

## Adatbázisok elmélete 19. előadás

Katona Gyula Y.  
Budapesti Műszaki és Gazdaságtudományi Egyetem  
Számítástudományi Tsz.  
I. B. 137/b  
kiskat@cs.bme.hu  
<http://www.cs.bme.hu/~kiskat>

2005

### Növelhető hash

#### Jellemzők:

- a  $d$  változó mindig a struktúra aktuális globális mélységét tárolja, ennyi bitig számít  $h(K)$  értéke
- $2^d$  bejegyzés lesz a vödörkatalógusban, mert  $2^d$  darab  $d$  hosszú bitsorozat van
- a vödörkatalógus most is egy mutatókból álló (most éppen  $2^d$  elemű) tömb
- a vödörök száma  $2^d$ -nél lehet kevesebb is, ha több mutató is mutat ugyanarra a vödörré

**A műveletek közös vonása:** kiszámoljuk  $h(K)$ -t  $d$  bitig és aztán a vödörkatalógus  $h(K)$ -hoz tartozó mutatója alapján elmegyünk abba a vödörbe (blokkhoz), ahol az ezen  $d$  bittel kezdődő  $h(K)$  értékű rekordok vannak.

Ha két mutató ugyanoda mutat, akkor különböző  $d$  hosszú bitsorozattal kezdődő  $h(K)$  értékű rekordok ugyanoda kerülnek.

**Minden vödörnek van egy lokális mélysége, ezt a  $d'$  változó tárolja.** Ez azt mutatja, hogy az ebbe a vödörbe kerülésnél az első hány darab bit számít. Természetesen  $d' \leq d$  áll minden vödörré,  $d' < d$  akkor van, ha több mutató ugyanoda mutat.

### Növelhető hash

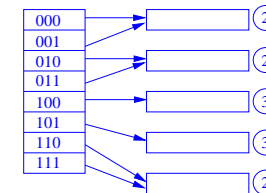
Kiküszöböli a vödörös hash azon hibáját, hogy nem tud alkalmazkodni a gyorsan növő állományhoz.

#### Fő elvek:

- kiszámoljuk  $h(K)$ -t, de az eredménynek csak az első (vagy utolsó) pár jegye számít abban, hogy melyik rekord hova kerül
- dinamikusan változik, hogy hány bit számít és ezzel együtt az is, hogy hány vödör lesz
- a vödörök mérete fix, tipikusan egy blokkból állnak. Így majd a műveletek 1 lapeléréssel menni fognak, egy kis belső memóriában való keresgélés után.

### A struktúra felépítése

A struktúra vázlatát  $d = 3$  esetén:



Mivel  $d = 3$ , ezért az első három bitig számoljuk ki  $h(K)$ -t. Most összesen öt vödör van a lehetséges maximális nyolc helyett, mert azok a rekordok, amiknek a  $h(K)$ -ja 00-val kezdődik elférnek egy vödörben, itt már az első két bit alapján tudjuk, hogy melyik vödörbe kerül a rekord. Hasonló a helyzet a 01 és 11 kezdetű  $h(K)$  értékekkel is. Egyedül az 10 első két bit esetén számít, hogy mi a harmadik (az 10 kezdetűek nem férnek el egy vödörben).

## Műveletek megvalósítása

- **keresés:** az adott  $K$  keresési kulcsra kiszámoljuk  $h(K)$  első  $d$  bitjét, a vödörkatalógus megfelelő mutatója alapján tudjuk, hogy honnan kell beolvasni az egy blokkból álló vödört, amiben a rekordnak lennie kell
- **beszúrás:** beszúrni kívánt rekord  $K$  értékére kiszámoljuk  $h(K)$ -t, vödörkatalógus mutatóját követve behozzuk a blokkot, ahova kerülnie kell. Ha **belefér** ez a rekord is, akkor belerakjuk és kiírjuk a blokkot.

Ha **nem fér bele**, akkor szét kell szedni a blokkot két részre, egy vödör helyett lesz kettő.

Két eset van:

- ★ Ha  $d' < d$ , a vödör lokális mélysége kisebb  $d$ -nél:  $d' := d' + 1 \leq d$  lesz, a vödörkatalógus nem változik, csak az eddig ugyanarra a vödörré mutató két mutató most majd két külön blokkra mutat:

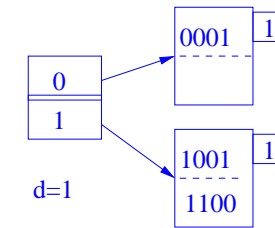


Természetesen az eddig egy vödörben levő rekordokat szét kell válogatni aszerint, hogy a  $d' + 1$ -edik bitje mi a  $h(K)$ -nak

## Példa

Növelhető hash segítségével akarjuk tárolni az adatainkat. Feltesszük, hogy egy lapra két rekord fér. A hash függvény 4 bites számot ad vissza, de jelenleg még csak egy bitet használunk ( $d = 1$ ), mivel eddig csak három elem (0001, 1001, 1100) van a táblázatban (az egyszerűség kedvéért az elemek helyett azt az értéket írjuk be, amit a hash függvény visszaad).

Tehát most így néz ki a tábla, van egy bejegyzés a 0-hoz és egy az 1-hez:



- ★ Ha  $d' = d$ , azaz már nem lehet növelni a lokális mélységet  $d$  változtatása nélkül:  $d := d + 1$ , a vödörkatalógust megduplázom egy  $x_1 \dots x_d$  bejegyzés helyett lesz kettő:  $x_1 \dots x_d 0$  és  $x_1 \dots x_d 1$ , az ezekből kiinduló két mutató ugyanoda megy, ahova a korábbi egy, a lokális mélység nem változik.

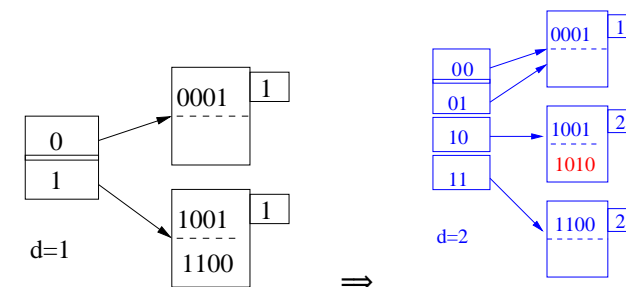


Ezek után már végrehajtható a vödörszétvágás úgy, mint az előbb, mert most már  $d' < d$  mindenhol az új  $d$ -vel.

- **törlés:** keresés, aztán pedig törlés a vödörből; ha ettől olyan helyzet áll elő, hogy a vödör összevonható a párjával (ahol az első  $d' - 1$  bit egyezik, de a  $d'$ -edik más), akkor **összevonás** (két mutató ugyanoda fog mutatni) és  $d'$  csökkentése eggyel. Ha minden  $d' < d$ , akkor  $d$ -t is csökkentjük eggyel.

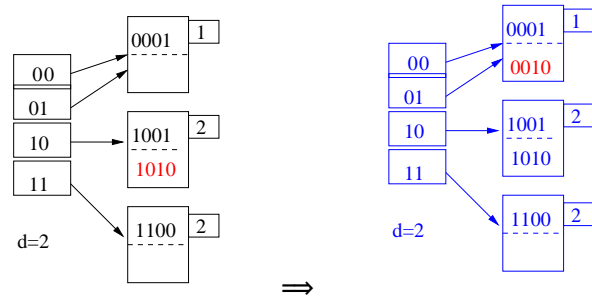
A fentiek miatt csak akkor mutathat két mutató ugyanarra a  $d'$  lokális mélységű vödörré, ha a két megfelelő érték első  $d'$  bitje megegyezik.

Ha most beszúrni szeretnénk egy olyan elemet, melyre a hash függvény az 1010 értéket adja, akkor az új tábla ilyen lesz:



Mivel az 1-hez tartozó lap már betelt, ezért itt a lokális mélységet növelni kellett 1-ről 2-re (két bitet akarunk figyelembe venni), de így a  $d$  értékét is növelnünk kellett.

Ha most jön egy 0010 hash-értékű elem, akkor azt simán be tudjuk rakni a helyére:



⇒

## A dinamikus hash értékelése

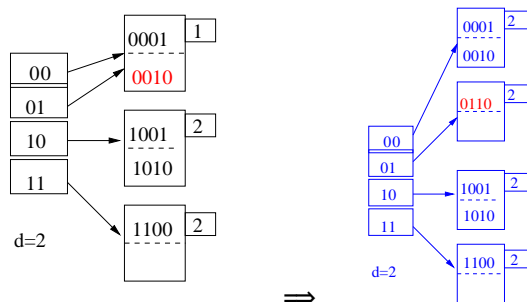
### Előnyök:

- egy I/O művelettel minden megvan, ha a vödörkatalógus a belső memóriában van
- a struktúra igazodik az állomány növekedéséhez

### Bajok:

- szerencsétlen esetben nagyon nagy vödörkatalógust kell építeni kevés rekord miatt (legrosszabb esetben lényegében a szekvenciális keresést kapjuk vissza)
- a  $h(K)$  értéket se lehet a végtelenségig továbbszámolni: ha  $h(K)$  már teljes hosszáig kiszámolandó, akkor nem lehet tovább növelni a struktúrát

De ha ezután olyan jön, ahol a hash függvény értéke 0110, akkor új lapot kell létrehozunk, de a  $d$  értékét nem kell növelnünk, mert elég lesz az eddig ugyanoda mutató két mutatót szétszedni és ezen új lapoknál a lokális mélységet 2-re állítani.



⇒

Ha ezután jönne egy olyan elem, aminek az értéke 00-val vagy 10-val kezdődik, akkor újabb lapra lenne szükségünk, de ehhez már a  $d$  értékét is növelni kellene.

## Indexek

Másik technika adatok jó tárolására. Továbbra is az a cél, hogy a műveletek gyorsan menjenek.

### Motiváló ötlet:

- könyvtárban úgy keresünk, hogy katalóguscetlik között keresünk és az alapján már megvan a könyv (sűrű indexre motiváció)
- ha a polcokon ábécé szerint vannak sorban a könyvek, akkor a polcok elején álló könyvek megvizsgálásával gyorsan eldönthető, hogy melyiken kell keresni (ritka indexre motiváció)

### Indexek általános jellemzői:

- rendezettséget figyelembe veszi (ellentétben a hash-sel)
- van korlát a legrosszabb esetre is (ellentétben a hash-el, ami viszont átlagosan jobb)
- jól bírja a dinamikus bővülést (ellentétben a hash-el)
- kulcs rendezett halmazból kerül ki (de nem féltétlenül egyedi)
- lehet ugyanarra az állományra több index is készítve

## Az indexstruktúra vázlatos felépítése



Az indexállományban vannak a keresést segítő infók, a főállományban vannak a tárolt adatok, a rekordok.

A főállomány lehet esetleg rendezett a keresési kulcs szerint, de nem feltétlenül az.

Az indexállományban indexrekordok vannak, ezek felépítése:

**kulcs** **mutató**

A kulcs valamelyik főállománybeli rekordhoz tartozó kulcsérték, a mutató pedig a főállományba mutat, oda, ahol ez a rekord van.

Aszerint, hogy minden főállománybeli rekordra van rá mutató indexbejegyzés vagy pedig csak néhányra, sűrű illetve ritka indexről beszélhetünk. (Majd látjuk, hogy ez két nagyon más helyzet lesz.)

## Ritka index, egyszintű

Feltesszük, hogy a főállomány szabad, azaz elmozgathatók a rekordok szabadon.

Jellemzők:

- **motiváló példa:** telefonkönyv vagy szótár sarkaiban a bejegyzések, ez alapján keresés
- **ritka index,** azaz nem minden főállománybeli rekordra van indexbejegyzés, csak blokkonként egyre
- a főállomány vagy rendezett a keresési kulcs szerint vagy lényegében rendezett (blokkokon belül esetleg nem rendezettek a rekordok, de a blokkok egymáshoz képest rendezetten vannak)
- az indexállomány rendezett a keresési kulcs szerint (az indexállomány is háttértéren van)
- az indexbejegyzésben szereplő kulcs a rendezett esetben a blokk legelső kulcsa, a lényegében rendezett esetben a blokk legkisebb kulcsa
- a mutató az indexbejegyzésben arra a blokkra mutat, ahol a bejegyzésben szereplő kulcshoz tartozó rekord van

A fentiek miatt a ritka index kivonata lesz a főállománynak, tartalmazza rendezetten a blokkok legkisebb kulcsait.

## Keresés

**Adott a keresési kulcs értéke, meg kell találni az ilyen rekordokat.**

1. Először az indexállományban megkeressük azt a legnagyobb indexbejegyzést, aminek kulcsa még éppen nem nagyobb, mint az adott keresési érték. Ez átlagosan  $\frac{n}{2}$  I/O művelet, ha  $n$  blokkból áll az indexállomány.
2. Az előbb megtalált indexbejegyzéshez tartozó blokkban (plusz egy I/O művelet) megkeressük a rekordunkat.

Megjegyzések:

1. Ha van valami plusz struktúra, ami segíti a gyors keresést az indexállományban, akkor jobbat is lehet, mint  $\frac{n}{2}$ .
2. A blokkon belül, a főállományban már bárhogy kereshetünk, de leginkább szekvenciálisan megy.

## További műveletek

- **beszúrás:** először egy keresés (hol kellene lennie), aztán blokkon belül a helyére tesszük:
  - ★ **ha rendezett a főállomány:** blokkon belül megkeressük a helyét, beillesztjük; **ha nem fér be**  $\Rightarrow$  blokkvágás két egyenlő részre, az új blokk minimális (legelső) kulcsára indexbejegyzés beszúrása az indexállományba
  - ★ **ha lényegében rendezett a főállomány:** ha van még hely a blokkban, akkor berakjuk, mindegy, hogy hova; **ha nem fér be**  $\Rightarrow$  blokkvágás két részre, mindkét részben a minimális elemhez új indexbejegyzés és a régi törlése az indexállományból
- **törlés:** hasonlóan, mint a beszúrás, először keresés, aztán törlés a blokkból; **ha épp a minimális rekordot töröltem a blokkból**  $\Rightarrow$  indexbejegyzés módosítása **ha nagyon ritkák a blokkok**  $\Rightarrow$  esetleg lapösszevonás (és persze az indexállomány módosítása is ilyenkor), de általában nem érdemes
- **tól-ig:** valahonnan valameddig terjedő értékű rekordok keresése: a "től" érték keresése, aztán a főállomány végigolvasása az "ig" értékig (a főállomány folyamatos elérése megoldott)
- **módosítás:** ha kulcsot nem érint: keresés, átírás; ha kulcsot is érint: törlés, beszúrás

## Megjegyzések:

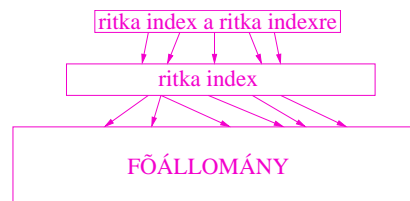
- Tényleg használtuk, hogy a főállomány szabad (pakoltuk a rekordokat össze-vissza). Lesz majd technika, amivel elérhető lesz, hogy a főállomány szabadnak látszódjon.
- Azt is erősen kihasználtuk, hogy (lényegében) rendezett a főállomány.
- Azért hasznos az indexállomány, mert sokkal kisebb, mint a főállomány, könnyebb benne keresni és a végen csak plusz egy I/O művelet kell a befejezéshez. De ennek ára van: karban kell tartani plusz egy struktúrát.

- eggyel több szint lesz, azaz eggyel több lapelérés kell a kezdeti keresés után, de a kezdeti keresés lerövidül
- nem csak kétszintű lehet a ritka index, hanem több is  $\implies$  dinamikusan is változhat  $\implies$  B-fa

## Többszintes ritka index

Az indexen belül keresni arányos az indexállomány blokkjainak számával. Ez jóval kisebb, mint a főállomány lapszáma, de még mindig nagyon nagy lehet.

Ezért: többszintű index, vagyis index az indexre:

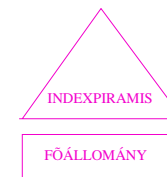


- a felső index még kisebb lesz, könnyebb lesz benne keresni
- a középső szint egyszerre indexe a főállománynak és "főállománya" a felső indexnek
- keresés: a legfelső szinten megkeressük a legnagyobb olyan bejegyzést, ami még kisebb a keresetnél és innen két lap beolvasásával (a megfelelő középső szintű indexlap és aztán a főállomány megfelelő lapja) megvan a keresett rekord
- a többi művelet hasonlóan megy (persze, ha módosul a főállomány, akkor esetleg mindegyik indexállományt is módosítani kell)

## B-fa

A ma ismert egyik legjobb és legelterjedtebb megoldás,  $m$  elágazásos B-fa vagy  $B_m$ -fa, lényegében ahogy algeból tanultuk:

- a fa levelei: a főállomány blokkjai
- a főállomány (a levelek) rendezett a keresési kulcs szerint
- minden levél ugyanolyan messze van a gyökértől
- a fa belső csúcsai: a különböző szintű indexek lapjai
- egy csúcs gyerekei: az indexlapon levő mutatóknak megfelelő eggyel lejjebb levő indexlapok (illetve alul levelek)
- $m$ : egy lapra  $m$  indexrekord fér rá (kicsit más lesz egy belső csúcs szerkezete, mint algeból volt)
- minden lap legalább félig kitöltött, kivéve esetleg a gyökeret (minden csúcsnak legalább  $\frac{m}{2}$  gyereke van, kivéve esetleg a gyökeret)



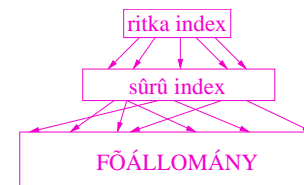
## Műveletek

- **keresés:** ahogy algeból volt, a csúcsokban található kulcs-bejegyzések és mutatók mentén; arányos a fa magasságával, ami  $O(\log_m n)$ , ha  $n$  blokkja van a főállománynak
- **beszúrás:** ahogy algeból volt, beszúrás után esetleg csúcsvágás(ok), de max.  $O(\log_m n)$
- **törlés:** ahogy algeból volt, törlés után esetleg csúcsösszevonás(ok), de max.  $O(\log_m n)$

Megjegyzések:

1. Ha  $m$  nagy  $\Rightarrow$  ritkán kell csúcsvágás/csúcsösszevonás.
2. általában  $m$  úgy van választva, hogy a fa magassága max. 4 legyen, ha az első lap a belső memóriában van, akkor elég 3 I/O művelet mindenhez

Tipikus használata:



A sűrű index ráépül a főállománnyra, erre építjük a valódi állományszervezést. A sűrű index miatt a főállomány szabadnak és rendezettnek tűnik.

## Sűrű index

Eddig feltettük, hogy a főállomány szabad és (lényegében) rendezett a keresési kulcs szerint. Hogyan érjük ezt el? Hogyan lehet több kulcs szerint is keresni?

Erre megoldás a sűrű index:

- a főállomány minden rekordjához van egy indexbejegyzés  $\Rightarrow$  ugyanannyi bejegyzés lesz a sűrű indexben, mint ahány rekord van a főállományban, csak persze kisebbek a bejegyzések  $\Rightarrow$  sűrű index = főállomány kicsiben
- ez nem önálló állományszervezési technika (ellentétben a ritka index változataival), hanem csak kiegészítés, ami lehetővé teszi, hogy a főállományt szabadnak és rendezettnek tételezhessük fel

Haszna:

- **szabaddá teszi a rekordokat** (a főállomány ugyan kötött, de a sűrű index bejegyzései szabadon mozgathatók: építhető rá ritka index)
- **rendezettnek mutatja a főállományt: a sűrű indexet úgy rendezzük, ahogy akarjuk**
- sokkal kisebb, mint a főállomány, mégis egy az egyben megfelel neki

## Műveletek ebben a struktúrában

Úgy dolgozunk, mintha a sűrű index lenne a főállomány, innen már csak egy plusz lapelérés a valódi főállomány.

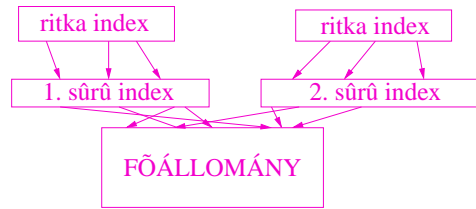
- **keresés:** keresés sűrűben, onnan egy lapelérés
- **beszúrás:** beszúrás sűrűbe, aztán valahova berakjuk a főállományba
- **törlés:** keresés, törlés a főállományból, törlés a sűrűből is

Hátrány

- plusz egy lapelérés kell a sűrű miatt
- karban kell tartani a sűrű indexet is mindig, amikor a főállomány változik

## Nagy előnye a sűrű indexnek

Lehetővé teszi, hogy egy főállományra több különböző kulcs szerint is legyen index:



Itt minden sűrű index rendezett a megfelelő kulcs szerint és persze ha változik a főállomány, akkor mindegyik sűrűt is változtatni kell.

Példa:

A Személy(név, telefonszám, személyszám, ...) sémában a személyszám az elsődleges kulcs, ezért a rendszer eszerint rendezetten tárolja az adatokat és erre biztosan létre is hoz valami keresési struktúrát.

De ha mi szeretnénk a név-re is: kell egy sűrű index: invertált állomány.

## Számolási példa

Egy állományt sűrű index, majd erre épített 1-szintes ritka index segítségével szeretnénk tárolni. Adjon értelmes alsó becslést a szükséges lapok számára az alábbi feltételek mellett:

- az állomány  $3 \cdot 10^6$  rekordból áll,
- egy rekord hossza 300 Byte,
- egy lap mérete 1000 Byte,
- a kulchossz 45 Byte,
- egy mutató hossza 5 Byte.

Megoldás:

A főállományban  $3 \cdot 10^6$  rekord van, mivel rekordok nem lóghatnak át laphatáron, ezért ehhez kell  $10^6$  lap.

A sűrű indexben annyi bejegyzés lesz, mint ahány rekord van a főállományban, azaz  $3 \cdot 10^6$ .

Egy lapra pontosan 20 bejegyzés fér: ez  $1,5 \cdot 10^5$  lap.

Ez azt is jelenti, hogy a ritka indexben lesz legalább  $1,5 \cdot 10^5$  bejegyzés, ehhez kell még  $7,5 \cdot 10^3$  lap.

Ez összesen 1 157 500 lap.

## Különböző technikák összevetése

- **hash**: konstans (gyakran 1) lapelérés átlagosan, de legrosszabb esetben lassú
- **ritka index**: korlátos viselkedés legrosszabb esetben is, dinamikus bővülés támogatása, rendezettség figyelembe vétele; B-fa esetén a gyakorlatban konstans lapelérés
- **sűrű index**: önmagában nem jó, csak kiegészítésként szolgál