Seminár o princípoch tvorby databáz

??????

18. marca 2008

Obsah

1 Disky a súbory **2**
1.1 Súborový manager (File manager) 2
1.2 Disk 3
1.2.1 RAID 4
1.3 Management diskového miesta 4
1.4 Buffer manager 5
1.5 Súbory a indexy 6

2 Organizácia súborov a indexy **8**
2.1 Výpočet ceny 8
2.2 Operácie 8
2.2.1 Heap 8
2.2.2 Zotriedený súbor 9
2.2.3 Hashovaný súbor 9
2.3 Indexy 9
2.3.1 Vlastnosti indexov 10

3 Stromové indexy **12**
3.1 ISAM (Index Sequencial Access Method) 12
3.2 B+ strom 13
3.2.1 Search 14
3.2.2 Insert 14
3.2.3 Delete 16
3.2.4 Duplicity 18
3.2.5 B+tree v reálnych systémoch 18
3.2.6 Key kompresia 18
3.2.7 Rád stromu 18
3.2.8 Bulk loading = dávkové pridávanie 19

4 Hashové Indexy **21**
4.1 Statické hashovanie 21
4.2 Extendible (Rozpínavé) hashovanie 22
4.3 Lineárne hashovanie 24

5 Výpočet dopytov **27**
5.1 Metadáta 27
5.2 Využitie indexov 27
5.3 Zádrhle relačných operátorov 28

6 Externé triedenie **28**
6.1 Jednoduchý dvoj-cestný merge sort 28
6.2 Externý merge-sort 29
6.3 Replacement Sort (pre 0-tý prechod) 31

7 Výpočet relačných operátorov **32**
7.1 Selekcia 33
7.2 Projekcia 34

8 Join **35**
8.1 Block Nested Loops Join 37
8.2 Index Nested Loops Join 37
8.3 Sort-merge join 38
8.4 Hash Join 39
8.5 Hybrid hash join 39
8.6 Porovnania 40
8.7 Hash join vs. Sort-merge join 40
8.8 Hash join vs. Block nested loops join 40
8.9 Iné podmienky pre join 40
8.10 Top-k join 41
8.11 Treshold algoritmus 41

9 3P-NRA **43**
9.1 fáza 1 43
9.2 fáza 2 43
9.3 fáza 3 43

10 Počítanie množinových operácií **44**
10.1 Počítanie agregačných operácií 44

11 Optimalizácia výpočtu dopytov **44**
11.1 Úvodné spracovanie selektu 44
11.2 Zistenie ceny plánu 45

12 Histogramy **47**
12.1 Motivačný príklad 47
12.2 Niečo o histogramoch 47
12.3 Histogram dištančný 49
12.4 Histogram ekvipotenčný 49
12.5 Histogram komprimačný 49

13 Stromy výpočtov **50**

14 Výpočet alternatívnych plánov nad jednou tabuľkou **52**
14.1 Dopyty nad viacerými tabuľkami 53
14.2 Algoritmus na left-deep plány (dynamické programovanie) . . 53
14.3 Vnorené selekty 54

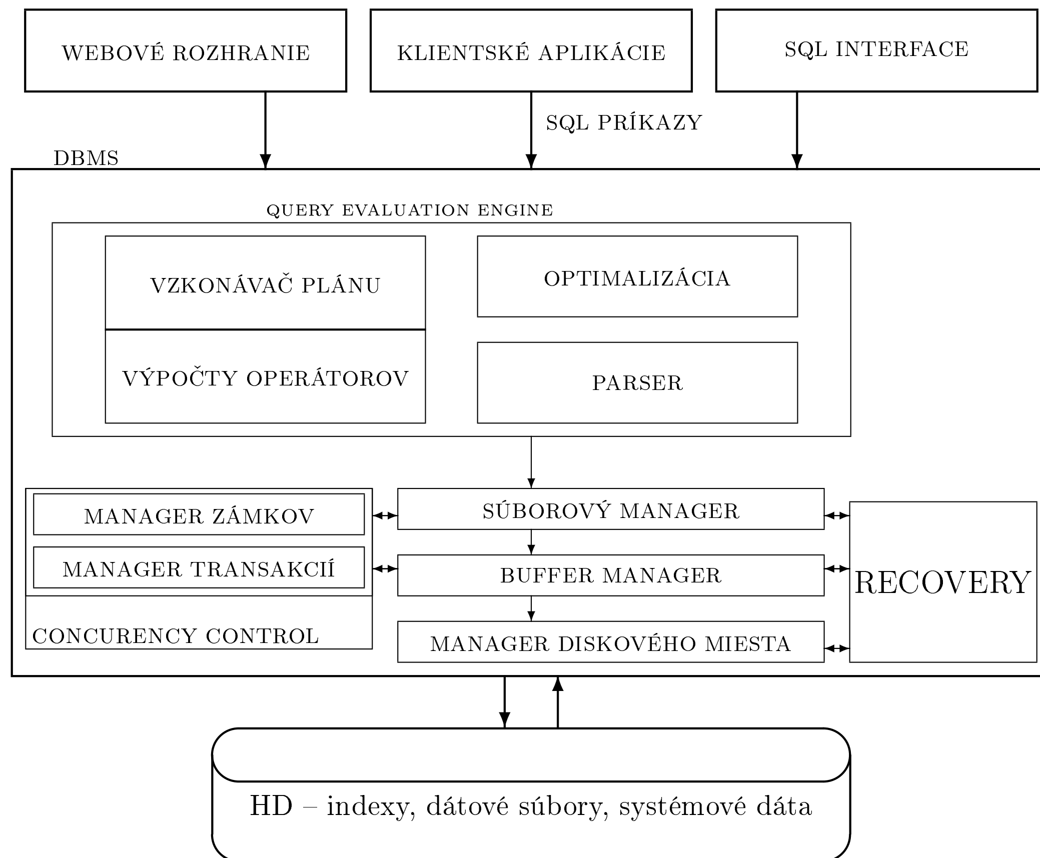
15 Distribuované databázové systémy **55**
15.1 Architektúry DBS 55
15.2 Typy distribuovaných databáz 56
15.3 Základné požiadavky pre distribuovaný DBM 57
15.4 Architektúry distribuovaných DBMS 57
15.5 Ukladanie dát v distribuovanom DBMS 58
15.5.1 Horizontálna 58
15.5.2 Vertikálna 59
15.6 Na čo si dať pri tvorbe databáz pozor - pre DB správcov . . . 59
15.7 Využitie redundancie 60

16 Paralelné výpočty **61**
16.1 Triedenie 61
16.2 Paralelny hash JOIN 62
16.3 Vylepšený paralelný hash join 62

17 Distribuované výpočty **63**
17.1 Distribuované joiny 63

18 Rozloženie dát s optimálnym rozdelením **66**

1 Disky a súbory



Obrázok 1: Schéma komunikácie.

Dáta sú na diskoch.

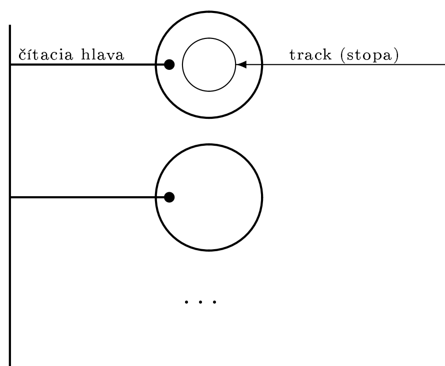
1.1 Súborový manager (File manager)

- Poskytuje abstrakciu súboru záznamov.
- Poskytuje a uvoľnuje miesto v stránkach (4kB alebo 8kB).

- Ak záznam potrebujeme spracovať disk–>operačná pamäť, identifikácia miesta na disku môže ísť cez ďalšie štruktúry.
  Po identifikácii stránky takto putuje do **buffer managera**, ten to uloží do buffer pool.
- Oslovuje buffer manager pre stránku.

1.2 Disk

Disk je zhruba 100 krát pomalší ako RAM (dnes ioDrive 1000 rýchlejšie ako disky).



Obrázok 2: Jednotlivé lamely pevného disku sú uložené pod sebou. Každej prislúcha jedna čítacia hlava - iba jedna hlava môže byť aktívna v jednom čase. Dáta sú potom rozdelené do trackov a tie sa ďalej delia na bloky (jednotka čítania/zapisovania) a bloky na sektory. Zjednotenie trackov s rovnakým polomerom sa nazýva cylinder.

Prístupový čas je suma nasledujúcich časov:

- Seek – čas potrebný na presun hlavy na danú stopu,
- rotačné oneskorenie – čakanie na správny blok na stope,
- transfer time – čas čítania/zapisovania bloku a poslania na zbernicu.

Sekvenčný prístup znižuje seek time a rotačné oneskorenie a teda je oveľa rýchlejší ako priamy prístup, lebo bloky sa pri zápise zapisovali blízko seba.

1.2.1 RAID

- zoskupenie niektorých diskov na zvýšenie výkonu a spoľahlivosti
- výkon sa zvyšuje rozdeľovaním dát (ilúzia rýchleho disku)
- spoľahlivosť sa zvyšuje redundanciou (redundant arrays of independent disks)
- 1 časť sa volá **stripping unit**. Ak máme D diskov, tak i-ta časť je uožená na disku $i \mod D$.

Ak je stripping unit = 1bit, tak ak chceme 1 blok, tak sa zapojí všetkých D diskov a načíta sa D blokov paralelne. Takže rýchlosť sa nezväčší ak je stripping unit = 1 blok a čítame veľa dát sekvenčne, tak sa to môže zrýchliť D krát.

Spoľahlivosť – ak priemerná chybovosť je 50000 hodín a máme 100 diskov, tak v priemere sa jeden pokazí každých 50000/100 = 500 hodín (zhruba 21 dní). Preto sa používajú napríklad paritné disky (10 paritných diskov zo 100 – spoľahlivosť 250 rokov)

1.3 Management diskového miesta

Pridávaním a odoberaním záznamov vznikajú diery. Riešenie: zoznam dier (linked list) alebo bitmapa obsadenia (lepšia na detekciu väčších dier - pre zápis).
Použitie OS – prečítaj bit i súboru f –>prečítaj blok f stopy t cylindra c disku d.

Nevýhody:

- pre 32-bitové systémy sú max. veľkosti súborov 46B,
- každý operačný systém je iný.

1.4 Buffer manager

- politika odstraňovania,
- zásobáreň buffera zložená z frame-ov (v každom frame-e je jedna stránka),
- vyššie vrstvy sa nestarajú či stránky sú na disku, ale musia informovať o tom či zmenili obsah stránky alebo ju prestali používať. Buffer manager sa potom postará, aby prislušná stránka bola zmenená na disku.

Každý frame si uchováva **pin-count** a **dirty**. Pin-count počíta počet požiadaviek na stránku v danom frame-e. Dirty je boolovská premenná, ktorá určuje či sa dáta v danom frame-e zmenili oproti tomu čo je na disku.

Pri požidavke sa vykonáva nasledovné:

1. Ak stranka v buffery je, tak ju vráti a zvýši pin-count daného frame-u o 1, ak tam nie je, tak:
   - vyberie frame podľa politiky odstraňovania a pin-count daného bloku zvýši o 1,
   - ak DIRTY=TRUE, tak zapíše stránku na disk,
   - nahradí stránku novou.
2. vráti adresu v RAM na frame.

Ak nadvrstva vezme daný frame, tak sa pin-count zníži o 1. Ak pin-count>0, tak neodstraňujeme. Ak daný frame chcú viac ako dva procesy, tak treba pamätať na zamykanie, aby nevznikli konkurenčné zápisy jednej stránky.

Politika odstraňovania LRU (last recently used)

- buď cez rad frame-ov s pin-count=0,
- clock odstraňovanie prechádza frame-y dookola $1 \ldots N$ a prvý =0 je odstránený,

- ak je zásobáreň buffera 10 a súbor, ktorý číta veľa ľudí, má 11 stránok, tak sa vždy všetky frame-y budú nahrádzať v cykle – najhoršie riešenie.

Ďalšie – FIFO, MRU (most recently used), random. Ešte sú rozšírenia, že je viac zásobární (. . . pre každú tabuľku).
Prefetching of pages – stiahne stránky predtým ako boli volané.

1.5 Súbory a indexy

Heap:

- nezotriedený súbor
- každý záznam má vlastné rid=<page_id, slot_number >(record id), kaťdá stránka má rovnakú veľkosť,
- operácie CREATE/DESTROY FILE, INSERT/DELETE RECORD s daným rid, GET(rid), SCAN,
- double linked list – jeden pre plné a jeden pre „prázdne"stránky (ak vložíme záznam do stránk, ešte stále ju označujeme za prázdnu – môže byť problém ak sú záznamy variabilnej dĺžky),
- adresár stránok – každý záznam v adresári identifikuje stránku, alebo sekvenciu stránok (adresár je oveľa menší ako veľkosť stránok dohromady, pre každú stránku si môžeme pamätať koľko má voľného miesta).

Formát súborov

- fixná dĺžka záznamov
    1. vždy zlepené záznamy (ľahko sa počíta offset) – problém že sa mení rid,
    2. bitová mapa slotov.
- rôzna dĺžka záznamov

- adresár slotov pre každú stránku v dvojiciach
  <offset záznamu, dĺžka>,
- vhodné aj pre fixné dĺžky, ak chceme uchovávať usporiadanie podľa nejakého atribútu (stačí triediť adresár namiesto záznamov v tom prípade netreba udržiavať dĺžku slotu),
- je dobré zlepšovať záznamy v stránke, aby voľný priestor bol pokope.

Formát záznamov

- fixná dĺžka – typy sa uchovávajú na jednom mieste v systémovom katalógu, umiestnenie sa ľahšie počíta
- rôzna dĺžka
  - a) oddeľovače (. . . vyžaduje skenovanie pokiaľ sa nájde),
    b) pole offsetov položiek aj s koncom záznamu, null sa reprezentuje rovnakým offsetom susedných položiek,
  - pozor na zmenu hodnôt, lebo môžu spôsobiť posun dát,
  - po zmene na väčší záznam sa tento pravdepodobne bude musieť presunúť do inej stránky (ak rid obsahuje číslo stránky, môžme nechať na stránke forwarding adresu),
  - záznam môže narásť nad veľkosť stránky, takže ho musíme deliť na menšie časti,
  - reálne DB:
    –>obmedzenie veľkosti záznamov na (2kB – 32kB) okrem BLOB, CLOB, ktoré sú typicky uložené mimo,
    –>Oracle umožňuje ľubovolnú veľkosť záznamov.

2 Organizácia súborov a indexy

2.1 Výpočet ceny

- B - počet stránok
- R - počet záznamov na stránku
- D - priemerný čas I/O jednej stránky $\sim 15ms$
- C - priemerný čas spracovania záznamu $\sim 100ms$
- H - čas výpočtu hash funkcie $\sim 100ms$

Okrem toho ešte treba vziať do úvahy čas výpočtu CPU, čas prenosu pri distribuovaných databázach a block access.

2.2 Operácie

- SCAN - vybratie všetkých záznamov v tabuľke
- ROVNOSŤ - hľadanie všetkých záznamov splňujucích podmienku rovnosti
- RANGE - získanie všetkých záznamov z intervalu
- INSERT - nájdenie, vybratie, zmena a zapísanie stránky
- DELETE - nájdenie, vybratie, zmena a zapísanie stránky (zmazanie záznamu s danným rid)

2.2.1 Heap

- SCAN $= B(D + R * C)$
- ROVNOSŤ $= \begin{cases} 1/2 * B(D + R * C), & \text{ak unique} \\ B(D + R * C), & \text{inak} \end{cases}$
- RANGE = ROVNOSŤ
- INSERT $= 2 * D + C$
- DELETE = ROVNOSŤ$+C + D$

2.2.2 Zotriedený súbor

- SCAN = $B(D + R * C)$
- ROVNOSŤ = $D * \log_2 B + C * \log_2 R +$ počet nájdených$*C$
- RANGE = ROVNOSŤ
- INSERT = ROVNOSŤ + POSUN = ROVNOSŤ$+B(D + R * C)$
- DELETE = INSERT

2.2.3 Hashovaný súbor

Stránky sú spájané do **bucket**-ov (oblasti). Primárnu stránku pre bucket nájdeme rýchlo.

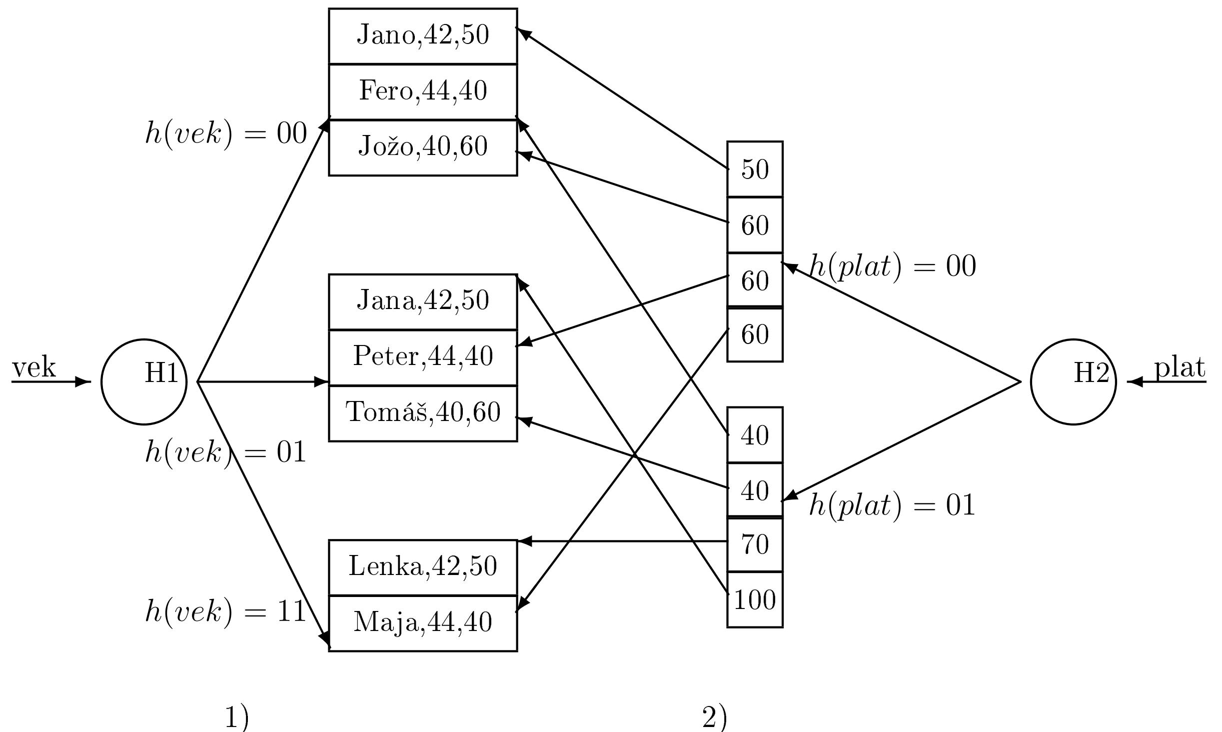
- SCAN = $1,25 * B(D + R * C)$ - lebo priemerné zaplnenie je 80%.
- ROVNOSŤ = $H+D+0,5*R*C$ - za predpokladu, že bucket je v prvej stránke a záznam nájdeme po prehľadaní polovice stránky. Rovnosť musí byť špecifikovaná pre všetky položky v search key.
- RANGE = SCAN
- INSERT = ROVNOSŤ $+C + D$ = DELETE

2.3 Indexy

- prídavná štruktúra zlepšujúca operácie, ktoré sú neefektívne v základnej štruktúre
- kolekcia dátových položiek umožňujúca efektívne nájsť dátové položky podľa kľúča k

typy dátových položiek

1. k_* - celý záznam(y) s kľúčom k
2. $< k, r_{id} >$ - vieme, aký záznam máme hľadať v základnej štruktúre
3. $< k, r_{id} - list >$ - tu vieme zoznam r_{id}-ov s kľúčom k



Typy 2 a 3 sú nezávislé na organizácii dát. Vždy je možné mať iba jeden index typu 1.

2.3.1 Vlastnosti indexov

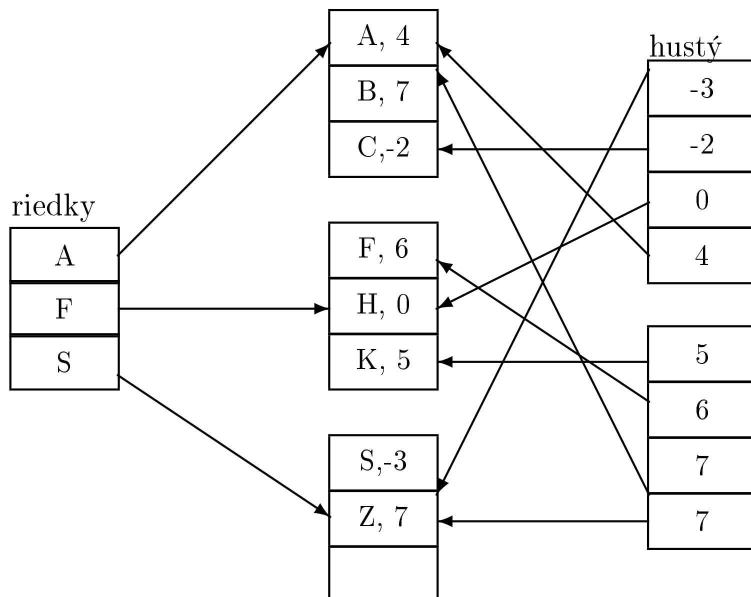
klastrovaný index - ak záznamy majú rovnaké, alebo takmer rovnaké poradie ako dátové položky v indexe. Ak nejaké záznami musíme presúvať medzi stránkami, tak sa mení ich r_{id} =<id stránky, slot> a musíme updatovať všetky indexy, čo je veľmi drahé!

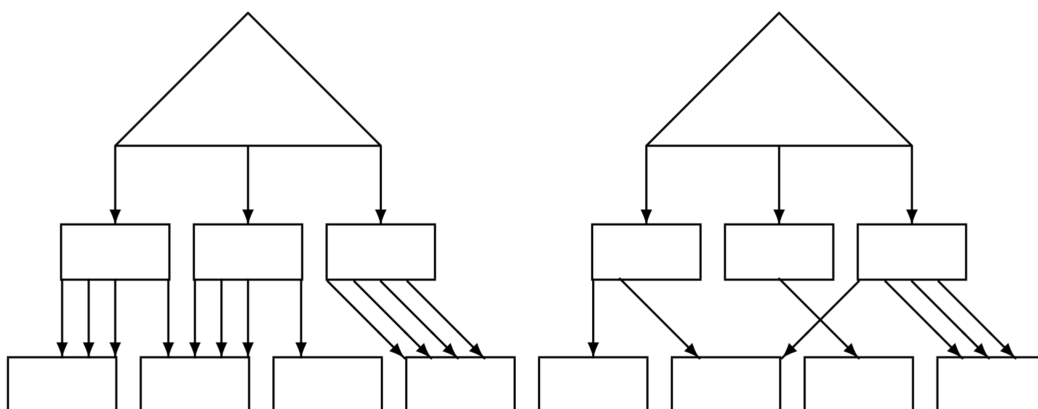
hustý (dense) index - obsahuje aspoň jednu dátovú položku pre každú hodnotu k aktuálnej domény.

riedky (sparse) index - obsahuje jednu dátovú položku pre každú stránku záznamov v dátovom súbore. Musí byť nad klastrovaným indexom.

zložené indexy - ak k obsahuje viac ako jeden atribút tabuľky. Vždy sa

hľadá v poradí, v akom boli atribúty uvedené pri definícii indexu.





klastrovaný index

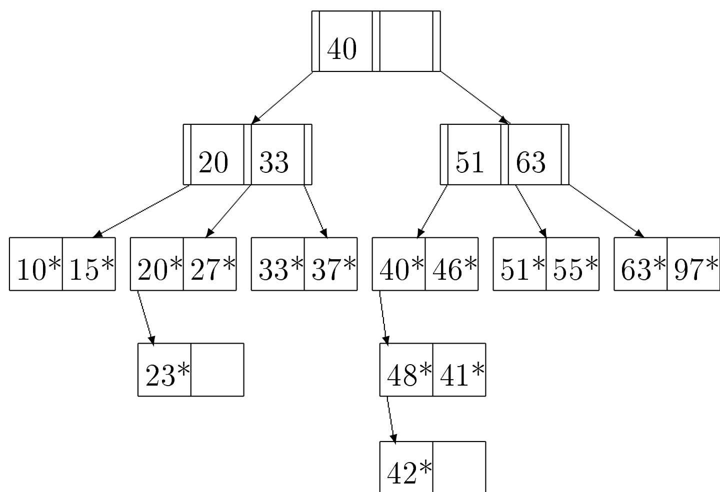
neklastrovaný index

3 Stromové indexy

- efektívne rozsahové dopyty
- majú efektívne pridávanie a mazanie na rozdiel od zotriedených súborov
- celkom dobré dopyty pre rovnosť (nie až také dobré ako hash)

3.1 ISAM (Index Sequencial Access Method)

- je statická štruktúra, kde na najvyššej úrovni je smerník
- málo efektívne pri častom pridávaní a mazaní
- každý uzol je stránka, všetky dáta sú v listoch
- primárne stránky sú umiestnené sekvenčne, lebo ich počet je jasný pri tvorbe stromu
- ak sa zmažú údaje z primárnych stránok, tak sa nepresúvajú dáta zo stránok pretečenia
- organizácia súboru: | index | primárne stránky | overflow |
- výhoda oprti B+ stromu
  - listové uzly sa nemusia zamykať, lebo je to statický index (zrýchlenie, lebo často pristupované uzly sú nezamknuté)



3.2 B+ strom

- dynamická štruktúra
- motivácia - zlepšiť čas binárneho vyhľadávia
- vytvoríme druhý súbor s jednou dátovou položkou pre každú stránku dátového súboru (riedky index)
- nema stránky pretečenia
- vyvážený strom
- primárne listové uzly nie sú sekvenčne vedľa seba, ale susedné listy majú na seba smerníky
- garantované min. zaplnenie uzlov 50% okrem roota
- výška stromu určuje na koľko krokov sa nájde záznam

- listy môžu obsahovať všetky druhy dátových položiek. Ak obsahujú jeden druh (k*), tak B+ strom obsahuje všetky dáta (je dátovým súborom).
- ak je jeden druh, treba myslieť na to, keď záznamy sú rôznej dĺžky

3.2.1 Search

`function` search(k): smerník na uzol

tree_search(root, k);

`function` tree_search(smerník na uzol p, hodota k): smerník na uzol

ak *p je list, tak vráť p;

ak $k < k_1$, tak vráť tree-search(p_0, k);

ak $k \geq k_m$, tak vráť tree-search(p_m, k);

nájdi i: $k_i \leq k < k_{i+1}$ a vráť tree-search(p_i, k);

3.2.2 Insert

- rekurzívne sa vnorí do listu, kde sa má záznam vložiť a vracia sa späť, niekedy je vrchol plný a musí sa rozdeliť

`procedure` insert(smerník na uzol p, záznam z, nová položka so synom);

ak *p je vnútorný uzol N, tak

nájdi p_i podstromu kam patrí z;
insert (p_i, z, n);
ak n = null skonči;
ak N má voľné miesto, vlož *n, nastav n = null a skonči;
rozdeľ:
prvých d položiek ostáva aj s prvými d+1 smerníkmi
posl. d položiek a d+1 smerníkov vložíme do nového uzla N2;
n := & (<minimálna hodnota v N2, smerník na N2>)

ak N bol root, tak urob nový root <smerník na N, n> a skonči;

ak *p je list L

ak L má boľné miesto, tak vlož z a skonči;

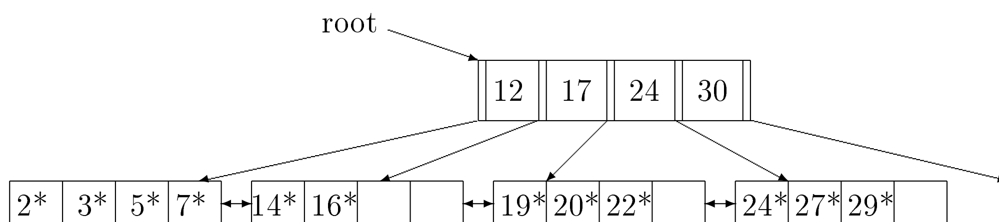
rozdeľ:

prvých d položiek ostáva aj s prvými d+1 smerníkmi

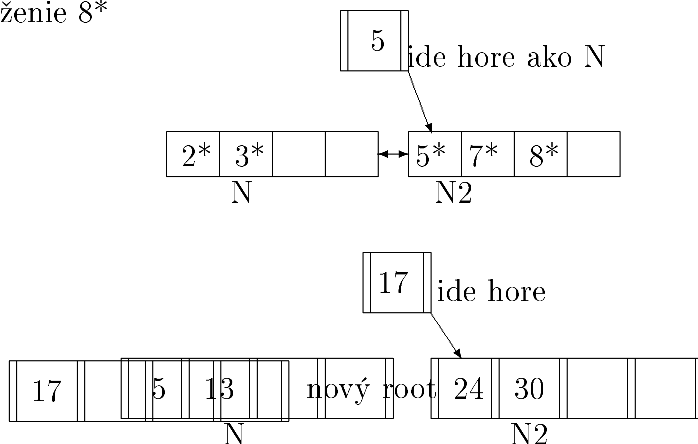
posl. d položiek a d+1 smerníkov vložíme do nového uzla N2;

n := & (<minimálna hodnota v N2, smerník na N2>)

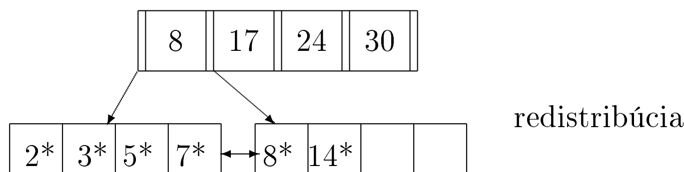
ak N bol root, tak urob nový root <smerník na N, n>, skonči a nastav smerníky so susedmi;



Vloženie 8*



- rozdiel medzi delenim listov a vnútorných uzlov je preto, že každý tak chceme mať v listoch, lebo chceme ľahšie odpovedať na rozsahové dopyty
- sú aj také varianty insertu, ktoré sa snažia hodiť záznam za bezprostredným súrodencom (majú rovnakého rodiča)



- to však zvyšuje počet I/O, ak súrodenci sú plní
- oplatí sa to, ak redistribúciu nerobíme na vnútorných uzloch
- ak delíme listy, tak potrebujeme aj jedného suseda, aby sme zmenili smerník, preto redistribúcia má zmysel

3.2.3 Delete

`procedure` delete(smerník na otca O, smerník na uzol P, záznam Z, záznam starého syna SS)

ak P* je uzol N, tak

nájdi P_i podstromu, kam patrí Z;

delete (P, P_i, Z, SS);

ak SS = null, skonči;

odstráň *SS z N;

ak počet položiek je menší ako D;

vezmi súrodenca S;
ak S má $<$ D položiek, tak
prerozdeľ položky do S a N cez otca;
nastav SS na null a skonči;

spoj S a N;

SS = & (aktuálna položka ukazujúca na zaniknutý uzol k otcovi);

vlož *SS do spojeného uzla;

ak P* je list L;

ak počet položiek je menší ako D

vezmi súrodenca S;

ak S má $<$ D položiek, tak

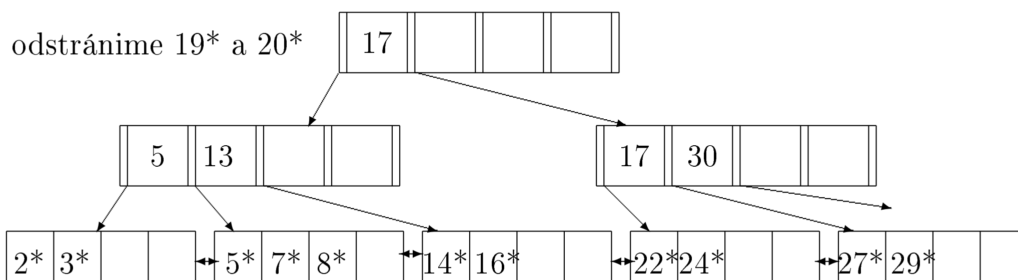
prerozdeľ položky do S a N cez otca;

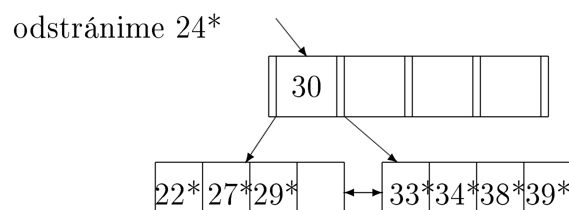
nastav SS na null a skonči;

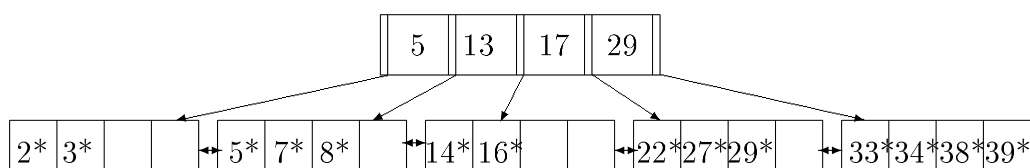
spoj S a N;

SS = & (aktuálna položka ukazujúca na zaniknutý uzol k otcovi);

nastav smerník so susedmi;







- redistribúcia má zmysel, lebo zmeny sa prejavia iba v otcovi
- tabuľky sa obvykle zväčšujú!

3.2.4 Duplicity

1. Overflow pages
2. prehľadávanie zmeniť na najľavejší záznam s kľúčom **k**
3. rozšíriť kľúč k na $\langle k, rid \rangle$ (máme unique) – pre aleternatívu (2)
4. robiť zoznamy

3.2.5 B+tree v reálnych systémoch

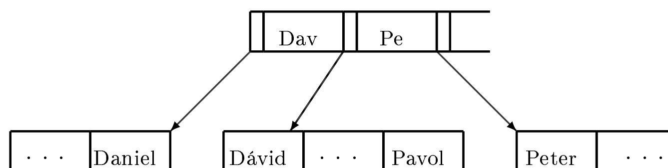
- často sa vymazanie deje iba tak, že sa záznam označí ako vymazaný
- Oracle umožňuje co-klastrovanie záznamov z viacerých relácií

3.2.6 Key kompresia

- výška stromu závisí od počtu dátových položiek a veľkosti ... položiek
  výška = log $_{\text{fan-out}}$(# dátových položiek)

  fan-out = rozvetvenie

- ak su kľúče vo vnútorných uzloch príliš dlhé stringy, tak ich veľa na stránku nevojde

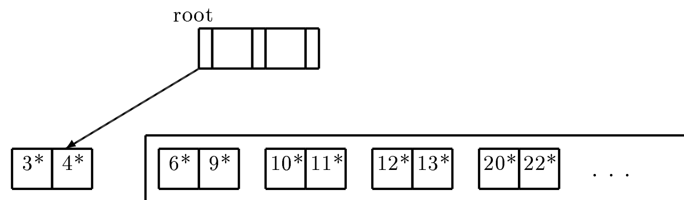


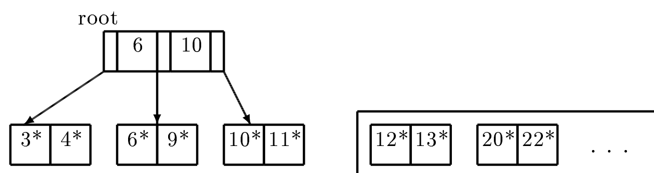
3.2.7 Rád stromu

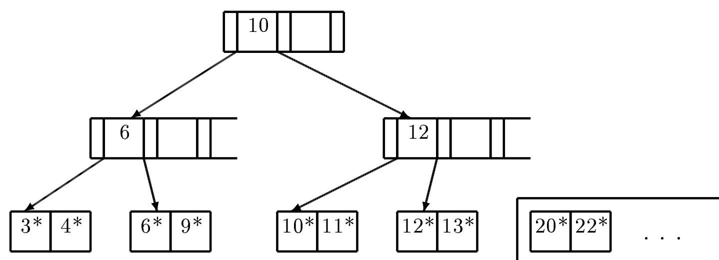
- rád stromu je štandardne rovný polovici zaplnenia uzla v strome
- v praxi stačí povedať, že stránka je z polovice zaplnená
- listové uzly obsahujú spravidla rôzne veľké položky
- ak je kľúč string, tak aj dátove aj indexové položky sú každá rôzne dlhá

3.2.8 Bulk loading = dávkové pridávanie

- pridávanie veľa dát po jednom môže byť drahé
- prvým krokom je usporiadať záznamy podľa k (alebo $\langle k, rid \rangle$ v prípade (2)) nech rád stromu d = 1 a usporiadané dáta máme v stránkach





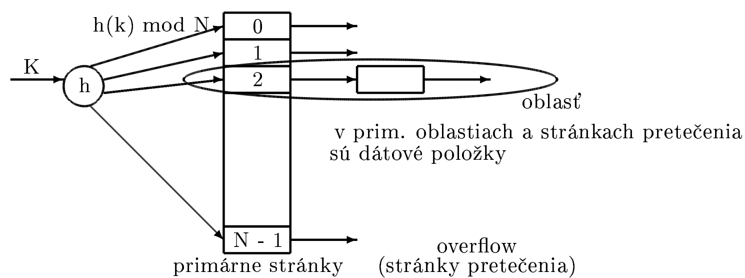


- ľavé podstromy sa už nikdy nemenia (v pamäti stačí uchovávať iba celú pravú vetvu stromu)
- celková cena dávkoveho pridávania
  1. zápis dátových položiek do stránok (# stránok so záznamami + #stránok s dátovymi položkami)
  2. sort = cca 3 * # stránok s dátovymi záložkami

4 Hashové Indexy

- vynikajúce pre dopyty rovnosti (napr. nested loops join)
- hashovacia funkcia mapuje hodnoty z domény kľúča do čísel buchetov (oblastí)

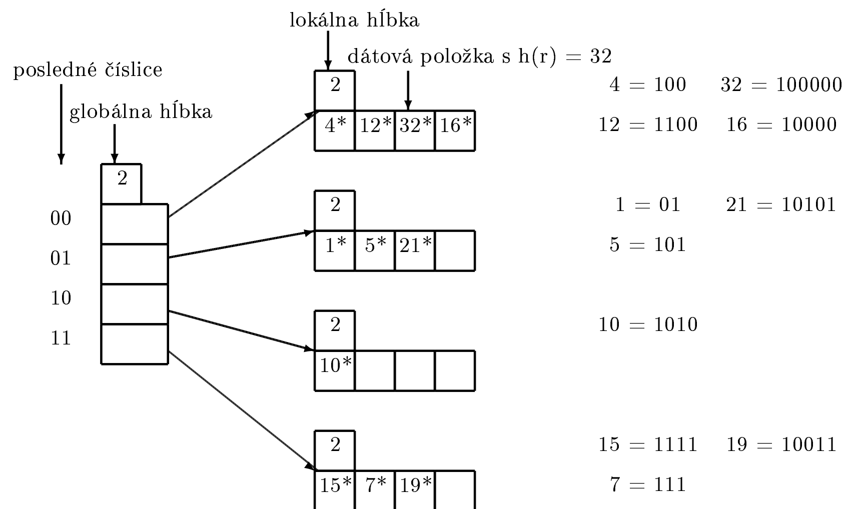
4.1 Statické hashovanie



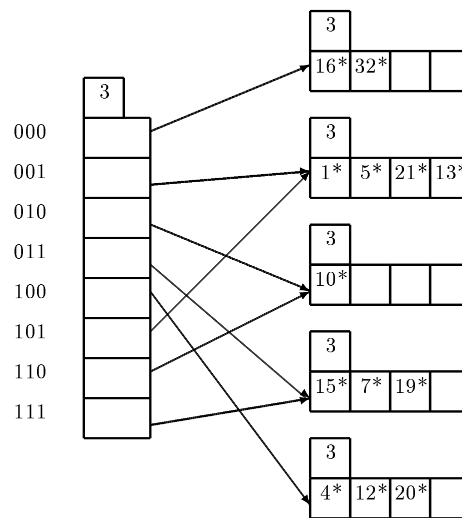
- hashovacia funkcia musí distribuovať doménu kľúča rovnomerne do oblastí
- príkladom môže byť $\mathbf{h(x) = ax + b}$, potom **oblasť** $= \mathbf{h(x)}$ **mod N** konšanty **a,b** sa nastavujú podľa rozdelenia hodnôt
- primárne stránky sú za sebou na disku takže sa ľahko určí, kde je i-ta oblasť a na nájdenie čisto stačí **1 I/O** operácia
- pri častom pridávaní / mazaní sa predlžujú oblasti pretečením a index začína byť zlý

4.2 Extendible (Rozpínavé) hashovanie

na pochopenie – ak chceme zabrániť overflow stránkam znásombíme počet oblastí a rozhodíme záznamy. Nameisto načítania a zapísania všetkých oblastí použijeme adresár pointrov na oblasti, zdvojnásobime ho a rozdelíme iba oblasť ktorá pretiekla.



- keď vložíme $\mathbf{h(r) = 13 = b(1101)}$ tak sa štruktúra nemení, lebo ide do 2. oblasti
- keď hceme vložiť $\mathbf{h(r) = 20 = b(10100)}$ musíme rozdeliť 1. oblasť podľa posl. troch číslic



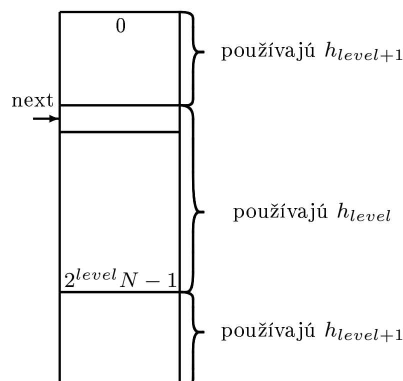
- rozpínanie adresára znamená skopírovať celý adresár a zmeniť v 2. kópii jeden smerník
- na zistenie či rozpínanie adresára nám stačí - porovnať lokálnu hĺbku delenej časti s globálnou hĺbkou
- na oblasť ukazuje 2^{g-l} adresárových položiek (**g** = # glob. Hĺbka, **l** = lokálna hĺbka)
- mohli by sa použiť aj prvé bity ale potom by rozpínanie nešlo cez kopírovanie
- pri mazaní – ak sa oblasť vyprázdni, mohla by sa spojiť (často to nerobí), zníži sa lokálna hĺbka. Ak sa znížia všetky lokálne hĺbky adresár sa môže zraziť na polovičný
- ak je adresár v pamäti tak dopyt rovnosti je na **1 I/O**
- pri asymetrických *(šikmých = skew)* rozdeleniach hodnôt, tak treba vybrať lepšiu hashovaciu funkciu, ktorá rozhodí dáta rovnomernejšie – v praxi sa to celkom dá
- treba pamätať aj na **kolízie**, keď $\mathbf{h(x)} = \mathbf{h(y)}$. Ak príliš veľa kolízií zaplní stránku, treba aj tak robiť stránky pretečenia.

4.3 Lineárne hashovanie

- nepotrebujeme adresár
- ak sú veľmi asymetrické dáta tak dáta v oblasti pretečenia môžu spoôsobiť neefektivitu využíva funkcie h_0, h_1,... také, že ak obor hodnôt h_i = **0...N**-1 tak obor hodnôt h_{i+1} = **0...2N-1**
- typicky ak máme hashovaciu funkciu h tak
  $h_i(\mathbf{k}) = \mathbf{h(k)\ mod}\ (2^N)$
- ak štartovací počet oblastí je **N**=2^{d0}, tak pre ľub. **i** sa pozeráme na posledných $d_{0+i} = d_i$ bitov hodnoty hashovacej funkcie **h**.

N = 32

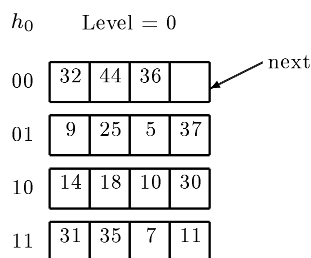
- h_0 = h mod (32) $\rightarrow$ číslo oblasti potrebuje 5 bitov
- h_1 = h mod (2*32) = h mod (64) $\rightarrow$ číslo oblasti potrebuje 6 bitov
- index používa globálnu premennú **level** a smerník **next**

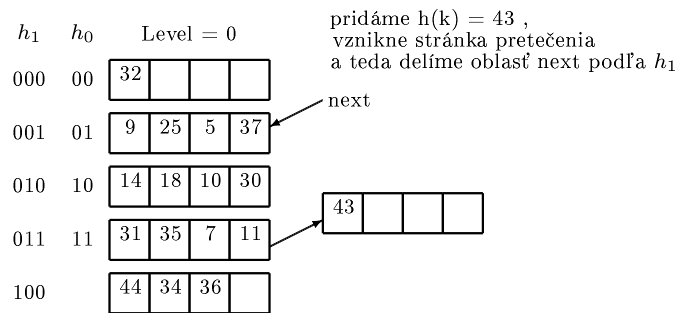


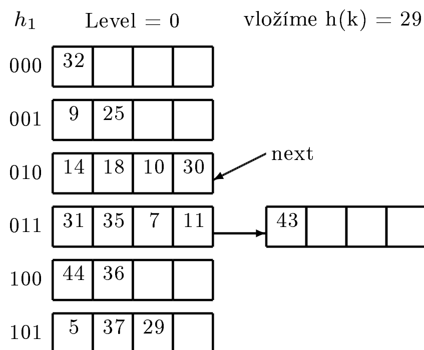
Hľadanie

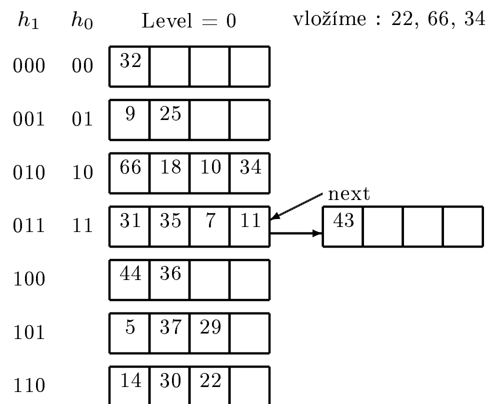
1. vypočítame $h_{level}(\mathbf{k}) = \mathbf{x}$, ak $\mathbf{x} \geq$ **next** čítame oblasť
2. ak $\mathbf{x} <$ **next** vypočítame $h_{level+1}(\mathbf{k})$ a čítame oblasť

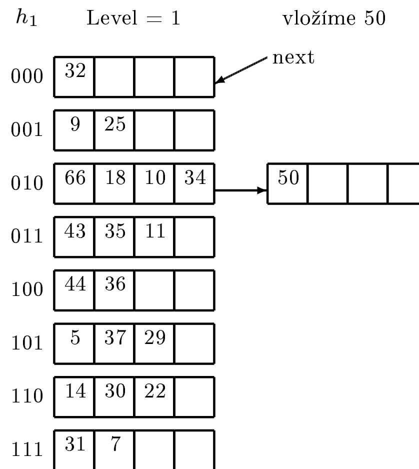
- ak pri lineárnom hashovaní chceme vložiť záznam do plnej stránky tak to nemusí byť tá stránka, ktorá sa rozdelí v rámci primárnych stránok, ale môže dôjsť k vytvoreniu stránok pretečenia
- oblasti sa rozdeľujú v poradí od **0-tej** až po (2^{level} **N-1)-vú**
- ak vznikne nová stránka pretečenia deľme oblasť označenú **next** a smerník **next** presunieme na nasledujúcu oblasť











- cena dopytu rovnosti je pri zhruba rovnomernom rozdelení asi **1,2 I/O**
- delete môže zmršťovať index ak je posledná oblasť prázdna (alebo vojdu spolu so stránkou $h_{level-1}$(k)
- prechod $h_i \rightarrow h_{i+1}$ vlastne zodpovedá zdvojnásobeniu adresára
- má menšie zaplnenie oblastí ako rozpínave hashovanie
- pre rovnomerné rozdelenia je však absencia adresára zrýchlením pre dopyty rovnosti
- pre šikmé rozdelenia je lepšie rozpínavé hashovanie, kvôli väčšiemu zaplneniu

5 Výpočet dopytov

5.1 Metadáta

Takmer všetky metadáta sú uložené v tabuľkách.

- o tabuľkách - meno, názov súboru, štruktúra súboru (napr: heap), pre každý atribút - meno a typ, indexy, integritné obmedzenia
- o indexoch - meno, štruktúra, kľúč
- o view-och - meno, definícia
- štatistiky o tabuľkách a indexoch - aktualizujú sa občas
  kardinalita (počet riadkov tabuľky), veľkosť (počet stránok), kardinalita indexu (počet rôznych hodnôt kľúčov), veľkosť indexu (počet stránok indexu, B+stromy - počet listov), výška indexu, max. kľúč, min, kľúč
- údaje o užívateľoch a právach

5.2 Využitie indexov

- hash index sa dá použiť, ak je podmienka v CNF a všetky atribúty kľúča sa vyskytujú vo forme atribút = hodnota
- stromový index sa dá použiť, ak podmienka v CNF je prefixom kľúča indexu: ak kľúč indexu je <a, b, c>, tak dobre sú <a>, <a, b>, <a, b, c> a zlé napr. <b, c>, <a, c>
- ak sa dá použiť viac indexov, treba vybrať jeden a zvyšné podmienky overovať pre každý vybraný riadok
- najlepšie je vybrať najselektívnejšiu cestu
  selektivita cesty - počet vrátených stránok na zistenie celej odpovede
  redukčný faktor - počet tíc spĺňajúcich jednu subpodmienku v CNF
- počet tíc spĺňajúcich celú podmienku sa dá aproximovať súčinom redukčných faktorov
- pre hash index je odhad počtu stránok spĺňajúcich podmienku rovnosti $\#stranok(tabulka)\frac{1}{\#klucov(index)}$ (ak je klastrovaný index)

- pre stromy je počet záznamov odhadovaný ako $\frac{1}{\#klucov}$, ak je podmienka rovnosti, ak je napr. $x > 4,2$, tak redukčný faktor $= \frac{max_x - 4,2}{max_x - min_x}$
- pre lepší odhad máme histogramy

5.3 Zádrhle relačných operátorov

σ selekcia - ak máme neklastrovaný index, tak počet I/O môže narásť až $\frac{\#zaznamov}{stranok}$ krát, napr. 200 krát, niekedy je potom lepšie urobiť table scan

π projekcia - problém nastáva, ak chceme distinct - usortíme výsledok a povyhadzujeme, čo je naviac - dá sa urýchliť, ak začneme sortiť už pri prvom čítaní a vypisovať pri poslednom čítaní
- najlepšie, ak máme už klastrovaný index

$\bowtie$ join - každý join má svoje výhody a nevýhody, je výhodné, ak joinujeme už osekané dáta

- každý operátor môže svoj výsledok uložiť = materializovať alebo poslať ako prúd dát ďalšiemu operátoru = on the fly (operácie open, get_next, close)

6 Externé triedenie

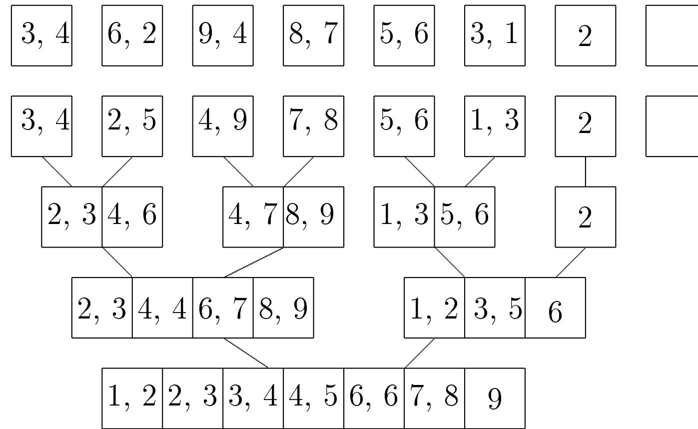
- zotriedenie výsledku selektu
- prvý krok pre bulk-loading do B+stromu
- odstránenie rovnakých riadkov pri projekcii
- niektoré joiny vyžadujú triedenie ako jeden z krokov

6.1 Jednoduchý dvoj-cestný merge sort

- ak nemáme takmer žiadnu RAM

1. 0-ty prechod: čítaj každú stránku, zotrieď jej zoznamy a zapíš
2. pokiaľ počet behov v predchádzajúcom prechode bol > 1, tak rob:

i-ty prechod: pre nespracované 2 behy z predchádzajúceho prechodu čítaj ich po stránkach a generuj výstup po stránkach ako nový beh



Zložitosť:

- 0-tý prechod vygeneruje 2^k behov
- 1-vý prechod vygeneruje 2^{k-1}
- k-tý prechod vygeneruje 1 beh
- teda počet I/O: $2N(\lceil log_2 N \rceil + 1)$ (R/W * počet prechodov), pamäť: 3 stránky

6.2 Externý merge-sort

- v 0-tom prechode zotriedime B stránok v pamäti (B = buffer)
- v i-tom prechode robíme (B-1) - cestný merge-sort (jedna stránka je na output)

Zložitosť:

- 0-tý prechod vygeneruje $\lceil N/B \rceil$ behov,
- počet I/O: $2N(\lceil log_{B-1}(\lceil N/B \rceil) \rceil + 1)$
- cena CPU rastie so zvyšujúcim sa B
- počet prechodov pre N=1000 000 a B = 256 (4kB/str.) = 1MB je 3 pre N = 1 000 000 000 je 4

6.3 Replacement Sort (pre 0-tý prechod)

- buffer sa rozdelí na zotriedenú množinu kandidátov o B-1 stránok a output buffer
- v prvom kroku sa naplní output záznamami s najmenším kľúčom, zapamätá sa maximálna hodnota M output stránky a tá sa vypíše na výstup
- v i-tom ktoku načítame 1 stránku zo vstupu do zotriedenej množiny a do output buffera sa vložia najmenšie hodnoty väčšie alebo rovné ako M a zapíšeme novú hodnotu M
- ak už nevieme naplniť output buffer, začneme ho plniť od najmenších záznamov zotriedenej množiny (ako v 1. kroku)

- podľa doteraz známych informácii sa to v komerčných databázach nepoužíva kvôli rôznej veľkosti záznamov (zatiaľ)

I/O po blokoch

- predpokladáme, že čítame/zapisujeme naraz, povedzme, 32 stránok
  - tým zmenšíme počet ciest v merge-sorte 32-krát
  - dá sa to riešiť tým, že použijeme 32-krát väčšiu pamäť
  - výhoda: stačí 1 seek time, 1 rotačné oneskorenie a 32-krát transfer time

Double buffering

- ak nechceme, aby procesor vykonával takmer nulovú činnosť zatiaľ čo sa čaká na načítanie ďalšieho bloku dát
- keď spracúvam i-ty blok a viem, že budem potom spracúvať (i+1)-ý, tak nechám načítať aj ten (producent - konzument)

Použitie B+ stromov na triedenie

- clastrovaný strom je výborný: na utriedenie všetkých záznamov treba prechod od roota do ľavého listu +N
- neklastrovaný je často nevýhodný:

  Máme f záznamov <kľúč, rid> v každom liste a p záznamov v dátových uzloch:
  - často P/f < 0,1
  - celková cena (P+P/f).N môžeme aproximovať na p*N (najhorší prípad)
  - teda, ak máme p=10, tka zložitosť triedenia všetkých záznamov je 10*N (na sort stráca cca 3N)
  - dokonca ak selektivita podmienky na daný kľúč je 10-20% a p ≅ 20, tak je často lepší sort

7 Výpočet relačných operátorov

Nasledujúce výpočty budú používať tieto dáta:

- Predavači (pid:int, pmeno: strint, rating: int, vek:real)
- Objednávky (pid:int,zid:int,den:date, meno:string)

	Predavači	Objednávky
n-tica	50 byteov	40 byteov
stránka	80 záznamov	100 záznamov
N	500 stránok	1000 stránok

- budeme ignorovať zložitosť zapísania výsledku dopytu a počítať len I/O cenu

7.1 Selekcia

```
SELECT * FROM Objednavky WHERE meno="peter"
```

- žiaden index, heap-file: môžeme prejsť všetky záznamy a overovať podmienky = 1000 I/O
- žiaden index, sorted file: binárne vyhľadávanie; nájdenie prvého je $\log_2 N$
  - v našom príklade $\log_2 1000 \cong 10$ I/O
- B+strom:
  - pre ľubovoľný OP je klastrovaný B+strom najrýchlejšia možnosť okrem rovnosti, kre je lepší hash
  - pre neklastrovaný index máme v listoch informácie kde sú spĺňajúce záznamy
  - ak už načítame nejakú dátovú stránku, tak by sme z nej mohli vypísať všetky spĺňajúce záznamy (listové dáta spĺňajúce podmienku sa usortia podľa smerníkov (rid)); potom cena je očet dátových stránok obsahujúcich spľňajúce záznamy
  - v našom príklade pri selektivite 10% má klastrovaný B+strom (100+cca 2) I/O; neklastrovaný môže mať viac ako 1000 I/O; scan je niekedy rýchlejší
- hash index a rovnosť:
  - klastrovaný je nájdenie prvého cca 1,2 I/O + počet stránok v prípadnej oblasti pretečenia ak ich je veľa
  - pre neklastrovaný je sitácia podobná ako neklastrovaný B+strom, akurát, že selektivita býva väčšia
- disjunkcie:
  - podmienky sa najprv hodia do CNF (napr. (a ∨ b) ∧ c ∧ (d ∨ c))
  - pokiaľ na jeden člen disjunkcie nemáme index najlepšie je table scan pre celú podmienky ak máme $(a_1 \vee a_2 \vee \cdots \vee a_n)$

  - ak je (a ∨ b) ∧ c a c je dobre selektívny a indexovaný, tak testujeme (a ∨ b) na vyselektovaných záznamoch
  - ak máme ($a_1 \vee \cdots\ a_n$) a každý atribút má index, môžeme použiť stratégiu, že budeme robiť zjednotenie výsledkov; ak sú všetky indexy neklastrované, tak sa robí iba zjednotenie záznamov podľa rid cez triedenie

- konjunkcie: podobne ako disjunkcie, len treba robiť prienik

7.2 Projekcia

Jediný problém je `DISTINCT`.

- treba spraviť 3 veci:
  1. vyprodukovať také tice, ktoré spĺňajú projekciu
  2. usporiadať tieto tice podľa kombinácie atribútov
  3. prejsť výsledok a vyhádzať duplicity
- dá sa to zlepšiť tak, že pri o-tom prechode sortu sa čítaju celé záznamy a generujú už utriedené osekané behy , pričom sa duplicity vyhadzujú už pri prechodoch

Hashová metóda:

1. delenie: aplikujeme hashovanú funkciu h na všetky orezané záznamy a rozdeľujeme ich do B-1 oblastí; ak sa v buffri zaplní stránka, ide na disk

2. odstránenie rovnakých za použitia hashovacej funkcie h2 urob hash tabuľku v buffri z jednej oblasti delenia a odstráň duplicity ak h2 vráti viac vecí do jednej oblasti

  - táto metóda vyžaduje veľký buffer
  - ak T je počet stránok s orezanými záznamami, tak každá oblasť obsahuje T/(b-1) . f stránok (f je koeficient zaplnenia) a to sa musí v 2. Kroku vôjsť do buffra, teda B> T/(B-1) . f -> B > odmocninaz(T.f)

- obe metódy (hash aj zlepšený sort) majú, zhruba, rovnakú I/O cenu
- ak máme B+ strom: na tých atribútoch, na ktorých robíme projekciu, tak triedenie nemusíme robiť a duplicity sa riešia ľahko
- zložitosť: Prechádzajme objednávky (záznam 40 byteov, 100 záznamov). Nech projekcia vyrába orezané záznamy 10 byteov.

  Triediaca metóda 1: 1000 I/O na scan, 250 I/O zápis orezaných, sort na 2 prechody, 2.2.250 I/O = 2500 I/O dokopy, lebo 250 I/O na odstránenie

  Triediaca metóda 2: 1000 I/O na scan, 25O I/O na zápis cca 7 behov, 250 I/O na dotriedenie s odstránením, dokopy 1500 I/O B+ strom: 1000 I/O

8 Join

```
SELECT * FROM Objednavky o, Predavaci p WHERE o.sid=p.sid
```

$$O \bowtie_{sid} P = \sigma_{o.sid=p.sid}(o * p)$$

Nested loops join

- prechádzame outer reláciu O a pre každú ticu r $\in$ O prechádzame celú inner reláciu B
- cena prechádzania O je M I/O (M stránok)
- majme počet tíc v jednej stránke O = p_o potom P prechádzame p_o . M krát vždy s cenou N I/O (N stránok) - celková cena je teda M p_o . M .N

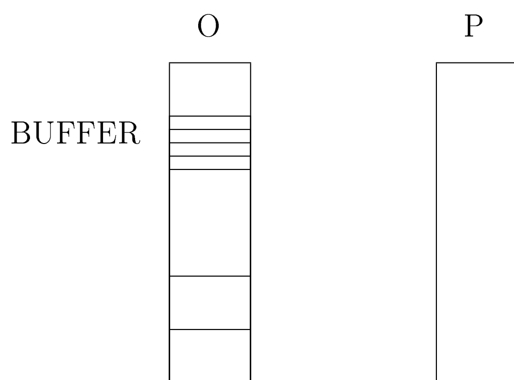
- ak joinujeme Objednavky a Predavacov: M=1000, p_o=100, N=500, tak celková cena je 1000+ 100. 1000. 500 = 50 001 000 I/O (pri cene jedného I/O = 10ms to vychádza na 140 hodín)
- dá sa zlepšiť ak pracujeme po stránkach: pre každú stránkuu O prechádzame každú stránku P a spravíme join stránok v pamäti
- cena prechádzania je už iba M+M.N, teda v našom príklade 1000 + 1000.500 = 501 000, čo zaberie 1,4 hodín
- ešte môžeme prehodiť poradie O a P, lebo join je symetrický, potom 500+500 .1000= 500 500

8.1 Block Nested Loops Join

Predpokladajme, že vieme menšiu reláciu dať celú do buffra, potom nám stačia už iba 2 extra framy v buffri. Jeden na stránku z druhých relácii a druhú na výstup: cena M+N=optimum (M-počet stránok O, N-počet stránok P).

Na zlepšenie ceny procesorového času je lepšie urobiť hash tabuľku z relácie v pamäti. Ak nám celá outer relácia nevojde do pamäte sčítame blok B-2 stránok a prejdeme celú reláciu. To opakujeme, kým celá outer relácia nebola v pamäti.

- Cena je: $M + \frac{M}{B-2} * N$, ak ignorujeme extra miesto pre hash tabuľku



Príklad 1 *Nech nám vojde naraz 100 stránok.* $O \ldots M = 1000$ $P \ldots N = 500 B = 102$ *Ak robíme*

- $O \bowtie P$: $1000 + 10 * 500 = 6000$ *I/O cca 1 minúta*
- $P \bowtie O$: $500 + 5 * 1000 = 5500$ *I/O*

8.2 Index Nested Loops Join

Vnútorná relácie je index, v prípade B+ stromu máme list na cca. 2-4 I/O, v prípade Hash máme list na cca 1-2 I/O.

- Ak robíme $O \bowtie P$:
  - $1000 + 100 * 1000(1 + 1,2) = 221000$ I/O ak (2)
  - $1000 + 100 * 1000 * 1,2 = 121000$ I/O ak (1)
- ak robíme $P \bowtie O$:
  - $500 + 80 * 500 * 1,2 = 48500$ I/O ak (1)

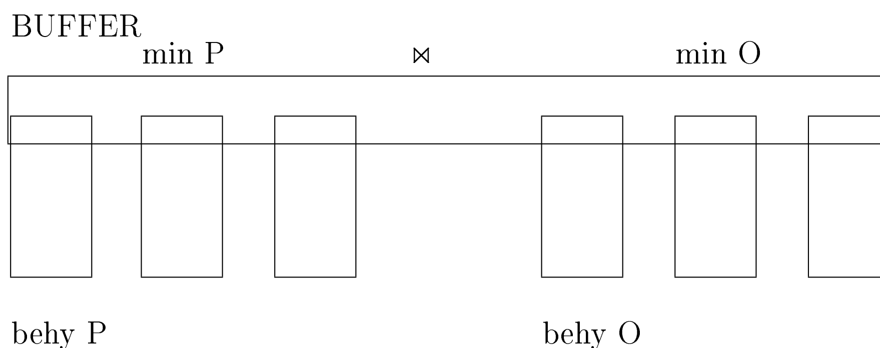
8.3 Sort-merge join

Najlepšie je ak netreba vôbec triediť (indexy). Prechádzame oboma zotriedenými reláciami, kým nenájdeme zhodu. Ak nájdeme zhodu robíme kartézsky súčin zhodných častí.

Cena spájanie je M+N ak joijnujeme relácie cez kľúč prvej relácie.

Príklad 2
- *Majme B=100,*
  - *objednávky zotriedime na 2 prechody=*$2 * 2 * 1000 = 4000$ *I/O,*
  - *predavačov zotriediete tiež na 2 prechody=*$2*2*500 = 2000$ *I/O,*
  - *celková cena* $4000 + 2000 + 1000 + 500 = 7500$ *I/O, čo je viac ako block nested loops join.*
- *majme B=35,*
  - *triedenia sú stále na 2 prechody a cena ostáva 7500 I/O, pre block nested loops join je to viac ako 15000 I/O (*$1000+28*500 = 15000$ *I/O,* $500 + 16 * 100 = 16500$ *I/O)*
- *majme B=30O*
  - *sort-merge join stále 7500 I/O, block nested loops join už za 2500 I/O (*$1000 + 4 * 500 = 3000$ *I/O,*$500 + 2 * 1000 = 2500$ *I/O)*
- ***vylepšenie:** ak buffer* $B > \sqrt{M} + \sqrt{N}$*, lebo ak robíme 0-tú frázu pre reláciu veľkosti M vznikne nám* $\frac{M}{B}$ *behov a tie buď chceme utriediť na 1 prechod treba nám* $B > \frac{M}{B}$ *potom* $B > \sqrt{M}$*. teba ak máme buffer* $B > \sqrt{M} + \sqrt{N}$ *vieme konfigurovať sortovací prechod s joijnom.*

- *cena:* $2000 + 1000 + 1000 = 500 = 4500\ I/O$



A ak použijeme na 0-tý krok replacement sort môžeme mať aj menej behov a teba menší buffer.

8.4 Hash Join

Má 2 fázy. Prvá je rovnaká ako pri odstraňovaní rovnakých riadkov pri projekcii, kde sa rozdelia obe relácie do oblasti podľa funkcie h. V druhej fáze načítame jednu oblasť outer relácii do hash tabuľky a pamäti podľa h2 a prechádzame zodpovedným oblasti inner relácie a hľadáme v hash tabuľke zodpovedajúce riadky.

- cena je $3(M + N)$ to je v našom príklade $3(1000 + 500) = 6500$ I/O

Pamäť treba opäť $B > \sqrt{min(M, N) * f}$ ak je menšia outer. Ak predsa len niektorá oblasť nevojde do buffra robíme rozdelenie rekurzívne.

8.5 Hybrid hash join

Ak máme viac pamäte rozdelíme outer reláciu na **k** oblastí, ktoré vojdú do buffra (jednom) a ešte nám ostane miesto na prvú oblasť v pamäti, pre ktorú už aj robíme s hash tabuľbou v pamäti t.j. ušetríme R/W prvých oblastí oboch tabuliek.

Príklad 3 *Majme*

- *B=300*

Jednu 250 stránkovú partíciu dáme na disk a jednu do hash tabuľby (=250+500), potom jednu 500 stránkovú na záver relácie na disk a druhú spájame (=1000+500). Potom už len spojenie z disku spolu $750 + 1500 + 250 + 500 = 3000$ *I/O*

- *B=500*

tak všetko vojde do buffra a stačí $500 + 1000 = 1500$ *I/O*

8.6 Porovnania

8.7 Hash join vs. Sort-merge join

- ak $B > \sqrt{x}$, kde $x = \max N, M$, tak cena oboch je $3(M + N)$.
- ak hash hunkcia nevie urobiť rovnomerné rozdelenie môže dôjsť k tomu, že niektorú oblasť nevojde do pamäte - sort-merge join je rezistentný.
- ak je $\sqrt{\min M, N} < B < \sqrt{\max M, N}$, tak hash je tým lepší čím je $\sqrt{minM, N}$ menšie
- ak potrebujeme výstup utriedený je často výhodnejší sort-erge join.

8.8 Hash join vs. Block nested loops join

- ak hash tabuľka outer relácie vojde do pomäte tak obe majú $R + S$
- ak potrebujeme materializovať viac oblastí,tak hash začína byť lepší lebo spájame už iba tú stránku, ktoré môžu byť spojené a nie systémo kartézského súčinu (každý blok s každým blokom).

8.9 Iné podmienky pre join

- P.pid=O.pid a P.pmeno=O.meno
  - pre index nested loops join je najlepšie ak máme na inner reláciu <p.1,meno> - index.

- Merge-sort je potrebné triediť podľa <pid,meno>
- ostatné rovnako.

- *P.pmeno* $<$ *S.meno* beginitemize
- pre index nested loops join musíme mať B+ stromu pre inner reláciu hash a sort sú nepoužiteľné
- ostatné rovnako

8.10 Top-k join

Select P.pmeno,
$3 * P.rating + 2 * (today() - 0.den)$
FROM Predavači P, Objednávky O
WHERE $p.pid = o.pid$
ORDER BY 2 DESC
LIMIT k

8.11 Treshold algoritmus

f-musí byť monotónna, pretože potrebuje málo pamäte.

1. Rob sekvenčný prístup do všetkých relácii a pre každý objekr rob priamy prístup do ostatných relácií na získanie celkovej hodnoty objektu. Ak je to jedna z k najlepších hodnôt zapamäj si objekt a ak si pamätáš viac ako (k+1), tak vyhoď najhorší
2. Ak k najlepších objektov má hodnotu väčšiu ako Threshold $\tau = f(x_1, \ldots, x_m)$, kde $x_1, \ldots, x_m$ sú posledné čítané hodnoty z relácií $R_1, \ldots, R_m$, tak skonči; inak rob 1

Cena: ak k=1 tak najhoršie $\sum(\frac{N_i}{2}) + \sum(\frac{N_i}{2}) * (m-1)$.

Počet záznamov v stránke i, zložitosť block nested loops join výpočtu zodpovedá kartézskému súčinu sa skorším ukončením.

Príklad 4

A 0,5	*B 0,4*	*C 0,8*
B 0,3	*A 0,2*	*A 0,5*
D 0,2	*D 0,1*	*B 0,4*
C 0,0	*C 0,1*	*D 0,0*

Hodnoty sú zotriedené od najlepších po najhoršie.
Riešenie:
$f = x_1 + 2x_2 + 4x_3$
h=2
T=<(A;2,9)>
T=<(A;2,9); (B;2,7)>
T=<(C;3,4); (A;2,9)>
$\tau = 0,5 + 2 * 0,4 + 4 * 0,8 = 4,5$

T=<(C;3,4); (A;2,9)> - ostáva
$\tau = 0,3 + 2 * 0,2 + 4 * 0,5 = 2,7$

9 3P-NRA

- máme dve hodnoty

- $W(X) = f(w_1(X), \dots, w_m(X))$, kde $w_i(X) = min_i$, ak nepoznáme hodnoty X_i, inak X_i
- $B(X) = f(b_1(X), \dots, b_m(X))$, kde $b_i(X) = X_i$, ak poznáme hodnoty X_i inak $\underline{X_i}$ tj. naposledy čítané X_i

9.1 fáza 1

- rob sekvenčný prístup do všetkých zoznamov (relácií) a získaj $\langle X^1, X^1_1 \rangle, \dots, \langle X^m, X^m_m \rangle$, pre všetky X^i vypočítaj $W(X^i)$
- ak $|T| <$ k vlož X^i do T
- inak, ak $W(X^i) > W(T_k)$ vlož X^i do T naspráνne miesto
- ak X^i bolo v C, tak presuň T_k do C

9.2 fáza 2

- pre všetky X patriace do C vypočítaj $B(X)$
- ak $B(X) \leq W(T_k)$ odstráň ho z C
- ak |C|=0 skonči, inak rob fázu 3

9.3 fáza 3

- rob sekvenčný prístup do zoznamov ktoré majú neznáme hodnoty objektov v C a T, pre každý objekt X videný v sekvenčných prístupoch
- ak $X \notin T \cup C$, ignoruj ho inak vypočítaj $W(X)$ a $B(X)$
- ak $B(X) \leq W(T_k)$ vyhoď X z C
- ak $|C| = 0$ skonči
- ak $W(X) \geq W(T_k)$, vlož X do T a ak X bolo v C, presuň T_k do C

- ak sa zvýšil $W(T_k)$ alebo znížil T rob fázu 2, inak rob fázu 3

Cena: iba nejaké percento z $R_1 + \ldots + R_m$, ak $C + T$ vojdú do pamäte

10 Počítanie množinových operácií

- $R \times S$ - ako join bez podmienok
- $R \cap S$ - ako join s podmienkou rovnosti na všetkých atribútoch
- $R \cup S$ - ako sort oboch + join s eliminovaním duplicít
- - hash: partície podla hash aj pre R aj pre S spájanie ako hash join s eliminovaním duplicít
- $R - S$ - ako $R \cup S$ len neeliminujeme duplicity ale tie riadky R, ktoré sú v S

10.1 Počítanie agregačných operácií

- AVG, SUM, MIN, MAX, COUNT – preleziem tabuľku a počítam v extra premenných
- GROUP BY
  - sort: utriedim podľa GROUP BY stĺpca a počítam agregačnú funkciu pre všetky rovnaké
  - hash: urobím hash tabuľku podľa GROUP BY stĺpca (obvykle vojde do pamäte)do tabuľky
  - dávam ⟨ GROUP BY stĺpec, hodnota agregačnej funkcie ⟩
  - - ak nevojde do pamäte tak najprv partitioning

11 Optimalizácia výpočtu dopytov

11.1 Úvodné spracovanie selektu

1. Rozdelenie vnorených dopytov na bloky jednoduchších dopytov

```
SELECT P.pid, min(O.deň)
FROM Predavači P, Objednávky O, Autá A
WHERE P.pid=O.pid AND O.oid=A.aid AND A.farba="žltá"
AND P.rating= (SELECT MAX(P2.rating) FROM Predavači P2)
GROUP BY P.pid
HAVING COUNT(*)>2
```

2. Vyjadrenie blokov dopytov v relačnej algebre

$\pi_{\texttt{P.pid}}$, min(O.deň) (

HAVING COUNT(*)>2 (

GROUP BY P.pid (

$\sigma_{\texttt{P.sid=O.pid} \wedge \texttt{O.aid=A.aid} \wedge \texttt{A.farba="žltá"} \wedge \texttt{P.rating=} \textit{hodnota vnoreného selectu}}$
(Predavači $\times$ Objednávky $\times$ Autá))))

3. Vybratie $\sigma\pi\times$ výrazu

$\pi_{\texttt{P.pid}}$, min(O.deň) (

HAVING COUNT(*)>2 (

GROUP BY P.pid ($\pi_{\texttt{P.pid, O.deň}}$

($\sigma_{\texttt{P.sid=O.pid} \wedge \texttt{O.aid=A.aid} \wedge \texttt{A.farba="žltá"} \wedge \texttt{P.rating=} \textit{hodnota vnoreného selectu}}$
(Predavači $\times$ Objednávky $\times$ Autá)))))

4. Nájdenie prvej aproximácie = alternatívne príkazy pre $\delta\pi\times$výraz

11.2 Zistenie ceny plánu

Ceny ovplyvňujú :

- veľkosť výsledku čiastočneh operácie
- spracovanie on - the - fly (materializovanie čiastkovej operácie)
- zotriedenie výsledku

Veľkosť výsledku stĺpec = hodnota

- je odhad $\frac{1}{\#klucov(I)}$ ak I je index nad stĺpcom (predpoklad zoznam dát)
- ak nemá index tak $\frac{1}{10}$ veľkosti tabuľky

stĺpec1=stĺpec2

- $\frac{1}{max(\#klucov(I_1),\#klucov(I_2)}$ ak vieme oba indexy
- $\frac{1}{max(\#klucov(I)}$ ak vieme 1 index
- $\frac{1}{10}$ ak nevieme nič

stĺpec>hodnota

- $\frac{max(I)-hodnota}{max(I)-min(I)}$ ak vieme index

stĺpec in (...)

- súčet hodnôt typu stĺpec=hodnota avšak max $\frac{1}{2}$ lebo predpokladáme dobrú selektivitu

Tabuľka 1: Počty detí podľa veku

Vek	**Počet zúčastnených**
0	2
1	3
2	3
3	1
4	2
5	1
6	3
7	8
8	4
9	2
10	0
11	1
12	2
13	4
14	9
Celkom	Σ =45

12 Histogramy

12.1 Motivačný príklad

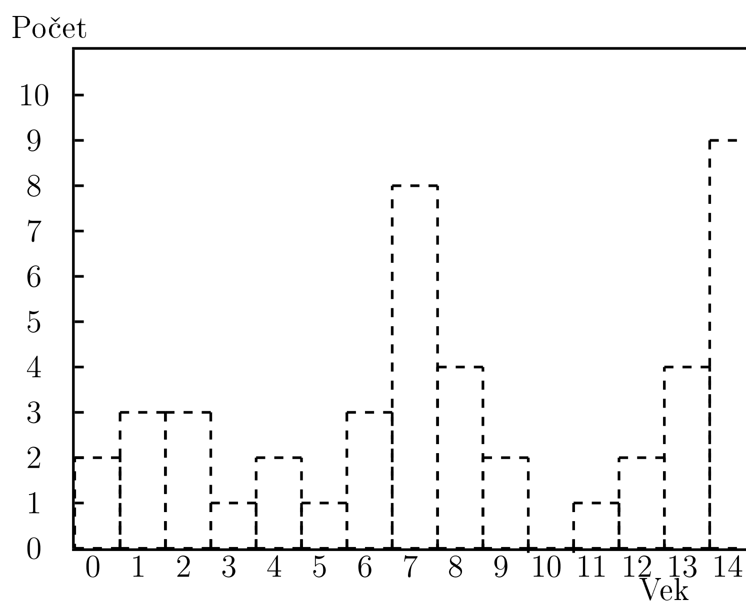
Bol robený prieskum o účasti detí na nedeľných športových aktivitách. Okrem iných otázok bol skúmaný aj vek zúčastnených detí. Zaznamenané počty detí podľa veku sa nachádzajú v nasledujúcej tabuľke.

12.2 Niečo o histogramoch

Histogram alebo histogram početností (z gr. histos - vzpriamený, gramma - kresba, zápis) je stĺpcový diagram (stĺpikový graf) tvorený pravidelnými rovnobežníkmi, ktorých základne (os "x") majú dĺžku zvolených intervalov, a ktorých výšky (os "y") majú veľkosť príslušných absolútnych alebo relatívnych početností zvolených tried.

Podrobné informácie o „štruktúre“ nameraných údajov je možžné získať až z grafického zobrazenia pomocou histogramu.

- Histogram plochého tvaru vzniká obvykle v prípadoch, keď údaje boli zhromaždené za premenlivých podmienok.
- Histogram hrebeňového typu je charakteristický pravidelným striedaním vyšších a nižších hodnôt. To značí, nevhodne stanovené hranice intervalov.
- Asymetrický tvar histogramu väčšinou signalizuje prípad, kde hodnoty sledovaného znaku ležia v blízkosti hranice, ktorá vymedzuje odbor hodnôt znaku. Príkladom je fyzikálna veličina (objem, hmotnosť).
- Histogram s izolovanými hodnotami obvykle signalizuje prítomnosť odľahlých hodnôt.
- Histogram s vyššou početnosťou hodnôt v krajnej triede signalizuje úmyselné skresľovanie nameraných údajov tak, aby neboli prekračované stanovené tolerančné medze.



Obrázok 3: Histogram k úvodnej úlohe

Pravidlá tvorby histogramu:

1. Šírka stĺpcov histogramu musí byť rovnaká
2. Kategórie musia by? vzájomne vyhradené a obsahova? všetko
3. Histogramy môžu byť použité na poukázanie rozdielnych zberov dát vrátane procesov, ktoré vyžadujú rôzne vzorky na stanovenie, či je proces vhodne vykonávaný

Možžnosti využitia histogramu sú ve?mi široké, od analýzy kvality vstupu cez hodnotenie úspešnosti aktivít zlepšovania kvality, analýzy spôsobilosti procesu apod.

Naším cieľom je z uvedených údajov získať čo najviac informácií. Jednou z možností je použitie histogramov nasledujúcich typov:

- Histogram dištančný
- Histogram ekvipotenčný
- Histogram komprimačný

12.3 Histogram dištančný

Rozdelí skúmanú doménu, t.j. vek detí na rovnako široké skúmané časti. V prípade, že budeme uvažovať 5 skúmaných častí, každá z nich bude obsahovať vek troch vekových skupín po sebe nasledujúcich, t. j. 0-2, 3-5, 6-8, 9-11, 12-15. Priemernú zaplnenosť vyjadríme pomocou histogramu na obrázku 2.

12.4 Histogram ekvipotenčný

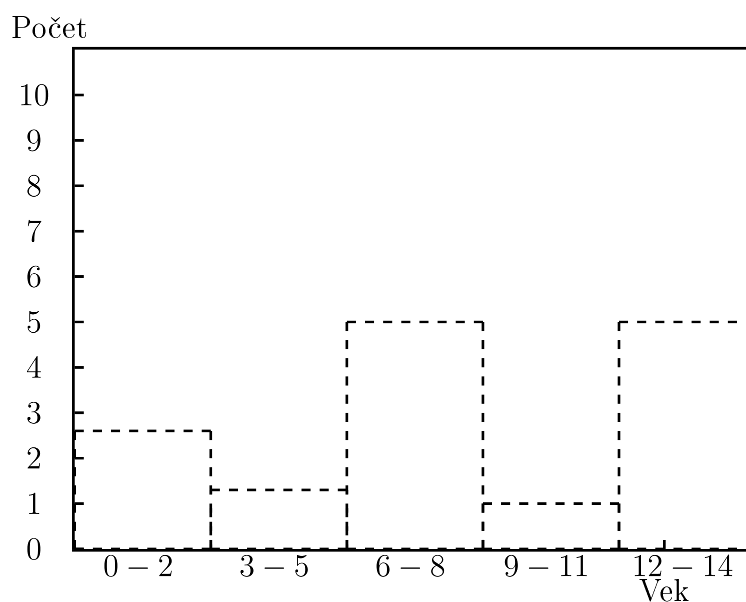
Tento histogram rozdeľuje doménu na rovnako zaplnené časti, t. j. výška je skoro rovnaká, šírka je rôzna. Pri rozdelení 0-3, 4-7, 8-9, 10-13, 14 sú nasledujúce počty: 9, 10, 10, 7, 9.

Vyskúšame podmienku: $vek > 13$, teda 14-ročné deti.

Keby bolo rozdelenie rovnomerné, tak v jednej vekovej skupine by boli 3 deti, pretože $\frac{45}{15} = 3$, takže aj 14-ročných je v priemere 3.

12.5 Histogram komprimačný

Zdôrazňuje, že niektoré hodnoty sú príliš prečnievajúce. Potom prispôsobí rozdelenie intervalov týmto hodnotám. Napríklad, 7- a 14-roční vykazujú veľkú účasť. Preto sú možné dva spôsoby rozdelenia:



Obrázok 4: Dištančný histogram

1. 0-6, 7, 8-13, 14
2. 7, 14, zvyšok

13 Stromy výpočtov

Bežný stroj generuje mnoho strojov výpočtu.

Ekvivalencia:

$$\sigma_{P_1 \wedge \cdots \wedge P_n}(R) \equiv \sigma_{P_1}(\sigma_{P_2}(\ldots \sigma_{P_n}(R)))$$

Komutatívnosť:

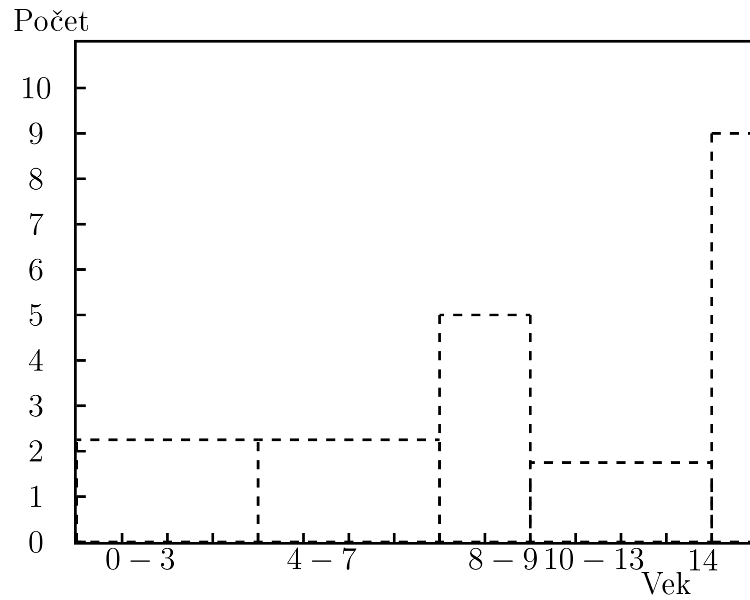
$$\sigma_{P_1}(\sigma_{P_2}(R)) \equiv \sigma_{P_2}(\sigma_{P_1}(R))$$

Projekcia:

$$\Pi_{a_1}(R) \equiv \Pi_{a_1}(\Pi_{a_2}(\ldots \Pi_{a_n}(R) \ldots)), a_i \subseteq a_{i+1}$$

Komutatívnosť a asociatívnosť karteziánskeho súčinu:

$$R \times S \equiv S \times R, \qquad R \times (S \times T) \equiv (R \times S) \times T$$



Obrázok 5: Ekvipotenčný histogram

Toto tiež platí o joinoch:

$$R \bowtie S \equiv S \bowtie R, \quad R \bowtie (S \bowtie T) \equiv (R \bowtie S) \bowtie T$$

Platí:

$$\Pi_a(\sigma_P(R)) \equiv \sigma_P(\Pi_a(R)),$$

ak každý atribút v P je medzi atribútmi a.

$$R \bowtie_p S \equiv \sigma_P(R \times S)$$

Veľmi dôležité:

$$\sigma_P(R \times S) \equiv \sigma(R) \times S$$

$$\sigma_P(R \times S) \equiv \sigma(R) \times S$$

ak P obsahuje iba atribúty v R.

$$Pi_a(R \times S) \equiv \Pi_{a_1} \times \Pi_{a_2}(S),$$

kde a_1 obsahuje atribúty z R, a_2 obsahuje atribúty z S.

$$\Pi_a(R \bowtie_p S) = \Pi_{a_1}(R) \bowtie_p \Pi_{a_2}(S),$$

ak $a_1 \cup a_2 = a, a_1 \in R, a_2 \in S$ a všetky atribúty sú medzi atribútmi v a.

$$\Pi_a(R \bowtie_P S) = \Pi_a(\Pi_{a_1}(R) \bowtie_P \Pi_{a_2}(S))$$

+ ďalšie operácie $\cup, \cap, \setminus$ z úvodu do informatiky.

14 Výpočet alternatívnych plánov nad jednou tabuľkou

Budeme pracovať s jednou tabuľkou:
Π**rating, count(*)**
Group By rating (
Π**rating, meno (**
$\sigma rating > 5 \vee vek = 2$ (Predaváči))))
Bez indexu!

1. file – scan $+\sigma + \Pi$ ... 500 I/O
2. uloženie výsledku ...... 20 I/O
3. zotriedenie ....3*20 I/O = 60

dokopy 580 I/O operácií je odhadovaná cena.

Single index ak máme hash index na vek.

- ak máme **klastrovaný** – nech redukčný faktor pre vek = 2 je 0,1, t.j. 10% všetkých riadkov obsahuje vek 2
- **neklastrovaný** – je to podobné jako bez indexu.

Multiple index

- má zmysel, ak sú neklastrované indexy

- cez prienik rid (row id) a získame len relevantné stránky

Sorted index

- dá sa použiť ak máme klastrovaný B+strom, tým ušetríme triedenie pre Group By
- napr. ak redukovaný faktor rating > 5 je 0,5 stačí 250 I/O a máme výsledok

Index only - nejdeme do záznamov

- ak všetky atribúty v SELECT, WHERE, GROUP BY, HAVING sú v indexoch
- indexy nemusia byť klastrované

Príklad: Majme index B+stromu s kľúčom
<rating, meno, vek>.
Ak veľkosť položky v liste je 80% zo zoznamu potom 500 * 0,8 * 0,5 ..... 200 I/O operácií

14.1 Dopyty nad viacerými tabuľkami

Majme A,B,C,D tabuľky.

Využívajú sa hlavne kvôli tomu, aby sa redukoval priestor plánov – aby výpočet všetkých plánov nebol drahší ako samotný výpočet najlepšieho plánu.

Vieme vyrobiť všetky plány, ktoré používajú on-the-fly výpočet.

14.2 Algoritmus na left-deep plány (dynamické programovanie)

Fáza 1. Vyrobíme všetky 1-relačné plány, použijeme všetky projekcie, ktoré sa nepoužívajú;

v podmienkách, kde sú aj cudzie atribúty použijeme všetky selekcie, ktoré nemajú cudzie atribúty;

pripravíme všetky plány, ktoré generujú vlastné usporiadanie;

Fáza 2. Vyrobíme všetky 2-relačné plány – plány z predchádzajúcej fázy sú ako outer relácie.

Majme $A \bowtie B$.

1. zistíme selekcie použité len pre B;
2. zistíme selekcie definujúce join $A \bowtie B$;
3. outer relácia sa vždy berie tak, že je spracovateľná;
4. opäť všetky plány s vlastným usporiadaním on-the-fly;

Fáza 3. Plány z predchádzajúcej fázy sú outer;

Nakoniec **Group By** a množinové operácie.

14.3 Vnorené selekty

- optimalizátory nevedia čo robiť
- ak vnútorný selekt vráti 1 hodnotu **vek = SELECT max (rating) FROM**
- ak vráti viac riadkov **p.idp In(SELECT o.idp)** cez nested-loops
- ak sú prepojené - typický sa vypočíta vnútorný selekt pre každý riadok vonkajšieho selektu

15 Distribuované databázové systémy

- **Paralelný DBS**
  - paralelizovanie výpočtu vo viacerých jadrách - load, výroba indexov
  - napojených viac diskov
  - spoločná pamäť

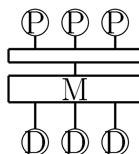
- **Distribuovaný DBS**
  - dáta sú uložené na viacerých uzloch z ktorých každý vie vykonávať samostatnú úlohu nezávisle na ostatných
  - lepšia dostupnosť (ak 1 PC nejde, ostatné môžu prebrať jeho úlohu)
  - veľké zvýšenie výkonu

Databázy sú jedny z najlepšie paralelizovateľných programov.

15.1 Architektúry DBS

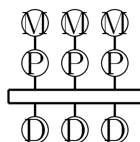
1. zdieľaná pamäť
   - komunikácia procesorov cez pamäť, netreba aby komunikovali medzi sebou
   - procesory však súperia o pamäť

zdieľaná pamäť



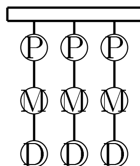
2. zdieľané disky

    - procesory súperia o zbernicu k disku a aj na komunikáciu medzi sebou



3. nič zdieľané

    - pridávanie nových strojov spôsobuje zrýchlenie aj lepsie škálovanie
    - zrýchlenie – čím viac strojov(uzlov), tým rýchlejšie
    - škálovanie – ak úmerne so zväčšovaním databázy pridávame stroje tak výkon ostáva



15.2 Typy distribuovaných databáz

1. homogénny DB systém

    - rovnaký software na každom stroji

2. heterogénny DB systém

    - rôzny DBMS (data base managment system)

    - používajú sa gateway protokoly JDBC, ODBC

- veľa optimalizácií sa robiť nedá (transakcie, fragmentácie...)

15.3 Základné požiadavky pre distribuovaný DBM

1. nezávislosť od distribúcie dát
   - v dopytoch nechceme špecifikovať kde sú uložené fragmenty tabuliek
   - nechceme špecifikovať optimálne počítanie dopytu
2. nedeliteľnosť transakcií
   - ak zmena úspešná tak ostáva vykonaná
   - ak sa zmena nepodarila tak na žiadnom uzle nesmú ostať čiastkové zmeny

15.4 Architektúry distribuovaných DBMS

1. klient - server
   - nespĺňa prvú základnú požiadavku
   - nevedia o sebe servre ani klienti
   - používa sa ak máme jeden silný server a veľa klientov
2. Middleware systémy
   - jeden stroj je manažér, obvykle neobsahuje žiadne DB dáta, rozdeľuje úlohy
   - použitie na heterogénne distribuované DB systémy
   - všetci klienti pristupujú iba k tomuto serveru
3. Spolupracujúce servre

- klient môže vyslať požiadavku na ľubovoľný server
- ak daný server nemá dáta lokálne, generuje sám požiadavku na ostatné servre

15.5 Ukladanie dát v distribuovanom DBMS

- veľké tabuľky sú obvykle fragmentované na viacerých serveroch pričom žiaden z nich nemá celú tabuľku Fragmentácia môže byť vertikálna alebo horizontálna.

15.5.1 Horizontálna

- dáta o zamestnancoch a podobne sa môžu ukladať v danej pobočke kde sa vytvorili a aj najviac používajú
- možné zrýchlenie joinov a selekcií

1. round - robin
   - ak máme n strojov tak i - tý záznam ide na (i mod n) - tý stroj
   - efektívne pre operácie vyžadujúce prechod celej tabuľky
2. hash
   - použijeme hashfunkciu, ktorá delí do n oblastí
   - ak je dopyt s podmienkou vek = 20 tak vieme na akom stroji sú dáta ak hashfunckia delila podľa veku.
3. rozsahové delenie
   - rozdelíme dáta podľa hodnôt nejakých atribútov tak, aby boli rozhodené rovnomerne
   - často veľmi ťažké rozhodiť rovnomerne, riešením môže byť že si to odskúšame na menšej vzorke náhodných dát.

15.5.2 Vertikálna

- zrýchlenie projekcií

Pri fragmentácii musíme jednoznačne vedieť spojiť fragmenty do 1 tabuľky. Malé tabuľky, ktoré sa často neupdatujú je praktické mať v niekoľkých kópiach na viacerých serveroch. Výhodou je lepšia dostupnosť dát a zrýchlenie výpočtu. Avšak pri update musí nastať zmena všade.

Dobrým delením môžeme dosiahnúť N^2 efekt

počet procesorov	počet uzlov	operácia	čas
1	1	30x30	900
3	1	(30x30)/3	300
	3	(10x10)*3/3	100
	30	(1x1)*30/30	1

V treťom a štvrtom prípade treba vykonať 3 resp. 30 takýchto operácií. Ale keďže máme 3 resp. 30 uzlov tak sa to opäť delí. Taktiež je tu podmienkou že príslušné tretiny tabuliek už musia byť rozhodené po uzloch s rovnakým horizontálnym delením.

15.6 Na čo si dať pri tvorbe databáz pozor - pre DB správcov

1. podľa akého kľúča deliť dáta - aby bolo rozdelenie rovnomerné, tak delenie podľa stĺpca, ktorý má oveľa väčší počet rôznych hodnôt ako je počet strojov kde chceme deliť

2. rozmiestnenie súvisiacich tabuliek - ak dáta nie sú dobre rozmiestnené tak je vysoký traffic. Pri joine tabuliek A a B je ideálne ak je potrebné vykonať iba lokálne čiastkové joiny bez prenosu dát cez komunikačný kanál. To sa dá zabezpečiť ak sú tabuľky delené podľa joinovacieho atribútu - maximalizuje sa N^2 efekt. Ak je joinovací atribút s malou doménou, treba deliť podľa viacerých atribútov.

15.7 Využitie redundancie

1. replikované tabuľky - všetky záznamy danej tabuľky sú na všetkých uzloch. N^2 efekt zaručený, treba však viac diskového priestoru. Vhodné pre malé tabuľky, ktoré sa často nemenia.

2. znovudelené materializované pohľady - ak máme tabuľku delenú podľa niektorých atribútov a materializovaný pohľad pomocou iných atribútov tej istej tabuľky, naviac môžeme niektoré stĺpce z pohľadu vynechať

3. replikovaé materializované pohľady - vyberieme iba stĺpce potrebné pre joiny. Spravíme view a pošleme ho na viac uzlov. Použiteľné hlavne na malé alebo stredne veľké tabuľky málo aktualizované.

4. znovudelené indexy - podobne ako znovudelené pohľady ale neuchovávame pohľady ale indexy, ktoré majú v sebe užitočné atribúty ako kľúč.

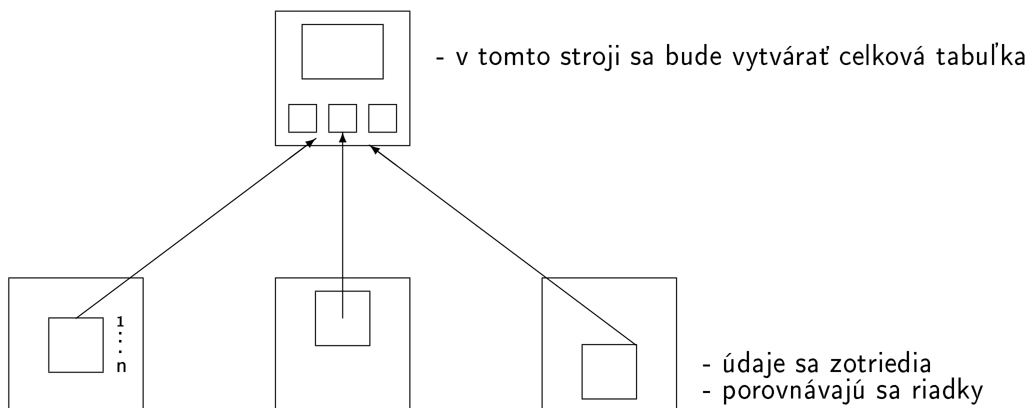
5. global join indexy - má v sebe smerníky aj na vzdialené uzly do pôvodnej tabuľky delenej podľa iného kľúča.

16 Paralelné výpočty

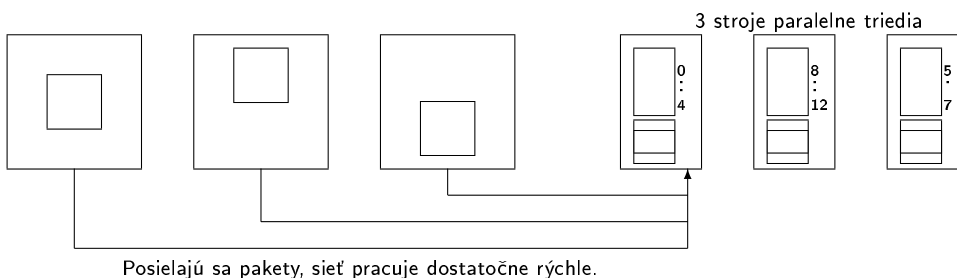
Pri paralelných výpočtoch narozdiel od distribuovaných nie je podstatná cena pri prenose.

16.1 Triedenie

Pri veľkom množstve tabuliek sa môže stať, že sa nezmestia do pamäte. Preto využijeme jeden z nasledujúcich spôsobov:
1. spôsob) - Výpočet sa deje v dvoch krokoch.

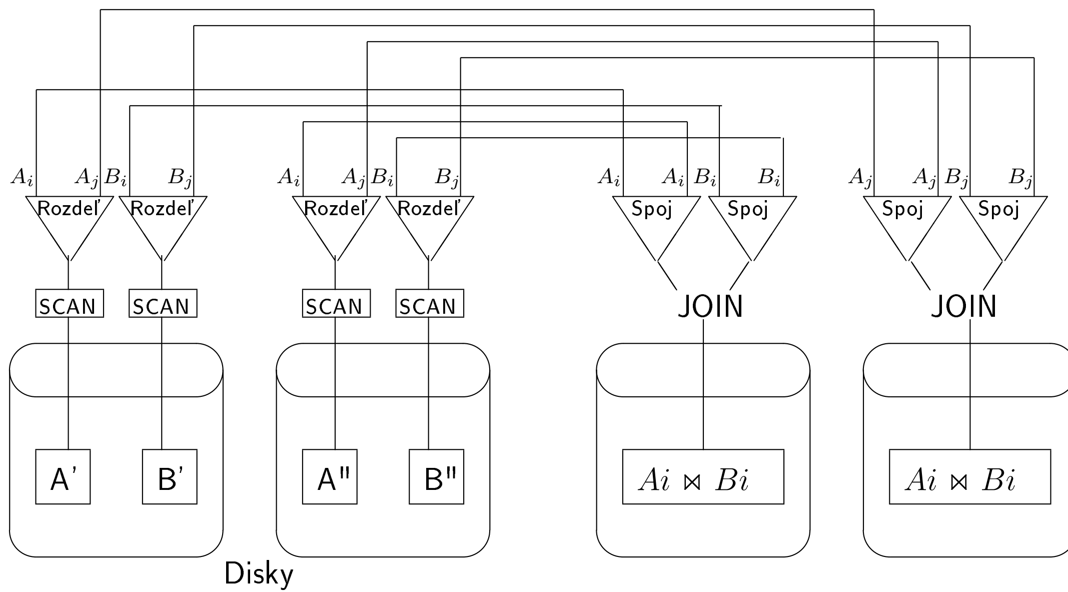


2. spôsob)
Njaprv sa robí rozsahové delenie, potom delenie dôjdených dát.



16.2 Paralelny hash JOIN

Hash hodnoty - hash funkcia musí byť na tých atribútoch, na ktorých ju joinujeme.



Predpokladajme, že máme tabuľku A rozdelenu horizontálne na A' a A" a tabuľku B na B' a B". Na riadky tabuľky aplikujeme hash funkciu h1. Ak hash hodnota je j tak zaznam pošleme na uzol, kde sa bude spracoavať join $Ai \bowtie Bi$, podobne aj hash hodnota je i tak zaznam pošleme na uzol, kde sa bude spracovavať join $Ai \bowtie Bi$.

16.3 Vylepšený paralelný hash join

Namiesto toho aby sme delili, povedzme na n oblasti, kde n je počet uzlov, tak delíme na viac ako n-oblasti: m $>$ n

1.krok: Na každom stroji aplikuj funkciu h1 na obe relácie A a B takú, aby menšia relácia vytvorila oblasti, z ktorých každá vojde do pamäte veľkosti súčtu pamäti na všetkých uzloch.

2.krok: Vezmi i-tu oblasť (nech A je menšia)

a) aplikuj h2 na Ai, ktorá určí ktorý záznam pôjde na ktorý uzol.
b) na každom uzle vytvor hash tabuľku v pamäti z dát, ktoré prišli.
c) aplikuj h2 na Bi a pošli príslušné záznamy na uzol, kde sa bude robiť join.
d) prichádzajúce záznamy z Bi joinuj s hash tabuľkou v pamäti. $Ai \bowtie Bi$

17 Distribuované výpočty

- záleží na vyššej cene prenosu po sieti.
- odpoveď sa sumarizuje na jednom mieste (uzol dopytu).
- ak je tabuľka rozdelená horizontálne, tak jedno-tabuľkové dopyty sa obvykle dajú vypočítať lokálne a výsledok sa sumarizuje. (najlepšie pre agregačné funkcie)
- ak máme tabuľku skopirvanú na viacerých uzloch, tak najlacnejší môže byť výpočet v uzle dopytu, alebo v uzle ktorý nie je vyťažený.

17.1 Distribuované joiny

Predpokladáme, že joinujeme dve tabuľky (**P**redavači * **O**bjednávky), pričom každá tabuľka je na inom uzle. (Predavači 500 stránok, Objednávky 1000 stránok). Budeme používať 2 čísla:

- t_d = čas V_0 na disku pre jednu stránku
- t_s = čas presunu stránky po sieti

Prvá metóda **Presun na prvý uzol:** Cenu 500 stránok prečítame z disku, tie prenesieme, potom robíme tabuľkový join. (predpokladáme 3(**P**+**O**))
$500*(2t_d + t_s) + 3(500 + 1000)t_d$=$5500t_d + 500t_s$

Algoritmus **Semijoin:** snaží sa znížiť hodnotu t_s v prvej metóde

1.) Vytvor projekciu na joinované atribúty z tabuľky Predavači(pid) a pošli projekciu na druhý uzol.

2.) Vytvor redukciu z tabuľky Objednávky iba na tie riadky, ktoré sa dajú spojiť joinom, a pošli túto redukciu po sieti.

3.) Vypočítaj join Predavačov s redukciou, ktorá prišla.

- **Cena:** potrebujeme 500 t_d na prečítanie, napr. 100 t_d na vzdialený zápis projekcie, ak to nie je *unique* stĺpec, tak ešte lokálny zápis 100 t_d, sort 400 t_d (2 prechodný) 100 t_d na elimináciu duplicít.
- Ak máme selekciu, ktorá nám oreže Predavačov na 20%, napr.σ_{rating}>8 tak:

  1.krok: vyžaduje (500+20)t_d v prípade, že je to unique alebo (500+20)t_d+(20+80+20)t_d+20t_s

  2.krok: povedzme, že redukcia je tiež 20%, teda zotriedime na 2 behy, t.j. 4*1000V0 a pošleme redukciu 1000t_d+200t_s

  3.krok: urobíme join 3*(500+200)t_d=2100t_d

  Dohromady máme 7940t_d+200t_s

Úvaha: Ak sú obe utriedené pred joinom tak:

1.) $500t_d + 20t_s + 20t_d$
2.) $1000t_d + 200t_s$
3.) $(500 + 200)t_d$
Dokopy to je $2220t_d + 220t_s$.

Bloom join:

- namiesto projekcie na joinované atribúty sa posiela bit-vektor veľkosti K, vytvorený počítaním hash funkcie z joinovaných atribútov do <0,k-

1>. Ak sa nejaká hodnota priradí číslu i <0,k-1>, tak v bit-vektore sa nastaví 1 na i-tom mieste. Inak 0.

- na druhom uzle sa spraví také isté počítanie podľa h. Ak niektorí riadok má hash hodnotu i a na i-tom mieste je nula, tak sa nezahrnie do redukcie.

Cena: opäť predpokladáme redukciu na 20%

1.) prechádzame Predavačov a generujeme bit-vektor $500t_d + \frac{k}{8*4096}t_s$ Predpokladajme, že x=2, potom k je dostatočne veľké, a tak je lepšia redukcia.

2.) prejdeme všetky objednávky a generujeme redukovanú tabuľku. (Predpokladáme chybu 10%) $1000t_d$+$220t_s$

3.) robíme join 3*(500+220)t_d=2160t_d

Dokopy to je $3660t_d + 240t_s$.

18 Rozloženie dát s optimálnym rozdelením

Metóda "All-beneficial Sites"

- umiestnime tabuľky a ich kópie na všetky uzly, kde ich umiestnenie je viac výhodné ako nevýhodné. (Benefit prevyšuje náklady).

$Benefit_{U,tab}$=(čas na dopyt vzdialeného uzla - čas na lokálny dopyt). Frekvencia dopytov na tabuľku tab z uzla U.

$Nklady_{U,tab}$=(čas na lokálny update + čas na vzdialený update). Frekvencia updatov.

Majme:

Tabuľka	Veľkosť	priem. čas dopytu	priem. čas vzdial. dopytu
Tab1	0,3 GB	100 (150)ms	500 (600)ms
Tab2	0,5 GB	150 (200)ms	650 (700)ms
Tab3	1GB	200 (250)ms	1000 (1100)ms

Transakcia	uzol dopytu	frekvencia	akcie
T1	S1, S4, S5	1	3*Read z Tab1, 1*Write do Tab1, 2*Read z Tab2
T2	S2, S4	2	2*Read z Tab1, 3*Read z Tab3 1*Write do Tab3
T3	S3, S5	3	3*Read z Tab2, 1*Write do Tab2, 2*Read z Tab3

Tabuľka: Cena a benefit pre každú tabuľku umiestnenú na piatich možných miestach

Table	Site	Remote Update (Local Update) Transaction	No. of Writes* Freq* Time (milliseconds)	Cost (milliseconds)
tab1	S1	T1 from S4 and S5 (T1 from S1)	2*1*600 ms+1*1*150 ms	1350 ms
	S2	T1 from S1, S4, S5	3*1*600ms	1800ms
	S3	T1 from S1, S4, S5	3*1*600ms	1800ms
	S4	T1 from S1 and S5 (T1 from S4)	2*1*600ms+1*1*150ms	1350ms
	S5	T1 from S1 and S4 (T1 from S5)	2*1*600ms+1*1*150ms	1350ms
tab2	S1	T3 from S3 and S5	2*3*700ms	4200ms
	S2	T3 from S3 and S5	2*3*700ms	4200
	S3	T3 from S5 (T3 from S3)	1*3*700ms+1*3*200ms	2700ms
	S4	T3 from S3 and S5	2*3*700ms	4200ms
	S5	T3 from S3 (T3 from S5)	1*3*700ms+1*3*200ms	2700ms
tab3	S1	T2 from S2 and S4	2*2*1100ms	4400ms
	S2	T2 from S2 and S4	1*2*1100ms+1*2*250ms	2700ms
	S3	T2 from S4 (T2 from S2)	2*2*1100ms	4400ms
	S4	T2 from S2 and S4	1*2*1100ms+1*2*250ms	2700ms
	S5	T2 from S2 and S4	2*2*1100ms	4400ms
Table	Site	Query (Read) Source	No. of Reads* Freq* Time (Remote-Local Time)	Benefit (milliseconds)
tab1	S1	T1 at S1	3*1*(500-100)	1200ms
	S2	T2 at S2	2*2*(500-100)	1600ms
	S3	None	0	0
	S4	T1 and T2 at S4	(3*1+2*2)*(500-100)	2800ms
	S5	T1 at S5	3*1*(500-100)	1200ms
tab2	S1	T1 at S1	2*1*(650-150)	1000ms
	S2	None	0	0
	S3	T3 at S3	3*3*(650-150)	4500ms
	S4	T1 at S4	2*1*(650-150)	1000ms
	S5	T1 at T3 at S5	(2*1+3*3)*(650-150)	5500ms
tab3	S1	None	0	0
	S2	T2 at S2	3*2*(1000-200)	4800ms
	S3	T3 at S3	2*3*(1000-200)	4800ms
	S4	T2 at S4	3*2*(1000-200)	4800ms
	S5	T3 at S5	2*3*(1000-200)	4800ms

Z toho nám vychádza, že benefity pre:

- tab1 presahujú náklady na S4.

- tab2 presahujú náklady na S3,S5

- tab3 presahujú náklady na S2, S3, S4, S5

Ak Benefity sú zhruba rovnaké ako Náklady tak sa oplatí tabuľku kopírovať aby sa zvýšila dostupnosť.
Ak sú Benefity všade horšie tak treba vybrať 1 uzol, kde je to najmenej zlé.