Kapitola 1

Státnice - Datové struktury ve vnější paměti

1.1 Základní pojmy

- Logickou jednotkou dat je záznam, který má atributy se jmény a doménami (uvažujeme max. jednu hodnotu pro každý atribut).
- (Homogenní) soubor je kolekce (multimnožina) záznamů. Na souboru jsou definovány operace **INSERT**, **DELETE**, **UPDATE** a **FETCH**.
- Pro soubory s neopakujícími se záznamy je klíč množina atributů, které záznam jednoznačně identifikují v souboru. Jeden nich z klíčů se označuje jako primární.
- Vyhledávací klíč je něco jiného – k jeho jedné hodnotě se dá najít množina odpovídajících záznamů. Jsou tři druhy vyhledávacích klíčů: hodnotový, hashovaný a relativní (přímo pozice v souboru).
- Logickému záznamu odpovídá fyzický záznam délky R, BÚNO na magnetickém disku; ten může obsahovat další data — oddělovače, ukazatele, hlavičky. Záznamy mají buď pevnou, nebo proměnnou délku.
- Fyzické záznamy jsou organizovány do bloků délky B – hlavní jednotky přenosu mezi RAM a HDD.
- $\frac{B}{R}$ se nazývá blokovací faktor ($\lfloor b \rfloor$).
- Schéma organizace souborů (SOS) je popis logické paměťové struktury, ve které se soubor nachází. Může obsahovat více logických souborů, ten který nese uživatelská data je primární, jeho délka v počtu záznamů se značí N. Další operace nad SOS jsou **BUILD, REORGANIZATION, OPEN, CLOSE**.
- Dotaz je každá funkce, která pro zadaný argument vrátí "odpověď" – množinu záznamů. Dotazy mohou být buď na úplnou, nebo jen částečnou shodu, popř. intervalovou shodu.
- Vyváženost struktury je zajištění omezení délky cesty při vyhledávání nějakým výrazem ($O(\log N)$ atp.), popř. rovnoměrnosti naplnění bloků (popisuje ji faktor naplnění stránek). Splnění se dosahuje štěpením a sléváním bloku.
- SOS, které splňuje vyváženost, se nazývá dynamické, jinak statické.

Fyzická média

Soubory se fyzicky ukládají hlavně na magnetické pásky nebo HDD.

- Pásky umožňují jen sekvenční přístup, bloky jsou při vyhledávání čteny a kontrolovány sekvenčně (vhodné setřídění dát podle klíče). Pro kapacitu pásky je důležitá hustota, jsou nutné meziblokové mezery. Sekvenční čtení trvá $t' = R \cdot t/(R + W)$, kde W jsou meziblokové mezery, t je transfer rate.
- HDD umožňuje přímý přístup k datům. Uvažují se hlavy, válce (cylindry) a stopy. Vždy jen jedna hlava může číst. Většinou se HDD dělí na sektory. Pro rychlost přístupu jsou důležité náledující proměnné:
  - seek (přesun na jiný válec) — s
  - rotate (doba 1 půlotáčky) — r
  - block transfer time — propustnost sběrnice — btt

Nové disky mají různý počet sektorů na 1 stopu a tím pádem složitý výpočet cylindru a hlavy z čísla bloku, který dělá řídící elektronika. Udávaná adresa cylindr-hlava-sektor tak nemusí mít nic společného se skutečností.

Př. nechť 1 stopa = 512 K. Načtení 1 MB pak trvá minimálně $s + r$ pro nalezení prvního sektoru a $2 \times 2r$ za načtení stop. Při čtení z náhodně rozmístěných 4 K bloků ale vyjde $256 \times (s + r + btt)$ — tj. až 100x pomalejší.

Časový odhad doby přístupu k záznamu T_F je složitější:

- $s + r + btt$ obecně
- $r + btt$ pro další záznam ve stejném cylindru

Máme i operaci **REWRITE** — přepis (během 1 otáčky disků): $T_{RW} = 2r$. Další časy se značí T_I (**INSERT**), T_D (**DELETE**), T_U (**REWRITE**), T_Y (**REORG**), T_X (načtení celého souboru).

1.2 Statické metody organizace souborů

Sekvenční soubor

Jsou dvě varianty:

- Neuspořádaný (hromada) – jenom fyzicky za sebe naplácané záznamy. Složitost nalezení $O(N)$.
- Uspořádaný – záznamy řazeny podle klíče. Aktualizované záznamy se umisťují do zvláštního neuspořádaného souboru, **REORGANIZATION** celek znova setřídí. Nalezení je opět $O(N)$ nebo $O(N/\lfloor b \rfloor)$. U médií s přímým přístupem ale $O(\log_2 N)$.

Index-sekvenční soubor

Primární soubor je sekvenční soubor setříděný podle primárního klíče a nad ním (i víceúrovňový) index: číslo bloku a minimální hodnota klíče v něm; pro vyšší úroveň to samé, ale na blocích indexu nižší úrovně.

Nejvyšší úroveň indexu je obvykle jen jeden blok (master). Počet úrovní se dá spočítat: $x = \lceil \log_p N/\lfloor b \rfloor \rceil$, kde $p = \lfloor B/(V + P) \rfloor$ a V je velikost klíče a P velikost pointeru na blok nebo záznam.

Zařazení nového záznamu – problémy: přidávání do oblasti přetečení a v ní řetězení záznamu za sebe (každý záznam tam má pointer na další záznam v oblasti přetečení, nejenom blok). Pro oddálení nutnosti vkládat do oblasti přetečení lze bloky plnit na míň než 100 %.

Indexovaný soubor

Máme primární soubor a indexy pro různé vyhledávací klíče. Indexují se přímo záznamy, ne jen bloky. Primární soubor už nemusí být setříděný. Varianta: clusterovaný index – záznamy se společnou hodnotou 1 atributů jsou blízko sebe (jde jen pro jeden atribut).

Index může být podobný jako u index-sekvenčního souboru, ale lepší je, když pro záznamy se stejným klíčem je ve všech úrovních až na první (nejnižší) jen jeden klíč, který pak odkazuje na seznam záznamů. Pro aktualizace se nepředpokládá oblast přetečení, ale změny v indexu. $T_F = (1 + x)(s + r + btt)$.

Lze použít i dotazy na kombinovanou částečnou shodu, ale trvají dlouho. Alternativa: kombinovaný index pro více atributů; to ale vyžaduje analýzu, na co jsou dotazy nejčastější.

Bitové mapy

Bitové mapy jsou efektivní způsob indexace pro atributy s malou doménou (max. stovky). Pro každou hodnotu této domény vyrobíme vektor bitů délky N, kde jednička na i-té pozici indikuje, že i-tý záznam má právě tuto hodnotu atributů.

Booleovské dotazy potom fungují jako bitové operace nad těmito vektory. Vektory bitů lze navíc komprimovat — nejsou tak velké, vhodné pro databáze s hodně záznamy.

Indexy v DIS (document information system)

Jedná se o tzv. invertované soubory pro fulltextové hledání – pro každé slovo máme uložen počet souboru, kde se vyskytuje, a pointer na soubor souřadnic, tj. dvojic [dokument, pozice]. Dotazy na shodu pro více slov pak jsou množinové průniky (začínající od slova s nejmenším počtem výskytu v databázi).

Získání invertovaného souboru: rozparsování na slova, setřídění, odstranění duplicit, výpočet frekvencí slov. Zipfův zákon distribuce slov $f = k/r$, kde k je konstanta, r je pořadí četnosti, f je četnost. Taky lze využít kompresi a nebo zmenšit indexy vyhozením nevýznamných slov.

Soubor s přímým přístupem

Záznamy v primárním souboru ("adresový prostor") jsou rozptýleny pomocí hashovací funkce.

Implementace: buď hash = číslo stránky; nebo hash = číslo stránky i relativní pozice v ní. Pro kolize se snažím umístit záznamy pokud možno do stejné stránky.

1.3 Statické hashovací metody

Jako perfektní hashování se označuje nejen stav, kdy funkce nedává pro danou množinu žádné kolize, ale i takový systém, kde máme zaručený čas $O(1)$ pro přístup k záznamu.

Cormackovo perfektní hashování

Hashování odpovídá "velké funkci a malé tabulce" ze sekce o hashování. Máme soubor a adresář. Krom primární hashovací funkce h jsou potřeba další hashovací funkce $h_i(k,r) = (k \mod (2i + 100r + 1)) \mod r$, kde:

- r je velikost oboru hodnot (počet kolizí)
- i je nějaký vhodný parametr — číslo hash fce (hledá se postupným zkoušením, dokud není výsledek bez kolizí).

V adresáři uchováváme pro každý záznam odkaz do primárního souboru, r a i. V primárním souboru máme klíč a data. Při hledání:

1. Zahashujeme klíč pomocí h, dostaneme se do adresáře, pokud je tam $r = 0$, nenašli jsme.
2. Pokud $r \geq 1$, spočítám podle uloženého i hash-fci h_i, vezmu odkaz do primárního souboru, přičtu výsledek h_i a v primárním souboru zkontroluju, jestli jsem našel co jsem hledal.

Pro **INSERT** spočítám h a:

1. Pokud na daném místě v adresáři nic není, najdu volné místo v primárním souboru délky 1 a vložím
2. Pokud tam už něco je, najdu volné místo délky $r + 1$, zvětším r a najdu i, aby kolizní od sebe rozházela a přeházím je na nové místo.

r nemusím zvětšovat o 1, ale rychleji, takže najdu požadované i lépe (pokud mám blbý h_i, někdy r prostě musím zvětšit o víc).

Toto hashování lze použít i pro dynamický paměťový prostor, když z primárního souboru uděláme nesouvislý prostor, kam lze přidávat stránky.

Perfektní hashování Larson-Kalja

Uchováváme menší pomocnou tabulku než předchozím případě – pro každou stránku adresového prostoru máme jen jednu hodnotu – separátor. Máme:

- Množinu M hashovacích funkcí h_i, které vytvářejí pro každý záznam posloupnost stránek, do kterých ho můžeme zkoušet vkládat
- Navíc druhou sadu jim jednozn. odpovídajících funkcí s_i, které počítají signaturu.

Pokud je separátor stránky větší než signatura záznamu, bude záznam v této stránce, jinak ho tam nemůžu vložit. Po přeplnění vyhodím záznamy s největší signaturou a zmenším separátor. Prvek se nemusí povést vložit (když mi dojdou obě sady funkcí).

Pokud vkládám do plné stránky, nemůžu prvek vložit, i když má prvek menší signaturu než separátor — separátor pak musím zmenšit a ze stránky vyhodit i něco dalšího, čímž dojde ke kaskádování.

Toto hashování je vhodné pro málo vkládání a hodně čtení – mám zaručený jen jeden přístup na disk, protože malý adresář se do paměti vejde.

Dotazy na částečnou shodu

Máme-li záznamy s n atributy, použijeme pro každý z atributů zvláštní hashovací funkci, která generuje d_i-bitový výsledek (signaturu atributu). Signatura (hash) celého záznamu je pak konkatenací signatur atributů. Pro adresový prostor o velikosti $M = 2^d$ stránek volíme délky signatur atributů tak, aby $\sum d_i = d$.

Dotaz na částečnou shodu je potom dotaz, kde bity některých atributů jsou nahrazeny "?". Nazveme s-bitovou signaturu dotazu, má-li zadaných s bitů. Takový dotaz je 2^{d-s}-krát dražší než přímý konkrétní.

Je-li pravděpodobnost dotazů na i-tý atribut rovna P_i, vyjde průměrná cena dotazů:

$$\sum_{q \subseteq \{1,..n\}} (P_q \cdot \Pi_{i \notin q} 2^{d_i})$$

Ideální rozvržení d_i proto vychází (Lagrangovými multiplikátory):

$$d_i = (d - \textstyle\sum_{j=1}^{n} \log_2 P_j)/n + \log_2 P_i.$$

Je nutné upravit případné extrémy a přepočítat.
Pro urychlení dotazů můžeme použít deskriptory stránek:

- Máme deskriptor záznamů: $w = w_1 + .. + w_n$-bitový řetězec, kde pro každý atribut $A_1, ..A_n$ máme právě jednu jedničku (může být zadáno jinak, ex. vzorce pro ideální w_i i počet jedniček k nim).
- Deskriptor stránek je bitový OR deskriptorů všech záznamů ve stránce.
- Při hledání vyrobím deskriptor dotazů: místo "?" dám samé nuly. Pokud má dotaz někde v deskriptoru "1" a stránka "0", nemusím tam hledat.

Lze udělat i dvouúrovňově – s deskriptory segmentů.
Další možnost urychlení jsou Grayovy kódy, které se snaží řešit nesouvislost oblasti stránek se záznamy vyhovujícími dotazům (např. 10???1010). Jde o jiný způsob kódování binárních čísel — dvě po sobě jdoucí čísla se liší vždy jen v jednom bitu. Max. zisk — o 50% lepší než binární. Konverze čísla do Grayova kódování z pozičního vypadá následovně:

$$G(x) = B(x) \text{ xor } B(\lfloor \frac{x}{2} \rfloor)$$

Zpětně (g_i je i-tý bit Grayova kódu, n-tý bit je nejvyšší):

$$b_n = g_n,\ b_i = g_i \text{ xor } b_{i+1}$$

1.4 Dynamické hashování

Rozšiřitelné hashování – Fagin (Koubkovo "externí hashování")

Kromě primárního souboru (nebo souborů, protože jde o dynamickou strukturu) mám pomocnou strukturu – adresář s pointery na stránky o velikosti 2^d (přičemž některé stránky mohou být v adresáři uvedeny víckrát) a používám hashovací funkci s rovnoměrným rozdělením.

Vždy použiju jen prvních d bitů hashe, minimum kolik potřebuju. Podle nich určím záznam v adresáři a z něj najdu stránku. Pro stránku se může používat míň bitů, proto na ní může ukazovat víc záznamů v adresáři. Když se stránka naplní, rozštěpím ji na 2 podle dalšího bitu hash-funkce.

Pokud už stránka používá stejně bitů jako adresář, musím zvětšit o 1 bit i adresář (a zdvojnásobit jeho velikost), s ostatními stránkami nic nedělám. Při vypouštění záznamů lze stránky slévat, pokud si pamatuju jejich zaplněnost (můžu slévat jen stránky, které se liší jen v posledním bitu).

Adresář můžeme ukládat na jedné stránce externí paměti. Potom **FETCH** jsou max. 3 operace a **INSERT** nebo **DELETE** max. 6 operací s externí pamětí. Očekávaný počet použitých stránek je $\frac{n}{b \ln 2}$, kde b je počet prvků v jedné stránce, a velikost adresáře $\frac{e}{b \ln 2} n^{1+\frac{1}{b}}$ – to je víc než lineární růst, tj. nelze používat donekonečna (bez důkazů).

Lineární hashování – Litwin

V tomto případě nemám adresář, jen oblast přetečení, přístupnou pointry z primárních stránek. Data vkládám do primárních stránek, po každých L **INSERT**ech se vynuceně štěpí určená stránka (v pořadí podle čísel stránek $0, 1, 00, 01, 10, 11, 000$). Podle hashe (jeho konec musí odpovídá číslu stránky) se rozdělují při štěpení prvky.

FETCH: když mám aktuálně n stránek, vezmu posledních $\lfloor \log_2 n \rfloor + 1$ bitů hashe (pokud je to $> n$, zanedbám horní bit) a tím dostanu číslo stránky. Když v ní prvek není a stránka je přetečená, podívám se ještě do oblasti přetečení.

Při štěpení je nutné vrátit do rozštěpených stránek i záznamy z oblasti přetečení. Problémem jsou více zaplněné stránky na konci, tedy ještě nerozštěpené. Řešením je buď nerovnoměrné rozdělení hash-fce, nebo skupinové štěpení stránek.

Skupinové štěpení stránek

Rozšíření Litwina pro lepší využití stránek:

- Stránky jsou vždy organizovány vždy v s skupinách po g stránkách (začínáme s s_0 skupinami, postupně se s zvětšuje, g zůstává).
- Mám inicializační hash-funkci h do $\{0 \dots (g \cdot s_0) - 1\}$.
- Štěpím také po L vložení, vždy g stránek do $g + 1$, na to mám nezávislé hashovací funkce $h_0, ...h_\infty$ do $\{0 \dots g\}$.
- Až dostanu s skupin po $g + 1$ stránkách, přeorganizuju stránky zpátky do skupin po g (můžu přidat nějaké prázdné, aby to vycházelo).

Při hledání se počítají všechna prehashování úplně od začátku — je to docela HW náročné, ale proti rychlosti disků se vyplatí.

Spirálová paměť

Tohle je další rozšíření Litwinova hashování, tentokrát pro exponenciální rozdělení klíčů. Místo rozdělení jedné stránky na starou a novou tu starou zahodí a přidá dvě nové.

Klíče jsou nejprve rovnoměrně rozděleny hash-funkcí $G(c,k) \to \langle c, c+1\rangle$ $(c \in \mathbb{R}_0^+)$, pak přepočteny do $\langle b^c, b^{c+1}\rangle$ a zaokrouhleny na konkrétní čísla stránek. Při změně c se mění velikost prostoru.

Při expanzi se nové c volí tak, aby zničilo stránku, která se štěpí (nejnižší číslo):

$$c_{n+1} := log_b(f+1) \text{ (kde } f \text{ je číslo 1. stránky).}$$

Aby se mi neposouvaly všechny stránky pořád dál od 0, první zahozené stránce dám nové číslo a znova ji použiju – přepočítávání log. stránek na fyzické se pak dělá rekurzivně:

1. Je-li $\lfloor$logická stránka$/b\rfloor < \lfloor$(1+logická stránka)$/b\rfloor$, pak fyzická stránka := fyzická stránka($\lfloor$logická stránka$/b\rfloor$)
2. Jinak fyzická stránka = $\lfloor$log. stránka$/b\rfloor$.)

Hashovací funkce zachovávající uspořádání

Kvůli urychlení intervalových dotazů se objevily hashovaci metody, zachovávající uspořádání klíčů:

- Lineární hashování pro intervalové dotazy – vychází z Litwina, využívá nejvýznamnější bity k určení stránky.
- Částečně lineární hashování zachovávající uspořádání – vychází ze znalosti rozdělení klíčů, dělí doménu klíčů na nestejně dlouhé intervaly, aby vyvažovalo nerovnoměrnost rozdělení. Nelze ale pořád reorganizovat, takže se upravují jenom přilehlé intervaly při vkládání a odebírání. Nehodí se pro dynamicky rostoucí doménu klíčů. Může být provedeno víceúrovňově.

1.5 Třídění na vnější paměti

Pro třídění něčeho, co se nevejde do operační paměti, se používá **MERGESORT**:

1. Ze dvou souborů vezmu dva prvky
2. Vyberu menší z nich, zapíšu ho na výstup
3. Ze souboru, odkud ten menší pocházel, načtu další.
4. Konec setříděného úseku poznám tak, že další načtený prvek je menší než ten vypsaný. Pokud dojdu na konec setříděného úseku na jednom ze vstupů, dokopíruju zbytek setříděného úseku i z druhého vstupu na výstup.

Postupně dostávám delší setříděné kusy. Původní použití — setříděni kousků, které se vešly do paměti, a slévání na páskách. Dnes je použitelné i na HDD.

N-cestný **MERGESORT** na vnější paměti: Mám-li $n+1$ stránek v paměti:

1. Vytvořit setříděné běhy velikosti n stránek (použít **HEAPSORT** nebo **QUICKSORT**).
2. Pak v každém kroku slévat (maximálně) n nejkratších běhů, výsledek ukládat jako 1 běh.

Pro M stránek v celém souboru je složitost $O(2M\lceil \log_n M/n\rceil)$ — celé projde $\log_n(M/n)$-krát procesem slévání.
HEAPSORT může vytvořit i delší běhy, než co se vejde do paměti:

1. Průběžně odebírám a vypisuju minimální prvky a načítám další.
2. Pokud je načtený prvek větší než minimum, hodím ho na haldu.
3. Pokud je menší, dám ho do aktuálně vznikající haldy na druhém konci pole.

Až se vyčerpá halda, začnu nový běh. V nejhorším případě dopadne stejně jako třídění v paměti, průměrně je 2x lepší, ideálně setřídí všechno.