Téma 10 – Přístup k datům

Obsah

1. Dokončení SQL dotazů
2. Organizace ukládání dat
   - Záznamy pevné a proměnné délky
   - Sekvenční organizace souborů
   - Organizace "*multi-table clustering*"
3. Indexování
   - Podstata indexování
   - Husté a řídké indexy
   - B+ stromy a jejich vlastnosti
4. Hašování
   - Princip hašování
   - Statické hašování
   - Dynamické rozšiřitelné hašování

Sql View - Pohled

- Vytvoření pohledu na tabulku, nebo spojení tabulek

`create [ or replace ] view` *v* **`as`** <formulace dotazu>
kde *v* je jméno pohledu

- Vytvořený pohled se tváří jako samostatná relace – tabulka
- Pokud pohled neobsahuje agregátní funkce, GROUP BY, DISTINCT, některé druhy spojení a podmínek, pak je možné přes pohled upravovat, nebo vkládat do původní tabulky

Sql alter – změna schématu relace

- SQL příkaz ALTER umožní přidat, smazat, změnit formát sloupce

`alter table` *t* **`add`** *`sloupec typ`*

kde *t* je jméno tabulky, *sloupec* jméno nově přidávaného sloupce a jeho *typ*

`alter table` *t* **`drop column`** *`sloupec`*

kde *t* je jméno tabulky, *sloupec* jméno sloupce, který bude smazán

`alter table` *t* **`modify column`** *`sloupec typ`*

kde *t* je jméno tabulky, *sloupec* jméno měněného sloupce na jeho nový *typ* (Oracle 10G a více vypustili klíčové slovo **column**, MS Access má klíčové slovo **alter column**)

Uspořádání řádků tabulky

- Příkaz select umožňuje definovat uspořádání řádků tabulky dodatkem

select * from *t* **order by** *sloupec* [**asc**|**desc**], *sloupec2* [**asc**|**desc**]

kde *t* je jméno tabulky, *sloupec*, *sloupec2* jméno sloupce/ů podle kterých se bude řadit, **asc** vzustupně, **desc** sestupně. Pokud se neuvede, je použito **asc**.

- Příkaz select umožní vybrat také jen prvních několik řádků

select * from *t* **limit** *počet*

kde *t* je jméno tabulky, *počet* je kolik řádků tabulky nás zajímá.
MS Access - **select top** *počet* *** from** *t*
Oracle - ***select * from*** *t* ***where*** **ROWNUM<=***počet*

Organizace ukládání dat

Databáze a soubory

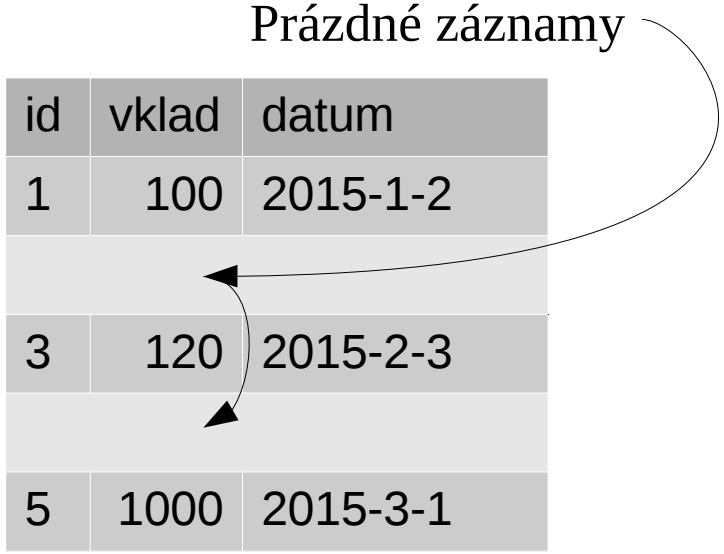
- Databáze je uložena jako soustava souborů (*files*). Každý soubor je posloupnost záznamů (*records*). Záznam je posloupnost polí (*fields*), obvykle odpovídajících atributům
- Základní přístup
  - Předpoklad: každý záznam má pevnou délku
  - Každý soubor obsahuje záznamy jediného konkrétního typu
  - Každá datová relace je v samostatném souboru

  Snadná implementace
- Záznamy pevné délky *m* bytů
  - *i*-tý záznam začíná na pozici $m * (i - 1)$
  - Přístup k záznamům je velmi jednoduchý
    - Viz služba **`lseek`** pro práci se soubory
- Výmaz záznamu *i* – alternativy:
  - $i+1, ..., n$ na $i, ..., n-1$
  - přesuň záznam *n* na místo *i*-tého
  - nepřesouvej nic, ale udržuj seznam smazaných záznamů, tj. volného místa

Seznamy volného místa

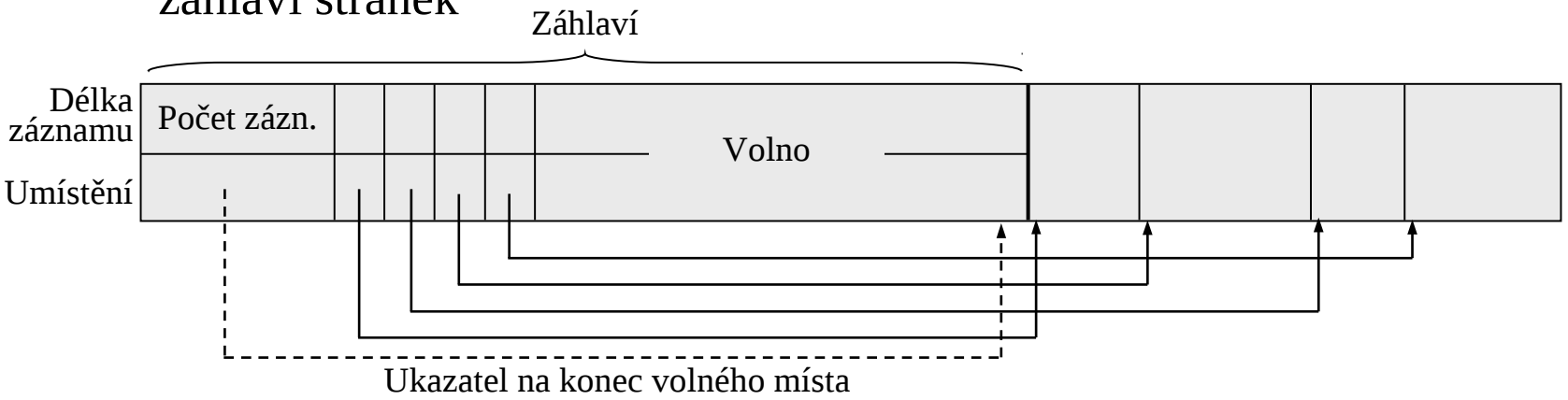
- Ulož adresu prvního smazaného záznamu v záhlaví souboru
- Použij místo po zrušeném záznamu k uložení adresy druhého smazaného záznamu, atd.
  - Uložené adresy jsou odkazy (*pointery*) – odkazují na příští zrušený záznam
- Při vkládání záznamů se použije místo po zrušených záznamech

id	vklad	datum
1	100	2015-1-2
2	150	2015-1-10
3	120	2015-2-3
4	250	2015-2-10
5	1000	2015-3-1



Záznamy s proměnnou délkou

- Záznamy s proměnnou délkou se používají zřídka. Mohou vzniknout při potřebě
  - uložení různých záznamů v jednom souboru
  - uložení záznamů s poli (atributy) proměnné délky
- Stránkované soubory
  - Soubor = množina stránek (bloků) fixní velikosti
  - Záhlaví stránky obsahuje
    - Počet záznamů ve stránce
    - Odkaz na konec volného místa
    - Pozici a délku každého záznamu
    - Záznamy lze uvnitř stránky libovolně přesouvat a šetřit tak místo
    - Odkazy na záznamy v souboru jsou tvořeny dvojicemi (*číslo_stránky*, *číslo_záznamu_ve_stránce*). Neodkazují se přímo záznamy, ale údaje v záhlaví stránek



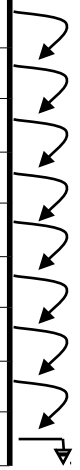
Organizace záznamů v souborech

- Umísťování záznamů může výrazně ovlivnit přístupové doby
  - zpravidla se vyhledává prostřednictvím klíče (asociativní paměť)
- Halda – záznam se umístí v souboru kamkoliv, kde je místo
- Sekvenční organizace – záznamy se ukládají v pořadí daném hodnotou vyhledávacího klíče záznamu
- Hašování – hašovací funkce počítaná z vhodného atributu záznamu (klíče) určí číslo bloku v souboru, kam bude záznam umístěn
- Záznamy každé relace se zpravidla ukládají do samostatných souborů. V organizaci označované jako "*multi-table clustering*" se záznamy různých relací ukládají do jednoho souboru
  - Motivace: ukládej logicky spolu související záznamy do jednoho bloku s cílem minimalizace I/O operací

Sekvenční organizace souboru

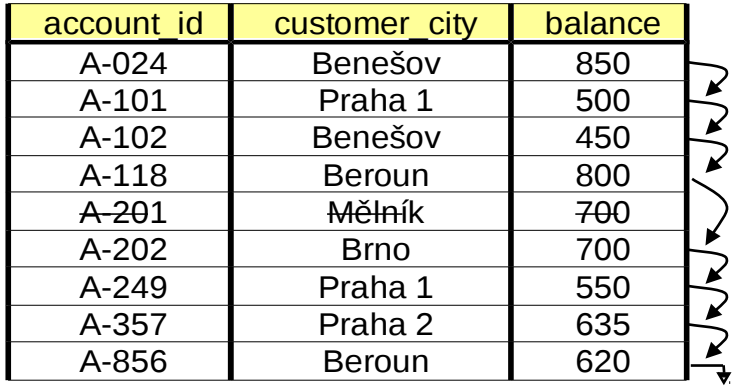
- Velmi vhodné pro aplikace, kdy se požaduje sekvenční zpracování celé tabulky
- Záznamy jsou řazeny podle vyhledávacího klíče
  - Uspořádané sekvenční soubory umožňují velmi rychlý přístup k datům
    - „Půlení intervalu“ – logaritmická (sub-lineární) složitost
  - Údržba je avšak extrémně obtížná
    - Proto je obvykle ke každému záznamu připojen odkaz na "logicky následující" záznam

account_id	customer_city	balance
A-024	Benešov	850
A-101	Praha 1	500
A-102	Benešov	450
A-118	Beroun	800
A-201	Mělník	700
A-202	Brno	700
A-249	Praha 6	550
A-357	Praha 2	635
A-856	Beroun	620



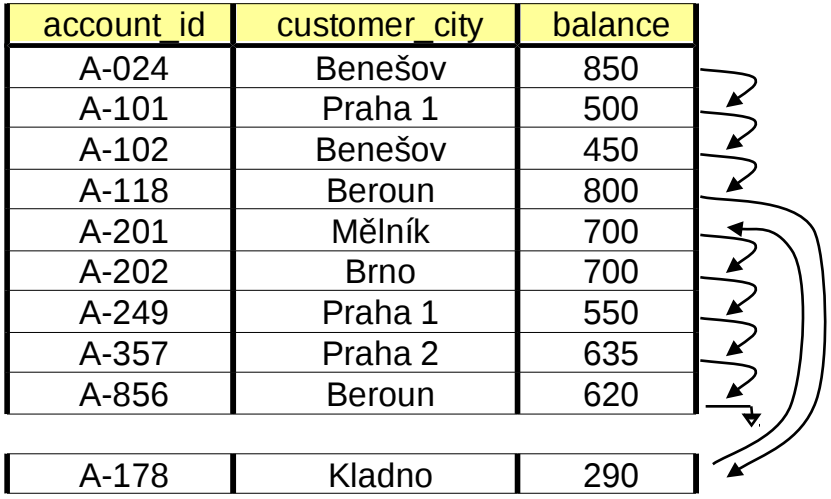
Sekvenční organizace souboru (pokr.)

- Výmaz
  - řetězec odkazů přeskočí smazaný záznam



account_id	customer_city	balance
A-024	Benešov	850
A-101	Praha 1	500
A-102	Benešov	450
A-118	Beroun	800
~~A-201~~	~~Mělník~~	~~700~~
A-202	Brno	700
A-249	Praha 1	550
A-357	Praha 2	635
A-856	Beroun	620

- Vložení
  - najdi pozici, kam záznam vložit
  - je-li tam volno, vlož ho tam
  - pokud ne, vlož záznam do tzv. bloku přeplnění
  - V každém případě je nutno aktualizovat řetězec odkazů



account_id	customer_city	balance
A-024	Benešov	850
A-101	Praha 1	500
A-102	Benešov	450
A-118	Beroun	800
A-201	Mělník	700
A-202	Brno	700
A-249	Praha 1	550
A-357	Praha 2	635
A-856	Beroun	620
A-178	Kladno	290

- Reorganizace souboru
  - Čas od času se musí soubor reorganizovat, aby se obnovilo plné sekvenční uspořádání, a tak bylo používat efektivní (sub-lineární) hledání

Organizace "*multi-table clustering*"

- Uložení několika relací v jednom souboru s organizací *multi-table clustering*
  - Relace *depositor* a *customer*

customer_name	account_id
Vrba	A-101
Vrba	A-202
Vrba	A-301
Votava	A-102

customer_name	customer_street	customer_city
Vrba	Spálená 20	Mělník
Novák	U vody 15	Beroun

  uložené v jednom souboru
  - Vhodné pro dotazy typu *depositor* ⋈ *customer* a dotazy týkající se jednoho zákazníka a jeho účtů
  - Neefektivní, když se dotaz bude týkat jen zákazníka

Vrba	Spálená 20	Mělník
Vrba	A-101	
Vrba	A-202	
Vrba	A-301	
Novák	U vody 15	Beroun
Novák	A-102	

- Vede na záznamy s proměnnou délkou
- Lze též přidat řetězce odkazů spojujících záznamy patřící k jedné ze zobrazovaných relací

Uložení slovníku dat

- Slovník dat (*data dictionary*, *system catalog*) obsahuje údaje o datech (*metadata*)
  - Informace o relacích
    - jména relací, jména a typy atributů každé relace, integritní omezení
    - jména a definice pohledů
  - Informace o právech uživatelů, včetně zakódovaných hesel
  - Údaje o fyzické organizaci souborů
    - Jak je která relace uložena (sekvenčně/hašovaná/...)
    - Fyzická lokalizace relace (v kterém souboru, ...)
    - Údaje o indexech a dalších strukturách ➔
- Struktura slovníku dat
  - Relační reprezentace na disku
  - Možná reprezentace soustavou relací (tabulek) podle schémat:

Relation_metadata = (*relation_name, number_of_attributes, storage_organization, location*)
Attribute_metadata = (*relation_name, attribute_name, domain_type, position, length*)
User_metadata = (*user_name, encrypted_password, group, ACL*)
View_metadata = (*view_name, definition*)
Index_metadata = (*index_name, relation_name, index_type, index_attributes*)

Indexace a indexní soubory

Základní myšlenka indexace

- Indexy zrychlují hledání v datech podle klíče
  - např. autorský index v knihovně
- Indexní soubor sestává ze záznamů ve tvaru

- Indexní soubory bývají *výrazně menší* než původní soubory s daty
  - mnohdy se celé vejdou do operační paměti
- Dva základní typy indexů:
  - Setříděné indexy: soubor je setříděn podle klíčů
  - Hašované indexy: klíče jsou rovnoměrně rozloženy v "blocích" referencovaných hašovací funkcí
- Měřítka pro hodnocení efektivní organizace indexů
  - Podporované typy vyhledávání, např.
    - záznamy s konkrétní hodnotou klíče (či jiného atributu) *versus* záznamy s hodnotou klíče v zadaném intervalu hodnot
  - Doba přístupu
  - Náročnost operací vkládání a výmazu dat
  - Prostorová (paměťová) náročnost (režie)

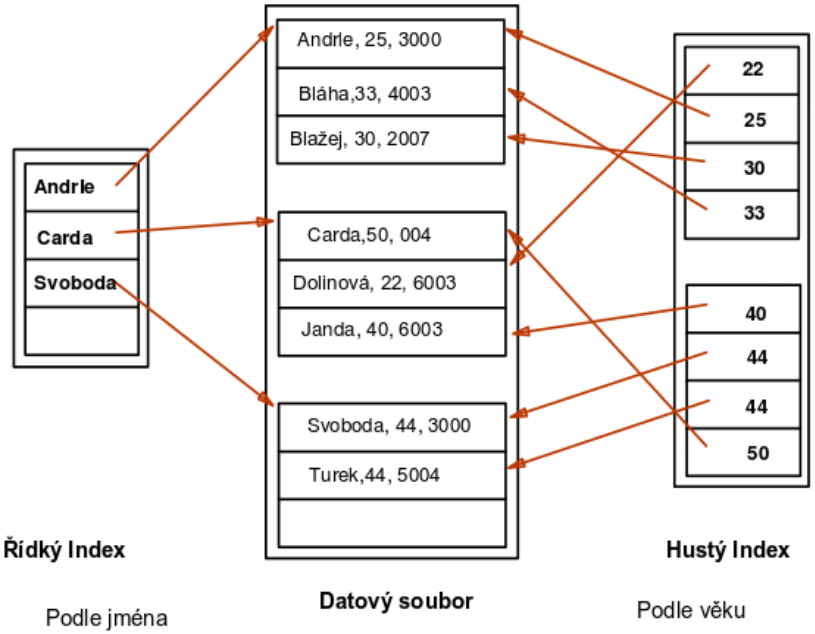
Setříděné indexy

- Při setříděných indexech jsou položky indexního souboru setříděny a uloženy podle hodnot prohledávacího klíče
  - Např. autorský katalog v knihovně
  - Index může být prohledáván iterovaným půlením
- Primární index: index určující pořadí záznamů v sekvenčně organizovaném souboru
  - Prohledávací klíč primárního indexu je obvykle (i když ne nutně) primárním klíčem relace
    - POZOR: Nejde-li o primární klíč relace, může k jedné hodnotě prohledávacího klíče existovat více datových záznamů
- Sekundární index(y) určují pořadí jiné, než je určeno primárním indexem
  - Vhodné pro setříděné výpisy či prohledávání odlišné od primárního indexu
- Index-sekvenční soubor označuje setříděný datový soubor s primárním indexem a vše je spojeno do jediného diskového souboru

Husté a řídké indexy

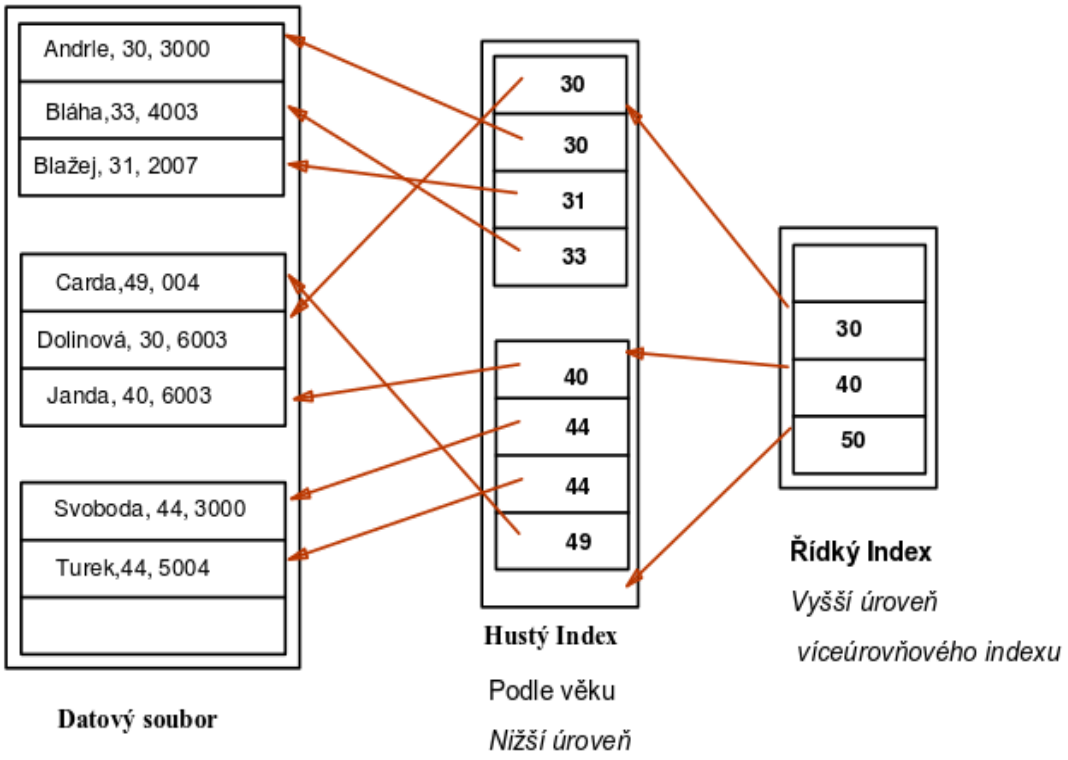
- Hustý index obsahuje záznamy pro všechny hodnoty prohledávacího klíče

- Řídké indexy obsahují odkazy pouze na některé hodnoty klíčů
  - Použitelné jen když datové záznamy jsou setříděn podle stejného klíče
- Řídké vs. husté indexy
  - Menší a méně režie spojené s rušením a vkládáním záznamů
  - Obecně pomalejší než husté indexy při vyhledávání
  - Výhodné a nejčastější užití: řídký index s položkami odkazujícími na první záznam podle klíče v každém alokačním bloku datového souboru



Víceúrovňové indexy

- Pokud se primární index nevejde do paměti, vyhledávání začne být nákladné
- Řešení:
  - považuj primární index uložený na disku za sekvenční soubor a vybuduj k němu řídký index
  - vnější index = řídký index primárního indexu
  - vnitřní index = primární index k datům
  - Je-li i pak vnější index příliš velký, vytvoř další úroveň, atd.
  - Problém: Při vkládání či mazání záznamů je nutno aktualizovat všechny nadřazené úrovně indexů



Aktualizace indexů

- Mazání záznamů
  - Pokud mazaný záznam je jediný záznam s danou hodnotou prohledávacího klíče, pak je nutno smazat i příslušnou indexní položku
  - Výmaz při jednoúrovňových indexech:
    - Husté indexy – analogické výmazu datového záznamu
    - Řídké indexy
      - jestliže hodnota prohledávacího klíče mazaného záznamu v řídkém indexu je, dojde k vymazání náhradou této hodnoty další hodnotou klíče
      - jestliže další hodnota prohledávacího klíče již v řídkém indexu existuje, pak místo náhrady jen klíč vymaž
- Vložení záznamu
  - Jednoúrovňové indexy:
    - Pomocí stávajícího indexu najdi, kam má vkládaný záznam přijít
    - Husté indexy – není-li vkládaná hodnota prohledávacího klíče v indexu, vlož ji
    - Řídké indexy (případ, kdy index odkazuje první záznam v alok. bloku)
      - je-li v indexu položka pro blok, kam se nový záznam umístí, není nutná žádná úprava, pokud se nevytváří další blok
      - je-li nutno vytvořit další blok, pak první hodnota v novém bloku se musí vložit do indexu
- Rušení a vkládání záznamů při víceúrovňových indexech je rekurzívním zobecněním jednoúrovňových operací

Sekundární indexy

- Často chceme nalézt všechny záznamy, jejichž hodnota určitého atributu (ne nutně primárního klíče) splňuje danou podmínku
  - Příklad 1: V relaci *account* uložené sekvenčně podle čísla účtu chceme najít všechny účty v dané pobočce
  - Příklad 2: stejně jako v předešle, avšak navíc chceme najít jen účty s určitým konkrétním zůstatkem či dokonce se zůstatkem v zadaném rozmezí
- Můžeme budovat sekundární indexy s položkami pro jiné než primární klíče
  - Sekundární indexy musí být husté (*Proč?*)
  - Indexní záznam ukazuje na skupinu obsahující reference na všechny záznamy s určitou hodnotou sekundárního prohledávacího klíče

Primární a sekundární indexy – vlastnosti

- Indexy přinášejí jasné výhody při hledání záznamů podle hodnot prohledávacích klíčů
- **Avšak**: Aktualizace indexů působí režijní náklady při aktualizaci databází
  - jakmile je změněn obsah datového souboru, musí být modifikovány **všechny** indexy přidružené k těmto datům
- Sekvenční prohlížení podle primárního indexu je velmi efektivní, ale prohlížení dle sekundárního indexu je mnohem nákladnější
  - každý přístup k záznamu může znamenat nahrání nového alokačního bloku souboru z disku
  - nahrání bloku potřebuje řádově milisekundy
    - oproti řádově 10 – 100 ns při přístupu do hlavní paměti

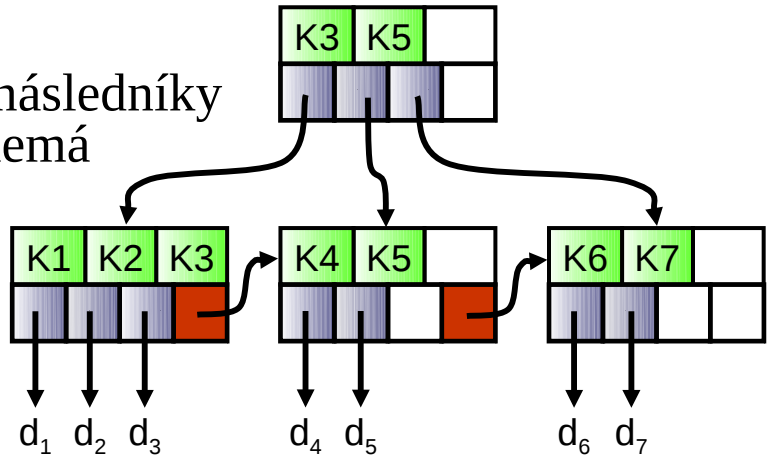
Indexní soubory s B+ stromy

Indexy s B+ stromy jsou alternativou k index-sekvenčním souborům

- Nevýhody index-sekvenčních souborů
  - Jak soubory rostou, degraduje se jejich účinnost – příliš mnoho bloků přeplnění => nutnost periodické reorganizace souborů
- Výhoda indexů s B+ stromy
  - Automatická reorganizace při malých, lokálních, změnách způsobených vkládáním a rušením jednotlivých záznamů
  - Reorganizace celého souboru za účelem zachování výkonnosti není nutná
- (Menší) nevýhoda B+stromů
  - vyšší režie při vkládání a rušení záznamů
  - větší paměťové (prostorové) nároky
- Demonstrace B+ stromů
  - https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html
  - http://slady.net/java/bt/view.php

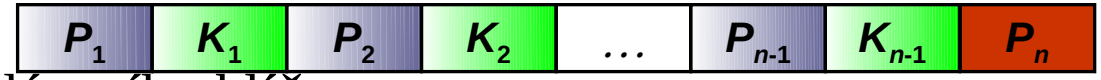
Indexy s B+ stromy

- B+ strom je stromová datová struktura reprezentující uspořádaná data a respektující existenci bloků dat
  - Umožňuje efektivní vyvolávání, vkládání a rušení záznamů identifikovaných klíčem
  - Jde o dynamický víceúrovňový stromově organizovaný index, přičemž existují omezení na maximální a minimální počet klíčů v každém uzlu stromu
- B+ strom má následující vlastnosti
  - Všechny cesty od kořene k listu stromu jsou stejně dlouhé
  - Každý vnitřní uzel stromu má mezi $\lceil n/2 \rceil$ a n následníky, kde n se nazývá řád stromu (též faktor větvení)
    - n závisí na velikosti klíče a velikosti alokačního bloku souboru; $n \approx 100$
  - List stromu obsahuje $\lceil (n-1)/2 \rceil$ až $n-1$ hodnot klíčů
  - Výjimky:
    - Pokud kořen není listem, má aspoň 2 následníky
    - Je-li kořen zároveň listem (tj., strom nemá vnitřní uzly), může obsahovat 0 až $(n-1)$ hodnot



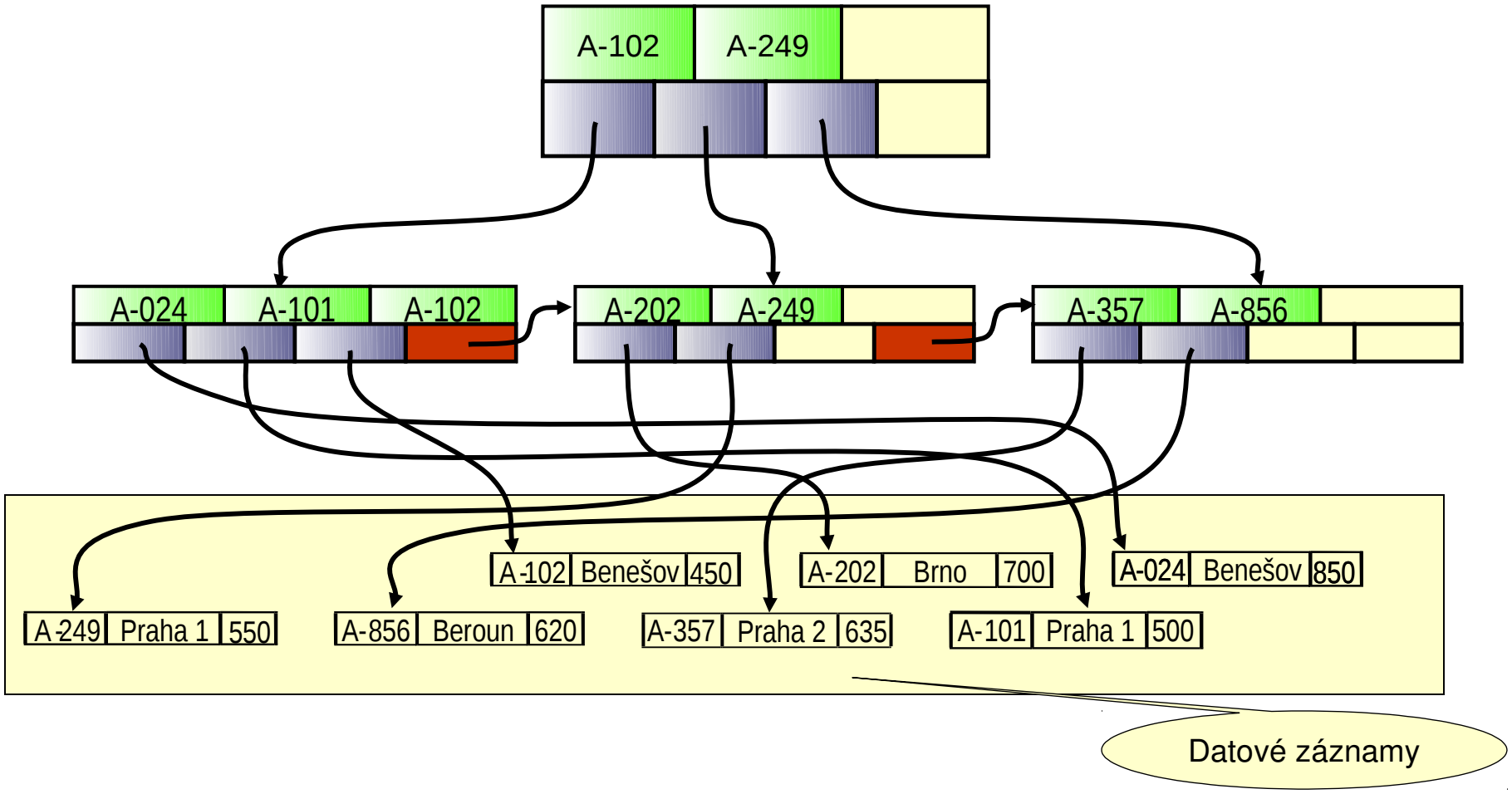
B+ strom při n=3

Struktura uzlů B+ stromu



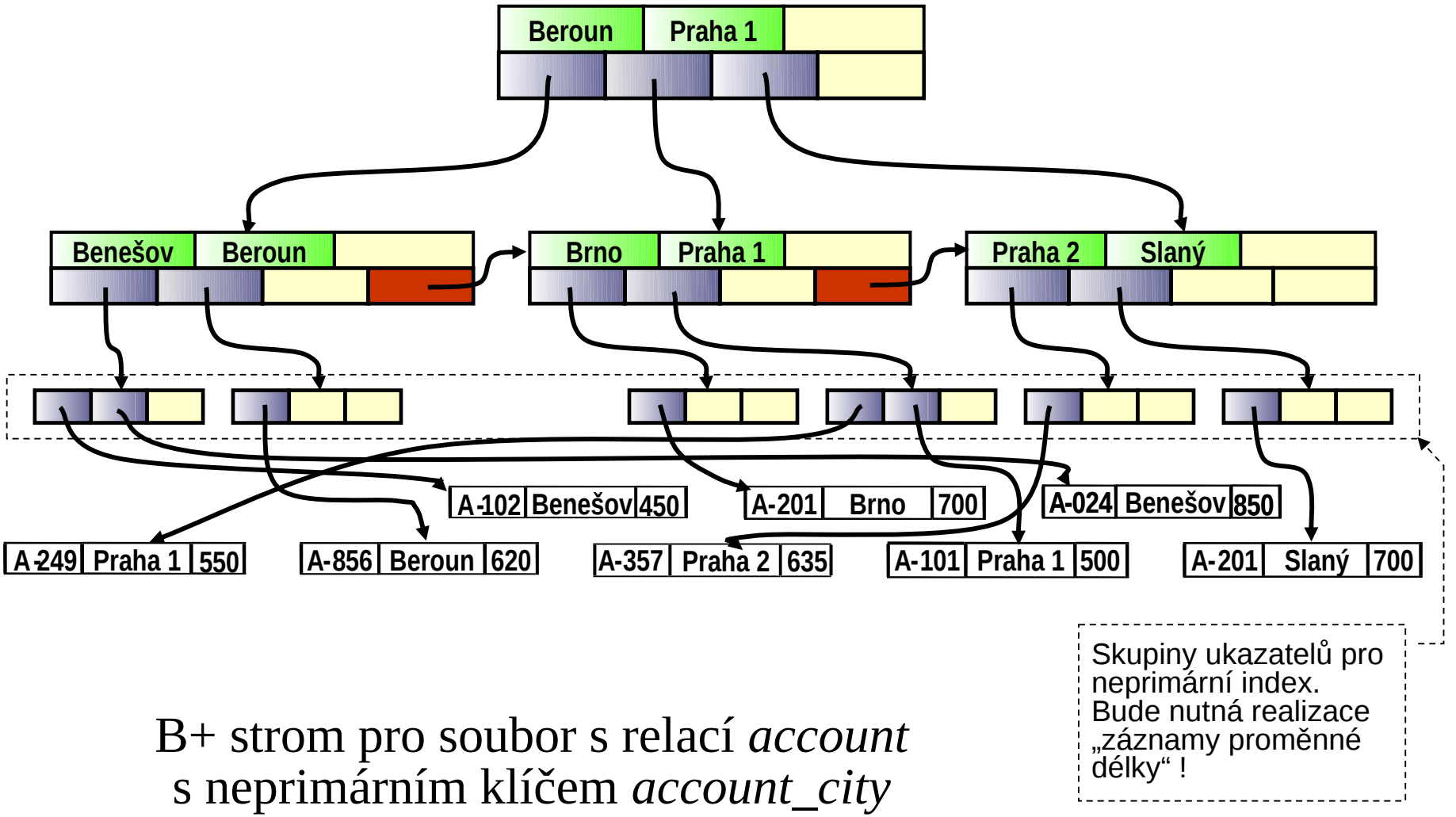
- Typický uzel stromu
  - K_i jsou hodnoty prohledávacího klíče
  - P_i jsou odkazy (ukazatele) na následníky (pro nelistové uzly) nebo odkazy na datové záznamy (či jejich skupiny) pro listy
  - Klíče v uzlu jsou uspořádány $K_1 < K_2 < K_3 < \ldots < K_{n-1}$
- Nelistové uzly tvoří víceúrovňový řídký index listových uzlů. Pro nelistové uzly s *m* odkazy platí,
  - že podstrom odkazovaný P_1 obsahuje klíče s hodnotou $\leq K_1$,
  - pro $2 \leq i \leq n-1$ platí, že podstromy odkazované P_i mají klíče s hodnotami λ, kde $K_{i-1} \leq \lambda < K_i$, a
  - uzel referencovaný P_n obsahuje hodnoty $> K_{n-1}$
- Pro listy platí
  - Pro $i = 1, 2, \ldots, n-1$, ukazatel P_i buď odkazuje přímo datový záznam s klíčem K_i, nebo odkazuje skupinu dalších ukazatelů na více datových záznamů s klíčem K_i.
    - Druhý případ je nutný, pokud nejde o primární klíč, kdy v datech je více záznamů se stejným prohledávacím klíčem
  - V listech L_i a L_j při $i < j$ platí, klíče v L_i jsou menší než klíče v L_j
  - P_n ukazuje na následující list
    - To umožňuje uspořádané výpisy bez procházení stromu

Příklad B+ stromu pro primární index



B+ strom pro soubor s relací *account* s primárním klíčem *account_id* při $n = 3$

Příklad B+ stromu pro neprimární index



B+ strom pro soubor s relací *account*
s neprimárním klíčem *account_city*
při $n = 3$

Poznatky o B+ stromech

- Protože vazby mezi uzly jsou realizovány ukazateli, logicky blízké bloky nemusí být blízké fyzicky
  - oddělení logiky od implementace
    - může ale snížit efektivitu
- Nelistové uzly tvoří hierarchii řídkých indexů
- B+ stromy jsou poměrně "mělké"
  - mají malý počet úrovní a od kořene k listu vede „krátká cesta“
    - V úrovni pod kořenem je minimálně 2 * ⌈n/2⌉ hodnot klíčů
    - Další úroveň obsahuje aspoň 2 * ⌈n/2⌉ * ⌈n/2⌉ hodnot
    - ... atd.
  - Je-li v datovém souboru *K* hodnot prohledávacího klíče, pak hloubka stromu není větší než $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$
    - pro K=1.000.000 a n=100 je hloubka maximálně $log_{50}(1.000.000) = 4$
- Vkládání a výmazy záznamů z datového souboru jsou relativně rychlé a mohou být prováděny v logaritmickém čase ➔

Prohledávání B+ stromů

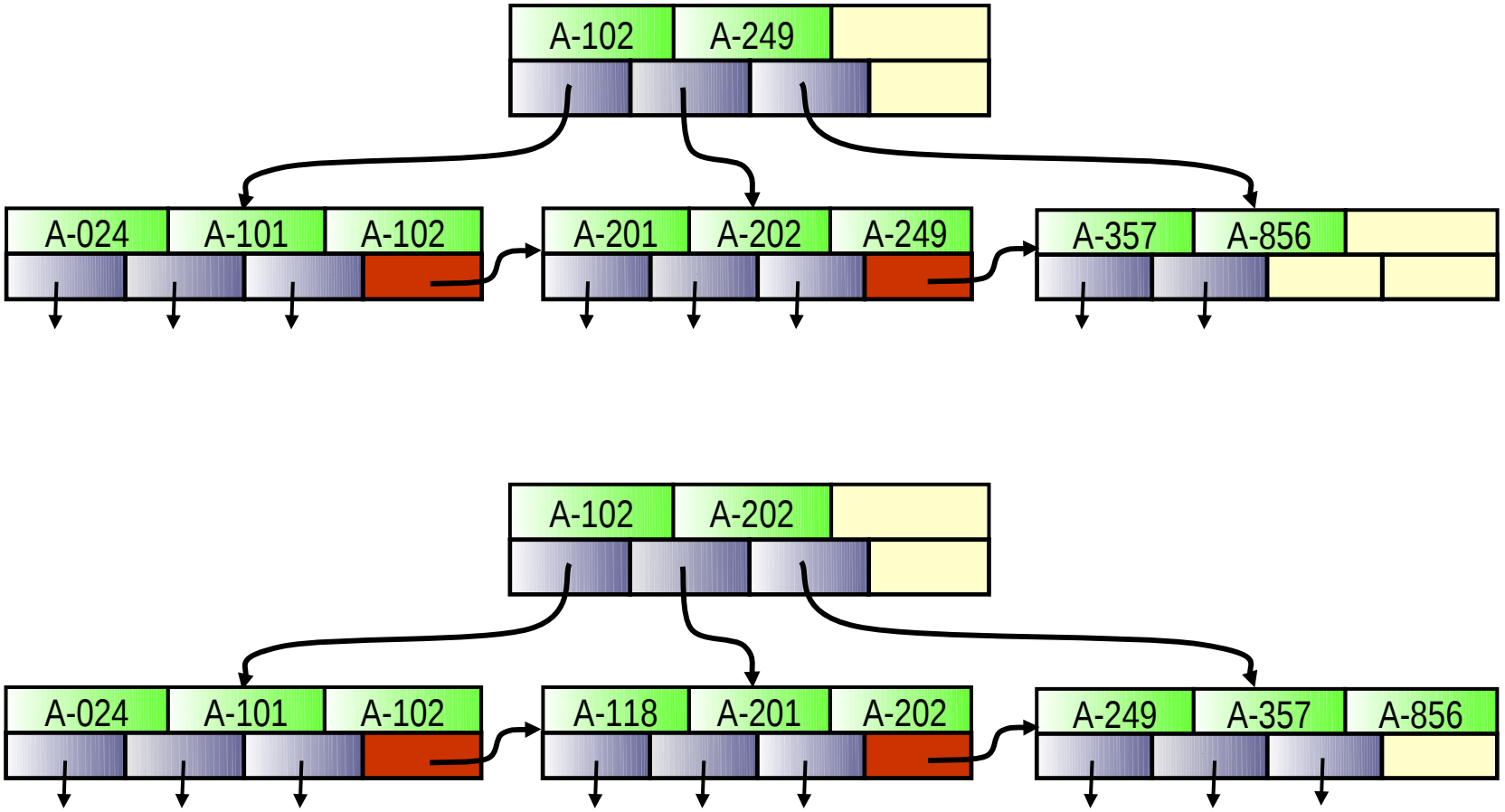
- Najdi všechny záznamy s hodnotou klíče = *k*
  1. *Node* ← *root*
  2. **`repeat`**
     1. Hledej v *Node* nejmenší klíč $K_i > k$
     2. Pokud taková hodnota v *Node* existuje *Node* ← P_i [krok ve stromu dolů]
     3. Jinak $k \geq K_{n-1}$ a pak *Node* ← P_n [krok ve stromu "vodorovně"]

     `until` *Node* je list stromu
  3. Prohledej list stromu
     1. Pokud pro nějaké *i* platí $K_i = k$, použij ukazatel P_i a přejdi na hledaný záznam nebo skupinu záznamů
     2. V opačném případě záznam s klíčem *k* neexistuje
- Velikost uzlu je obvykle shodná s velikostí bloku souboru
  - typicky 4 KiB a *n* je obvykle okolo 100 (≈40 bytů na položku)
  - Je-li hodnot klíče 1 milión a *n* = 100, pak
    1. do paměti budou nahrány nejvýše $log_{50}(1.000.000) = 4$ uzly (bloky) a budou prohledány
    2. Pokud by byly použity klasické vyvážené ***binární*** stromy, potřebovali bychom binární strom s hloubkou 20 a tedy 20 přístupů na disk
    3. A každý přístup na disk je v řádu ≈ 10 ms

Vkládání do B+ stromů

- Základní algoritmus
  1. Najdi list, kam by vkládaný záznam měl patřit
  2. Pokud hodnota klíče už v listu existuje
     1. jde-li o index k primárnímu klíči => CHYBA
     2. přidej záznam do dat a je-li to třeba, přidej ukazatel do skupiny
  3. Neexistuje-li hodnota klíče v listu, pak
     1. přidej záznam do dat a případně vytvoř skupinu ukazatelů
     2. Je-li v listu místo, přidej pár (***klíč***, ***ukazatel***)
     3. Není-li místo, vzniká problém: Uzel je třeba rozdělit na dva
- Rozdělení listového uzlu
  - Vezmi *n* párů (klíč, ukazatel), včetně nově vkládaného, a setřiď je. Prvních $\lceil n/2 \rceil$ vlož do původního uzlu a zbytek umísti do bloku tvořícího nový uzel
  - Nechť ukazatel na nový uzel je *p* a *k* je nejmenší klíč v novém uzlu. Vlož (*k*, *p*) do uzlu nadřazeného rozdělovanému listu
  - Je-li nadřazený uzel plný, je nutno propagovat rozdělování směrem ke kořeni, dokud není nalezen uzel, v němž je volno
    - V nejhorším případě je nutno rozdělit kořenový uzel a zvětšit hloubku stromu o 1

Úprava B+-stromu: příklad vložení

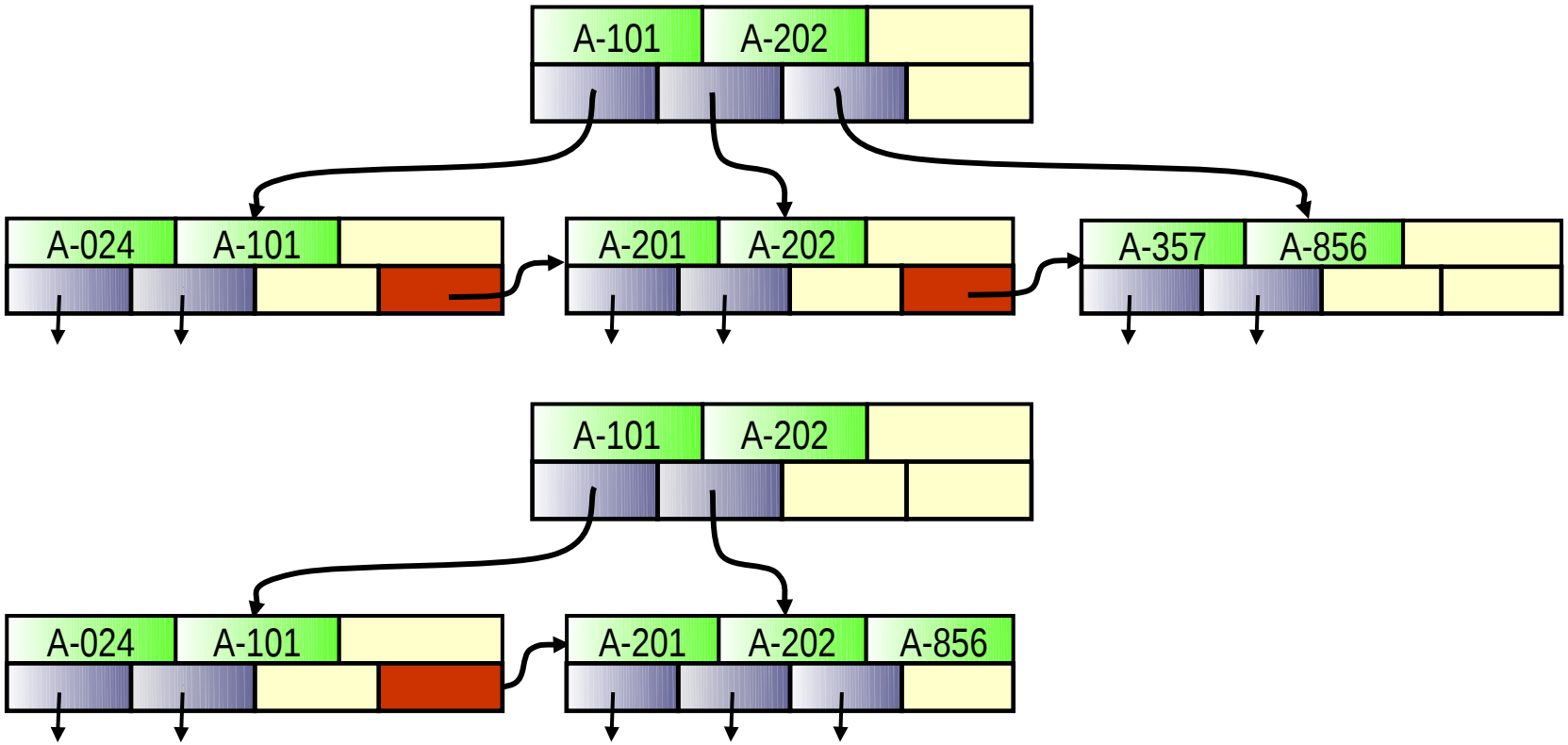


B+ strom před a po vložení záznamu s klíčem “A-118”

Výmazy z B+-stromů

- Najdi rušený záznam
- Vymaž ho z datového souboru a popř. ze skupiny ukazatelů
- Jestliže skupina ukazatelů je prázdná (nebo vůbec neexistuje), odstraň pár (klíč, ukazatel) z listového uzlu
- Má-li list po odstranění páru příliš málo položek a jeho "horizontální" soused jich také nemá dostatek, je nutno spojit dva "horizontálně" sousední uzly v jeden:
  - Vlož páry z obou uzlů do prvního a zruš druhý uzel
  - Vymaž pár (K_{i-1}, P_i), kde P_i je ukazatel na smazaný uzel v nadřazeném uzlu; rekurzívně aplikuj tuto proceduru směrem ke kořeni
- Má-li list po odstranění páru příliš málo položek a jeho "horizontální" soused jich má dost, pak přerozděl položky v sousedních uzlech
  - Přerozděl položky tak, aby oba uzly měly přibližně stejně položek
  - Aktualizuj odpovídající klíč v nadřazeném uzlu
- Rušení uzlů může kaskádně propagovat, dokud není nalezen uzel mající aspoň $\lceil n/2 \rceil$ položek
  - V krajním případě se zlikviduje celý strom a zbude samotný kořen neobsahující žádnou položku (prázdný index)

Úprava B+-stromu: příklad výmazu

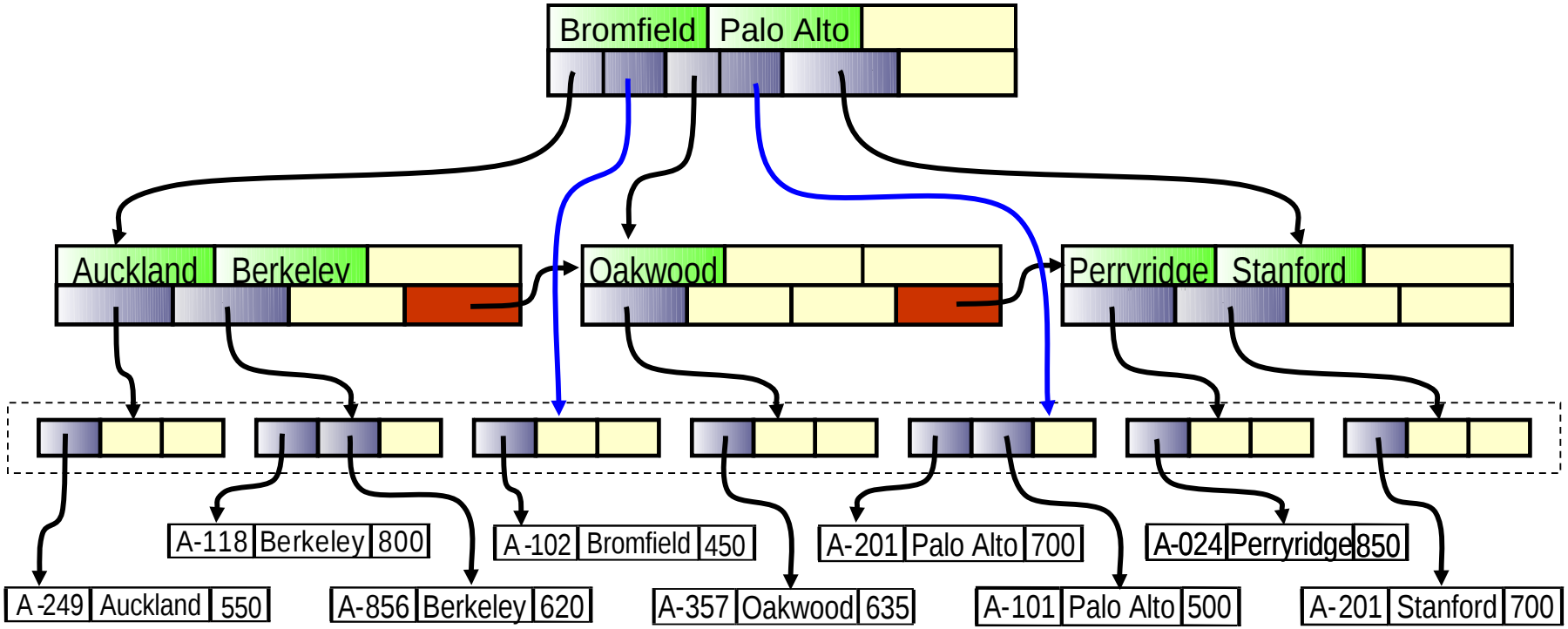


B+-strom před a po zrušení záznamu s klíčem “A-357”

Indexy s B-stromy

- B-stromy jsou podobné B+-stromům, avšak hodnoty klíčů se neopakují
  - eliminace redundantního ukládání klíčů
  - Hodnoty klíče, které se nacházejí v nelistových uzlech, už se nevyskytují nikde níže v podstromech
  - V nelistových uzlech je však nutno přidat další ukazatele
- Výhody B-stromových indexů:
  - Mohou potřebovat méně uzlů než odpovídající B+-strom
  - Občas lze najít záznam bez nutnosti projít až do listu
- Nevýhody B-stromových indexů :
  - Položky nelistových uzlů jsou větší, takže uzel má méně následníků, což může způsobit nutnost větší hloubky stromu
  - Aktualizace při vkládání a mazání záznamů jsou komplikovanější než u B+-stromů
- Typicky, výhody B-stromů nepřevažují nad nevýhodami

Příklad indexu s B-stromem



B-strom pro soubor s relací *account* (n=3)

Přístup přes několik klíčů

- Příklad:

  select *account_number*
  from *account*
  where *branch_name* = “Benešov” **and** *balance* = 1000

  – Možné strategie s použitím klíčů dle jednotlivých atributů
    1. Použij index pro *branch_name* a najdi záznamy pro "Benešov"; pak testuj na *balance* = 1000
    2. Použij index pro *balance* k nalezení účtů se zůstatkem 1000; pak testuj *branch_name* = "Benešov".
    3. Použij index pro *branch_name* k nalezení ukazatelů na záznamy z benešovské pobočky. Podobně použij index pro *balance*. Závěrem urči průnik obou množin získaných ukazatelů

- Indexy podle vícenásobných klíčů
  – Složené prohledávací klíče jsou klíče tvořené více než jedním atributem
    - Např. (*branch_name*, *balance*)
  – U složených klíčů je problém s porovnáváním a řazením
  – Nejčastější je lexikografické řazení: $(a_1, a_2) < (b_1, b_2)$, pokud
    - $a_1 < b_1$, nebo
    - $a_1 = b_1$ a $a_2 < b_2$

Hašování

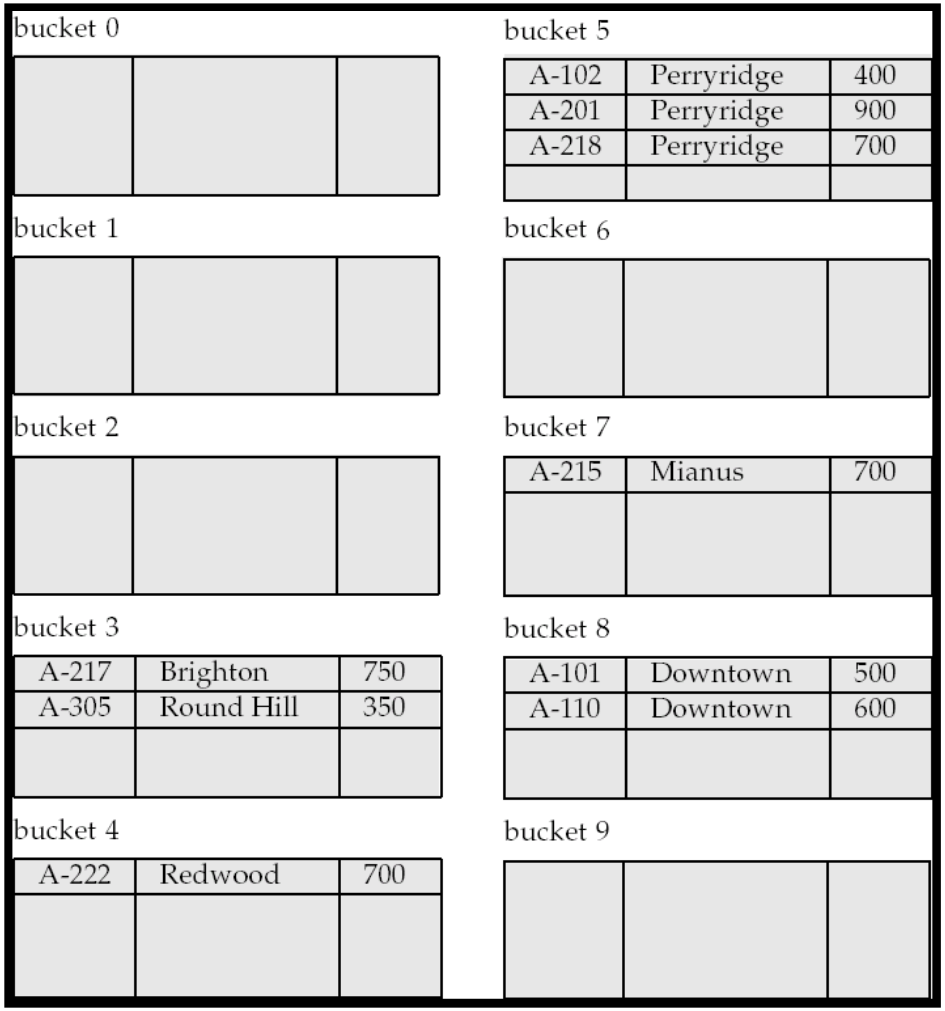
Statické hašování

- Sekce (*bucket* = kýbl, koreček) je paměťová jednotka obsahující jeden či několik záznamů
  - typicky je to jeden alokační blok diskového souboru
- Při hašované organizaci souboru získáme číslo sekce obsahující záznam s daným klíčem přímo jako funkční hodnotu hašovací funkce
- Hašovací funkce *h* je funkcí nad množinou hodnot prohledávacího klíče

  $B = h(k)$, kde k je hodnota klíče a B je adresa (číslo) sekce
- Hašovací funkce se užívá k lokalizaci záznamů při vyvolávání, vkládání i mazání záznamů
- Záznamy s různými klíči mohou být hašovací funkcí mapovány do téže sekce, která se pak musí prohledávat sekvenčně

Příklad hašované organizace souboru

- Soubor *account* s klíčem *branch_name*
  - Máme 10 sekcí
  - Binární reprezentace znaku je chápána jako celé číslo
  - Hašovací funkce vrací jako svoji funkční hodnotu součet binárních reprezentací znaků modulo 10
    - Např.
      h(Perryridge) = 5
      h(Round Hill) = 3
      h(Brighton) = 3

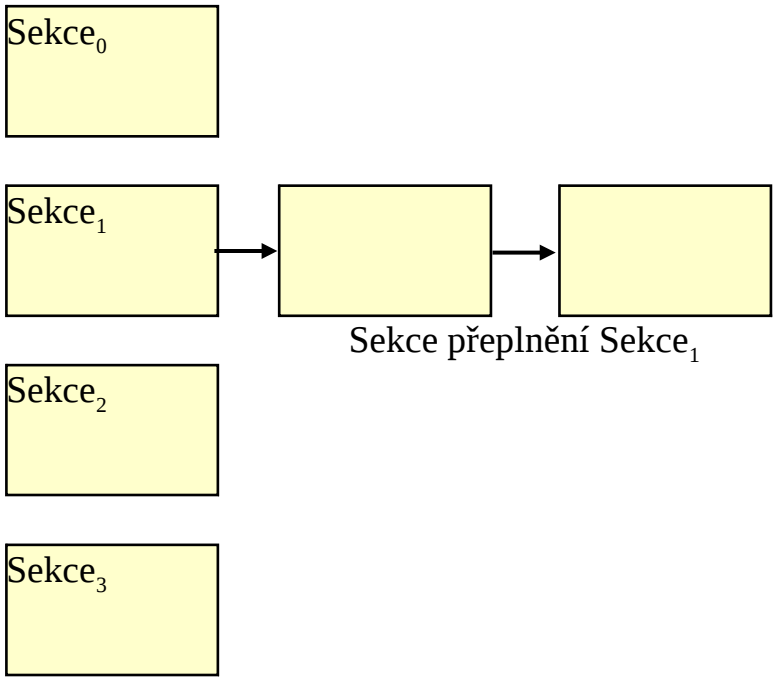


Hašovací funkce

- Nejhorší hašovací funkce mapuje všechny hodnoty klíče do téže sekce
  - nemá žádný přínos – vše se prohledává sekvenčně
- Ideální hašovací funkce rozděluje klíče rovnoměrně
  - Všem sekcím je přiřazen stejný počet záznamů pro *všechny možné* hodnoty klíče
  - Ideální hašovací funkce by měla být náhodná, protože každá sekce má obsahovat přibližně stejný počet záznamů bez ohledu na aktuální rozložení hodnot klíče v souboru
  - Náhodná funkce je však nereprodukovatelná a tedy nepoužitelná
- Typické hašovací funkce počítají s vnitřní binární reprezentací klíčů
  - Viz předchozí příklad

Přeplnění sekcí

- Přeplnění sekcí vzniká kvůli
  - nedostatečnému počtu sekcí
  - nerovnoměrnosti v distribuci záznamů. To má dva důvody:
    - více záznamů má stejné hodnoty klíče
    - zvolená hašovací funkce nerovnoměrně mapuje klíče
- Přeplnění sekcí lze redukovat, nikoliv eliminovat
  - Řeší se pomocí sekcí přeplnění
  - Je-li zapotřebí více sekcí, sekce přeplnění dané „hlavní“ sekce jsou zřetězeny



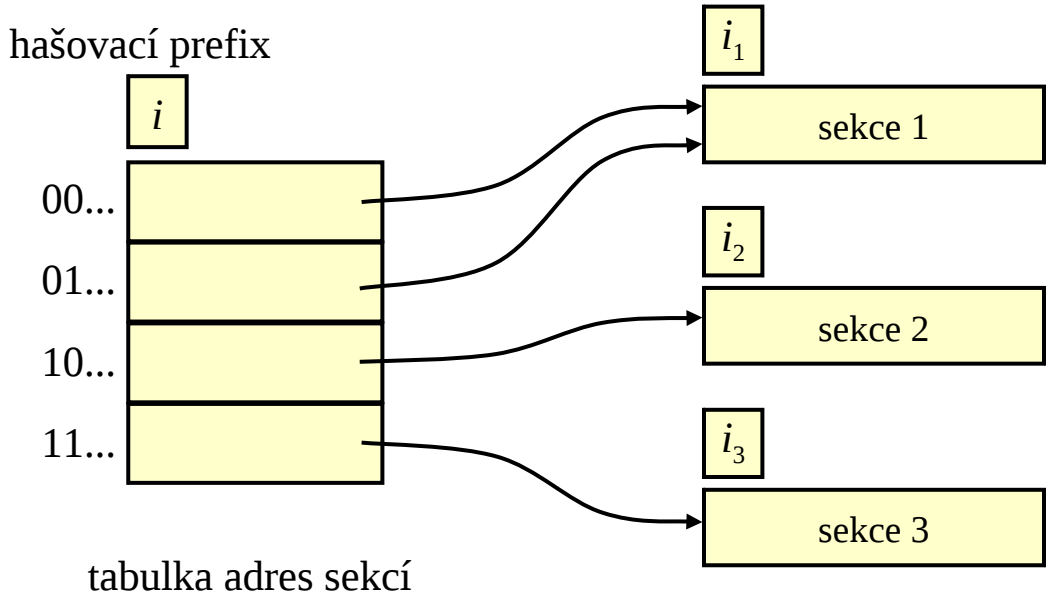
Nedostatky statického hašování

- Statická hašovací funkce *h* mapuje hodnoty klíče do pevné množiny čísel (adres) sekcí. Databáze v čase rostou nebo se zmenšují
  - Když je na počátku zvolen malý počet sekcí a soubor roste, účinnost rychle klesne kvůli velkému množství přeplnění
  - Je-li alokován velký prostor s ohledem na očekávaný růst, mrhá se kapacitou paměti a sekce jsou nevyužité
  - Když se databáze zmenší, opět se bude plýtvat prostorem
- Potenciálně lze čas od času soubor reorganizovat za použití jiné hašovací funkce
  - Nákladné a narušuje normální operace s databází
    - po dobu reorganizace nelze k datům přistupovat
- Lepší řešení: počet sekcí se bude měnit dynamicky podle potřeby

Dynamické hašování

- Vhodné pro databázové soubory, jejichž velikost se výrazně mění
- Umožňuje dynamickou modifikaci hašovací funkce
- Rozšiřitelné (*extendible*) hašování – jedna z forem dynamického hašování
  - Hašovací funkce generuje hodnoty ve velkém rozsahu – typicky *b*-bitová celá čísla (uvažujme $b = 32$)
  - Pouze několik bitů zleva (prefix) se používá jako index do tabulky s čísly (adresami) sekcí
  - Nechť délka prefixu je i bitů, $0 \leq i \leq 32$.
    - Tabulka adres sekcí má délku = 2^i. Na počátku $i = 0$
    - Hodnota i roste a klesá podle velikosti databázového souboru
  - Několik položek v tabulce adres může ukazovat na tutéž sekci
  - Tudíž skutečný počet sekcí je $< 2^i$
    - Počet sekcí se též dynamicky mění v důsledku spojování či rozdělování sekcí

Obecná struktura rozšiřitelného hašování



- *j*-tá položka tabulky adres sekcí obsahuje hodnotu i_j
  - Všechny položky tabulky ukazující na tutéž sekci mají shodných prvních i_j bitů
  - K nalezení sekce obsahující záznam s daným klíčem K_j je třeba
    - Určit $X = h(K_j)$ a použít prvních i bitů X jako index do tabulky adres sekcí a následovat ukazatel na příslušnou sekci

Užití rozšiřitelného hašování – vkládání

- Při vkládání záznamu s klíčem K_j
  - Najdi sekci *j*, kam má záznam přijít
  - Je-li tam místo, vlož záznam; jinak sekce musí být rozdělena a vložení se opakuje
- Rozdělení sekce *j* při vkládání záznamu s klíčem K_j:
  - Pokud $i > i_j$ (více než jeden ukazatel na sekci *j*)
    - alokuj novou sekci *z* a nastav $i_j = i_z = (i_j + 1)$
    - Aktualizuj druhou část položky v tabulce adres sekcí, původně ukazující na *j* tak, aby ukazovala na *z*
    - vyjmi záznamy ze sekce *j* a vlož je zpět (do *j* nebo *z*)
    - urči znovu adresu sekce pro K_j a vlož záznam
  - Když $i = i_j$ (jen jeden ukazatel na sekci *j*)
    - Inkrementuj *i* a zdvojnásob velikost tabulky adres sekcí
    - Nahraď každou položku v tabulce dvěma položkami ukazujícími na tutéž sekci
    - Nyní $i > i_j$, takže se užije předchozí přístup

Rozšiřitelné hašování – jednoduchý příklad

- Kapacita sekce = 1 záznam
- První dva záznamy s klíči k_1 a k_2

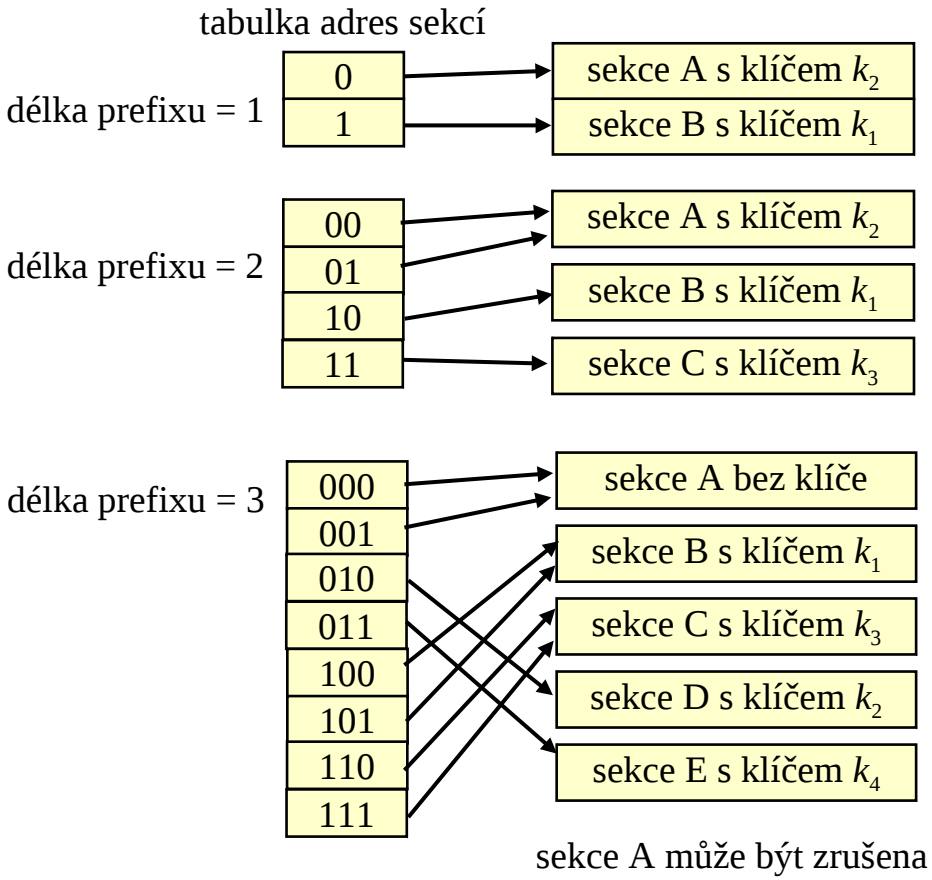
  $h(k_1) = 100100$

  $h(k_2) = 010110$
- Třetí záznam s k_3 dává

  $h(k_3) = 110110$
- Záznam 4 s k_4 má

  $h(k_4) = 011110$,

  což způsobí, že se sekce A přeplní a musí být rozdělena na A a D. Pokus o znovu-vložení záznamu s k_4 do D způsobí opět přeplnění a D se dále dělí na D a E za současného přidání dalšího bitu k prefixu



Užití rozšiřitelného hašování – výmaz

- Výmaz záznamu s daným klíčem
  - najdi záznam v sekci a vymaž ho
  - Je-li sekce prázdná, zruš ji (za současné aktualizace tabulky adres)
  - Sekce mohou být spojeny v jednu
    - Lze spojit jen sekce mající stejné i_j a současně existuje prefix i_j–1
  - Lze též zmenšit tabulku adres sekcí
    - Je to ale náročná a drahá operace a má smysl ji dělat jen když se počet sekcí výrazně zmenší anebo narážíme na problémy s kapacitou paměťového média

Porovnání indexace a hašování

- Jednoznačné výhody nejsou u žádného ze způsobů. Vše závisí na
  - ceně periodické reorganizace souborů a indexů
  - relativní frekvenci vkládání a rušení záznamů
  - Je důležité optimalizovat průměrnou dobu přístupu i za cenu podstatného prodloužení přístupové doby v nejhorším případě?
- Očekávané typy dotazů:
  - Hašování je obecně lepší při vyvolávání záznamů s přesnou hodnotou klíče
  - Jsou-li časté dotazy na interval hodnot klíče, pak indexace je výhodnější
- Praxe:
  - PostgreSQL podporuje hašované indexy, avšak jejich použití se nedoporučuje kvůli špatné efektivitě
  - Oracle podporuje statické hašování, ale ne hašované indexy
  - Microsoft SQLServer podporuje jen B+ stromy
  - ...

Cena SQL dotazu

Cena dotazu

- Cena dotazu je určena časem, jak dlouho trvá dotaz vyřídit
  - Čas dotazu určuje mnoho faktorů
    - Doba čtení dat z disku, vytížení a rychlost CPU, rychlost síťové komunikace pro distribuované databáze
- Čtení a zápis na disk je nejdůležitější údaj, který ovlivňuje rychlost zpracování dotazu (také je relativně snadné odhadnout počet přenesených dat).
  - Počet hledání v souboru - seeks * průměrná-cena-hledání
  - Počet přečtených bloků * průměrná-cena-čtení-bloku
  - Počat zapsaných bloků * průměrná-cena-zápisu-bloku
    - Cena zápisu je větší než cena čtení, nevíc některé systémy kontrolují, zda bylo zapsáno to co mělo (zápis + čtení)

Určení ceny dotazu

- Pro jednoduchost budeme uvažovat v ceně SQL dotazu pouze *počet transferů bloku dat z disku a počet vyhledávání - seek*
  - t_T – čas pro přesun bloku dat do paměti
  - t_S – čas pro vyhledání dat
  - Cena b přesunů dat a S hledání je
    $$b * t_T + S * t_S$$
- Pro jednoduchost zanedbáme cenu CPU
  - Realné SŘBD zohledňují i odhad ceny CPU
- Pro jednoduchost atké zanedbáváme cenu zápisu výstupních dat
- Rychlost přístupu na disk se zlepší využitím vyrovnávacích bufferů
  - Velikost bufferu je dána volným místem v paměti při vykonávání SQL dotazu
  - Požadovaná data mohou být k dispozici ve vyrovnávací paměti, ale to je velmi složité zohlednit při odhadu ceny SQL dotazu

Cena operace selekce

- **File scan** – vyhledávací algoritmus, který prochází soubor a hledá záznamy, které splňují zadanou podmínku.
- **A1** (*lineární prohledávání*). Postupné procházení všech n-tic relace a porovnávání, zda splňují zadanou podmínku.
  - Odhad ceny = b_r přesunů + 1 vyhledávání
    - b_r označuje počet bloků dat, které obsahují všechny záznamy z prohledávané tabulky *r*
    - 1 – vyhledávání, neboť bloky jsou řazeny za sebou
  - Pokud je podmínka rovnost primárního klíče a soubor je podle něj uspořádán
    - Průmerná cena = (b_r/2) přesunů + 1 vyhledávání
  - Lineární prohledávání může být použito bez ohledu na:
    - typ zadané podmínky
    - uspořádání záznamů v souboru
    - existenci indexů
-

Cena operace selekce

- **A2** *(binární vyhledávání).* - je použitelné pouze tehdy pokud je podmínka rovnost na atribut, který určuje uspořádání souboru.
  - Předpokládejme, že bloky jsou uloženy sekvenčně
  - Cena dotazu (Počet bloků, které se musí projít):
    - Cena dotazu $\lceil \log_2(b_r) \rceil * (t_T + t_S)$
    - Pokud více záznamů odpovídá hledané hodnotě je potřeba přidat cenu transferu bloku dat obsahující odpovídající záznamy

- **Prohledávání s indexy –** vyhledávací algoritmy využívající indexy
  - Podmínky musí být založena na atributech z indexu.
- **A3** (*primární index, rovnost pro klíč*). Cena nalezení jednoho záznamu pomocí indexu
  - $Cost = (h_i + 1) * (t_T + t_S)$

Cena operace selekce

A4 (*primární index, rovnost atributu, který není klíčem) Rozdíl oproti A3 je nutnost získat více záznamů.*

- Předpokládejme, že záznamy budou v souvislých blocích dat, b = počet bloků obsahujících hledaná data
- Cena = $h_i * (t_T + t_S) + t_S + t_T * b$

- **A5** (*rovnost na vyhledávacím klíči sekundárního indexu).*
  - Pokud je atribut kandidátním klíčem – existuje pouze jeden hledaný záznam
    - Cena = $(h_i + 1) * (t_T + t_S)$
  - Pokud atribut není kandidátním klíčem – může existovat až n záznamů
    - Cena = $(h_i + n) * (t_T + t_S)$
      - Může být velmi náročně!

Selekce na nerovnost

- Podmínka ve tvaru $\sigma_{A \leq V}(r)$ nebo $\sigma_{A \geq V}(r)$ lze použít
  - lineární vyhledávání
  - binární vyhledávání, pokud jsou záznamy setříděny,
  - nebo využít indexů:
- **A6** (*primární index, porovnání).* - předpokládáme, ře tabulka je setříděna podle primárního klíče
  - Pro $\sigma_{A \geq V}(r)$ použít index k nalezení prvního záznamu splňujícího $\geq v$ a dále pak sekvenční kopírování záznamů na výstup
  - Pro $\sigma_{A \leq V}(r)$ sekvenční kopírování záznamů dokud platí podmínka $\leq v$; *není potřeba index*
- **A7** (*sekundární index, porovnání*).
  - Pro $\sigma_{A \geq V}(r)$ použít index pro nalezení prvního záznamu, který splňuje podmínku $\geq v$ a pak sekvenčně procházet index a kopírovat záznamy z odkazů indexu.
  - Pro $\sigma_{A \leq V}(r)$ procházet od začátku index a kopírovat záznamy z odkazů dokud nenastane záznam $> v$
  - V každém případě pro kopírování záznamů je nutná I/O pro každý záznam

Spojování tabulek - Join

- Spojování tabulek je základní operace databázového systému.
- Existují následující algoritmy
  - Vnořené cykly
  - Vnořené bloky
  - Vnořený cyklus s indexem
  - Merge-join
  - Hash-join
- Výběr konkrétního algoritmu na základě porovnání cen algoritmů
- Pro následující příklad, předpokládejme
  - *Počet zázkazníků je*: 10,000 *účtů*: 5000
  - Počet bloků zákazníků je: 400 *účtů*: 100

Vnořené cykly

- Pro výpočet spojení $r \bowtie_\theta s$

  for *všechny n-tice* t_r **z** r
  for *všechny m-tice* t_s **z** s
  testuj (t_r,t_s) zda splňuje θ
  pokud ano zařaď $t_r \bullet t_s$ do výsledné tabulky.
  end
  end
- r se nazývá vnější relace a s vnitřní relace spojení.
- Nepoužívají se indexy a lze spojovat dvě tabulky na základě libovolné podmínky
- Velmi, velmi náročná, zkoumá všechny dvojice záznamů.

Cena vnořených cyklů

- V nejhorším případě, pokud paměť umožňuje uložit pouze jeden blok dat z relace t a jeden blok data z relace s

$$(n_r * b_s + b_r) * t_T + (n_r + b_r) * t_S$$

- Pokud se menší tabulka vejde do paměti celá, pak je nutné tuto tabulku použít jako vnitřní tabulku
  - Sníží to cenu na $(b_s + b_r) * t_T + 2 * t_S$
- Při nejhorším předpokladu s omezenou pamětí
  - Pokud je účet vnější relace:
    - 5000 ∗ 400 + 100 = 2,000,100 přesunů,
    - 5000 + 100 = 5100 vyhledávání
  - Pokud je zákazník vnější relace
    - 10000 ∗ 100 + 400 = 1,000,400 přesunů
    - 10000 + 400 = 10,400 vyhledávání
- Pokud se menší relace (účet) vejde do paměti celá pak je cena spojení:
  500 přesunů
  2 vyhledávání
- Lepší algoritmus jsou vnořené bloky.

Algoritmus vnořených bloků

- Varianta vnořených cyklů, kdy se využijí všechny dvojice z již načtených bloků

for každý blok B_r **z** r
  for každý blok B_s **z** s
    for každou n-tici t_r **z** B_r
      for každou m-tici t_s **z** B_s
        Pokud (t_r,t_s) *splňuje přidej* $t_r \bullet t_s$ *do výsledku*
      end
    end
  end
end

Cena alg. vnořených bloků

- Nejhorší případ při nedostatku paměti:
  $b_r * b_s + b_r$ block transfers + 2 * b_r seeks
  - Každý blok vnitřní relace s je přečten pro každý blok vnější relace
- Pokud se vnitřní relace vejde do paměti, pak stejné jako u vnořených cyklů:
  $b_r + b_s$ block transfers + 2 seeks.
- Při nejhorším předpokladu s omezenou pamětí
  - Pokud je účet vnější relace:
    - 100 ∗ 400 + 100 = 40,100 přesunů,
    - 2 * 100 = 200 vyhledávání
  - Pokud je zákazník vnější relace
    - 400 ∗ 100 + 400 = 40,400 přesunů
    - 2*400 = 800 vyhledávání

Algoritmus vnořených bloků

- Vylepšení algoritmu vnořených cyklů i vnořených bloků:
  - Pro algoritmus vnořených bloků: pokud je k dispozici vyrovnávací paměť pro *M* bloků
    - Cena = $\lceil b_r / M \rceil * b_s + b_r$ přesunů + $2 \lceil b_r / M \rceil$ vyhledávání
    - Pro náš příklad při vyrovnávací paměti 10 bloků, snížení počtu přesunů z 40,100 na 4,100.
  - Pokud je slučováno podle rovnosti na atribut/y tvořící kandidátní klíč vnitřní relace, pak lze vnitřní cyklus zastavit při nalezení shody
  - Použít index na vnitřní relaci, pokud je k dispozici

Indexed Nested-Loop Join

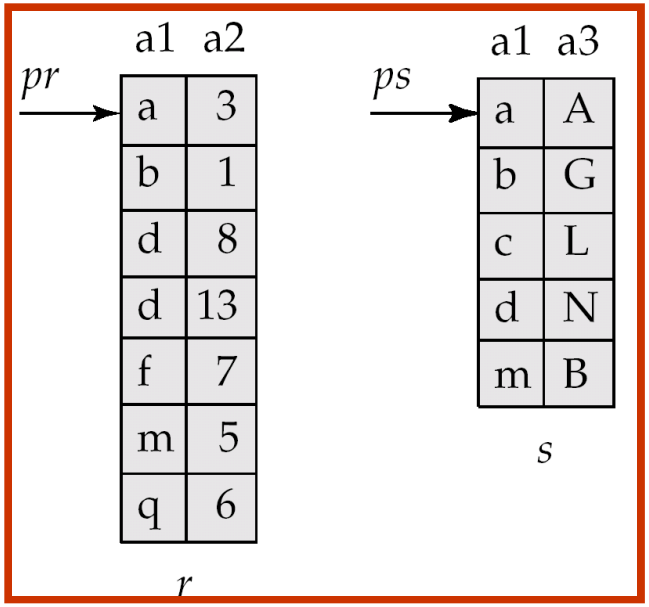
- Index může vylepšit spojování pokud
  - Pokud se tabulky spojují na základě rovností
  - Je index k dispozici
    - Někdy je výhodné vytvořit index jen kvůli spojování tabulek.
- Pro každou n-tici t_r z vnější relace *r* se použije index k nalezení m-tice z relace *s*, která splňuje podmínku spojení.
- Nejhorší případ: v paměti je místo pouze pro jeden blok dat z relace *r*, a pro každou n-tici z *r* se vykonává vyhledávání pomocí indexu v *s*.
- Cena spojení: $(b_r + n_r * c)\,(t_T + t_S)$
  - kde *c* je cena nalezení všech korespondujících řádku v *s* pomocí indexu
  - *c* lze odhadnout jakocenu jednoho příkazu select v relaci *s*.
- Pokud jsou indexy dostupné pro obě relace *r* a *s*, použije se realce s méně řádky jako vnější relace.

Příklad použití indexu pro spojení

- Spojení *zákazník* ⋈ *účet*, kde *účet* je vnější relace.
- Předpokládejme, že relace *zákazník* má primární index B^+-strom, který obsahuje 20 položek v uzlu.
- Protože relace *zákazník* má 10,000 řádků, hloubka vyhledávacího stromu je 4 a žádný další přístup do relace není nutný
- *účet* má 5000 řádků
- Nejhorší případ vnořených bloků má
  - 400*100 + 100 = 40,100 přesunů + 2 * 100 = 200 hledání
- Cena spojení s využitím indexu má
  - 100 + 5000 * 5 = 25,100 přesunů i hledání.
  - navíc cena CPU bude při použití indexů menší než u vnořených bloků

Merge-Join

1. Sloučení relací inspirované třídicím algoritmem Merge-sort.
2. Merge využívá setříděné obě relace pro sloučení
   1. Postupně se prochází obě relace tak, aby byly nalezeny všechny odpovídající páry.
   2. Posouvá se vždy ukazatel, který má menší hodnotu
   3. Složitější postup při vícenásbných shodných hodnotách atributů



Merge-Join

- Lze použít pouze pro spojování podle rovnosti sloupců
- Každý blok se čte pouze jednou
- Cena Merge-Join:

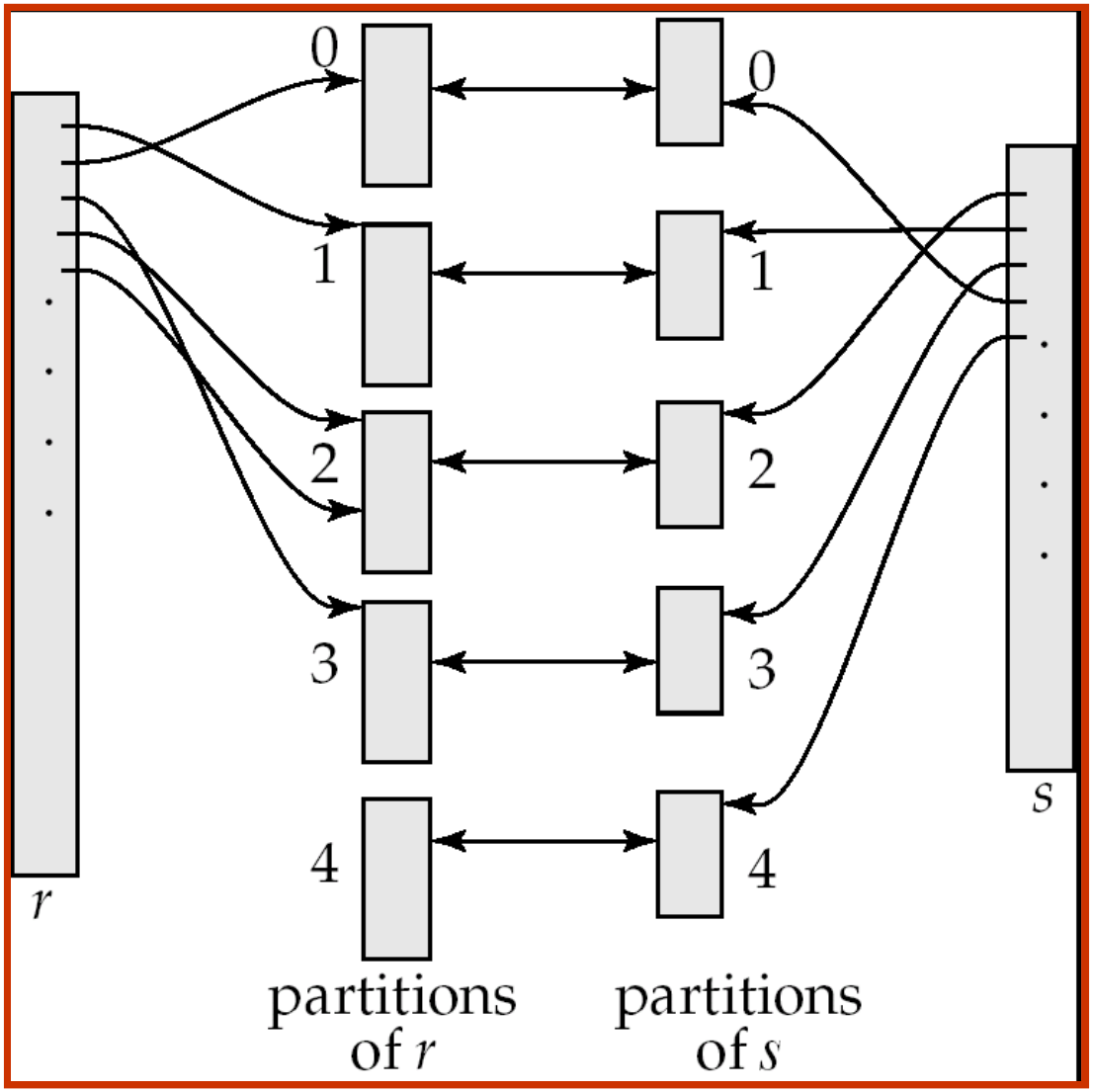
$$b_r + b_s \text{ přesunů } + \lceil b_r / b_b \rceil + \lceil b_s / b_b \rceil \text{ hledání}$$

  - + cena setřídění, pokud relace není setříděná.
- **Hybridní merge-join:** Pokud je jedna relace stříděná a pro druhou existuje sekundární vyhledávací index B^+-strom
  - Spojuje se setříděná relace a listy indexu druhé relace.
  - Počet přesunů pro relaci roste až k n_s

Hash-Join

- Použitelný setjně jako Merge-Join pouze pro spojení relací na rovnost hodnot sloupců.
- Hašovací funkce *h* je použita pro rozdělení řádků obou relací
- *h* mapuje *sloupce* hodnoty do {0, 1, ..., *n*}, kde *sloupce* označují společné atributy obou relací *r* a *s*.
  - $r_0, r_1, \ldots, r_n$ označují skupiny řádků *r* se stejnou hodnotou hašovací funkce.
  - $s_0,, s_1. \ldots, s_n$ označují skupiny řádků *s* se stejnou hodnotou hašovací funkce.

Hash-Join (Cont.)



Hash-Join (Cont.)

- n-tice z r_i se porovnávají pouze s m-ticemi z s_i Není nutné porovnávat s jinými řádky, protože:
  - n-tice z *r* musí mít stejné hodnoty jako m-tice z *s* protože spojování je na základě stejných hodnot atributů.
  - Pokud je hodnota těchot atributů zobrazena hašovací funkcí na hodnotu I, pak se n-tice z relace *r* musí nacházet v bloku r_i a m-tice z realce *s* se musí nacházet v bloku s_i.

Hash-Join

- Hodnota *n* dynamického hašování je volena tak, aby každý blok s_i se vešel do paměti.
  - Pro M velikost paměti v blocích, se typicky n volí tak aby velikost hašovacího indexu byla $\lceil b_s/M \rceil * f$, kdy f je konstanta pro reservování více prostoru, f=1.2 (20% rezerva)
  - The probe relation partitions s_i need not fit in memory
- **Rekurzivní hašování** se používá pokud velikost hašovacího indexu *n* je větší než velikost paměti *M*.
  - Místo hašovací funkce *n* potřebujeme pouze *M – 1* částí
  - To se udělá novou hašovací funkcí, která se použije na výsledek předchozí hašovací funkce a tím se získá *M – 1* bloků
  - Stejná rekurzivní hašovací funkce se použije pro *r* – není třeba vytvářet index, pouze se pro každou n-tici z *r* spočte rekurzivní hašovací funkce

Dotazy