

Baze de Date

Cap. 9. Optimizare acces la date. Indexare.



Textbook: Ramakrishnan, Gehrke, "Database Management Systems", McGraw Hill, C15

2023 UPT

Conf.Dr. Dan Pescaru

Stocarea datelor

1. RAM

- + Procesare eficientă
- Capacitate redusă, cost mare, volatilă

2. HDD

- + Capacitate medie/mare. Preț mediu. Permanentă
- Timp acces mare ($> 10^3$ RAM)

3. DVD, optic, bandă magnetică

- + Capacitate medie/mare. Preț mic
- Acces lent la date, bandă - acces secvențial

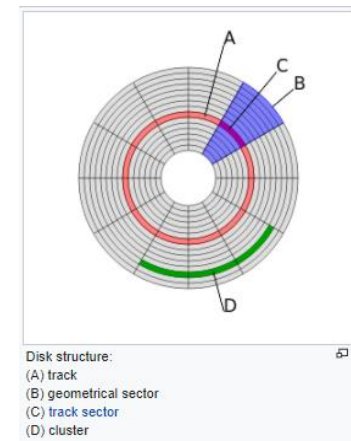
Stocarea pe HDD

1. Sistem de fișiere (S.O.)

- Organizare pe sectoare / blocuri
- Acces lent datorat părților mecanice (cap RW)
- Citirea paginilor consecutive este mai eficientă decât citirea lor în ordine aleatoare
- Poate fi gestionat de SO sau SGBD

2. Citirea/scrierea datelor

- Citire/scriere sectoare + utilizare buffere
- Management buffere: SO sau SGBD?



Fișier de date

1. O secvență de înregistrări

- Înregistrările sunt mapate pe sectoarele discului
- Înregistrări de lungime fixă sau variabilă (căutare eficientă vs. optimizare stocare)
- Înregistrări de lungime variabilă presupun organizarea într-o listă

2. Structura fișierului de date

- File header (ex. la dBase)
- Catalog al bazei de date (ex. Oracle, Microsoft SQL Server, IBM DB2, MySQL etc.)

Optimizarea accesului la date

1. Fișiere neordonate (heap files)

- Înregistrările sunt în ordine aleatoare
- Sunt potrivite dacă este necesar doar parcurgerea secvențială a tuturor înregistrărilor
- Căutările bazate pe condiții logice sunt ineficiente

2. Soluții pentru accelerarea căutării

- Sortarea înregistrărilor
- Indexare (arbori B+, tehnici hashing, clusterizare, virtualizare spațiu etc.)

Sortare fișiere

1. Eficiente pentru

- Extragerea înregistrărilor în ordinea respectivă
- Extragerea unui subinterval ordonat
- Căutare binară (complexitate $O(\log_2(N))$)

2. Probleme

- Adăugare/ștergere/modificare ineficiente
- Un singur criteriu de sortare este posibil la un moment dat
- Resortarea după alt criteriu este foarte ineficientă

Sortare fișiere - xBase

SORT TO fisier

ON exp1 [/A][/C][/D], exp2 [/A], ...

[FOR conditie]

[ASCEN/DESC]

- Creează un fișier nou (spațiu necesar dublu)
- Parametrii
 - /A – sortare ascendentă
 - /C – nu ține cont de litere mari/mici
 - /D – sortare descendentă
 - ASCE/DESC – ordine sortare globală (pentru toate expresiile de sortare)

Indexare

1. Alternativă mai bună la sortare

2. Eficientă pentru

- Extragerea înregistrărilor în ordinea indexului
- Extragerea unui subinterval ordonat
- Căutare pe arbore B+ (complexitate $O(\log_k(N))$) sau structură hash (acces în medie 1.2 citiri)
- Mai mulți indecși simultan (criterii de sortare)





3. Probleme

- Adăugare/ștergere/modificare ineficiente
- Spațiu suplimentar pe disc pentru indecși

Indexare – idea de bază

1. Index simplu

index

cheie	referință
K_1	
K_2	
K_3	
...	
K_N	

tablelă

cheie	câmp ₁	...	câmp _M
K_3	Val ₃₁		Val _{3M}
K_N	Val _{N1}		Val _{NM}
K_2	Val ₁₁		Val _{1M}
...			
K_1	Val ₂₁		Val _{2M}

Index - organizare

1. Un index pe un fișier accelerează selecțiile în câmpurile cheie de căutare pentru index

- Orice subset de câmpuri ale unei relații poate fi cheia de căutare pentru un index al relației
- Cheia de căutare nu este aceeași cu o cheie a unei relații (set minim de câmpuri care identifică în mod unic o înregistrare într-o relație)

2. Un index asigură regăsirea tuturor intrărilor de date k^* pentru cheia k

Tipuri de indecși

1. Primar vs. secundar vs. unic

- Dacă cheia de căutare conține cheia primară este index primar, dacă conține o cheie este unic

2. Clusterizat vs. neclusterizat

- Dacă ordinea înregistrărilor relației este la fel cu cea a intrărilor indexul este tip cluster
- Alternativa 1 implică clusterizare
- O tabelă poate fi clusterizată după o singură cheie
- Costul extragerii informației depinde dacă indexul este tip cluster sau nu (+1 citire spre tabelă)

Index – alternative pentru noduri

1. Sunt trei alternative

- Chiar înregistrarea cu cheia k
- $\langle k, rid \rangle$ al înregistrării cu cheia k
- $\langle k, \text{listă de } rid \rangle$ al înregistrărilor cu cheia k

2. Oricare alternativă este posibilă pentru oricare din tehnicile de indexare tip arbore B+, structură hashing etc.

- Pe lângă date și legături cu tabela indexată intrările vor conține și informații pentru direcționarea căutării

Index tip cluster

1. Prima alternativă reprezintă un index tip cluster

- Fișierul index reprezintă doar o formă de organizare a datelor din tabelă (tabelă = index)

2. Cel mult un index de tip cluster poate fi creat pentru o tabelă (altfel datele sunt duplicate integral)

- Trebuie ales cu grijă pentru că poate reduce timpul de căutare cu 20% - 66%

Alternative pentru informația din nod

1. Alternativele 2 și 3:

- Cheile de obicei mult mai mici decât înregistrările din tabelă. Deci în avantaj față de Alternativa 1 dacă înregistrările sunt mari și cheia este redusă
- Porțiunea structurii indexului utilizată pentru a direcționa căutarea, care depinde de dimensiunea nodului, este mult mai mică decât în cazul Alternativei 1
- Alternativa 3 mai compactă decât Alternativa 2, dar duce la noduri de dimensiuni variabile chiar dacă cheile de căutare sunt de lungime fixă

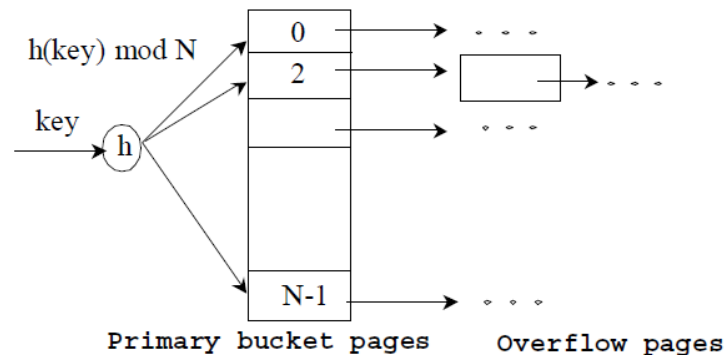
Indecși bazați pe hashing

1. Cheile sunt uniform distribuite în grupuri denumite buckets cu ajutorul unei funcții de hashing

- Bucket este o unitate de stocare ce poate conține 0..N înregistrări
- Funcție de hashing sau funcție de dispersie face o mapare a unor date la un interval numeric întreg
- Cheile care au aceeași valoare de hash sunt stocate în același bucket. Dacă nu mai încap se creează pagini adiționale legate într-o listă (overflow pages) pentru respectivul bucket

Hashing static

1. Numărul N de bucket-uri este fix, se creează pagini adiționale când este cazul
2. $h(k) \bmod N$ - va indica către înregistrarea care are cheia k (N este numărul de bucket-uri)



Hashing static - probleme

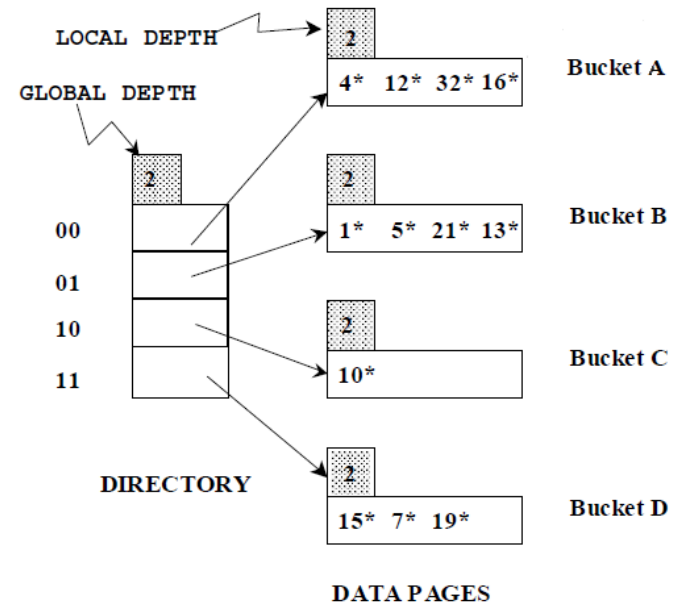
1. Bucket-ele conțin nodurile din index
2. Funcția hash trebuie să distribuie valorile în intervalul $0..N-1$: $h(k) = (a * k + b)$ este obținută prin optimizarea constantelor a, b
3. Prin adăugarea de date se pot dezvolta lanțuri lungi de pagini adiționale ceea va degrada semnificativ performanța căutării
4. Rezolvarea problemei: tehnici de hashing dinamic precum hash extensibil sau liniar

Hashing extensibil

1. Situație: bucket-ul (pagina principală) devine plină. De ce să nu reorganizăm indexul dublând numărul de bucket-uri?
 - Citirea/scrierea tuturor paginilor este costisitoare
 - Modificare: se utilizează un director de pointeri la bucket-uri. Se dublează virtual capacitatea indexului prin dublarea directorului, rupând doar bucket-urile pline, care necesită pagini adiționale
 - Directorul ocupă puțin loc comparativ cu bucket-urile, dublarea este eficientă. Evită paginile adiționale. Pentru acces se ajustează funcția hash

Hashing extensibil - implementare

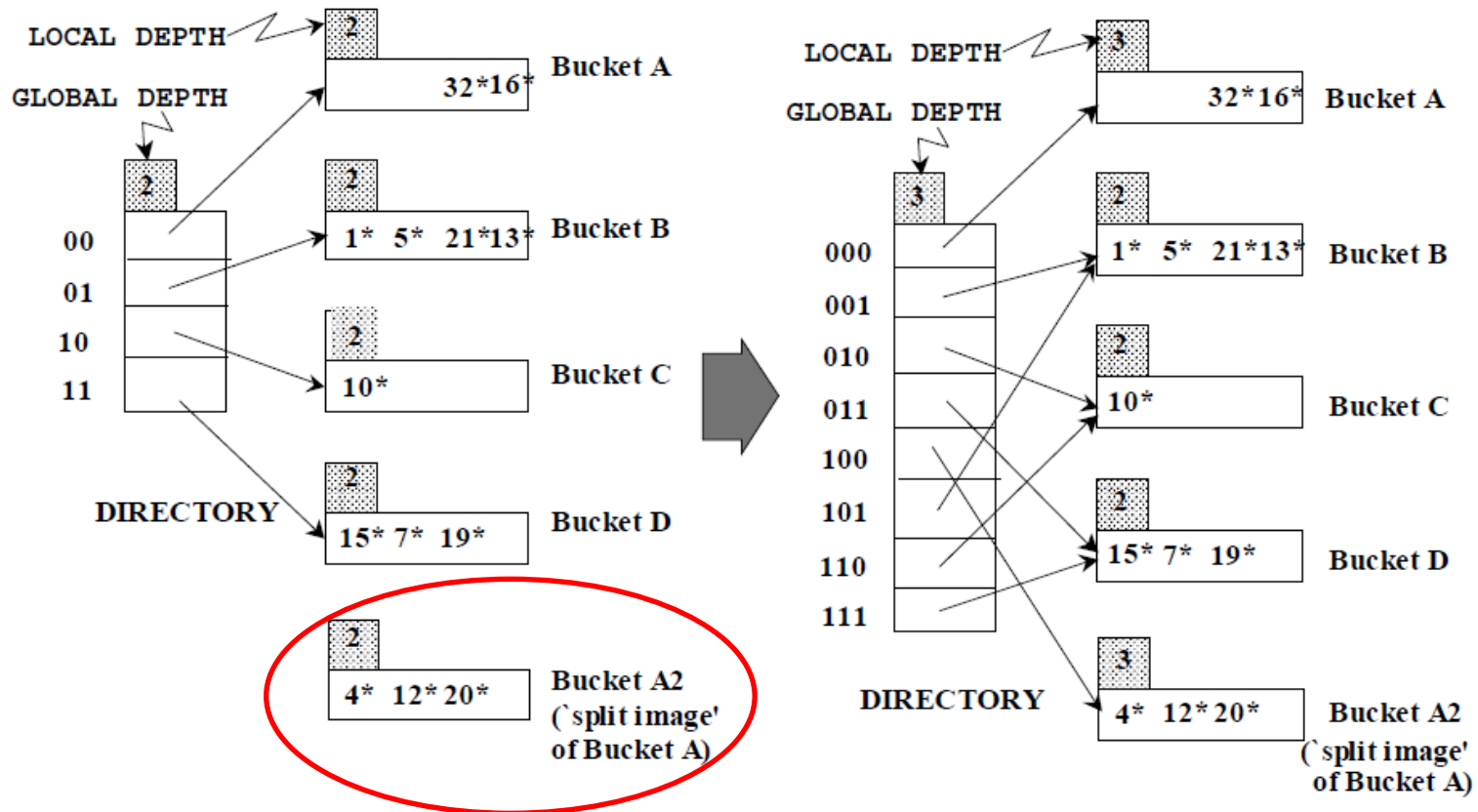
- Directorul este un tablou de 4 intrări
- Pentru a găsi bucket-ul r , se ia numărul de biți din $h(r)$ indicați de adâncimea globală (global depth)
- Dacă $h(r) = 5$ (binar 101), atunci este bucket-ul indicat de intrarea 01



- Inserare: dacă bucket-ul este plin se divide (se alocă o nouă pagină și se redistribuie cheile)
- Dacă este necesar se va dubla directorul (doar dacă adâncimea locală este egală cu cea globală)

Hashing extensibil - exemplu

- Inserarea $h(r)=20$ (cauzează dublarea)



Hashing extensibil - inserarea

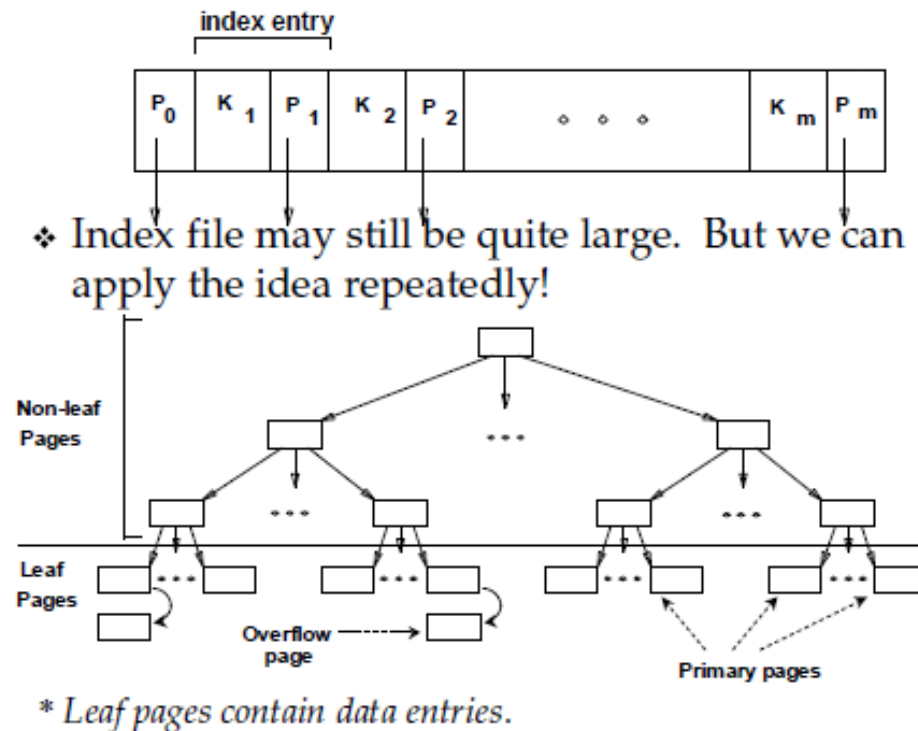
- 1. Global depth** (al directorului): numărul maxim de biți necesar pentru a identifica bucket-ul în care ajunge cheia
 - 2. Local depth** (al bucket-ului): numărul de biți utilizat pentru a determina dacă cheia aparține de bucket
- Când diviziunea bucket-ului determină dublarea?
 - Înainte de inserare, local depth = global depth. Inserarea cauzează local depth să devină $>$ global depth; directorul se dublează prin copierea pointerilor existenți și ajustarea celor spre bucket-ul divizat.

Hashing extensibil - concluzii

- Dacă directorul încapă în memorie, egalitatea testată print-un singur acces la disc; dacă nu prin două. Media este ~ 1.2
 - Fișier de 100MB, 100 octeți/înregistrare, 4K pagini conținând 1M înregistrări și 25K intrări în director; în majoritatea cazurilor va încăpea în memorie
 - Directorul crește în salturi, și există șansa să crească neuniform (același bucket se tot divide)
 - Cheile cu valori repetitive cauzează probleme!
- **Ștergere:** Dacă ștergerea unei intrări lasă un bucket gol acesta poate fi reunit cu perechea sa. Dacă fiecare intrarea arată spre același bucket ca și perechea, se poate înjumătăți directorul

Indecşi ISAM

- Indexed Sequential Access Method – ISAM (ex. Berkeley DB, Paradox, MySQL, Ms Access, dBase)

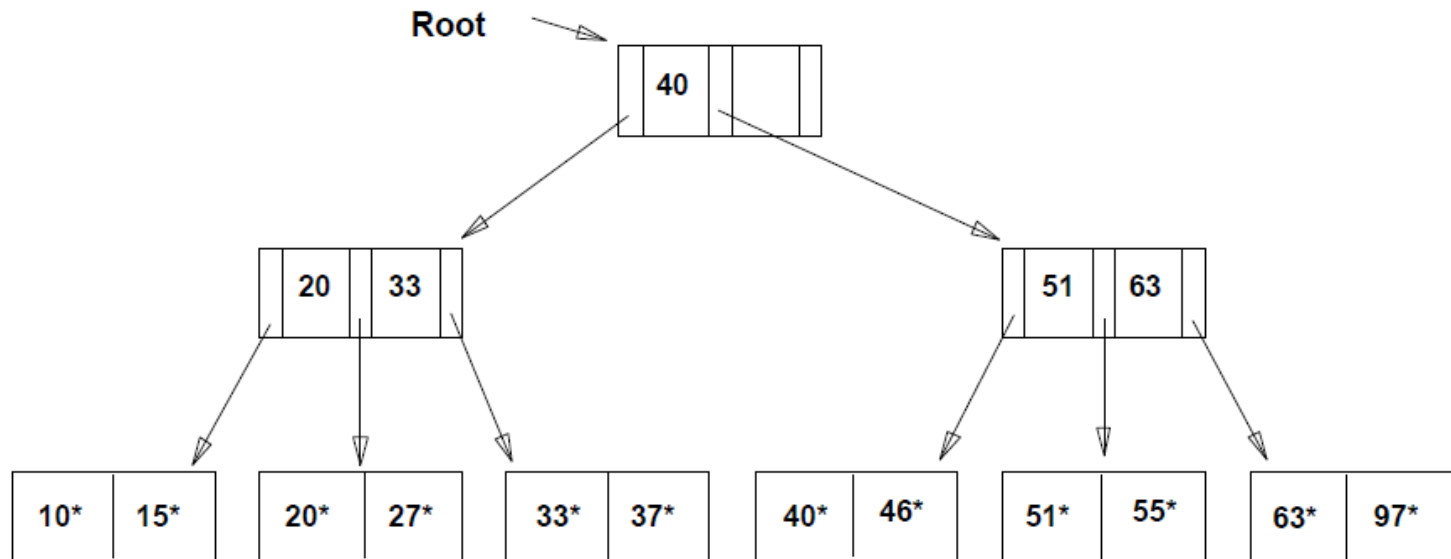


ISAM

- 1. Creare fișier:** frunzele (pagini cu înregistrări) sunt alocate secvențial și sortate după cheia de indexare; apoi nodurile index și spațiu pentru pagini adiționale
 - 2. Noduri index:** $\langle \text{valoare cheie}, \text{id pagină} \rangle$; direcționează căutarea către frunzele cu înregistrări
 - 3. Căutarea:** pornește de la rădăcină; merge pe comparație. Cost: $\log_F(N)$; $F = \text{nr intrări}$, $N = \text{nr frunze}$
 - 4. Inserare:** caută frunza și adaugă înregistrarea
 - 5. Ștergere:** caută frunza și șterge înregistrarea; dacă se golește o pagină adițională se va dealoca
- * *Structura de arbore este statică:* inserarea/ștergerea afectează doar nodurile frunză

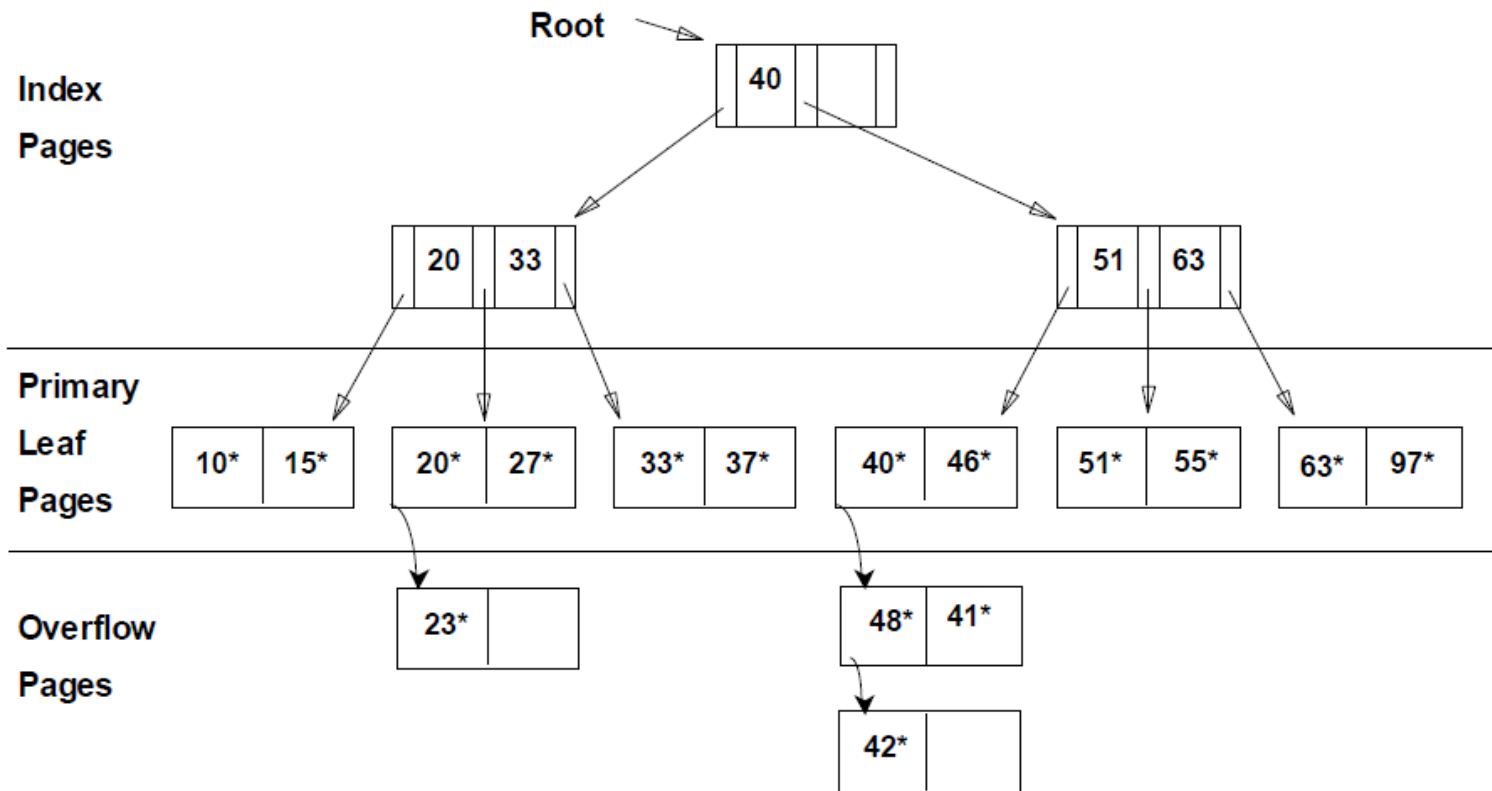
ISAM - exemplu

- Ex: fiecare nod are 2 intrări



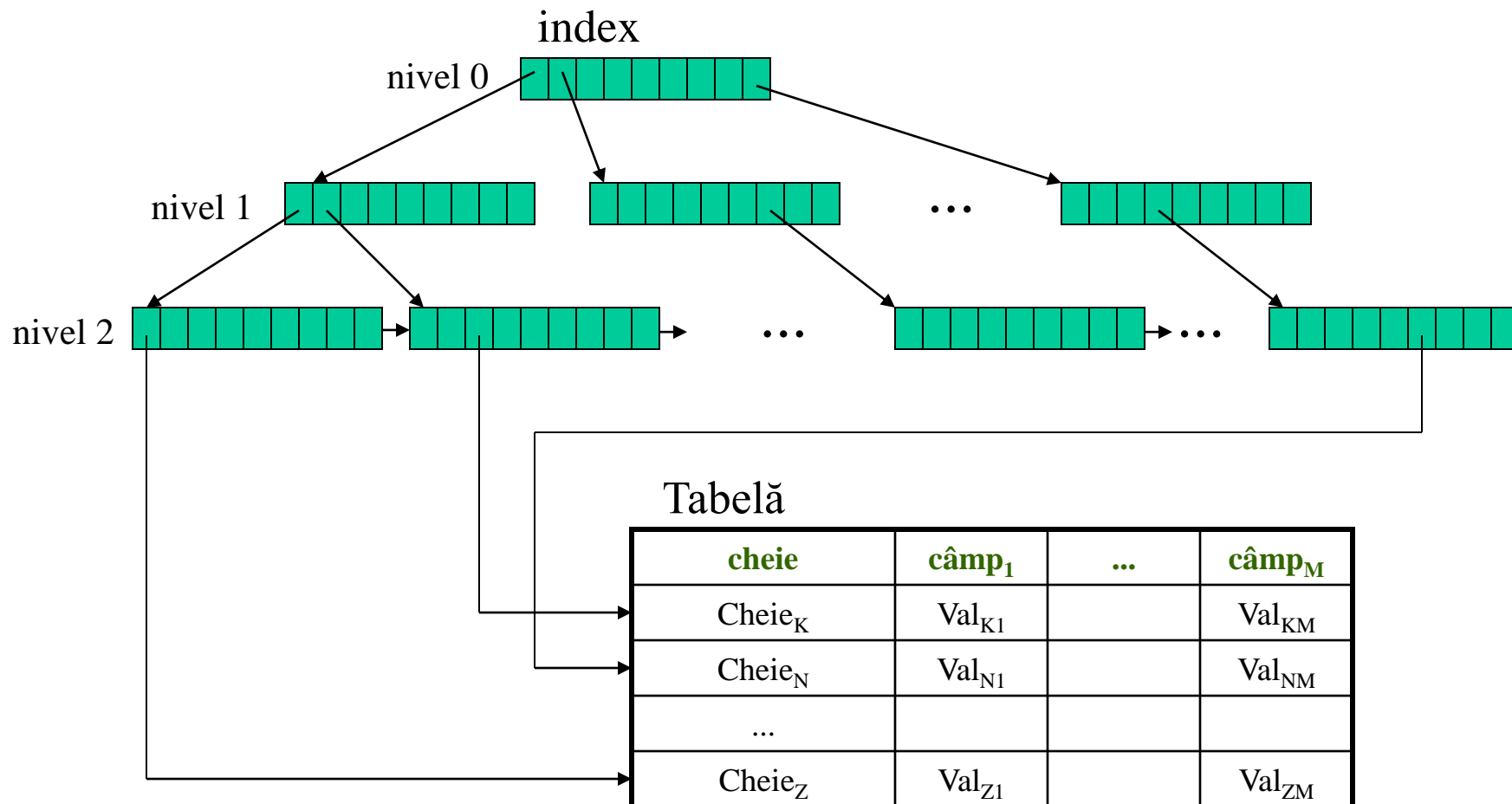
ISAM - exemplu

- După inserarea cheilor 23*, 48*, 41*, 42*



Indecși tip arbore B+

- Cei mai des utilizați în practică



Indecși B+

- Cost inserare/ștergere $\log_F(N)$; păstrează înălțimea balansată între subarbori
 - (F = încărcare noduri - fanout, N = nr frunze)
- Minim 50% ocupare nod (cu excepția rădăcinii). Fiecare nod conține $d \leq m \leq 2 \times d$ intrări, unde parametrul d este numit ordinul arborelui
- Asigură eficiență la căutări de egalități/inegalități
- Încărcare noduri ridicată (F : nr de copii/nod) ce asigură adâncime în jur de 3 sau 4
- Față de hashing permite căutarea de intervale (bazate pe comparații de inegalitate $<$, $>$)

Arbori B+ - inserare date

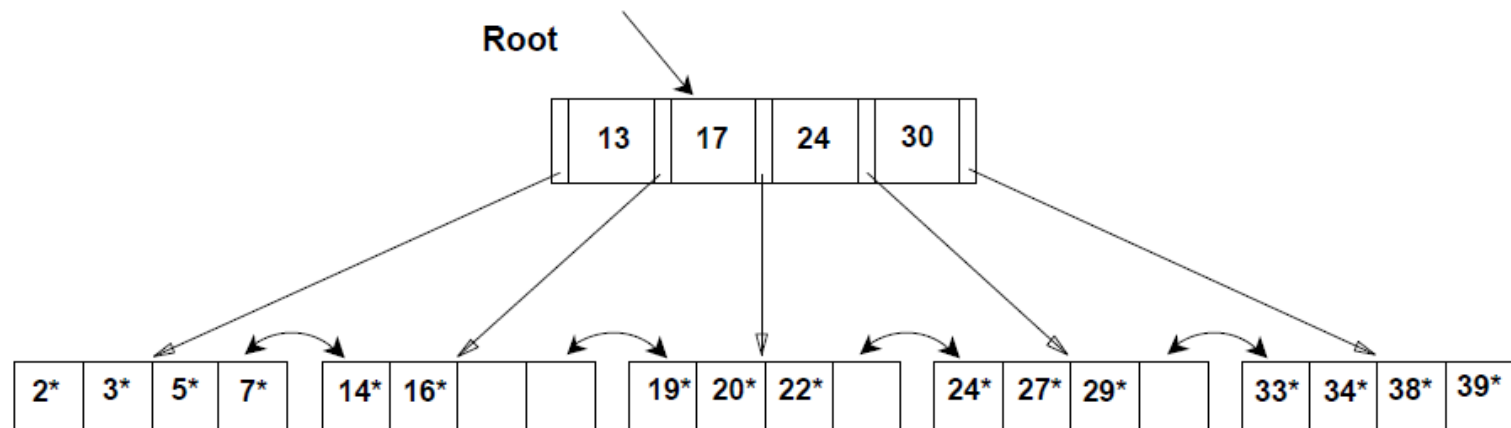
1. Caută frunza destinație L
2. Scrie cheia în L
 - Dacă L are spațiu suficient, gata!
 - Dacă nu, trebuie divizat L (în L și un nou nod L')
 - Redistribue intrările
 - Inserează o intrare în părintele lui L care arată spre L'
3. Acest lucru se poate întâmpla recursiv (în sus)
 - Pentru a diviza un nod index se redistribue cheile și se împinge în sus cheia mediană
4. Divizările *măresc* arborele – divizarea rădăcinii crește adâncimea arborelui
 - Creștere arbore: pe lățime sau un nivel pe adâncime

Arbori B+ - ștergere date

1. Pornește din rădăcină, caută frunza L cu acea cheie
2. Șterge intrarea din L
 - Dacă L este mai mult de jumătate plină, gata!
 - dacă L are doar **d-1** intrări,
 - Încearcă redistribuirea, împrumută de la frați (noduri cu același părinte ca și L)
 - Dacă nu se poate, reunește L cu un frate al său
3. Dacă se reunesc noduri, trebuie ștearsă referința din părintele L
4. Când se ajunge la rădăcină, scade adâncimea

Arbori B+ - exemplu

- Căutarea pornește din rădăcină, și se merge spre frunze (ca la ISAM)
 - Ex. de căutări: 5*, 15*, chei $\geq 24^*$...



Arbori B+ în practică

1. Ordin tipic: 100. Ocupare medie: 67%.
Fanout mediu = 133
2. Capacități tipice:
 - Înălțime 4: $133^4 = 312,900,700$ înregistrări
 - Înălțime 3: $133^3 = 2,352,637$ înregistrări
3. Se pot păstra nivelele superioare în memorie:
 - Nivel 1 = 1 pagină = 8 KB
 - Nivel 2 = 133 pagini = 1 MB
 - Nivel 3 = 17,689 pagini = 133 MB

Posibilități indexare

- Dacă considerăm o tabelă Angajați putem avea:
 1. Fișier simplu (ordine aleatoare, inserare la sfârșit)
 2. Fișier sortat, ex. după $\langle \text{vârsta}, \text{salar} \rangle$
 3. Index B+ tip cluster, cheie $\langle \text{vârsta}, \text{salar} \rangle$
 4. Fișier simplu și un index B + cu cheia $\langle \text{vârsta}, \text{salar} \rangle$
 5. Fișier simplu și un index hash cu cheia $\langle \text{vârsta}, \text{salar} \rangle$

Proiectarea indecșilor

1. Care indecși sunt potriviți?

- Care tabele trebuie indexate?
- Care sunt cheile necesare?
- Câți indecși sunt utili?

2. Pentru fiecare index ce tip trebuie folosit?

- Simplu sau cluster?
- Hash/B+?

Sugestii de proiectare

1. O abordare bună: se pornește de la cele mai folosite interogări. Se consideră cel mai bun plan cu indecșii existenți apoi se caută un dacă e posibil un plan mai bun cu indecși adiționali. Dacă da, se vor crea și aceștia
2. Înainte de crearea unui index se va studia impactul asupra inserării/ștergerii datelor!
 - Compromis: indecși accelerează căutările dar încetinesc modificările.
 - Se va considera cerințele de spațiu suplimentar

criterii de selecție (I)

- Atributele din WHERE sunt primele candidate pentru cheile de indexare
 - Condițiile de egalitate sugerează indecși de tip hash
 - Extragerea de intervale sugerează B+
 - Clusterizarea este bună în special pentru extragerea de intervale; poate însă ajuta la testarea de egalitate dacă sunt multe duplicate ale cheii de căutare

Criterii de selecție (II)

- Cheile multi-atribut se vor considera dacă WHERE conține condiții multiple (AND, OR)
 - Ordinea atributelor în cheie este importantă la extragerea de intervale
 - Astfel de indecși pot favoriza strategii de execuție tip index-only pentru interogările cele mai frecvente
- Se vor alege indecșii care acoperă cele mai multe interogări. Deoarece un singur index poate fi clusterizat per tabelă, alegerea lui va ține cont de cea mai interogare care implică acea tabelă

Gestionare indecși în xBase

1. Creare

- **INDEX ON** cheie **TO** index_file[.ndx]
[**UNIQUE**][**DESCENDING**]

2. Deschidere

- **USE** file **INDEX** ndx_file_list
- **SET INDEX TO** ndx_file

3. Schimbare index activ

- **SET ORDER TO** ndx_file

4. Căutare folosind index

- **SEEK** expr, **FIND** valoare

Creare indecși în Oracle

- Indecșii sunt utilizați în mod automat de optimizor și interpretorul de SQL
- Pentru cheia primară se creează implicit
- Creare explicită a unor indecși:

```
CREATE INDEX [UNIQUE] nume_index  
    ON tabelă(câmp1 ASC, câmp2 DESC ...)  
    [COMPUTE STATISTICS]; // utilizate de optimizor  
DROP INDEX nume_index;
```

- Pentru a crea indecși clusterizați se va folosi **CREATE TABLE** cu opțiunea **ORGANIZATION INDEX INCLUDING *cheie*** (la final, după paranteză închisă)