominar MOLAP. Los datos se almacenan en estructuras de almacenamiento basadas en arrays. Las herramientas MOLAP brindan acceso directo a estructuras de datos de este tipo.
- **ROLAP (Relational OLAP):** Es un conjunto de interfases de usuario y aplicaciones que le dan a una base de datos relacional un aspecto de base de datos multidimensional. Es decir, se simula un cubo dimensional en una base de datos relacional.
- **HOLAP (Hybrid OLAP):** es una combinación de las dos anteriores.

MODELO STAR: El modelo se utiliza en las tecnologías OLAP y se basa en almacenar los datos en una “**tabla de hechos**” y tantas “tablas de dimensiones” como dimensiones haya – 1 (una dimensión es el tiempo, y no tiene tabla propia).

- ✓ El star schema no captura las jerarquías directamente. La performance se mejora reduciendo el número de tablas implicada en la consulta. Este diseño simple hace a los datos menos complicados. Todas las tablas de dimensión hacen join a la tabla fact en una forma claramente definida.
- ✓ Alrededor de esta tabla central se definen otras tablas ligadas a esta mediante claves foráneas, y se denominan tablas de dimensiones y son aquellas que definen cualidades del atributo representado por la tabla de hechos.

- ✓ La clave primaria de la tabla de hechos será una clave compuesta por todas las claves foráneas que representan las claves primarias de cada una de las dimensiones asociadas.

Las dimensiones, aunque no es necesario ni parte del modelo, pueden estar entre si normalizadas (aunque independientes una de las otras), dando origen al modelo **COPO DE NIEVE**. El mismo es es más fácil de mantener y reduce el espacio de almacenamiento a través de la normalización de sus tablas. Sin embargo, algunos autores establecen que las búsquedas son mas lentas.

BD MULTIDIMENSIONAL: es aquella que almacena sus datos con varias dimensiones, es decir que en vez de un valor, encontramos varios dependiendo de los "ejes" definidos. Veamos un ejemplo:

Una tabla relacional de productos podría tener 2 campos ID, Nombre del producto y existencias. En un entorno multidimensional con dos dimensiones, tiempo y espacio, tendríamos por cada entrada N valores dependiendo de estos dos ejes y asi podríamos observar el número de existencias en el tiempo (histórico) y en las diferentes sedes de la empresa.

Existen 2 grandes tipos:

- ✓ **Hipercubo:** Dimensiona a partir de un cubo de grandes dimensiones. Tiene dispersión de datos (Ésta se produce cuando no todas las celdas del cubo se rellenan con datos (escasez de datos o valores nulos) y estos están alejados. (estático)
Se guardan todas las dimensiones en un cubo.
Cada intersección del cubo es otro cubo y así sucesivamente.
- ✓ **Multicubo:** Dinamiza el modelo, si una cara no existe se tiene un puntero a la próxima cara.
Es bueno cuando hay alta dispersión de datos. (dinámico)

¿En qué consiste la técnica de Data Mining? ¿Para qué se utiliza? Indique alguna de las técnicas algorítmicas aplicadas en el mismo.

- ✓ Consiste en la extracción de información oculta y predecible de grandes bases de datos.
- ✓ Las herramientas de datamining predicen futuras tendencias y comportamientos, permitiendo tomar decisiones conducidas por un conocimiento acabado de la información (knowledge-driven).
- ✓ Predicción automatizada de tendencias y comportamientos.
- ✓ Descubrimientos automatizados de modelos previamente desconocidos.
- ✓ Las técnicas más comúnmente usadas son: redes neuronales, árboles de decisión, algoritmos genéticos, etc.

Apuntes incluidos:

- Bases de datos del Ing Zaffaroni
- Modelo Relacional del Ing Zaffaroni
- Transacciones y Niveles de Aislamiento del Ing Restagno
- Bases de Datos Orientadas a Objetos del Ing Reinoso
- Datawarehouse- Ing Zaffaroni + Ing Reinoso

Apuntes no incluidos

- Data Base Administration del Ing Zaffaroni

Anexo Preguntas o Conceptos de Final

FUNCIONALIDADES Y CARACTERISTICAS DE UN DBMS:

- ✓ **Concurrencia:** permite el acceso simultáneo de muchos usuarios/procesos.
- ✓ **Atomicidad:** asegura que un conjunto de operaciones se ejecuta todo o falla en su conjunto.
- ✓ **Consistencia:** asegura que la integridad de los datos se mantiene antes y después de realizar una operación.
- ✓ **Backup y Restore:** provee herramientas para realizarlos.
- ✓ **Gestión de almacenamiento:** independiza al usuario de los detalles de bajo nivel de cómo se almacenan los datos. (tratamiento con FS)
- ✓ **Aislamiento:** usuario se abstrae de la implementación física del modelo.
- ✓ **Facilidad de consultas.**
- ✓ **Transaccionalidad.**
- ✓ **Abstracción.**
- ✓ **Procesamiento distribuido:** las diferentes capas o sistemas de un DBMS interactúan entre sí para llevar a cabo los objetivos generales de un DBMS.

QUE DEBE CUMPLIR UN MOTOR DE BD

- ✓ Relación costo/beneficio.
- ✓ Volúmenes de datos a manejar.
- ✓ Tipos de bloqueos.
- ✓ Concurrencia.
- ✓ Si el sistema es o no orientado a objetos.
- ✓ Capacidad del equipo de trabajo.
- ✓ Plataforma (Hardware y SO)

Características de los DSS Queries (Data Warehouse) -> OLAP:

- ✓ Muchas filas son leídas y el resultado no está en una transacción
- ✓ Los datos son leídos secuencialmente
- ✓ SQL complejos son ejecutados
- ✓ Grandes archivos temporarios son creados
- ✓ El tiempo de respuesta es medido en horas y minutos
- ✓ Hay relativamente poca concurrencia de queries

CARACTERISTICAS DEL Ambiente OLTP: On-Line Transaction Processing

- ✓ Relativamente pocas filas leídas
- ✓ Alta frecuencia de transacciones
- ✓ Acceso a los datos a través de índices
- ✓ Simple Operaciones SQL
- ✓ Respuesta de scan medida en segundos
- ✓ Muchos queries concurrentes

Existen 3 niveles de Seguridad:

- ✓ a) Base de datos
- ✓ b) Tabla
- ✓ c) Columna (depende del motor de bd)

PLAN DE CONSULTA: Consiste en que el SGBD tiene armado un plan de cómo va a acceder a los datos, esto hace que el acceso sea más rápido. Los SP, Vistas y Funciones implementan un plan de ejecución, es por esto que son más veloces que un query normal.

-El optimizador de consultas es el componente del sistema de gestión de base de -datos que intenta determinar la forma más eficiente de ejecutar una consulta SQL.

-Se decide cuál de los posibles planes de ejecución de una consulta dada es el más eficiente.

-Los optimizadores basados en costos asignan un costo (operaciones de E/S requeridas, CPU, etc.) a cada uno de esos planes y se elige el que tiene menor costo.

Por qué se establece que las Reglas del Negocio deben estar en el Motor de Base de Datos y no en la aplicación cliente?

- ✓ Reglas de negocio: cada aplicación debe reflejar las restricciones que existen en el negocio dado, de modo que nunca sea posible realizar acciones no válidas.
- ✓ La información puede ser manipulada por muchos programas distintos que podrán variar de acuerdo a los departamentos de la organización, los cuales tendrán distintas visiones y necesidades pero que en cualquier caso, siempre deberán respetar las reglas de negocio.
- ✓ Es por lo anterior expuesto que las reglas del negocio deben estar en el motor de base de datos.

Describe los componentes básicos de un DBMS. Teniendo en cuenta la administración de datos, la interfaz con el usuario y el procesamiento cliente-servidor.

- ✓ Dos lenguajes: DML y DDL.
- ✓ 3 capas: externa (usuario), lógica (conceptual) e interna (física).
- ✓ Disk manager (recibe peticiones del file manager y las envía al SO), file manager (decide que página contiene el registro deseado y la solicita el disk manager) y user manager (permite la interacción con el usuario).
- ✓ Arquitectura cliente-servidor: las aplicaciones corren en el cliente y generan solicitudes para y reciben respuestas del servidor. El servidor realiza el procesamiento de datos y aplica las reglas de negocio establecidas.
- ✓ Clustering: es la forma de agrupar la información. Hay dos formas, intra-file clustering (todas las tablas conviven con sus elementos) e inter-file clustering (se guarda cada objeto con sus relaciones).

¿Qué beneficios brinda la aplicación de la normalización al diseño de un modelo de base de datos?

- ✓ Evitar la redundancia de los datos.
- ✓ Evitar problemas de actualización de los datos en las tablas.

- ✓ Proteger la integridad de los datos.

¿Qué entiende por diccionario de datos, catálogo de datos o metadata?

- ✓ Para representar una base de datos relacional, necesitamos almacenar no sólo las relaciones mismas, sino también una cantidad de información adicional acerca de las relaciones.
- ✓ El conjunto de tal información se llama diccionario de datos o metainformación.
- ✓ El diccionario de datos debe contener información como: nombre de las relaciones en la base de datos; nombre de los atributos de cada relación; dominio de cada atributo; tamaño de cada relación; método de almacenamiento de cada relación; claves y otras restricciones de seguridad; nombres y definiciones de vistas.
- ✓ Además debe contener información sobre los usuarios de la base de datos y los poderes que éstos tienen: nombre de los usuarios; costo del uso efectuado por cada usuario; nivel de privilegio de cada usuario.