Kapitola 1

Fyzická organizácia dát

Je prirodzené, že vzhľadom k veľkému množstvu dát, ktoré obyčajne databázový systém spracováva, budú tieto dáta uložené na vonkajších pamäťových médiách. Nejde pritom len o to, že systém pri svojom štarte musí načítať veľké množstvo dát a pri eventuálnom ukončení práce všetky dáta na pamäťové médium zapísať. Množstvo spracovávaných dát je tak veľké, že výrazne prekračuje veľkosť dostupnej operačnej pamäti. To znamená, že databázový systém sa bude pri práci pomerne často obracať na vonkajšie pamäťové médium. Tieto médiá sú však výrazne pomalšie ako operačná pamäť počítača. Rýchlosť takéhoto systému preto vo veľkej miere závisí od vhodnosti fyzickej organizácie dát vo vonkajšej pamäti. To vysúva do popredia problematiku optimalizácie organizácie dát na vonkajších pamäťových médiách.

1.1 Fyzické parametre vonkajších pamäťových médií

Aby sme pochopili obmedzenia, ktoré práca s vonkajšou pamäťou prináša, uvedieme najprv základné typy externých médií a upozorníme na parametre, ktoré v rozhodujúcej miere ovplyvňujú ich rýchlosť.

1.1.1 Magnetické páskové pamäte

V prvom rade treba povedať, že on-line prevádzka databázového systému nad magnetickými páskami patrí minulosti. Využitie páskových pamätí je prevažne v off-line prevádzke, a to najmä na zálohovanie dát. To však nijakú zvláštnu optimalizáciu nevyžaduje.

Na magnetických páskach sa zväčša nachádza niekoľko záznamových stôp. (Bežná organizácia je 9 stôp: 8 údajových a jedna obsahujúca kontrolný súčet parity.) Údaje sú zaznamenávané po blokoch.[1] Magnetická páska je už svojou povahou rýdzo sekvenčným médiom,[2] t.j. v každom okamihu sme schopní pristúpiť len k nasledujúcemu (prípadne predošlému — podľa smeru pásky) bloku údajov. Akýkoľvek iný prístup by požadoval zdĺhavé prevíjanie pásky

[1] Ako neskôr uvidíme, organizácia údajov do blokov je bežná aj u iných pamäťových médií.
[2] SAM - sequential access memory

s postupným prechádzaním pásky blok po bloku až k požadovanému miestu. Aj samotné čítanie, resp. zapisovanie na pásku vyžaduje, aby sa páska najprv rozbehla na prevádzkovú rýchlosť a po skončení práce s blokom je nutné brzdenie, ktoré pásku opäť zastaví. Medzi blokmi údajov preto na páske musia byť medzery, umožňujúce toto štartovanie a brzdenie. Za zmienku preto stojí potreba vhodne zvoliť veľkosť bloku tak, aby medzery medzi nimi nezaberali priveľa miesta. Uvedený mechanizmus štartovania a zastavovania pásky má mnohé vylepšenia a optimalizácie, nebudeme sa tým však zaoberať.

Napriek možnosti blokového prístupu je vzhľadom na fyzikálne vlastnosti pásky (tepelná rozťažnosť) a vlastnosti mechanických zariadení (nedostatočná presnosť) páska typicky sekvenčné médium. Typická schéma spracovania je: *(kmeňový súbor, zmeny)* $\longrightarrow$ *(nový kmeňový súbor, správy o spracovaní).*

1.1.2 Rotačné pamäte

Pod rotačnými pamäťami rozumieme zvyčajne disky, príp. bubny. Disk sa skladá z niekoľkých na spoločnej osi rotujúcich povrchov a z čítacích hláv (pre každý povrch jedna) na spoločnom ramienku. Údaje sú na povrchoch zapísané v sústredných kružniciach — *stopách* (angl. tracks). Skupina stôp na jednotlivých povrchoch, ktoré majú ten istý polomer, sa označuje ako *cylinder*. Napokon, každý povrch je rozdelený na kruhové výseky — *sektory*. Určiť fyzickú polohu bloku údajov na disku teda značí určiť povrch, cylinder a sektor, na ktorom sa blok údajov nachádza. Vieme teda v podstate pristúpiť priamo k ľubovoľnému identifikovanému bloku údajov. Prístupová doba je daná časom na presun hlavičiek na príslušný cylinder (*seek-time*), natočením povrchov na príslušný sektor (*latency-time*) a samotným prenosom dát (*transmission-time*). Hovoríme o *pseudonáhodnom* prístupe. Z uvedených časov je pritom najvýznamnejší seek-time.

Bubny sa od diskov odlišujú len tým, že každý povrch má osobitnú hlavičku pre každú stopu. Časovo najnáročnejšia položka — seek-time teda odpadá.

1.2 Súbory a ich organizácia

Dáta sú na externých médiách uchovávané vo forme súborov (obyčajne viac súborov na jednom médiu). Konkrétne umiestňovanie údajových blokov súboru na vonkajšie pamäťové médium, údržba voľného priestoru a distribúcia diskového priestoru medzi jednotlivé súbory je zvyčajne záležitosťou operačného systému, nad ktorým databázový systém beží. Je však možné (napr. ORACLE pod UNIXom), že databázový systém má vyhradenú časť disku len pre seba. Táto časť disku nie je pod správou operačného systému, teda databázový systém k nej pristupuje priamo a jej konkrétna fyzická organizácia je čisto záležitosťou databázového systému. Tým sa však nebudeme zaoberať.

Pri práci so súborom teda pracujeme jednak s pomalšou, ale kapacitne väčšou vonkajšou pamäťou (t.j. externým médiom), jednak s (relatívne) rýchlou, avšak kapacitne podstatne menšou vnútornou (t.j. operačnou) pamäťou. Hovoríme o práci v *dvojúrovňovej pamäti.*

Počítače často obsahujú špecializované vstupno-výstupné procesory alebo vstupno-výstupné kanály, ktorých úlohou je ovládanie externých zariadení, aby od týchto činností bol oslobodený hlavný procesor počítača. Vstupno-výstupný procesor potom presúva údaje medzi vyrovnávacou pamäťou (*buffrom*) a vonkajším pamäťovým médiom, resp. naopak. Je však výhodné použiť dve vyrovnávacie pamäte. Kým hlavný procesor pracuje s jednou vyrovnávacou pamäťou, vstupno-výstupný procesor môže pracovať s druhou a presúvať dáta. Tejto technike sa hovorí *double-buffering*. Samozrejme, je možné použiť aj viac buffrov. Táto technika umožňuje plne využívať pri práci s externým médiom možnosti vstupno-výstupného procesora.

Buffrovanie však umožňuje držať vo vyrovnávacích pamätiach niekoľko blokov súčasne. Potom sa môže stať, že blok, s ktorým sme pracovali pred chvíľou, je ešte stále v pamäti a nemusí sa znovu do pamäte prenášať. Tak isto, ak postupne urobíme niekoľko zmien v bloku, zrealizujú sa všetky naraz až pri prenose bloku na externé médium, teda na jeden prístup. To sú nesporné výhody. Musíme však riešiť otázku, ktoré bloky pri preplnení vyrovnávacích pamätí odsunúť na disk. Tento problém je identický s problémom identifikácie „obete“ pri výpadku stránky virtuálnej pamäti známy z operačných systémov. Na ukážku spomeňme aspoň niekoľko stratégií:

- odsúvať bloky v poradí, v akom boli do pamäte umiestňované
- odsunúť blok, ktorý je v pamäti najdlhšie
- odsunúť blok, s ktorým sa najdlhšie nepracovalo

Čitateľa v tomto smere odkazujeme na niektorú z učebníc operačných systémov.

Pozrime sa teraz na súbor z pohľadu aplikačného programu. Tomu sa súbor javí len ako veľký (potenciálne nekonečný) adresný priestor.[3] Hovoríme o tzv. *logickom* súbore. Z hľadiska typológie je logický súbor dátovou abstrakciou: elementárne operácie s ním sú čítanie a zápis (resp. zmazanie) údaja z adresy logického súboru. To, ako, či a za akých okolností sme schopní k jednotlivým adresám pristúpiť, je problematikou *organizácie súboru*. To, aby sa čítania a zápisy v logickom adresnom priestore skutočne realizovali súborovými operáciami na externom médiu, je záležitosťou operačného systému alebo podobného mechanizmu. Podobne je záležitosťou operačného systému metóda, ktorou sa adresné pozície logického súboru zobrazujú napr. do čísiel povrchov, cylindrov a sektorov disku. Až na malé výnimky sa touto problematikou nebudeme zaoberať a opäť odkazujeme čitateľa na niektorú z učebníc operačných systémov. V ďalšom budeme predpokladať, že pracujeme len s jedným súborom nesúcim dáta a všetky operácie s ním budeme chápať ako operácie v adresnom priestore logického súboru.

1.2.1 Údajová štruktúra súboru

Údaje fyzickej databázy sú obyčajne organizované do údajových štruktúr nazývaných *záznamy* alebo niekedy *vety* (angl. records). Záznam je rozčlenený na údajové položky — *polia* alebo tiež *atribúty*. Polia majú určený svoj typ. Tento typ je v zásade „jednoduchým“ typom. To v tomto prípade značí číselné typy, reťazce pevnej, ale aj premenlivej dĺžky, smerníky a podobne. Do jedného záznamu sa zvyčajne zobrazuje jedna n-tica relácie (t.j. riadok tabuľky relačnej databázy) alebo záznam v chápaní sieťového či hierarchického databázového modelu.

[3] jeho kapacita je samozrejme v skutočnosti obmedzená kapacitou pamäťového média

Záznamy sú zoskupované do *súborov* (angl. files). Súbor je teda „kolekcia“ záznamov. Úmyselne sme použili termín „kolekcia“ a nie napr. množina či postupnosť — by mohlo evokovať spôsob, akým sa k záznamom pristupuje. V skutočnosti je spôsob prístupu k záznamom vecou *organizácie súboru* a môže byť principiálne rôzny.

V princípe by mohli byť záznamy v súbore rôznych typov (tzv. *nehomogénne* záznamy). Budeme sa zaoberať len homogénnymi záznamami. V takom prípade je súbor adekvátnou implementáciou relácie relačného modelu. V súboroch je zmysluplné definovať pojem *kľúča*[4] ako pole (množinu polí) záznamu, podľa ktorého sa k záznamu pristupuje. Kľúču podľa ktorého je súbor zorganizovaný sa zvyčajne hovorí *primárny kľúč*. Možno výstižnejšie než kľúč by bolo slovo index, to sa však zaužívalo v trochu posunutom význame. O poliach, ktoré nie sú súčasťou žiadneho kľúča, sa niekedy hovorí ako o *dodatočnej informácii.*

Ako sme už spomenuli, každá operácia s externým médiom je náročná na čas. Bolo by preto neefektívne obracať sa na vonkajšie médium kvôli každému jednému záznamu. Preto sa záznamy zoskupujú do blokov. Blok je najmenšia jednotka prístupu vonkajšej pamäte. To značí, že každá operácia so súborom na vonkajšom médiu sa týka vždy celého bloku. Spomenuli sme už tiež techniku buffrovania blokov.

1.2.2 Údajová štruktúra záznamu

Záznam je zväčša organizovaný ako postupnosť hodnôt jeho polí, vhodným spôsobom doplnená o pomocnú informáciu. Väčšina pomocnej informácie býva sústredená v samostatnej časti — *hlavičke záznamu.*

Ak sú polia záznamu pevnej dĺžky, stačí na identifikáciu začiatku poľa v zázname informácia o poradí, v ktorom sa polia uvádzajú, pričom toto poradie je pri homogénnych záznamoch fixné pre celý súbor. Je tak v globálne známa informácia, o aký počet bytov sú jednotlivé polia posunuté voči začiatku záznamu (tzv. *offset*). Spomeňme, čo napríklad môže byť v prípade polí s pevnou dĺžkou chápané ako pomocná informácia:

- *stavová informácia* záznamu: bity charakterizujúce stav záznamu
  - *used/unused bit:* informácia o tom, či záznam skutočne obsahuje údajové informácie alebo ide o nezaplnený záznam, ktorý obsahuje náhodné dáta.
  - *deleted bit:* informácia o tom, že záznam bol zmazaný — o význame a funkcii tejto informácie budeme hovoriť v súvislosti so smerníkmi.
- *formát záznamu:* informácia je potrebná len v prípade nehomogénnych záznamov, keď hovorí, o ktorý z uvažovaných formátov záznamu ide.
- *výplň:* nenesie žiadnu informáciu, ide jednoducho o nevyužitý priestor. Výplň vkladáme do záznamu medzi polia väčšinou vtedy, ak zo softwarových, ale častejšie z hardwarových dôvodov je nutné alebo vhodnejšie, aby jednotlivé polia začínali na špecifických pozíciách, napr. práca s poliami začínajúcimi na adresách, ktoré sú násobkom 4 je v niektorých prípadoch lepšie podporovaná hardwarom.

[4]Kľúč v zmysle organizácie súboru môže a nemusí byť kľúčom v zmysle relačnej teórie.

V prípade polí premenlivej dĺžky však potrebujeme aj pomocnú informáciu, ktorá určí začiatky jednotlivých polí. Zvyčajne sa pritom uplatňuje jedna z nasledujúcich stratégií:

1. Polia budú oddelené špeciálnym symbolom (napr. #), o ktorom je zaručené, že sa nevyskytuje v žiadnom z polí.

2. Každé pole bude začínať informáciou o svojej dĺžke.

3. Záznam bude na začiatku obsahovať systém smerníkov ukazujúcich na začiatky polí. To je vo svojej podstate ekvivalentné tomu, že začiatok záznamu obsahuje systém posunov (offsetov) jednotlivých polí vzhľadom k začiatku záznamu pre tento konkrétny záznam.

V prípade, že sa veľkosť záznamu prispôsobuje veľkosti svojich polí (záznam premenlivej dĺžky) môže byť praktické pridať k pomocnej informácii aj dĺžku záznamu. No a napokon, je potrebné zvoliť vhodný kompromis, ak niektoré polia záznamu sú pevnej a niektoré premenlivej dĺžky (vtedy sa zvyčajne v zázname nachádzajú najprv tie jeho polia, ktoré majú pevnú dĺžku a až za nimi polia premenlivej dĺžky).

Štruktúra záznamu, ako sme ju práve opísali, je štruktúrou fyzického záznamu — teda popisuje jeho fyzickú organizáciu. Pri pohľade z vyššej úrovne však už nevidíme pomocnú informáciu vo fyzickom zázname ani problémy s organizáciou záznamu pri poliach premenlivej dĺžky. Záznam sa nám javí len ako zoskupenie polí, pričom obsah jednotlivých polí môžeme čítať, mazať, prepisovať, resp. zmazať celý záznam, pričom sa už nemusíme zaoberať úpravou pomocnej informácie. Hovoríme o *logickom pohľade* na záznam, resp. o *logickom* zázname.

1.2.3 Údajová štruktúra bloku

Prenos údajov po blokoch značí súčasný prenos viacerých záznamov. Sme potom postavení pred problém nájsť polohu príslušného záznamu v rámci bloku. Opäť sa prípady záznamov s pevnou dĺžkou a premenlivou dĺžkou líšia a blok opäť obsahuje pomocnú informáciu, ktorá management záznamov v rámci bloku umožňuje.

V prípade záznamov s pevnou dĺžkou je zvyčajne blok rozdelený na úseky (angl. *slots*) zodpovedajúce veľkosťou veľkosti záznamov. Niektoré pozície sú neobsadené a obsahujú náhodné dáta. Na identifikáciu tejto situácie nám slúži used/unused bit. Zväčša je však výhodné zhromaždiť všetky used/unused bity (a podobne deleted bity) z jednotlivých záznamov na jedno miesto ako pomocnú informáciu bloku vo forme bitového vektora. Ak sú totiž used/unused bity rozptýlené po jednotlivých záznamoch a my hľadáme voľnú pozíciu v bloku na uloženie nového záznamu, musíme prezrieť všetky pozície bloku, až kým nenájdeme pozíciu označenú ako nepoužitú. Ak sú všetky used/unused bity sústredené na jednom mieste, identifikujeme voľnú pozíciu hneď. Pomocná imformácia bloku býva sústredená na jednom mieste — v *hlavičke bloku.*

Blok ďalej zvyčajne obsahuje smerník určujúci nasledujúci, príp. predchádzajúci blok, príp. niekoľko nasledujúcich blokov — v závislosti od organizácie súboru.

V prípade záznamov premenlivej dĺžky musíme byť navyše schopní určiť začiatky záznamov v bloku. Možnosti oddeľovačov alebo uvedenia dĺžky pred každým záznamom, ktoré sme uvádzali pri poliach premenlivej dĺžky, však už pri záznamoch nie sú výhodné. Prehľadať postupne celý blok kvôli nájdeniu jedného záznamu je totiž podstatne náročnejšie než prehľadať celý záznam kvôli jednému poľu. Výhodnejšia je preto možnosť umiestniť na začiatok systém smerníkov, tzv. *adresár* (angl. block directory), ktoré ukazujú na začiatky záznamov. Treba však zvážiť vyhradenie dostatočne veľkej časti bloku na tento účel alebo zabezpečiť mechanizmus umožňujúci dynamické zmeny veľkosti adresára.

Podobne ako pri záznamoch a súboroch má zmysel hovoriť o logickom pohľade na blok, keď abstrahujeme od pomocnej informácie bloku a mechanizmu jej údržby.

V ďalšej časti sa bližšie pozrieme na niekoľkokrát zmienené smerníky.

Smerníky a pripichnuté záznamy

Spomínali sme smerníky ukazujúce na začiatky jednotlivých polí v rámci záznamu, či smerníky adresára bloku ukazujúce na začiatky jednotlivých záznamov. Toto sú smerníky prislúchajúce pomocnej informácii záznamu, resp. bloku. Ich management nerobí vzhľadom k obmedzenému použitiu väčšie problémy. Implementované môžu byť napr. ako offset poľa, resp. záznamu k začiatku záznamu, resp. bloku. Pri logickom pohľade na záznamy či bloky sa s nimi vlastne nestretneme.

Do inej kategórie patria smerníky, ktoré pre blok určujú nasledujúci či predchádzajúci blok. Nadväznosť blokov je totiž v rozhodujúcej miere daná organizáciou súboru a s rozhodovaním o nej sa ešte stretneme v ďalšom texte. Tieto smerníky možno implementovať ako odkaz na číslo bloku v súbore (nech by už toto číslo znamenalo jednoznačný identifikátor bloku alebo jeho adresu v rámci logického súboru). Ani management tejto kategórie smerníkov nerobí zvyčajne veľké problémy. Zmeny nadväzností blokov, a teda aj management týchto smerníkov, je totiž plne určený organizáciou súboru a je preto aj plne pod kontrolou mechanizmu, ktorý organizáciu zabezpečuje, teda pod internou kontrolou databázového systému. To, že by tieto smerníky prestavoval aplikačný program priamo, sa vylučuje.

Smerníky, ktoré predstavujú problém, sú smerníky, s ktorými priamo manipuluje aplikačný program ako s údajom, teda ide o situáciu, keď je dátovým typom poľa záznamu smerník. Vzhľadom k aplikačnej povahe má zmysel uvažovať len smerníky ukazujúce na nejaký iný záznam. Smerník ukazujúci na pole alebo blok totiž nie je odzrkadlením vzťahu z aplikačnej oblasti; smerníky na polia a bloky sú len čisto implementačnou pomocnou informáciou.

Záznam, na ktorý ukazuje smerník z iného záznamu však nemožno jednoducho presúvať. Presunutie takéhoto záznamu totiž na zachovanie konzistencie súboru vyžaduje upraviť aj všetky naň ukazujúce smerníky. Hovoríme o *pripichnutých* (angl. pinned) záznamoch. Ba čo viac, pripichnutý záznam nemožno ani jednoducho vymazať. Keby totiž neskôr, po jeho vymazaní, bol na toto miesto umiestnený iný záznam, smerník by zrazu ukazoval na tento nový záznam namiesto toho, aby ukazoval na neexistujúci záznam. Na vyriešenie tejto situácie sa používa už skôr zmienený deleted-bit. Pri zmazaní záznamu nastavíme jeho deleted-bit, aby sme túto pozíciu už v budúcnosti na umiestnenie záznamu nepoužili. Závisí od konkrétneho prípadu, či takéto riešenie neznačí signifikantnú stratu alebo naopak ide o značné plytvanie priestorom.

Aby sme posúdili lepšie riešenia, zvážme najprv možnosti implementácie aplikačných smerníkov.

- Najjednoduchšie je implementovať smerník ako adresu adresného priestoru logického súboru. V takom prípade je teda smerník implementovaný *absolútnou adresou.* Pripichnutý záznam by v tomto prípade bol pripichnutý úplne.

- Opačným extrémom je implementovať smerník ako hodnotu kľúča záznamu, na ktorý ukazuje (keďže kľúč identifikuje záznam jednoznačne). Pristúpiť k záznamu, na ktorý smerník ukazuje v tomto prípade potom značí prejsť procesom vyhľadávania záznamu podľa primárneho kľúča. Tento proces je daný organizáciou súboru a môže vyžadovať niekoľko ďalších prístupov na externé pamäťové médium.

- Kompromisné riešenie je implementovať aplikačný smerník ako dvojicu (b, k), kde b je blok, v ktorom sa záznam nachádza a k je jeho kľúč. V tomto prípade na jeden prístup získame blok s uvažovaným záznamom a jeho prehľadaním na primárny kľúč k príslušný záznam, resp. informáciu, že taký záznam neexistuje. Prehľadanie bloku sa však už deje len v operačnej pamäti (záznamy v rámci bloku môžu byť napr. utriedené alebo aspoň zreťazené podľa hodnôt kľúčov[5]). To potom umožňuje presúvať záznam ľubovoľne v rámci bloku, v ktorom sa nachádzal.

V prípade záznamov premenlivej dĺžky a adresára smerníkov na začiatky záznamov v bloku možno značne ušetriť miesto plytvané zmazanými záznamami technikou *polopripichnutých* (angl. semi-pinned) záznamov. Aplikačný smerník nebude ukazovať priamo na začiatok záznamu, na ktorý by ukazovať mal, ale do adresára v hlavičke bloku. K požadovanému záznamu pristúpime potom na dvakrát: najprv sledujeme smerník do adresára a tam nájdený smerník nás už zavedie k požadovanému záznamu. Ide teda o nepriamu referenciu. Aj toto umožňuje ľubovoľné premiestňovanie záznamov v rámci bloku (dokonca sa nevyžaduje ani prístup pomocou primárneho kľúča — musíme len vždy patrične upraviť adresár smerníkov). Ak teraz zmažeme niektorý záznam, nemusíme nastavovať jeho deleted-bit a plytvať priestorom tým, že miesto po zmazanom zázname už pre istotu nepoužijeme. Deleted-bit môže byť vo forme bitového vektora súčasťou adresára smerníkov na začiatku bloku. Ak je záznam zmazaný, vyznačí sa táto skutočnosť nastavením deleted-bitu v adresári a smerník z adresára, ktorý na záznam ukazoval, sa už viac nepoužije. Miesto po zmazanom zázname však môžeme použiť opätovne. Vyplytvaný priestor je teraz len priestor na uloženie smerníka, ktorý sa už v budúcnosti nemôže použiť — to je však podstatne menej než nepoužiteľný priestor, ktorý by vznikol nepoužiteľnosťou miesta po zmazanom zázname.

1.2.4 Fragmentácia

Už sme videli, že polia aj záznamy môžu mať pevnú alebo premenlivú dĺžku. Do úvahy pripadajú tieto možnosti:

[5] pravda ak typ kľúča umožňuje porovnávanie hodnôt — to je však zvyčajne splnené, niekedy dokonca priamo vyžadované

- záznamy pevnej dĺžky s poliami pevnej dĺžky
- záznamy pevnej dĺžky s poliami premenlivej dĺžky
- záznamy premenlivej dĺžky s poliami premenlivej dĺžky

Možnosť záznamov premenlivej dĺžky s poliami pevnej dĺžky je totiž nezmyselná. Všimnime si najprv záznamy premenlivej dĺžky. Tie sú postupne umiestňované na rôzne pozície bloku a zase zmazávané. Časom vzniknú medzi záznamami medzery primalé na umiestnenie záznamu. Môže sa tak stať, že do bloku už nemožno vložiť ďalší záznam, aj keď celkový súčet voľného miesta by na to dostačoval. Tejto situácii sa hovorí *vonkajšia fragmentácia*. Zdá sa, že pri záznamoch premenlivej dĺžky sa jej skôr či neskôr nemožno vyhnúť. Naproti tomu pri záznamoch pevnej dĺžky toto nebezpečenstvo vôbec nehrozí. Blok je rozdelený na sloty rovnakej veľkosti ako veľkosť záznamu, a tak vznikajúce medzery majú veľkosť násobku veľkosti záznamu. Ak však má záznam polia premenlivej dĺžky, hrozí nebezpečenstvo, že polia nedostatočne využívajú celkovú pevnú dĺžku záznamu. Tejto situácii sa hovorí *vnútorná fragmentácia.* O vnútornej fragmentácii hovoríme aj v prípade záznamov pevnej dĺžky s poliami pevnej dĺžky, ak obsah polí nedostatočne využíva ich celkovú pevnú dĺžku (napr. ak typ poľa je reťazec pevnej dĺžky 255, ale väčšina záznamov neobsahuje reťazce dlhšie ako 10 znakov).

S vnútornou fragmentáciou počas prevádzky súboru vlastne nemožno nič urobiť, je vecou vhodného návrhu formátu záznamov a polí, aby sa tento efekt podľa možnosti minimalizoval. Iná je situácia s vonkajšou fragmentáciou. Ak k nej už došlo, môžeme vykonať *reorganizáciu* bloku zameranú na *kompaktovanie*, teda na zhromaždenie voľných úsekov do jedného alebo niekoľkých veľkých úsekov. Pritom je mimoriadne dôležité, aby sa záznamy v rámci bloku mohli presúvať. O súvise tohto problému a smerníkov sme hovorili v predošlej časti.

Treba povedať, že ak sa fragmentácia týka mnohých blokov, teda vlastne celého súboru alebo dokonca celého vonkajšieho pamäťového média (typické pre disky), je vhodnejšia off-line reorganizácia súboru či disku, nazývaná aj *defragmentácia,* v čase, keď systém nie je v prevádzke.

Je však potrebné zaoberať sa aj tým, ako dosiahnuť, aby k fragmentácii nedochádzalo, resp. aby k nej dochádzalo čo najneskôr. Na to sa najprv musíme zamyslieť nad správou voľného priestoru v bloku. Poznamenajme, že čo sa týka záznamov pevnej dĺžky, spomenuli sme už na správu voľného priestoru techniku used/unused bitu. Problematika vonkajšej fragmentácie je však záležitosťou záznamov premenlivej dĺžky.

Voľné úseky bloku sa vtedy zväčša udržujú v spájanom zozname. Smerník na prvý prvok takéhoto zoznamu sa nachádza v hlavičke bloku (ako súčasť pomocnej informácie). Spôsob prideľovania voľného miesta potom závisí od organizácie tohto zoznamu.[6] Management tohto zoznamu by pritom nemal zabúdať na skutočnosť, že ak sa dva menšie úseky nachádzajú v bloku pri sebe, ide v skutočnosti o jeden väčší úsek a takýmto spôsobom má byť aj registrovaný v spájanom zozname. Spomenieme tri základné techniky prideľovania voľného miesta:

[6] Spájaný zoznam ako spôsob udržiavania prehľadu o voľnom mieste možno použiť aj v prípade záznamov pevnej dĺžky. Pretože tam výber konkrétneho úseku nemôže fragmentáciu ovplyvniť, možno voľné miesto prideľovať v podstate ľubovoľným spôsobom. Najjednoduchšie je organizovať spájaný zoznam voľného miesta ako zásobník.

- First Fit — Záznamu je pridelený prvý nájdený dostatočne veľký voľný úsek.
- Best Fit — Záznamu je pridelený najmenší dostatočne veľký voľný úsek, teda úsek, do ktorého sa najviac hodí.
- Worst Fit — Záznamu je pridelený najväčší, avšak dostatočne veľký, voľný úsek, teda úsek, do ktorého sa najmenej hodí.

V metóde First Fit môže byť spájaný zoznam voľného miesta organizovaný v podstate ľubovoľne. Pri prideľovaní miesta sa zoznam prezerá, až kým sa nenájde dostatočne veľký úsek. Ak dostatočne veľký úsek neexistuje, zistíme to až prehľadaním celého zoznamu. Ak aj dostatočne veľký úsek existuje, môže sa stať, že ho nájdeme až po prehľadaní celého zoznamu.

V metóde Best Fit je spájaný zoznam voľného miesta utriedený vzostupne podľa veľkosti udržovaného miesta. Zoznam sa teda prezerá od začiatku, až kým sa nenájde dostatočne veľký voľný úsek. To, že takýto úsek vôbec neexistuje, môžeme zistiť hneď, ak sa pozrieme na posledný prvok spájaného zoznamu (ak nemá dostatočnú veľkosť ani on, hľadaný úsek neexistuje). Samozrejme, opäť sa môže stať, že ak hľadaný úsek existuje, nájdeme ho až prehľadaním celého zoznamu.

V metóde Worst Fit je spájaný zoznam voľného miesta opäť utriedený podľa veľkosti udržovaného miesta, avšak zostupne. Záznamu sa pridelí automaticky prvý prvok zoznamu. Ak nie je dostatočne veľký, požadovaný úsek neexistuje.

Metóda First Fit je najjednoduchšia a najmenej nákladná na údržbu. Zdalo by sa, že metóda Best Fit je omnoho vhodnejšia a Worst Fit nezmyselná vzhľadom k zámeru, ktorý sme si vytýčili, teda minimalizácia fragmentácie. Je to však práve naopak. V metóde Best Fit totiž po prehľadaní spájaného zoznamu voľného miesta, nájdení najvhodnejšieho voľného úseku a umiestnení uvažovaného záznamu na túto pozíciu zostane pravdepodobne z pôvodného voľného úseku malý kúsok neobsadený (tento kúsok je najmenší spomedzi všetkých možných — to je podstata metódy Best Fit). Vznikne teda nový voľný úsek, ktorý však už bude pravdepodobne primalý na to, aby sa doň umiestnil ďalší záznam. Fragmentácia sa tým práve podporuje. Pritom v metóde Worst Fit, ktorá už okrem udržiavania utriedeného spájaného zoznamu voľného miesta ďalšie náklady nevyžaduje, pridelíme najväčší voľný úsek. Tá jeho časť, ktorá zostane voľná po umiestnení záznamu, bude mať pravdepodobne ešte dostatočnú veľkosť na to, aby sa v budúcnosti použila na umiestnenie ďalších záznamov.

Možno ešte jednoduchšie ako metóda Worst Fit je prideľovať miesto na disku cyklicky. Celý disk vidíme ako cyklicky zreťazený zoznam blokov a prideľuje sa vždy ďalšie nasledujúce voľné miesto. V prípade, že nie je už k dispozícii dostatočne veľký úsek diskového priestoru je potrebné vykonať operáciu skompaktnenia disku. Nevýhodou je, že operácia skompaktnenia, je časove a výpočtovo dosť náročná. Na druhej strane, dáta zrušené uvoľnením miesta možno do nasledujúceho skompaktnenia, ešte vždy zachrániť, lebo neboli prepísané.

1.3 Schémy organizácie súborov

Niekoľkokrát sme sa dotkli pojmu *organizácia súboru* alebo aj *schéma organizácie súboru.* Pozrieme sa teraz na túto problematiku bližšie. Schéma organizácie súboru je logická štruktúra. Je tvorená blokmi súboru, pomocnými štruktúrami vzájomne zväzujúcimi bloky (napr. systém smerníkov) a algoritmami, ktoré slúžia na operácie so súborom a zachovávajú vlastnosti tohto previazania. Celý mechanizmus môže značiť, že okrem hlavného súboru obsahujúceho dáta (tzv. *primárneho súboru*) máme do činenia aj s ďalšími súbormi obsahujúcimi pomocnú informáciu. Pomocnej dátovej štruktúre zjednodušujúcej prístup do súboru sa hovorí *index*. Index je príkladom informácie, ktorá môže byť v niektorej zo schém organizácie súboru uložená v samostatnom súbore (tzv. *indexovom súbore*).

Zhrňme teraz operácie, ktoré sa so súborom vykonávajú a ktoré by mala každá schéma organizácie súboru podľa možnosti čo najefektívnejšie podporovať.

1. *Vyhľadanie záznamu* (angl. Look-Up, Search). Ide o najtypickejšiu operáciu so súborom zameranú na nájdenie konkrétneho záznamu pomocou informácie, ktorá ho dostatočne identifikuje, t.j. prevažne o známu hodnotu primárneho kľúča.

2. *Vloženie záznamu* (angl. Insertion). Ide o vloženie záznamu na miesto, ktoré mu určuje organizácia súboru. Aby sa zabránilo duplicitnému vloženiu záznamu (teda situácii, že v súbore už existuje záznam s týmto kľúčom), je treba pred operáciu Insert vykonať operáciu Look-Up.

3. *Zmazanie záznamu* (angl. Deletion). Predpokladá sa, že zmazávaný záznam v súbore existuje.

4. *Modifikácia záznamu* (angl. Modification, Update). Uvažovaný záznam sa najprv nájde operáciou Look-Up. Blok s pozmeneným záznamom sa potom zapíše na externé pamäťové médium.

5. *Vytvorenie súboru* (angl. Creation) a *Ukončenie práce so súborom* (angl. Close). Ide o technicky potrebné, avšak z hľadiska optimalizácie nezaujímavé operácie. Create musí inicializovať súborovú schému, Close zasa upraviť všetky štruktúry tak, aby sa kompletná informácia ocitla na externom pamäťovom médiu.

Medzi operácie so súborom by sme mohli zaradiť aj *off-line reorganizáciu* (napr. za účelom odstránenia fragmentácie a podobne). Touto operáciou sa obzvlášť zaoberať nebudeme. Pripomíname, že aj keď operácie sú formulované ako operácie so záznamami, systém pracuje v skutočnosti s blokmi. Nad fyzickým súborom sa teda vykonávajú operácie: read(B) — načítanie bloku do vyrovnávacej pamäti, write(B) — zápis bloku na externé pamäťové médium, locate(B) — priamy prístup k bloku na pamäťových médiách, ktoré priamy prístup umožňujú.

Budeme sa zaoberať týmito schémami organizácie súborov:

- Sekvenčné súbory a haldy
- Indexovo-sekvenčné súbory
- Stromové súbory
- Hašované súbory

Vo všetkých týchto prípadoch sa zameriame na dotazy na úplnú zhodu (resp. na dotazy s úplne zadaným kľúčom). O možnostiach ostatných typov dotazov pohovoríme neskôr.

1.4 Sekvenčné súbory a haldy

Sekvenčným súborom nazývame súbor organizovaný tak, že k jeho blokom možno pristupovať len v rastúcom alebo klesajúcom poradí, teda po každom bloku sa dá čítať len nasledujúci, resp. predchádzajúci blok. Táto organizácia môže byť daná buď povahou vonkajšieho pamäťového média (napr. magnetická páska) alebo jednoducho tým, že sme sa rozhodli súbory implementovať ako sekvenčné. V skutočnosti sa pod sekvenčnými súbormi rozumejú väčšinou tzv. *kľúčovo-sekvenčné* súbory, teda sekvenčné súbory udržiavané utriedené podľa hodnôt primárneho kľúča. Ak sekvenčný súbor nie je vôbec nijako utriedený, hovorí sa niekedy o *halde*.[7] Čo sa týka sekvenčnosti uvažovaných súborov, striktne vzaté by nemalo byť možné v ľubovoľnom okamihu pristúpiť napr. na koniec súboru. Prísne vzatý koncept sekvenčného súboru by totiž mal vyžadovať postupné čítanie súboru až na koniec. To je úplná pravda, ak je použité externé pamäťové médium sekvenčné. Ak však ide len o implementáciu sekvenčného súboru na pamäťovom médiu umožňujúcom aj priamy prístup (čo je v súčasnosti omnoho pravdepodobnejšie), nebolo by praktické nevyužiť to. Ak si to teda bude situácia vyžadovať, povolíme pristúpiť do súboru aj priamo (napr. na koniec súboru) — hovoríme o *sekvenčných súboroch s priamym prístupom*. Sekvenčnosť takéhoto súboru potom referuje na organizáciu jeho blokov. Každý blok pozná totiž len svojho nasledovníka a predchodcu. Bloky teda tvoria spájaný zoznam a sekvenčný pohyb v ňom je prirodzený. V prípade potreby dokážeme síce pristúpiť k akémukoľvek bloku priamo — prostredníctvom jeho čísla, ale to tento blok nedáva do logického súvisu s blokom, s ktorým sme pracovali predtým. Na takéto porušenie striktnej sekvenčnosti súboru v ďalšom výslovne upozorníme.

Halda je triviálnou schémou organizácie súboru. Ak chceme pristúpiť k záznamu v halde podľa primárneho kľúča, musíme postupne sekvenčne prehľadať celý súbor, až kým nenájdeme požadovaný záznam alebo po prezretí celého súboru neskonštatujeme, že sa taký záznam v súbore nenachádza. Pri úspešnom hľadaní teda prezrieme priemerne polovicu súboru a pri neúspešnom celý.

Časová náročnosť operácie Look-Up má za následok vysokú časovú náročnosť ostatných súborových operácií, pokiaľ musí byť pred nimi vykonaná. Vložiť vetu (operácia Insert) by sme mohli aj do bloku s ktorým práve pracujeme, ak je v ňom voľné miesto. Inak musíme pridať ďalší blok alebo záznam zapísať do posledného bloku. Ak je posledný blok plný, súbor sa zväčší o ďalší blok.

Zmazanie vyžaduje najprv vyhľadanie záznamu a nastavenie deleted-bitu pri pripichnutých záznamoch. Pri nepripichnutých záznamoch pevnej dĺžky môžeme dokonca vzniknutý priestor po zmazanom zázname zaplniť posledným záznamom súboru a tým súbor udržiavať skompaktovaný.

Modifikácia tiež vyžaduje operáciu Look-Up a následný zápis modifikovaných údajov na externé pamäťové médium. Ak však ide o záznamy premenlivej dĺžky a modifikáciou sa niektorý z nich zväčšil, môže sa stať, že musíme robiť presuny (týkajúce sa aj viacerých blokov), aby sme vytvorili dostatok miesta.

[7] Slovo halda — heap — má v informatike aj množstvo iných významov.

Venujme sa teraz kľúčovo-sekvenčným, teda utriedeným sekvenčným súborom. Najväčšiu ťažkosť zrejme predstavuje udržať takýto súbor utriedený. Vložiť záznam do súboru v tomto prípade totiž znamená vložiť ho na správne miesto. Ak v príslušnom bloku na to nie je dostatok priestoru, musel by sa celý súbor sekvenčne prekopírovať, pričom počas kopírovania by sa diskutovaný záznam vložil na správne miesto. To je prirodzene neúnosné. Problém sa zvyčajne rieši zavedením tzv. *oblasti pretečenia.* Ak je potrebné vložiť záznam, nie je na správnom mieste dostatok miesta a nemožno to vyriešiť lokálnymi zmenami v súbore, záznam sa vloží do oblasti pretečenia. Reorganizácia súboru sa potom spraví off-line, ak sa oblasť pretečenia preplní (či už úplne alebo percentuálne). Treba prirodzene doriešiť spôsob odkazu do oblasti pretečenia. Základné sú dve techniky:

- *Nepriamy prístup cez primárny súbor:* Každý záznam bude v rámci svojej pomocnej informácie obsahovať smerník potenciálne ukazujúci do oblasti pretečenia. Uvažujme záznamy R_1 a R_2 nachádzajúce sa v bloku primárneho súboru za sebou. Ak smerník záznamu R_1 neukazuje nikam (má nilovú hodnotu), značí to, že v utriedení po ňom nasledujúci záznam je skutočne záznam R_2. Ak však smerník záznamu R_1 ukazuje na záznam R v oblasti pretečenia, značí to, že v utriedení sa záznam R nachádza medzi záznamami R_1 a R_2. Ak by sa medzi záznamami R_1 a R_2 malo nachádzať viacero záznamov v oblasti pretečenia, budú tieto záznamy zreťazené do spájaného zoznamu. Smerník záznamu R_1 potom bude ukazovať na začiatok tohto zoznamu.

- *Prístup s posunom:* Smerník do oblasti pretečenia nebude obsahovať každý záznam. Tento smerník bude jeden pre celý blok. Ak je potrebné vložiť záznam R medzi dva po sebe idúce záznamy R_1 a R_2, všetky záznamy počnúc R_2 sa posunú o jednu pozíciu dozadu, čím sa uprázdni pozícia pre záznam R. Posledný záznam uvažovaného bloku je pritom vytlačený do oblasti pretečenia a bude naň ukazovať zmienený smerník. Ak je v oblastu pretečenia viacero záznamov, sú zreťazené do spájaného zoznamu a smerník ukazuje na jeho začiatok.

Použitie týchto techník je potrebné zohľadniť pri operácii Look-Up. Ak už totiž identifikujeme blok, v ktorom by sa mal záznam nachádzať, môže sa stať, že budeme musieť sledovať smerník do oblasti pretečenia, a teda budeme potrebovať ďalší prístup do externej pamäti. Oblasť pretečenia sa môže skladať z viacerých blokov a keďže záznamy v oblasti pretečenia sú zreťazené do spájaných zoznamov, môže sa stať, že budeme potrebovať aj viacero prístupov do oblasti pretečenia. Pri prístupe do oblasti pretečenia sa nám napriek sekvenčnosti súboru veľmi hodí priamy prístup do vonkajšej pamäti.

Skúmajme teraz operáciu Look-Up. Keďže súbor je utriedený, môžeme ho sekvenčne prehľadávať, až kým požadovaný záznam nenájdeme, resp. nenájdeme pozíciu, kde by sa mal nachádzať. Na identifikáciu neprítomnosti záznamu v súbore teda nepotrebujeme prezrieť celý súbor. V priemernom prípade prezrieme polovicu súboru aj pri úspešnom, aj pri neúspešnom hľadaní. Hovoríme o *lineárnom vyhľadávaní.*

Ak teraz opäť využijeme priamy prístup do vonkajšej pamäti, môžeme použiť dobre známe *binárne vyhľadávanie.* Najprv pristupíme doprostred súboru a zistíme, v ktorej polovici súboru by sa mal požadovaný záznam nachádzať. Ďalej pokračujeme len s takto získanou polovicou. Pristúpime do jej stredu, opäť zistíme, v ktorej časti by sme mali pokračovať atď. Potenciálna oblasť, v ktorej by sa mal hľadaný záznam nachádzať, sa tak s každým prístupom zmenší na polovicu. V najhoršom prípade teda potrebujeme počet prístupov logaritmický od dĺžky súboru. Výsledok je prirodzene rovnaký aj v prípade úspešného, aj v prípade neúspešného hľadania.

Ešte lepšie výsledky prináša *interpolačné vyhľadávanie*. Použiť sa dá vtedy, ak máme predstavu o pravdepodobnostnej distribúcii hodnôt kľúča. Interpolačné vyhľadávanie prebieha rovnako ako binárne, avšak nepristupujeme doprostred zostávajúcej časti súboru, ale rozdelíme ju v štatisticky danom pomere. Ak napr. vieme, že ľudí s menom začínajúcim písmenom zo začiatku abecedy nie je veľa, môžeme pri takom mene začať vyhľadávanie nie rozdelením súboru napoly, ale rozdelením na 1/4 a 3/4. Binárne vyhľadávanie je potom špeciálnym prípadom interpolačného, keď sú všetky hodnoty kľúčov rovnako pravdepodobné.

Problematika operácií Delete a Update je podobná ako v prípade haldy. Podstatnú časť ich časovej náročnosti tvorí úvodné vykonanie operácie LookUp.

Záverom poznamenajme, akú výhodu nám poskytli sekvenčné súbory s priamym prístupom.[8] Využili sme ich pri priamom prístupe na koniec súboru, do oblasti pretečenia a najmä pri binárnom vyhľadávaní v utriedenom súbore.

1.5 Indexovo-sekvenčné súbory

Uvažujme teraz možnosť vytvorenia pomocných (zväčša v samostatných súboroch umiestnených) údajových štruktúr, ktoré by urýchlili prístup k záznamom. Pridaním takejto pomocnej štruktúry — *indexu* — k sekvenčne utriedenému súboru vznikne indexovo-sekvenčná schéma organizácie súboru.[9] Cieľom je urýchlenie vyhľadávania, musíme však počítať s tým, že budeme musieť navyše udržiavať index. Praktickým príkladom indexovo-sekvenčného súboru je prekladový slovník medzi dvoma jazykmi. V ňom sú totiž slová utriedené podľa abecedy (čo je primárny sekvenčný súbor) a na každej strane sa hore nachádza slovo, ktoré je na strane prvé (index). Ak chceme v takom slovníku niečo nájsť, najprv v ňom listujeme sledujúc slová na stranách hore a keď nájdeme patričnú stranu, prezrieme ju a nájdeme v nej hľadané slovo. Niekedy index ani nemusí obsahovať celé slová, stačia niekoľkopísmenové začiatky slov — tak je to napr. v telefónnom zozname.

Spomínaný dvojúrovňový prístup — najprv hľadať v indexe a až potom v primárnom súbore dát — je pre indexovo-sekvenčnú organizáciu dát typický. Primárny súbor dát je utriedený podľa hodnôt primárneho kľúča. Index k nemu je zväčša samostatný súbor, ktorého záznamy sú tvaru „(hodnota kľúča, smerník do primárneho súboru)“. Index obsahuje jeden takýto indexačný záznam pre každý blok primárneho súboru. Indexačný záznam pritom obsahuje hodnotu kľúča záznamu, ktorý je v uvažovanom bloku primárneho súboru v zotriedení prvý a smerník na začiatok tohto bloku. Niekedy je možné, aby indexačný záznam neobsahoval celé hodnoty kľúča, ale len ich pre prevádzku dostačujúce časti (podobne ako v prípade telefónneho zoznamu).

[8] názov je vlastne paradoxný

[9] ISAM — index-sequential access method

Techniky práce s indexovo-sekvenčným súborom sú veľmi podobné technikám pre kľúčovo-sekvenčné súbory. Dodanie indexu však urýchľuje prístup. Pretože primárny súbor je utriedený, uvažujeme na uľahčenie operácie Insert podobne ako pri kľúčovo-sekvenčných súboroch s oblasťou pretečenia a jej offline reorganizáciou, ak nemožno kolíziu pri Inserte riešiť lokálnymi zmenami v súbore (napr. presunom niektorých záznamov do iných blokov).

Operáciu Look-Up začíname hľadaním v indexe, či už lineárnym, binárnym alebo interpolačným vyhľadávaním a po detekcii príslušného indexačného záznamu pristúpime do bloku primárneho dátového súboru. Pri operáciách Insert, Delete, Update treba navyše pamätať na to, že niektoré úpravy vyžadujú príslušné úpravy v indexovom súbore.

Urýchlenie pri vyhľadávaní je prirodzené. Index je totiž omnoho menší než primárny súbor. V ideálnom prípade môže byť natoľko malý, že ho možno umiestniť do operačnej pamäti a vyhľadávanie realizovať v nej.

Ak zvážime prípad pripichnutých záznamov, môže sa reorganizácia súboru značne skomplikovať. Záznamy totiž nemožno ľubovoľne presúvať, a tak je utriedenie primárneho súboru problematické. Určité riešenie prináša zoskupiť bloky do väčších celkov — *bucketov*. Indexujú sa potom nie bloky, ale buckety. Navyše upustíme od podmienky utriedenosti primárneho súboru. Budeme len požadovať, že ak K_1, K_2 sú dve po sebe idúce hodnoty kľúča v indexe, tak v buckete, na ktorý ukazuje index od K_1 sú práve záznamy s hodnotou kľúča medzi K_1 a K_2, čo umožňuje používať index tak, ako je zvykom. Súborové operácie sa teraz prirodzene prenášajú na buckety. Ak napríklad chceme vykonať operáciu Insert a v príslušnom buckete nie je blok s dostatkom miesta, zoberieme nový blok a bucket oň rozšírime. Keďže záznamy v rámci bucketu nie sú utriedené, môžeme pridať dodatočné smerníky, ktoré ich zreťazia podľa hodnôt kľúča, čím sa uľahčí hľadanie záznamu v buckete.

Aby sme sa iniciálne vyhli problému reorganizácie či už pri pripichnutých alebo nepripichnutých záznamoch, môžeme začať pracovať s primárnym súborom, ktorý nemá bloky zaplnené úplne. Pomerne rozumnú dobu tak k preplneniu bloku nebude dochádzať a zatiaľ teda nebude nutné používanie oblasť pretečenia ani reorganizáciu súboru.

Ak porovnáme kľúčovo-sekvenčnú schému organizácie súboru s indexovo-sekvenčnou, zistíme, že zostrojenie primárneho indexu prináša výhody, ktoré jednoznačne hovoria v prospech schémy ISAM. Len pri malých súboroch sa dá uvažovať o nákladnosti údržby primárneho indexu.

Do schémy ISAM sa niekedy zaraďujú aj tzv. *sekundárne indexy.* My sa o sekundárnych indexoch zmienime v samostatnej časti.

Zmienili sme sa, že indexový súbor je menší než primárny a za istých okolností by sa mohol dať umiestniť celý do pamäte. Opačným extrémom je situácia, keď je primárny súbor tak veľký, že dokonca aj indexový súbor je priveľký na vyhľadávanie. Prichádza do úvahy zaviesť druhú úroveň indexácie — indexáciu indexu, resp. tretiu a ďalšie úrovne. Vzniká tak *hierarchický index.*

Táto snaha môže byť motivovaná tzv. *hardwarovo závislým* indexovaním: V prípade diskov môžeme mať v prvej úrovni indexované jednotlivé bloky, pričom druhá úroveň indexácie bude indexovať cylindre a tretia povedzme disky v rámci istej množiny diskov. Znamená to, že pri vyhľadávaní najprv prístupom do indexového súboru tretej úrovne určíme disk, na ktorom sa záznam nachádza, tým sa dostaneme indexu druhej úrovne pre tento disk, čím identifikujeme príslušný cylinder a v najnižšej úrovni už nájdeme blok s hľadaným záznamom.

Uvedomme si však komplikáciu, ktorá so zavedením hierarchického indexu vzniká. Primárny súbor je sekvenčný a utriedený. Operácie s ním prinášajú problémy súvisiace s utriedením (napr. preplnenie bloku a následné umiestnenie záznamu do oblasti pretečenia). Hovorili sme, že zmeny v primárnom súbore sa musia prejaviť zmenami v indexe a nepokladali sme rozšírenie o zmeny v indexe za podstatné sťaženie údržby. Aj index prvej úrovne je však v našom ponímaní sekvenčný utriedený súbor so všetkými komplikáciami úprav v ňom. Zmeny v ňom sa musia prejaviť zmenami v indexe druhej úrovne atď. Zatiaľ čo robiť úpravy v jednoúrovňovej indexácii nebolo príliš problematické, údržba hierarchického indexu je s pribúdajúcimi úrovňami čoraz ťažkopádnejšia a náročnejšia.

Schéma hierarchických indexovo-sekvenčných súborov[10] sa svojou povahou začína čoraz viac podobať stromovej štruktúre. Jej sekvenčnosť je pritom daná (čoraz menej adekvátnou) požiadavkou sekvenčnosti a utriedenosti použitých súborov vyžadujúcou potenciálne off-line reorganizácie. Je preto ďalším logickým krokom (aspoň čiastočne) sa jej zbaviť. Najmä snaha robiť úpravy súboru len on-line je rozumná. Dostávame sa tak k stromovým súborom.

1.6 Stromy ako dátové štruktúry

Aby sme sa mohli zaoberať aplikáciami stromových štruktúr na efektívnu organizáciu súborov, musíme sa s príslušnými štruktúrami oboznámiť najprv ako s dátovými štruktúrami pre prácu v operačnej pamäti. O aplikáciách pre súbory budeme hovoriť neskôr.

Zhrňme najprv terminológiu. Z teoretického hľadiska sú stromy implementáciu grafovej štruktúry — súvislého acyklického grafu.[11] O vrcholoch grafu sa hovorí ako o *uzloch.*

Terminológia týchto údajových štruktúr vychádza z časti so záhradníckych pojmov a z časti z genealogických. Každý uzol môže mať niekoľkých nasledovníkov — *synov* (angl. children) a jedného predchodcu — *otca* (angl. parent), pričom len jediný uzol predchodcu nemá — tento uzol sa nazýva *koreň* (angl. root). Uzly, ktoré nemajú žiadnych nasledovníkov sú *listy* (angl. leaves). Hrany explicitne zobrazujú vetvenie stromu, aj keď strom sa zvyčajne zakresľuje s koreňom navrchu a listami dole. Uzly stromu sú tým rozčlenené do *úrovní* (angl. levels): koreň je jediný strom nultej úrovne, jeho synovia sú uzly prvej úrovne, synovia týchto synov uzly druhej úrovne atď. Úroveň s najväčším číslom, na ktorej má strom ešte uzly sa nazýva jeho *výškou* (angl. height). Výška stromu často figuruje ako parameter pri odhadoch časových zložitostí. O uzloch, ktoré majú spoločného otca, sa hovorí ako o *súrodencoch* alebo *bratoch* (angl. siblings). Pod *vetvou* stromu rozumieme ľubovoľnú cestu od koreňa k listu. Ak je počet synov vrchola stanovený na najviac q, hovoríme o *q-árnom* strome. Ak je tento počet stanovený na práve q, hovoríme o *striktne q-árnom* strome. Elementárnym prípadom sú binárne stromy pre $q = 2$. Ak počet synov uzlov nie je ohraničený, hovoríme o *všeobecnom* strome.

Budeme sa zaoberať len smerníkovou implementáciu stromov, teda situáciou, keď vzťahy medzi uzlami sú vyjadrené pospájaním pomocou smerníkov. Ak pritom ide o q-árny strom, môže každý uzol obsahovať pole smerníkov na jednotli-

[10] niekedy označovaná ako HISAM — hierarchical index-sequential access method

[11] Ak chceme byť presní v grafárskej terminológii, stromy, ako údajové štruktúry, zodpovedajú grafárskemu pojmu „pestovaný koreňový strom“.

vých synov. Ak ide o všeobecný strom, môžu byť synovia zviazaní do spájaného zoznamu a uzol obsahovať len smerník na jeho začiatok. Každý uzol s výnimkou koreňa môže ďalej obsahovať smerník na svojho otca. Pri zmenách v strome je potrebné adekvátne upravovať všetky tieto smerníky.

Poznamenajme, že pri použití niektorých rekurzívnych algoritmov, môže byť informácia o otcovi redundantná. Pre účely rekurzívnych algoritmov je možné strom definovať ako rekurzívnu údajovú štruktúru.

Nech už nesie strom akúkoľvek informáciu, naším cieľom je rýchle vyhľadávanie, resp. údržba stromu — konkrétny postup závisí od organizácie stromu. Spomeňme námatkovo koncept *binárneho vyhľadávacieho stromu*. Binárny vyhľadávací strom je striktne binárny strom. Každý uzol v ňom nesie práve jeden údaj — pre jednoduchosť predpokladajme, že ide o číselné údaje. O údajoch v strome sa hovorí ako o *kľúčoch* Organizácia uzlov do stromu je taká, že ak uzol N nesie hodnotu x, tak podstrom, ktorého koreňom je ľavý syn uzla N, obsahuje len uzly s hodnotami kľúča menšími alebo rovnými ako x a podstrom, ktorého koreňom je pravý syn uzla N, obsahuje len uzly s hodnotami kľúča väčšími ako x. Nebudeme sa zaoberať tým, ako uzly do stromu pridávať a odoberať tak, aby táto podmienka bola splnená. Na to odkazujeme čitateľa na učebnice o dátových štruktúrach. Všimnime si len, ako je tým uľahčené vyhľadávanie.

Vyhľadávanie znamená začať pristupovať k stromu od jeho koreňa a postupovať smerom k listom — teda prechádzať najviac jednou vetvou — a v priebehu tohto prechodu nájsť hľadanú informáciu. Konkrétne pri hľadaní čísla y, začneme koreňom. Ten nesie hodnotu x. Ak $x = y$, hľadaný kľúč sme našli. Ak $x \neq y$, pokračujeme v hľadaní v prípade $y \leq x$ v ľavom podstrome a v prípade $y > x$ v pravom podstrome, až kým hľadanú hodnotu nenájdeme. Ak sme došli až do listu a v postupe nemožno pokračovať, hľadaný kľúč sa v strome nenachádza.

Požiadavka postupného prehľadania najviac jednej vetvy zostáva v platnosti pri takmer každej rozumnej organizácii stromu. Podstatné na postupe vyhľadávania potom je, že hľadaný uzol nájdeme v najhoršom prípade v čase zodpovedajúcom dĺžke najdlhšej vetvy stromu, konkrétne v čase $h + 1$, kde h je výška stromu. Našou snahou preto je výšku stromu minimalizovať. To pri danom množstve dát, teda pri danom počte uzlov značí, že strom musí byť „čo najkošatejší", teda musí rásť do šírky. Ideálny je *úplný binárny strom*, teda striktne binárny strom, ktorého všetky listy ležia na tej istej úrovni. Ak má takýto strom výšku h, tak má $n = 2^h - 1$ uzlov, teda $h = \lceil \log(n + 1) \rceil$. Opačný extrém je strom obsahujúci len jedinú vetvu (*degenerovaný strom*), teda spájaný zoznam. Pri počte uzlov n má výšku $h = n$. V prvom prípade bude teda vyhľadávanie v najhoršom prípade logaritmické od počtu kľúč, kým v druhom lineárne (čo je sekvenčný prípad). Snažíme sa preto tendenciu k linearite potláčať a čo najviac podporiť košatosť stromu.

Nemôžeme si však dovoliť po každej operácii so stromom jeho úpravu na (skoro) úplný binárny strom — to by bolo príliš nákladné. Zväčša sa preto na organizáciu stromu uvalia nejaké dodatočné podmienky, ktoré zabezpečia, že strom je v istom zmysle *vyvážený*. Nepracujeme teda s úplnými stromami, ale vyváženosť aj tak zaručuje, že budú mať logaritmickú výšku a teda vyhľadávať v nich budeme na logaritmicky veľa prístupov. Operácie vloženia kľúča do stromu a zmazania kľúča musia pritom tieto dodatočné podmienky zachovávať. Obyčajným vložením (alebo zmazaním) kľúča v strome sa tieto podmienky zväčša porušia, preto operácie vloženia a zmazania musia vykonať v strome následnú úpravu, ktorá opäť zabezpečí ich platnosť — *rotáciu.* Je pritom prirodzená požiadavka, že rotácia nesmie byť nákladanou operáciou, inak by sme to, čo získame rýchlym logaritmickým vyhľadávaním, zasa stratili pri pomalom vkladaní a mazaní. Požaduje sa preto, aby celkový čas vkladania a mazania vrátane rotácií bol celkovo tiež najviac ak logaritmický.

To, čo sme o vyvažovaní stromov doteraz povedali, nás oprávňuje predpokladať, že dodatočné podmienky uvalené na organizáciu stromu a zabezpečujúce jeho vyváženosť budú pomerne sofistikované a podobne to bude aj s rotáciami. Ak je stupeň sofistikovanosti vysoký (čo je len subjektívny názor), hovorí sa v anglicky písanej literatúre o *advanced data structures.*

Aby sme získali predstavu o výške vyvažovaných stromov, budeme uvažovať AVL-stromy, pre ktoré podmienka vyvažovania hovorí, že pre každý uzol, výška ľavého a pravého podstromu sa líši najviac o jedna. Je zrejmé, že výška AVL-stromu nemôže byť menšia ako výška úplného vyhľadávacieho stromu. To nám dáva dolný odhad výšky. Horný odhad dostaneme pre krajný prípad, splňujúci podmienku vyvažovania výška ľavého a pravého podstromu sa líši práve o jedna. Pre počet uzlov $T(h)$ takéhoto stromu o výške h platí rekurentná rovnica:

$$T(h) = T(h-1) + T(h-2) + 1 \quad \text{s počiatočnou podmienkou } T(0) = 1.$$

Riešením tejto rovnice je $T(h) = F(h+2) - 1$, kde $F(i)$ je i-té Fibonacciho číslo. Pre i-té Fibonacciho platí:

$$F(i) > \frac{1}{\sqrt{5}}\Phi^i - 1, \text{ kde } \quad \Phi = \frac{1}{2}(1+\sqrt{5}) \ \text{ je pomer zlatého rezu.}$$

Dosadením $n = T(h)$, po úprave a výpočte numerických hodnôt dostávame:

$$\lg(n+1) \leq h < 1.4404 \lg(n+2) - 0.328.$$

Znamená to, že počet operácií pre vyhľadanie sa v porovnaní s úplným vyhľadávacím stromom zväčší v najhoršom prípade 1.5 krát.

Samotný koncept binárnych vyhľadávacích stromov bol v smere vyvažovania rôznym spôsobom rozšírený mnohými autormi. Spomeňme tu AVL-stromy, RB-stromy, či 2–3-stromy.

Pre účely súborovej organizácie však musíme upustiť od podmienky, že uzol nesie iba jeden kľúč. S najväčšou pravdepodobnosťou totiž budeme chcieť, aby uzly stromu boli bloky súboru a tie nesú väčšie množstvo záznamov. Tak isto, z čisto praktických dôvodov sa nemôžeme uspokojiť len s tým, že na vyhľadanie kľúča potrebujeme logaritmicky veľa prístupov — v praxi sú dôležité aj konštanty. To nás vedie k myšlienke neobmedzovať sa len na binárne stromy, ale žiadať vyššie *stupne vetvenia* použitého stromu, teda vyššiu *aritu.* Podrobnejšie si preto popíšeme na tieto účely vhodné údajové štruktúry — B-stromy a ich modifikácie: B^*-stromy a B^+-stromy.

1.6.1 B-stromy

Uzly B-stromu nesú väčšie množstvo informácie. Konkrétne, m-cestný B-strom ($m > 2$) obsahuje vo svojom uzle $m-1$ pozícií pre kľúče a m pozícií pre smerníky na synov. m-Cestný B-strom ($m > 2$) je potom vyhľadávací strom, ktorý spĺňa nasledujúce podmienky:

1. Každý uzol má najviac m synov.
2. Každý uzol okrem koreňa a listov má aspoň $\lceil m/2 \rceil$ synov.
3. Koreň, ak nie je listom, má aspoň 2 synov.
4. Všetky listy sú na tej istej úrovni.
5. Každý nelistový uzol s k synmi obsahuje práve $k-1$ kľúčov.
6. Listový uzol obsahuje aspoň $\lceil m/2 \rceil - 1$ a najviac $m-1$ kľúčov.

Pod tým, že B-strom je vyhľadávacím stromom rozumieme, že ak má (nelistový) uzol k synov (a teda obsahuje $k-1$ kľúčov), je organizovaný v tvare $(p_0, u_1, p_1, u_2, p_2, \ldots, p_{k-1}, u_{k-1}, p_k)$, kde p_i sú smerníky na synov a u_i sú kľúče. Pritom podstrom, na koreň ktorého ukazuje smerník p_i, obsahuje len kľúče u, pre ktoré platí $u_i \leq u < u_{i+1}$, pričom $u_0 = -\infty$ a $u_k = +\infty$. (Pre zjednodušenie sme tu opäť predpokladali, že kľúče u sú číselné. Toto obmedzenie nie je podstatné, môžeme pracovať s kľúčmi ľubovoľného typu, na ktorom je zavedené totálne usporiadanie.) Ak máme zaručené, že v strome nemôže byť uložený dvakrát ten istý kľúč, môžeme podmienku upraviť na $u_i < u < u_{i+1}$.

Napriek zložitejšej definícii je myšlienka B-stromov pomerne jednoduchá: Každý uzol okrem koreňa je zaplnený aspoň na 50% svojej kapacity. Výnimku tvorí len koreň, ktorý možno zaplniť podľa potreby. Strom je ďalej úplne vyvážený — všetky vetvy majú rovnakú dĺžku. Variabilita dátovej štruktúry je v počte synov jednotlivých uzlov. Poznamenajme, že existujú rôzne definície B-stromov, definujúce buď túto istú štruktúru alebo dátovú štruktúru veľmi podobnú.[12]

Pozrime sa teraz najprv na vyhľadávanie a potom na údržbu B-stromov. Vyhľadávanie v B-stromoch je jednoduché a pridŕža sa klasickej vyhľadávacej schémy. Vyhľadávanie začneme v koreni. Postupne prehľadávame kľúče v koreni, až kým nenájdeme hľadaný kľúč alebo smerník, ktorý budeme ďalej sledovať. Postup potom opakujeme v príslušnom podstrome. Ak dorazíme do listu a kľúč sme ani tam nenašli, nenachádza sa v strome.

Zvážme, aká je výška m-cestného B-stromu, ak obsahuje n kľúčov. Najprv preskúmajme minimálny počet kľúčov v strome pri výške h. Tabuľka 1.1 uvádza minimálne počty uzlov a kľúčov danej úrovne h v m-cestnom B-strome. Pritom t označuje hodnotu $\lceil m/2 \rceil$. Z tabuľky potom dostávame, že celkovo m-cestný B-strom výšky h obsahuje aspoň

$$1 + 2(t-1)\sum_{k=0}^{h-1} t^k = 1 + 2(t^h - 1) = 2t^h - 1 = 2\lceil m/2 \rceil^h - 1$$

kľúčov. Preto

$$h \leq \log_{\lceil m/2 \rceil} \left(\frac{n+1}{2} \right)$$

V najhoršom prípade preto na vyhľadávanie potrebujeme najviac $1 + h \leq 1 + \log_{\lceil m/2 \rceil} \left(\frac{n+1}{2}\right)$ prístupov.

Skúmajme teraz údržbu B-stromov, teda operácie vloženia a zmazania kľúča. Vložiť kľúč značí najprv nájsť listový uzol, ktorý by kľúč mal obsahovať a potom

[12]Definícia u Knutha napr. uvažuje s prázdnymi fiktívnymi listami, čo definíciu trochu skracuje, ale je na úkor praktickej predstavy.

Úroveň	Minimálny počet uzlov v úrovni	Minimálny počet kľúčov v úrovni
0	1	1
1	2	$2(t-1)$
2	$2t$	$2t(t-1)$
3	$2t^2$	$2t^2(t-1)$
$\vdots$	$\vdots$	$\vdots$
h	$2t^{h-1}$	$2t^{h-1}(t-1)$

Tabuľka 1.1: Min. parametre m-cestného B-stromu.

doň príslušný kľúč vložiť. Tým sa operácia vloženia končí, pravda, ak tento listový uzol obsahuje voľnú pozíciu, teda ak obsahuje najviac $m-2$ kľúčov. Ak list obsahuje $m-1$ kľúčov, voľné miesto na uloženie nového kľúča v ňom už nie je. V takom prípade sa uzol rozdelí na dva uzly, majúce pre nepárne m po $\lceil m/2 \rceil - 1$ a pre párne m $\lceil m/2 \rceil - 1$ a $\lceil m/2 \rceil$ kľúčov. Prostredný kľúč rozdeľovaného uzla je vysunutý do vyššej úrovne. Ak aj tam uzol pretečie, proces rozdeľovania sa opakuje. Ak pretečie dokonca aj koreň, rozdelí sa a vytvorí sa nový uzol, ktorý bude koreňom.

Mazanie kľúča predstavuje opačný proces. Ak mažeme kľúč z listu a ten obsahuje dostatočný počet kľúčov, je operácia bezproblémová. Ak mažeme kľúč z listu, ktorý obsahuje len $\lceil m/2 \rceil - 1$ kľúčov, zmazaním by podtiekol. Preto najskôr preskúmame bratov uvažovaného listu. Ak niektorý z nich nie je na dolnej hranici počtu kľúčov, možno si od neho posunutím jeden kľúč požičať, čím zabezpečíme potrebný počet kľúčov v pôvodnom liste. Ak sa takáto „pôžička“ pre nedostatočný počet kľúčov v súrodencoch nemôže uskutočniť, spojí sa uvažovaný uzol (ktorý je na dolnej hranici počtu kľúčov) po zmazaní so svojím pravým alebo ľavým bratom (ktorý je tiež na dolnej hranici počtu kľúčov). K týmto kľúčom ešte pristúpi kľúč z uzla o úroveň vyššie, a spoločne sformujú nový uzol na hornej hranici počtu kľúčov. To má za následok zmazanie uzla o úroveň vyššie a postup sa rekurzívne opakuje. V maximálnom prípade sa môže zaniknúť koreň.

Treba ešte zvážiť situáciu, v ktorej primárne zmazávaný kľúč nie je v listovom uzle. V tom prípade sa najprv nahradí v strome nasledujúcim kľúčom (ktorý sa nájde ako najľavejší kľúč pravého podstromu) a potom sa vyvolá operácia zmazania tohto náhradného kľúča z listu, kde sa nachádzala.

Uvedené postupy je možné rôznymi spôsobmi lokálne vylepšovať, t.j. robiť také presuny, ktoré umožnia, že nemusíme robiť rozdeľovania a spájania uzlov.

Presný algoritmus na operácie s B-stromami uvádza napr. Wirth.

1.6.2 B^*-stromy

Nemožno poprieť efektivitu s akou B-stromy narábajú s relatívne veľkým množstvom kľúčov, avšak situáciu, v ktorej sú uzly zaplnené len na 50%[13] nemožno považovať za priaznivú. Preto boli B-stromy v rôznych smeroch vylepšené. Na väčšiu zaplnenosť uzlov dávajú dôraz B^*-stromy. Ich minimálne zaplnenie uzlov je zhruba 66%. B^*-strom je potom vyhľadávací strom, ktorý spĺňa nasledujúce podmienky:

1. Každý uzol okrem koreňa má najviac m synov.
2. Každý uzol okrem koreňa a listov má aspoň $(2m-1)/3$ synov.
3. Koreň, ak nie je listom, má aspoň 2, avšak nie viac ako $2\lfloor(2m-2)/3\rfloor+1$ synov.
4. Všetky listy sú na tej istej úrovni.
5. Každý nelistový uzol s k synmi obsahuje práve $k-1$ kľúčov.
6. Listový uzol, ak nie je koreňom, obsahuje aspoň $(2m-1)/3-1$ a najviac $m-1$ kľúčov. Ak je listový uzol zároveň koreňom, obsahuje najviac $2\lfloor(2m-2)/3\rfloor$ kľúčov.

Nebudeme sa zaoberať detailmi ani údržbou tejto dátovej štruktúry. Podstatná informácia je, že pri v princípe rovnakých operáciách vyhľadávania a údržby ako u B-stromov dosahujú B^*-stromy vyšší *faktor zaplnenia.*

1.6.3 B^+-stromy

B^+-stromy sa od B-stromov a B^*-stromov odlišujú tým, že kľúče ukladajú iba do listov. Údaje vo vnútorných uzloch sa považujú len za navigačnú informáciu, ktorá vyhľadávanie nakoniec privedie do správneho listu. Ukladanie kľúčov len do listov nemá na efektivitu stromových operácií podstatný vplyv. Listové uzly totiž tvoria väčšinu všetkých uzlov stromu.

Informáciu vo vnútorných uzloch nemusia tvoriť kompletné hodnoty kľúčov. Môžu obsahovať len na navigovanie dostačujúcu informáciu. Napr. ak sú kľúče reťazce, môže ísť len o rozlišujúce prefixy (ako v prípade telefónneho zoznamu). Ak ide o najkratšie možné jednoznačne rozlišujúce prefixy (tzv. *minimálne separátory*), hovorí sa v anglicky písanej literatúre o *simple-prefix* B^+*-trees.* Vzhľadom k navigačnej funkcii vnútorných uzlov sa B^+-stromy hodia na implementáciu indexov. O praktickej realizácii takéhoto indexu sa bližšie zmienime v ďalšom texte.

1.7 Stromové súbory

O stromových súboroch hovoríme vtedy, ak ich bloky sú zviazané do stromovej štruktúry. Najčastejšie sa pod stromovými súbormi rozumejú B-stromové súbory. Na všetky ďalej uvedené účely by sa rovnako dobre ako B-stromy dali použiť aj B^*-stromy s tým rozdielom, že majú vyšší faktor zaplnenia. Nebudeme však už o B^*-stromoch hovoriť, aplikácie si vysvetlíme na B-stromoch.

[13] Podrobná analýza ukazuje, že očakávane využitie priestoru je $\ln 2 \approx 69\%$.

Nie je prekvapujúce, že stromovo možno súbory organizovať len na pamäťových médiách s priamym prístupom.

1.7.1 B-stromová organizácia súboru

Priamočiara aplikáciu B-stromov na organizáciu súboru je veľmi jednoduchá. Bloky B-stromovo organizovaného súboru budú tvoriť uzly B-stromu. Bloky budú obsahovať jednotlivé záznamy súboru a smerníky na synovské bloky. Veľkosť blokov a záznamov určuje, koľko záznamov (a smerníkov) bude obsahovať jeden blok, teda aký bude parameter m uvažovaného B-stromu. Pretože sa akýkoľvek prístup do takéhoto súboru začína prístupom do koreňového bloku, je rozumné udržiavať koreňový uzol permanentne v pamäti, čím sa takmer vždy ušetrí jeden prístup na externé pamäťové médium. Ďalej, rozdeľovania a spájania uzlov sú náročné operácie, takže je dobré sa im vyhnúť, ak sa dá. Treba preto vhodne zvoliť iniciálny faktor naplnenia blokov súboru, aby po optimálnu dobu nebolo potrebné rozdeľovania a spájania uzlov robiť. Doporučuje sa iniciálny faktor naplnenia $\ln 2 \approx 69\%$.

Pri takto poňatej aplikácii B-stromov na organizáciu súboru si však musíme uvedomiť nasledujúce skutočnosti:

- Údržba B-stromu vyžaduje pomerne časté presúvanie záznamov z jedného uzla do druhého, teda z jedného bloku do druhého. To znamená, že uvedeným spôsobom možno organizovať len súbory s nepripichnutými záznamami.

- Záznam môže byť celkovo podstatne väčší než kľúč v ňom obsiahnutý. Ak potom hľadáme v B-stromovo organizovanom súbore záznam, pri prechode vnútornými vrcholmi sme nútení zaoberať sa záznamami, z ktorých nás zaujíma len kľúč navigujúci nás v ďalšom postupe. Navyše veľké záznamy značia, že sa ich do jedného uzla vojde menej, čo znižuje parameter m.

Riešenie druhého problému nie je náročné: Pre vyhľadávanie nie je dôležité, aby sa v uzloch nachádzali celé záznamy. Postačí, ak sa v nich budú nachádzať kľúče záznamov a s každým kľúčom smerník na dodatočnú informáciu záznamu. Dodatočná informácia potom môže byť udržiavaná v inej časti súboru, resp. v úplne inom súbore v kompaktnej podobe (teda bez zbytočných medzier). Pri vyhľadávaní teda vyhľadáme príslušný kľúč a pomocou smerníka pristúpime aj k zvyšnej dodatočnej informácii záznamu (čo je dodatočný prístup do externej pamäti).

Ako sekundárnu sme dostali možnosť dodatočnú informáciu skompaktniť. To je nezanedbateľná výhoda. Bloky predstavujúce uzly stromu totiž zväčša nie sú zaplnené úplne. Ako sme uviedli, pre optimálnu prevádzku sa B-strom iniciálne plní na 69%. Súbor je teda prirodzene pomerne fragmentovaný. Ak oddelíme dodatočnú informáciu od kľúčov a budeme ju držať v kompaktnej podobe, príslušný B-strom bude pri nezmenenom počte záznamov menší, pretože kľúče zaberajú menej miesta ako celé záznamy. To značí, že je signifikantne menší počet uzlov B-stromu, ktoré sú prirodzeným spôsobom fragmentované. Celkovo tak pri rovnako výhodnej prevádzke B-stromu súbor zaberá menej miesta. Cena, ktorú za to platíme, je ďalší prístup do externej pamäti po dodatočnú informáciu záznamu.

Za istých okolností sme však už vyriešili aj prvý uvedený problém. Oddelením dodatočnej informácie do samostatnej oblasti sa táto stala polopripichnutou. Jediný smerník ukazujúci na takúto dodatočnú informáciu je totiž smerník od kľúča v B-strome a ten máme pod kontrolou. Dodatočnú informáciu teda možno ľubovoľne kompaktovať a reorganizovať, ak pritom udržiavame zmienený smerník. Kľúč v B-strome by však aj naďalej mohol byť pripichnutý, čo by znemožnilo úpravy v B-strome. Ako riešenie nariadime implementáciu smerníka hodnotou kľúča. Teda, ak by mal nejaký aplikačný smerník ukazovať na záznam, ponesie len hodnotu kľúča tohto záznamu. Pri pristupovaní k záznamu pomocou smerníka potom vyhľadáme kľúč v B-strome a v ďalšom kroku jeho dodatočnú informáciu. Kľúče v B-strome sa tak stanú nepripichnutými za dodatočnú cenu, že pri prístupe pomocou smerníka sme nútení urobiť vyhľadávanie v B-strome. Avšak parametre B-stromu sú aj pri veľkých súboroch také dobré, že sme schopní kľúč v B-strome lokalizovať na veľmi malý počet prístupov.

Urobme ešte terminologickú poznámku. Od B-stromu nemusí byť do samostatnej oblasti vydelená len dodatočná informácia záznamov. V tejto oblasti sa totiž môžu z rôznych dôvodov vyskytovať záznamy aj celé (to znamená, že hodnota kľúča sa nachádza aj v B-strome, kde potenciálne nemusí byť celá a aj v samostatnej oblasti spolu so zvyškom záznamu). Ak je navyše uvedená oblasť uložená v samostatnom súbore, je prakticky prístupový B-strom oddelený od dát.

Opísanej štruktúre sa niekedy hovorí *hustý index*(angl. dense index). Presne *o hustom indexe* hovoríme vtedy, keď indexová štruktúra obsahuje *index smerník na každú vetu*, v opačnom prípade sa index nazýva riedky (angl. sparse index). Všetky doteraz popísané indexy boli riedke. Riedky index vyžaduje, aby dátová štruktúra bola „organizovaná“ na základe jeho hodnôt. Preto každá štruktúra má obvykle najviac jeden riedky index.

Objektom indexácie hustého indexu totiž nie sú záznamy samotné, ale dvojice (k, p), kde k je kľúč, podľa ktorého sa indexuje a p je smerník na skutočný výskyt záznamu. V konečnom dôsledku je teda indexovaný každý záznam osobitne. Treba si uvedomiť, že hustý index obsahuje každý kľúč primárneho dátového súboru zatiaľ, čo primárny index (napr. ISAM-index) obyčajne všetky kľúče neobsahuje (ISAM-index obsahuje len kľúče, ktoré sú v každom bloku prvé). Ako neskôr uvidíme, aj iné prístupové mechanizmy možno koncipovať ako husté indexy.

Jedno použitie hustého indexu je v situácii, keď umiestňovanie celých záznamov do štruktúr je nevýhodné kvôli priestorovej náročnosti a je vhodnejšie do týchto štruktúr umiestňovať len kľúče a záznamy udržiavať v samostatnom súbore. Druhé použitie hustého indexu je ako technika na odstránenie pripichnutia záznamov. Ďalej (a to je asi najdôležitejšie) — to, že záznamy sú uložené v samostatnej oblasti (resp. súbore) a to, že je indexovaný každý záznam osobitne, umožňuje, aby boli záznamy v tejto samostatnej oblasti organizované ešte podľa nejakých iných kritérií alebo úplne inou súborovou organizáciou (napr. utriedené podľa iných polí ako podľa primárneho kľúča a pod.), čo umožňuje pristupovať k nim pomocou inej výberovej stratégie než primárny kľúč.

1.7.2 B^+-stromová organizácia súboru

Videli sme, že oddelenie prístupového B-stromu a dátového súboru prinieslo určité výhody v zavedení hustého indexu. Pokúsime sa teraz podobnú myšlienku použiť na indexovanie blokov v zmysle ISAM-indexovania. Možnosť na to nám poskytujú B^+-stromy. Ako sme povedali B^+-stromy obsahujú dátovú informáciu len v listoch. Informácia vo vnútorných uzloch slúži len ako navigácia pri vyhľadávaní, teda ako forma (primárneho) indexu.

Dátový (primárny) súbor bude oddelený od prístupového B^+-stromu. Záznamy v ňom budú organizované do blokov a v rámci blokov utriedené podľa hodnôt primárneho kľúča. Ak to z nejakých príčin potrebujeme, mohol by byť primárny súbor dokonca celý utriedený podľa hodnôt primárneho kľúča, čím by sme plne skĺzli späť do ISAM-schémy. To však nie je potrebné. Jediné, čo požadujeme je, že ak niektorý blok obsahuje záznamy s kľúčmi od hodnoty K_1 po hodnotu K_2, tak už žiaden iný blok neobsahuje záznamy s hodnotami kľúča K, pre ktoré $K_1 \leq K \leq K_2$. Inými slovami, primárny súbor nemusí byť utriedený, ale konzistentný stav je len taký, ktorý dostaneme z ľubovoľne fragmentovaného utriedeného súboru ľubovoľným poprehadzovaním blokov. B^+-strom teraz slúži ako efektívny hierarchický index k primárnemu súboru. Vnútorné uzly obsahujú navigačnú informáciu a listy namiesto konkrétnej hodnoty smerníky na začiatky blokov primárneho súboru.

Ako sme už povedali, primárny súbor nemusí byť utriedený. Všimnime si však, že keby sme toto utriedenie z nejakých príčin požadovali, dostali by sme veľmi efektívnu implementáciu hierarchického indexu schémy HISAM.

Pri použití B^+-stromov je ďalej nezanedbateľná možnosť použiť namiesto štandardných B^+-stromov simple-prefix B^+-stromy. Uvedomme si ďalej, že nie každá operácia s primárnym súborom musí nutne pozmeniť B^+-stromový index. Mnohé aktualizácie si teda nebudú vyžadovať žiadne operácie nad B^+-stromom. B^+-stromová organizácia súboru preto spája výhody stromovej a sekvenčnej organizácie.

Poznamenajme ešte, že aj B^+-stromy možno použiť na vytvorenie hustého indexu tým, že budeme indexovať jednotlivé záznamy.

1.7.3 Virtuálne B-stromy

Vráťme sa ešte v krátkej poznámke k myšlienke zmienenej na začiatku tejto kapitoly — totiž k myšlienke buffrovania blokov. Použitie vyrovnávacích pamätí na udržanie viacerých blokov v pamäti súčasne sa v prípade B-stromov a ich variácií ukazuje ako veľmi výhodné. Dokonca sme navrhovali, aby blok predstavujúci koreňový uzol B-stromu bol umiestnený v pamäti permanentne. Aj niektoré nižšie položené bloky budú používané pomerne často, aj keď nie tak často ako koreňový. Má teda zmysel udržiavať vo vyrovnávacích pamätiach viacero často pristupovaných blokov pre potenciálne budúce použitia. Keď sa vyrovnávacie pamäte preplnia, vhodnou stratégiou sa niektorý blok odsunie na disk. O takejto organizácii sa v súvislosti s B-stromami hovorí ako o *virtuálnych* B-stromoch. Virtuálne B-stromy dosahujú efektívnosť zhruba porovnateľnú s technikou rozšíriteľného hašovania, o ktorej sa zmienime neskôr.

1.8 Hašované súbory

Vysokú efektívnosť prístupu sme pri stromovej organizácii súboru dosahovali vyváženosťou použitých stromových štruktúr. To zabezpečovalo dobrý „vyvážený“ čas prístupu k hľadanému záznamu. V hašovaných súboroch sa vyváženosť štruktúr nahrádza štatistickou vyváženosťou. Tá je založená na tom, že hašovacia schéma bude v priemernom prípade záznamy do súboru *distribuovať* rovnomerne.

Treba poznamenať, že štatistická vyváženosť hašovania značí len dobrý priemerný prípad, teda dobrý celkový čas práce systému. O jednom konkrétnom prípade prístupu k dátam sa nehovorí nič. Najhorší prípad prístupu preto môže byť veľmi nevyhovujúci, a tak môže byť hašovanie nevhodné pre systémy, ktoré si v každom konkrétnom prípade vyžadujú rýchlu odozvu.

Podobne ako v prípade stromov, aj hašovanie má využitie pri dátových štruktúrach v operačnej pamäti, najmä pri realizácii rozsiahlych tabuliek. V konečnom dôsledku, logický súbor vlastne nie je nič iné ako veľmi rozsiahla tabuľka. V podstate teda budeme k súboru pristupovať ako k tabuľke. O hašovaní sa niekedy hovorí aj ako o *technike rozptýlených tabuliek.*

Podstata hašovania je veľmi jednoduchá. Každému kľúču z priestoru potenciálnych kľúčov U (*univerzum*) predpíšeme „adresu“, na ktorej bude v súbore umiestnený záznam s týmto kľúčom. Triviálne by preto bolo mať rovnako veľký adresný priestor ako veľkosť univerza. V reálnom prípade však v súbore nie sú naraz umiestnené záznamy so všetkými kľúčmi, ktoré by sa v ňom nachádzať mohli. Mať preto vyhradenú pozíciu pre každý z nich by bolo obrovské plytvanie súborovým priestorom. (Aj keby sme to chceli dopustiť, v nejednom prípade by veľkosť univerza presahovala reálne možnosti externej pamäti.) Adresný priestor pre umiestňovanie záznamov je preto podstatne menší ako mohutnosť univerza. Adresa umiestnenia záznamu sa z hodnoty kľúča musí vypočítať *hašovacou funkciou.* Hovoríme o *transformácii kľúča na adresu.*

Aby sme boli konkrétni, súbor bude obsahovať M základných blokov očíslovaných $0, \ldots, M-1$. Hašovacia funkcia bude funkcia $h\colon U \to \{0, \ldots, M-1\}$ a celkom zjavne nebude prostá. O množinách záznamov, pre ktoré dáva hašovacia funkcia rovnaké hodnoty, sa hovorí ako o *triedach.* Ak budeme chcieť do súboru uložiť záznam s kľúčom k, uložíme ho do bloku s číslom $h(k)$. Ak je už však tento blok plný, hovoríme o *kolízii.* Jednotlivé prístupy k hašovaniu sa líšia najmä tým, ako riešia problematiku kolízií.

Z uvedeného vyplýva, že organizácia súboru hašovaním si vyžaduje externé pamäťové médium s priamym prístupom. Ďalej, budeme hovoriť o ukladaní záznamov do blokov, ale z rovnakých dôvodov ako u stromov, môžu celé záznamy zaberať podstatne viac miesta ako samotné kľúče, s ktorými prídeme do styku pri vyhľadávaní. Je preto namieste zvážiť možnosť použitia hašovacích štruktúr ako hustého indexu a ukladať do nich len kľúče a smerníky na skutočné výskyty záznamov.

1.8.1 Vlastnosti hašovacích funkcií

Uveďme najprv, aké požiadavky má spĺňať dobrá hašovacia funkcia:

- Hašovacia funkcia má distribuovať kľúče do tried rovnomerne.
- Hašovacia funkcia má vykryť celý adresný priestor $\{0, \ldots, M-1\}$, teda má byť surjektívna.
- Hodnoty hašovacej funkcie sa majú dať jednoducho počítať.

Rovnomerná distribúcia kľúčov do tried je podstatná pre štatistickú vyváženosť schémy a tým dobrý priemerný prístupový čas. Navyše, kumulovanie hodnôt zvyšuje riziko kolízií. Požiadavka surjektívnosti je prirodzená, aby nedochádzalo k plytvaniu priestorom. Jednoduchá vypočítateľnosť hašovacej funkcie požaduje, aby sme pri prístupe nestrácali čas zdĺhavým výpočtom transformovanej adresy.

Hašovacie funkcie možno rozčleniť podľa toho, či majú znalosť o štatistickej distribúcii kľúčov, ktoré hašujú (tzv. *distribučne závislé* hašovacie funkcie) alebo predpokladajú, že hašované kľúče sú distribuované rovnomerne (tzv. *distribučne nezávislé* hašovacie funkcie). Hašovacie funkcie sa nekonštruujú pre ľubovoľný typ prvkov univerza U. Zväčša sa predpokladá, že prvky univerza U sú čísla alebo bitové reťazce alebo ich na takýto druh možno previesť. Najmä prevod ľubovoľného dátového typu na postupnosť bitov je priamočiary. Pri bitových reťazcoch je pritom dôležitá požiadavka, aby výsledná hodnota adresy závisela od všetkých bitov kľúča (a to podľa možnosti od všetkých rovnako). Vznikajú tak aj niektoré ad-hoc metódy, ktoré dávaju štatisticky dobré výsledky, ale nie sú viditeľne podložené teóriou.

Uvedieme teraz niektoré konkrétne hašovacie techniky. Aby sa nám v ďalšom ľahšie vyjadrovalo, označme dĺžku adresy $m = \log_2 M$.

1.8.2 Distribučne nezávislé hašovacie funkcie

Logický súčet

Bitový reťazec kľúča sa rozdelí na úseky zodpovedajúce dĺžke adresy (t.j. na úseky dĺžky $m = \log_2 M$, pričom M by mala byť mocnina 2). Tieto úseky sa potom sčítajú bit po bite modulo 2 (teda operáciou exkluzívneho súčtu XOR). Súčet určuje transformovanú adresu.

Súčtová transformačná funkcia

Reťazec bitov kľúča sa opäť rozdelí na úseky zodpovedajúce dĺžke adresy m (M by opäť malo byť mocninou 2). Tieto úseky sa sčítajú obyčajným aritmetickým súčtom v dvojkovej sústave. Ako hodnota adresy sa použije m bitov súčtu s najmenšou váhou.

Metóda stredu mocniny

Kľúč sa vynásobí sám sebou a ako adresa sa použije niekoľko vopred stanovených bitov zhruba v strede výsledku. Vychádza sa pritom z toho, že tieto bity závisia od všetkých bitov kľúča.

Metóda delenia

V tejto metóde sa predpokladá, že sú kľúče (prvky univerza U) číselné. Hašovacia funkcia je určená predpisom:

$$h(k) = k \bmod M$$

Aby naozaj dochádzalo k rovnomernému rozdeľovaniu adries, požaduje sa všeobecne, aby M bolo prvočíslo.

Linova transformačná metóda

Reťazec bitov reprezentujúcich kľúč rozdelíme na úseky takej dĺžky, aby sme číslo reprezentované úsekom mohli interpretovať ako číslicu p-árnej sústavy. Takto získané číslo v p-árnej sústave potom prevedieme do 10-kovej sústavy a ako transformovanú adresu vezmeme zvyšok, ktorý dostaneme jeho vydelením číslom q^t, pričom p, q, t spĺňajú podmienky: p a q sú nesúdeliteľné, $p = q + 1$, t je kladné celé také, že q^t aproximuje počet adries, ktoré sú k dispozícii.

Príklad: Ak transformujeme kľúč $k = 975$ a máme k dispozícii 48 adries, zvolíme $p = 8$, $q = 7$ a $m = 2$. Číslo 975 je zapísané binárne 100101110101, čo po rozdelení na 3-ice (pretože $2^3 = 8 = p$) dáva $(4565)_8$. Pritom $(4565)_8 = (2421)_{10}$. Transformovaná adresa kľúča 975 preto bude $20 = 2421 \bmod 49$.

Multiplikatívna transformačná funkcia

Multiplikatívna transformačná funkcia je jednou z najefektívnejších. Nech A je číslo z intervalu $(0, 1)$. Hašovaciu funkciu h definujeme predpisom

$$h(k) = \lfloor M\big((k \cdot A) \bmod 1\big)\rfloor$$

kde $x \bmod 1 = x - \lfloor x \rfloor$, teda zlomková časť čísla x. Túto funkciu možno dobre realizovať hardwarovo, keď číslo A chápeme ako podiel $\frac{A'}{w}$, kde $w = 2^{w'}$ reprezentuje použitú presnosť (w býva zvyčajne „najväčšie v počítači zapamätateľné“ číslo). Poznamenajme, že je vhodné zvoliť A' a w nesúdeliteľné. Operáciu

$$\left\lfloor M\left(\left(k \cdot \frac{A'}{2^{w'}}\right) \bmod 1\right)\right\rfloor$$

teraz možno realizovať tak, že číslo k vynásobíme číslom A'. Z výsledku použijeme len posledných w' bitov, ostatné ignorujeme. Tým sme dostali číslo $\big((k \cdot A) \bmod 1\big)$ s presnosťou na w' bitov. Získané číslo teraz vynásobíme číslom M a posledných w' bitov ignorujeme. Tým sme vlastne zrealizovali vonkajšiu celú časť.

Ak je navyše m mocninou 2, napr. $M = 2^p$, je zmienené násobenie len bitovým posunom a postup možno zhrnúť nasledovne: Číslo k vynásobíme číslom A'. Vo výsledku identifikujeme posledných w' bitov a použijeme prvých p z nich ako transformovanú adresu.

Často sa za číslo A volí pomer zlatého rezu, teda

$$A = \frac{\sqrt{5} - 1}{2}$$

Multiplikatívna transformačná funkcia sa vtedy nazýva Fibonacciho transformačnou funkciou.

Polynomiálna transformačná funkcia

Polynomiálna transformačná funkcia rozširuje metódu delenia tým, že namiesto okruhu celých čísel sa numerické operácie budú vykonávať v okruhu polynómov. Kľúč $k = (k_{n-1}k_{n-2}\dots k_0)$ budeme reprezentovať polynómom $K(x) = k_{n-1}x^{n-1} + \cdots + k_1x + k_0$ nad vhodným konečným poľom F. Ten budeme deliť pevne zvoleným polynómom $P(x)$. Dostaneme tak zvyšok

$$K(x) \bmod P(x) = h_{s-1}x^{s-1} + \cdots + h_1x + h_0$$

Hašovaciu funkciu popíšeme vzťahom

$$h(k) = h_{s-1}q^{s-1} + \cdots + k_1 q + k_0$$

kde q je počet prvkov poľa F. Veľkosť priestoru adries je v tomto prípade $M = q^s$.

Z algebry je známe, že konečné pole existuje len pre počet prvkov $q = p^t$, kde p je prvočíslo a toto pole je izomorfné s Galoisovým poľom $GF(q)$. Problematická a náročná je len voľba polynómu $P(x)$. Teória kódovania pritom za istých okolností dokáže pri voľbe polynómu $P(x)$ ako vhodného ireducibilného polynómu zaručiť, že kľúče hašované na tú istú adresu (teda kľúče tej istej triedy) sa líšia v aspoň vopred predpísanom počte bitov.

1.8.3 Distribučne závislé hašovacie funkcie

Tieto techniky predpokladajú, že máme v nejakej podobe znalosti o štatických vlastnostiach a distribúcii transformovaných kľúčov. Dobré výsledky sa dosahujú aj urobením štatistickej analýzy na vzorke kľúčov.

Znaková analýza

Metóda sa používa, ak možno kľúč charakterizovať ako reťazec. Urobením štatistickej analýzy na vzorke kľúčov zistíme, že znaky na niektorých pozíciách reťazca kľúče distribuujú rovnomerne a niektoré sa vymykajú priemeru. Ako adresu potom zoberieme patričný počet tých pozícií reťazcov, ktoré majú rovnomernú distribúciu.

Interpolačné hašovanie

Táto technika je založená na vete z teórie pravdepodobnosti:

> Ak $X > 0$ je spojitá náhodná premenná a $F(x)$ je jej distribučná funkcia, tak náhodná premenná $F(X)$ má rovnomerné rozdelenie pravdepodobnosti na intervale $\langle 0, 1)$.

Pre bližší popis použitých pojmov odkazujeme čitateľa na ľubovoľnú základnú učebnicu teórie pravdepodobnosti.

V technike interpolačného hašovania budeme predpokladať, že hašované kľúče sú číselné. Znalosť distribučnej funkcie $F(x)$ (teda znalosť pravdepodobnosti, že $X < x$) pritom značí znalosť distribúcie hašovaných kľúčov. Hašovaciu funkciu teraz popíšeme vzťahom

$$h(k) = \lfloor M \cdot F(k) \rfloor$$

Tento druh hašovania je mimoriadne významný — zachováva totiž usporiadanie, a to v nasledujúcom zmysle: Ak $k_1 \le k_2$, tak $h(k_1) \le h(k_2)$. Navyše ak $h(k_1) < h(k_2)$, tak $k_1 < k_2$. (Tieto vlastnosti sú dané skutočnosťou, že distribučná funkcia $F(x)$ je na celom obore definície neklesajúca.)

1.8.4 Riešenie kolízií oddeleným reťazením

Oddelené reťazenie (angl. separate chaining) je najjednoduchšou metódou riešenia kolízií. V prípade, že ku kolízii dôjde, teda keď chceme do súboru vložiť záznam s hodnotou kľúča k, ale blok číslo $h(k)$ je už plný, zoberieme nový blok a prireťazíme ho k bloku číslo $h(k)$ do tvaru spájaného zoznamu. V konečnom dôsledku ide o druh oblasti preplnenia, a to samostatnej pre každý blok $0, \ldots, M-1$. Každý s blokov číslo $0, \ldots, M-1$ teda udržiava smerník na začiatok spájaného zoznamu ďalších blokov.

Vyhľadávanie záznamu s kľúčom k teraz možno popísať ako vyhľadávanie v záznamu s kľúčom k v spájanom zozname číslo $h(k)$. Vkladanie záznamu s kľúčom k nás najprv núti urobiť vyhľadávanie a v prípade, že záznam v súbore ešte neexistuje, umiestnenie záznamu do posledného bloku spájaného zoznamu s číslom $h(k)$ (ak je posledný blok preplnený, zoberieme nový a prireťazíme ho na koniec spájaného zoznamu). Ak máme zaručené, že záznam v súbore neexistuje, mohli by sme hneď pristúpiť na koniec spájaného zoznamu (bez postupného prehľadávania) a záznam tam umiestniť. Na tieto účely sa nám môže hodiť udržiavať v blokoch číslo $0, \ldots, M-1$ smerník na prvú voľnú pozíciu na konci zoznamu. Zmazávanie pri pripichnutých záznamoch značí len nastavenie deleted-bitu, čo má z hľadiska dlhodobejšej práce hašovacej schémy nepriaznivý dopad na priemernú vyhľadávaciu dobu. Pri nepripichnutých záznamoch môžeme po odstránení záznamu urobiť kompaktovanie spájaného zoznamu.

Z hľadiska výkonu schémy je dôležitým parametrom *faktor naplnenia* $\alpha = \frac{N}{M}$, kde N je počet záznamov momentálne sa nachádzajúcich v súbore pri nepripichnutých záznamoch, resp. celkový počet záznamov do súboru vkladaných pri pripichnutých záznamoch. Ďalším dôležitým parametrom je tzv. *blokovací faktor* b — počet záznamov v bloku. Analýza pre $b > 1$ je však pomerne náročná. Pre $b = 1$ rovnomerná distribúcia záznamov značí, že v priemernom prípade budú mať jednotlivé spájané zoznamy dĺžku α. Priemerný prípad vyhľadávania vtedy je $1 + \frac{\alpha}{2}$ prístupov pri úspešnom a $e^{-\alpha} + \alpha$ prístupov pri neúspešnom hľadaní záznamu v súbore.

Metóda je dostatočne efektívna pre malé α ($\alpha < 1$) z hľadiska počtu prístupov. V tomto prípade však vyvoláva značnú nenaplnenosť blokov spájaných zoznamov. Nevýhoda sa odstraňuje tým, že prvotné bloky sa volia dostatočne veľké a bloky preplnenia primerane menšie.

Má zmysel pri realizácii jednotlivých spájaných zoznamoch uvažovať o hardwarovo podporovanej implementácii — napr. umiestňovať spájaný zoznam blokov preplnenia na tom istom cylindri disku ako prvotný blok tohto zoznamu. Pri hľadaní v zoznamoch preplnenia sa tak zabráni pohybu hlavičiek disku.

1.8.5 Riešenie kolízií spoločným reťazením

Spoločné reťazenie (angl. coalesced chaining) sa snaží odstrániť nenaplnenosť blokov. Ak je kolízií málo, sú osobitné zoznamy preplnenia pre každý prvotný blok plytvaním priestoru. Je preto možné udržiavať len jeden zoznam preplnení, pričom sú v ňom reťazené do spájaného zoznamu kolízie so všetkých prvotných blokov súčasne.

Ďalším vylepšením je nepoužívať na zoznam preplnení samostatné bloky, ale využiť na tento účel nezaplnenú časť prvotných blokov. Prvotný blok potom okrem záznamov, ktoré sú doň hašované, obsahuje aj úseky zoznamu preplnení

Hlavička bloku	Záznamy hašované do bloku	→ Voľné miesto ←	Preplnenia z iných blokov

Obr. 1.1: Využívanie prvotných blokov aj na preplnenia

```
procedure insert(k);
begin
  a:=h(k); i:=0; j:=a;
  repeat
    if T[j] contains record with key k then
       exit;                          {Záznam nájdený}
    if T[j] contains free place then
     begin
       insert record into T[j];       {Vloženie záznamu}
       exit
     end
    else
     begin
       j:=G(i,j,a,k);                 {Výpočet nového
                                       alternatívneho bloku}
       i:=i+1;
     end
  until j=a;                          {Detekcia cyklu}
  error('Table is full');
end;
```

Obr. 1.2: Algoritmus vkladania pri otvorenej adresácii

z iných blokov. Samozrejme, treba zariadiť, aby sa na zoznam preplnení používali ešte nezaplnené prvotné bloky rovnomerne, teda aby sa prvotný blok nepreplňoval pretečeniami z iných blokov. Prvotný blok potom môže byť realizovaný napr. ako na obr. 1.1.

1.8.6 Riešenie kolízií otvorenou adresáciou

Názov *otvorená adresácia* (angl. open adressing) je odvodený od toho, že pri kolízii sa hľadá alternatívne umiestnenie v inom prvotnom bloku. Po prvom zahašovaní je teda ešte určenie prvotného bloku, v ktorom bude záznam uložený, otvorené. Metóda teda podobne ako vylepšenie spoločného reťazenia využíva ešte nezaplnený priestor iných prvotných blokov. Rozdiel je však v tom, že zatiaľ čo spoločné reťazenie spája alternatívne umiestnenia do spájaného zoznamu, pri otvorenej adresácii používame na určenie alternatívneho umiestnenia výpočtové techniky. Všeobecná schéma vkladania do súboru je potom daná algoritmom na obr. 1.2.

V tomto algoritme označuje k vkladaný kľúč, a počiatočný blok, j alternatívny blok, i číslo pokusu a pole T[ ] je poľom prvotných blokov. Potrebujeme ešte došpecifikovať funkciu $G(i, j, a, k)$, ktorá určuje alternatívne bloky. Funkcia G môže mať tvar od najjednoduchších predpisov až po zložité schémy zahrňujúce ďalšie hašovacie funkcie. Číslo bloku, na ktorý je kľúč k zahašovaný pri i-tom alternatívnom pokuse sa zvykne označovať $h_i(k)$, pričom $h_0(k) = h(k)$.

Algoritmus vyhľadávania sa od algoritmu vkladania príliš neodlišuje. Jediný rozdiel je v tom, že pri detekcii voľného miesta alebo cyklu vyhľadávanie zastavíme s tým, že bolo neúspešné a záznam sa v súbore nenachádza.

Mazanie zo súboru technikou deleted-bitu neprináša žiadne komplikácie, problematické však je, ak chceme priestor po zmazaných záznamoch opätovne použiť. Musíme si totiž uvedomiť, že ak zmažeme záznam, ktorý nie je na konci vyhľadávacej postupnosti a neskôr budeme v súbore hľadať záznam, ktorý sa vo vyhľadávacej postupnosti nachádza až za zmazaným záznamom, vyhľadávací algoritmus najprv nájde voľné miesto po zmazanom zázname a mylne tak detekuje koniec vyhľadávacej postupnosti a neprístomnosť hľadaného záznamu. Situácia je o to komplikovanejšia, že jednotlivé vyhľadávacie postupnosti alternatívnych blokov sa môžu nepredvídateľným spôsobom krížiť a spájať. Prakticky teda nevieme povedať, kedy mažeme záznam na konci vyhľadávacej postupnosti alternatívnych blokov, čiže zmazanie je bezpečné a kedy to tak nie je.

Jedna možnosť, ako znovu použiť miesto po zmazanom zázname, je nasledovná: V prípade, že blok, z ktorého záznam mažeme, je plný, označíme ho špeciálnym symbolom (t.j. nastavením nejakého signalizačného bitu). Ak potom vyhľadávací algoritmus pri vyhľadávaní narazí na takto označený blok, nebude ho napriek prítomnosti voľného miesta považovať v reťazi alternatívnych blokov za posledný a bude v hľadaní pokračovať. Nevýhodou je ťažká detekovateľnosť možnosti blok zasa odznačiť. Môže sa potom stať, že po čase bude väčšina blokov označených a prevádzka súboru sa stane neefektívnou. V takom prípade už nezostáva iná možnosť než súbor reorganizovať — *prehašovať*. Uvedieme ešte niekoľko možností ako zvoliť funkciu G.

Lineárne pokusy

Funkciu G volíme

$$G(i, j, a, k) = (j + c) \bmod M$$

kde c a M sú nesúdeliteľné. Často sa pod lineárnymi pokusmi rozumie rovno prípad $c = 1$. Lineárne pokusy sú najjednoduchšou možnosťou voľby funkcie G. Nevýhodou lineárnych pokusov je však *primárne zhlukovanie* (angl. primary clustering). Je to vlastnosť, že ak sa dva rôzne kľúče k_1 a k_2 zahašujú v niektorých krokoch do toho istého bloku, budú ich vyhľadávacie postupnosti alternatívnych blokov od tohto okamihu rovnaké, teda ak pre $i \neq j$ platí $h_i(k_1) = h_j(k_2)$, tak $h_{i+t}(k_1) = h_{j+t}(k_2)$ pre všetky kladné t. Prakticky to znamená, že sa vyhľadávacie postupnosti alternatívnych blokov týchto kľúčov v istom momente spojili a budú sa od tohto okamihu zhodovať aj napriek tomu, že $h_0(k_1) \neq h_0(k_2)$. Vzniká tak tendencia, že sa tieto postupnosti budú viac predlžovať. V skutočnosti tvar funkcie G v metóde lineárnych pokusov zhlukovanie priamo vyrába.

Napriek zmieneným nedostatkom sa metóda lineárnych pokusov dá implementovať za podpory hardwaru na diskoch s dobrými výsledkami. Musíme len zvoliť takú konštantu c a také rozmiestnenie blokov na disku, aby sa bloky kongruentné podľa modulu c nachádzali na tom istom cylindri. Prehľadávanie postupnosti alternatívnych blokov potom takmer nevyžaduje pohyb hlavičiek disku a prístup je napriek zhlukovaniu rýchly.

Existuje aj pojem *sekundárneho zhlukovania* (angl. secundary clustering). Je to prípad, keď $k_1 \neq k_2$, ale $h_0(k_1) = h_0(k_2)$ a vyhľadávacie postupnosti kľúčov k_1 a k_2 sa zhodujú. Je to teda prípad, keď funkcia G nezávisí od svojho argumentu a. Vyhnúť sa sekundárnemu zhlukovaniu nie je vo všeobecnosti jednoduché.

Lineárna podielová metóda

Lineárna podielová metóda je miernou modifikáciou lineárnych pokusov. Robí totiž postupné prehľadávanie, ale s premenným krokom. Funkciu G volíme nasledovne

$$G(i,j,a,k) = \begin{cases} \big(j + (k \text{ div } M)\big) \bmod M, & \text{ak } (k \text{ div } M) \neq 0, \\ (j+1) \bmod M, & \text{inak} \end{cases}$$

Premennosť kroku odstraňuje oproti metóde lineárnych pokusov primárne zhlukovanie pre kľúče nekongruentné modulo M. Je zaručené, že ak M je prvočíslo, tak prehľadáme každú pozíciu za M krokov.

Lineárna metóda s váženým prírastkom

$$G(i,j,a,k) = \big(j + i \cdot (2a+1)\big) \bmod M$$

Dá sa ukázať, že takáto voľba funkcie G takmer úplne odstraňuje primárne zhlukovanie.

Lineárna metóda s dvojitou transformáciou (dvojité hašovanie)

V tomto prípade sa na voľbu kroku používa druhá hašovacia funkcia h', od ktorej sa požaduje, aby dávala len hodnoty nesúdeliteľné s M (krok súdeliteľný s M totiž nevedie k prehľadaniu všetkých pozícií). Funkciu G teraz popíšeme

$$G(i,j,a,k) = \big(j + h'(k)\big) \bmod M$$

Metóda prvočísel

Ak prime(x) je funkcia vracajúca x-té prvočíslo, popíšeme funkciu G ako

$$G(i,j,a,k) = \big(j + \text{prime}(a)\big) \bmod M$$

Kvadratické pokusy

Všetky predošlé metódy prehľadávali alternatívne bloky lineárne, len s rôzne volenou dĺžkou kroku (závislou od kľúča, či čísla pokusu). Teraz však použijeme kvadratické prehľadanie. Volíme pritom

$$G(i,j,a,k) = \big(j + c_1 + c_2(2i+1)\big) \bmod M$$

V konečnom dôsledku preto $h_i(k) = \big(h_0(k) + c_1 i + c_2 i^2\big) \bmod M$. Pri takto sa meniacom kroku nedochádza k primárnemu zhlukovaniu.

Kvadratická podielová metóda

Kvadratická podielová metóda vylepšuje kvadratické pokusy tým, že odstraňuje aj sekundárne zhlukovanie. Funkciu G volíme

$$G(i, j, a, k) = \big(j + c_1 + c_2(k) \cdot (2i + 1)\big) \bmod M$$

kde $c_2(k)$ je vhodná funkcia. Konkrétne, pri kvadratickej podielovej metóde sa volí $h(k) = k \bmod M$ a $c_2(k) = k \operatorname{div} M = k - h(k)$. Výpočet funkcie $c_2(k)$ nás potom nestojí prakticky nič navyše.

1.8.7 Rozšíriteľné hašovanie

Riešenia kolízií, ako sme ich uviedli v predchádzajúcom texte, boli statické v tom, zmysle, že sa pri nich nemenil počet prvotných blokov. Ak bolo množstvo uložených záznamov také veľké, že ich uloženie začalo neúnosne zvyšovať faktor naplnenia α a tým zvyšovať riziko kolízií, bolo potrebné celý súbor reorganizovať, teda prehašovať pri zväčšenom počte prvotných blokov M. Ak bol naopak počet uložených záznamov primalý, bloky boli nenaplnené a zefektívnenie práce so súborom si vyžadovalo prehašovanie pri menšej hodnote parametra M. Prehašovať súbor je však závažné rozhodnutie, pretože ide o časovo veľmi náročnú operáciu. Existujú však techniky, ktoré umožňujú zmenu parametra M počas prevádzky bez nutnosti prehašovania. Takým technikám sa hovorí *dynamické hašovanie.* Spomedzi dynamických techník si uvedieme tzv. *rozšíriteľné hašovanie.*

Rozšíriteľné hašovanie je založené na možnosti rozdeliť blok v prípade preplnenia na dva (a podobne zlievania blokov pri podtečení). Do hašovacej schémy sa pridáva dodatočná štruktúra — *katalóg* blokov, alebo tiež *adresár.* Hašovacia funkcia nebude určovať blok priamo, ale určí len položku — *index* v katalógu blokov. Táto položka už potom obsahuje smerník na príslušný blok so záznamami, pričom viacero položiek môže obsahovať smerník na ten istý blok záznamov. Aby sme túto predstavu skonkretizovali, predpokladajme, že hašovacia funkcia vracia ako hodnotu číslo, ktoré budeme chápať ako bitový reťazec. Pre momentálny stav adresára je podstatný parameter d — *hĺbka adresára* — pričom $1 \leq d$. Parameter d hovorí, koľko z bitov hašovacej funkcie je pre určenie indexu v katalógu blokov podstatných. V princípe by mohlo ísť o ľubovoľných d bitov — je však rozumné voliť prvých d alebo posledných d bitov. My sa budeme zaoberať prípadom, keď je index v katalógu blokov určený poslednými d bitmi. Ak potom hašovacou funkciou určíme pre kľúč k hodnotu $h(k)$, je príslušný index v katalógu blokov $\big(h(k) \bmod 2^{d-1}\big)$. Pomocou smerníka, ktorý je v katlógu blokov na tomto indexe teraz už pristúpime do príslušného bloku. Vyššie uvedené znamená, že pri hĺbke d má katalóg 2^{d-1} pozícií.

Každý blok so záznamami zároveň udržuje informáciu o svojej hĺbke d', pričom $1 \leq d' \leq d$. Hĺbka bloku d' určuje, koľko posledných bitov hodnoty hašovacej funkcie je podstatných pre záznamy v tomto bloku. Hodnota d' môže byť menšia než hodnota d. Táto skutočnosť je daná tým, že katalóg môže obsahovať smerníky pre hĺbku d (a tým formálne rozlišovať bloky podľa posledných d bitov), ale viacero smerníkov môže ukazovať na ten istý blok záznamov. Tak sa môže stať, že skutočný rozlišovací počet bitov pre blok je menší. Ak napr. pre $d = 3$ ukazujú smerníky katalógu, zodpovedajúce koncovej 3-ici bitov 010 a 110, na ten istý blok záznamov, je pre tento blok podstatné len rozlíšenie poslednej dvojice bitov 10 a hĺbka tohto bloku je $d' = 2$.

Skúmajme, čo sa teraz stane, ak sa pokúsime vložiť do súboru záznam a smerník z katalógu blokov nás privedie do bloku B, ktorý je plný. Situáciu vyriešime, tým, že sa blok B rozdelí na dva bloky B_0 a B_1. Rozlišujeme pritom dva prípady:

(a) $d' < d$ — V tomto prípade blok B nevyužíva rozlišovaciu schopnosť katalógu úplne a ukazuje naň z neho viacero smerníkov. Polovica z nich bude odteraz ukazovať na blok B_0 a druhá polovica na blok B_1. Konkrétne, ak blok B obsahoval záznamy, ktoré mali ako poslednú d'-ticu bitov reťazec w, budú mať bloky B_0, resp. B_1 hĺbku $d'+1$ a rozlišovacie koncové reťazce $0w$, resp. $1w$. Na blok B_0 budú z katalógu ukazovať smerníky, ktoré zodpovedajú indexom končiacim $0w$ a na blok B_1 smerníky zodpovedajúce indexom končiacim $(d'+1)$-ticou $1w$. Bloky B_0, resp. B_1 budú obsahovať tie záznamy bloku B, ktorých hodnota hašovacej funkcie končí bitmi $0w$, resp. $1w$.

(b) $d' = d$ — V tomto prípade blok B využíva rozlišovaciu schopnosť katalógu úplne a ukazuje naň z neho len jeden smerník. Aby sa mohol blok B rozdeliť, musí sa najprv zväčšiť hĺbka katalógu — katalóg sa zdvojnásobí zväčšním hĺbky na $(d+1)$. Každý index s koncovými d bitmi w pôvodného katalógu sa tým rozdelí na dva indexy, totiž $0w$ a $1w$. Smerník z nového indexu $0w$ aj smerník z nového indexu $1w$ budú ukazovať na ten istý blok, na ktorý ukazoval smerník od pôvodného indexu w. Zväčšenie hĺbky katalógu pritom spôsobilo, že každý blok záznamov má teraz hĺbku ostro menšiu než je hĺbka katalógu. Vo vkladaní preto možno pokračovať rozdelením bloku B tak, ako v bode (a).

Uviedli sme tým spôsob, akým sa bloky rozdeľujú na požiadanie — teda vtedy, keď je to potrebné. Musíme ešte uviesť spôsob, akým sa zasa bloky spájajú. Treba však poznamenať, že bloky sa nemusia spájať (resp. adresár zmenšovať) hneď, ako na to vznikne príležitosť. V závislosti od nami zvolenej stratégie možno túto úpravu pozdržať pre prípad čoskorého opätovného rastu.

Zlučovanie blokov prebieha inverzne k procesu rozdeľovania. Dva susedné bloky B_0 a B_1 sa môžu zlúčiť do jedného bloku B, ak majú rovnakú hĺbku $(d'+1)$, prvý z nich obsahuje záznamy, ktorých hodnota hašovacej funkcie končí $(d'+1)$-ticou bitov $0w$ a druhý $(d'+1)$-ticou $1w$. Vzniknutý blok B bude mať hĺbku d' a bude obsahovať záznamy, ktorých hodnota hašovacej funkcie končí d'-ticou bitov w. Budú naň ukazovať všetky smerníky, ktoré predtým ukazovali na bloky B_0, resp. B_1. Pravdaže, bloky B_1 a B_2 môžeme zlúčiť len vtedy, ak sa celkový počet v nich obsiahnutých záznamov vojde do nového bloku B.

Z konštrukcie katalógu blokov vyplýva, že ak sa stane, že na každý blok záznamov ukazujú z katalógu hĺbky $(d+1)$ aspoň dva smerníky,[14] tak pre každú d-ticu bitov w platí, že smerníky z indexov $0w$ a $1w$ ukazujú na ten istý blok záznamov. Katalóg sa teraz môže dvakrát zmenšiť na hĺbku d tým, že indexy $0w$ a $1w$ splynú pre každé w do nového indexu w. Smerník tohto nového indexu bude pritom ukazovať na ten istý blok, na ktorý ukazovali oba smerníky od pôvodných indexov $0w$ a $1w$.

[14] Ľahko vidieť, že v každom okamihu ukazuje na každý blok záznamov počet smerníkov, ktorý je mocninou 2.

Poznamenajme ešte, že ak je katalóg malý, je vhodné ho udržiavať v operačnej pamäti. Ak katalóg nie je umiestnený v operačnej pamäti potrebujeme na dosiahnutie hľadaného záznamu 2 prístupy: jeden do katalógu a jeden do bloku záznamov. Uveďme ďalej, že očakávaný faktor naplnenia blokov je pri rozšíriteľnom hašovaní $\ln 2 \approx 69\%$.

1.9 Dotazy

V tejto časti sa budeme zaoberať databázovým aspektom súborov. Dôležitá akcia, ktorá sa nad súborom vykonáva, je vyhodnotenie *dotazu*. Pod dotazom budeme rozumieť ľubovoľnú totálnu vyčísliteľnú funkciu, ktorá aktuálnemu obsahu súboru (t.j. *stavu* súboru) priradí *odpoveď* na dotaz, t.j. množinu záznamov, ktoré dotazu vyhovujú. Uvedená definícia dotazu nám umožňuje formulovať v princípe takmer ľubovoľný dotaz, t.j. ľubovoľnú otázku o záznamoch súboru. Našou snahou je, aby systém dokázal na dotaz odpovedať čo najrýchlejšie, teda aby nebolo potrebné čítanie celého súboru, prípadne viacnásobné čítanie alebo dokonca backtrackovanie cez súbor. Je však nemysliteľné, aby systém podporoval skutočne ľubovoľné dotazy. Zameriame sa preto len na podporu základných, často sa vyskytujúcich typov dotazov.

- *Dotaz na úplnú zhodu.* Sú zadané všetky hodnoty polí záznamu, ktorý hľadáme. V princípe tak ide iba o Áno/Nie dotaz, teda o zistenie, či sa daný záznam v súbore nachádza alebo nie.
- *Dotaz na čiastočnú zhodu* (angl. partial-match query). Sú zadané len niektoré hodnoty polí záznamu. Odpoveďou sú všetky záznamy, ktoré majú hodnoty polí nastavené zadaným spôsobom.
- *Dotaz na intervalovú zhodu* (angl. range-query). Pre každé alebo len niektoré polia je zadaný interval hodnôt (ide teda o úplnú alebo čiastočnú intervalovú zhodu). Odpoveďou sú všetky záznamy, ktoré majú hodnoty polí v zadaných intervaloch.
- *Zložený dotaz.* Ide o dotaz zložený z dotazov predošlých typov pomocou logických spojok (AND, OR) a pod.

1.9.1 Dotazy na úplnú zhodu

Dotaz na úplnú zhodu je v skutočnosti prístupom do súboru podľa hodnoty primárneho kľúča. Práve preto — až na malé výnimky — nie je podstatné, či sú pri dotaze na úplnú zhodu naozaj zadané všetky polia záznamu. Keďže kľúč identifikuje záznam jednoznačne, postačuje plne zadať polia kľúča.[15] Všetky prístupové mechanizmy, ktoré sme doteraz skúmali, t.j. primárny index, B-stromy i hašovanie umožňujú práve prístup pomocou primárneho kľúča a riešia teda problematiku dotazov na úplnú zhodu. Aby sme umožnili vyhodnocovanie aj iných typov dotazov, budeme sa v ďalšom zaoberať možnosťou dodania

[15] V tejto súvislosti poznamenajme, že niekedy sa od primárneho kľúča nevyžaduje, aby identifikoval záznam jednoznačne, ale aby identifikoval nejaké malé množstvo záznamov. Táto zmena by ale vyžadovala revidovanie všetkých doteraz zmienených metód a techník. My sa naďalej budeme držať toho, že primárny kľúč určuje záznam jednoznačne.

ďalších pomocných štruktúr na tieto účely, resp. možnosťou modifikácie štruktúr vyhľadávajúcich podľa primárneho kľúča tak, aby umožňovali aj iné typy dotazov.

1.9.2 Dotazy na čiastočnú zhodu

V princípe nám ide o možnosť vyhľadávať v súbore podľa hodnôt poľa, ktoré neidentifikuje záznam jednoznačne. Je však špecifikovaná presná hodnota tohto poľa a naším cieľom je nájsť záznamy, ktoré majú toto pole naplnené uvedenou hodnotou. Typický príklad takého dotazu je

```
Zamestnanie='Študent'
```

Odpoveďou naň bude mnoho záznamov. Kľúču, ktorý neidentifikuje záznamy jednoznačne (napr. `Zamestnanie` v predchádzajúcom príklade) sa hovorí *sekundárny* kľúč a prístupovému indexu (či už sekvenčnému, stromovému alebo hašovanému), ktorý umožňuje vyhľadávanie podľa sekundárneho kľúča sa hovorí *sekundárny index.*

Hustý index ako sekundárny index

Jedna z možností, ako organizovať sekundárny index, je zostrojiť hustý index vyhľadávajúci podľa sekundárneho kľúča a nebrať ohľad na to, že sekundárnym kľúčom záznamy jednoznačne určené nie sú. Táto technika je vhodná najmä vtedy, ak každou konkrétnou hodnotou sekundárneho kľúča je určený len veľmi malý počet záznamov. Vo vzniknutej štruktúre (či už sekvenčnej, stromovej alebo hašovanej) sa potom nachádzajú niektoré hodnoty sekundárneho kľúča viackrát. Pri každom výskyte hodnoty sekundárneho kľúča sa nachádza smerník na skutočný záznam dátového súboru. Tento prístup si však vyžaduje modifikáciu doteraz uvedených algoritmov.

Majme napr. sekundárny index obsahujúci dvojice (k, p), kde k je hodnota sekundárneho kľúča a p smerník na záznam a nad ním máme ako prístupovú štruktúru zostrojený ISAM-index. Keď hľadáme záznamy s hodnotou sekundárneho kľúča X_2 a v ISAM-indexe za sebou nasledujú položky X_1 a X_2 indexujúce bloky B_1 a B_2, neznamená to, že máme hľadať v bloku B_2 (čo by bola pravda, keby kľúč bol primárny) — musíme sa pozrieť aj do bloku B_1. Odkazy na záznamy s hodnotou sekundárneho kľúča X_2 sa totiž môžu nachádzať aj na konci bloku B_1.

Invertované zoznamy a invertované súbory

Ak môže byť v súbore záznamov s tou istou hodnotou sekundárneho kľúča podstatne viac, náročnosť vyhľadávania podľa sekundárneho kľúča predošlou metódou značne stúpne. Bude totiž vyžadovať značné úsilie vyhľadať všetky výskyty sekundárneho kľúča. Vtedy je výhodnejšie, ak sekundárny index obsahuje pre každú možnú hodnotu sekundárneho kľúča spájaný zoznam smerníkov na záznamy s touto hodnotou sekundárneho kľúča. Takémuto zoznamu sa hovorí *invertovaný zoznam* a sekundárnemu indexu obsahujúcemu invertované zoznamy *invertovaný súbor.*

Používanie invertovaných zoznamov je bežné aj v každodennej praxi. Ako príklad možno uviesť register pojmov na konci knihy. Ku každému pojmu (t.j.

hodnote sekundárneho kľúča) je uvedený zoznam strán, na ktorých sa tento pojem spomína (t.j. invertovaný zoznam). Podobne, invertovaným súborom ku katalógu kníh knižnice, zoradenému podľa identifikačného čísla, je menný katalóg, zoradený podľa mien autorov, v ktorom sú pre každého autora uvedené identifikačné čísla (resp. názvy) kníh, ktoré od neho knižnica vlastní. Invertované súbory teda neexistujú samy o sebe. Sú iba odzrkadlením obráteného — „invertovaného" — pohľadu na dáta obsiahnuté v primárnom dátovom súbore.

Zostáva zodpovedať otázku, akým spôsobom sú invertované zoznamy v invertovanom súbore organizované. Triviálna možnosť je uchovávať každý invertovaný zoznam v osobitnom súbore. Názov súboru by potom musel vyjadrovať, o ktorý sekundárny kľúč a o ktorú jeho hodnotu ide v invertovanom zozname tohto súboru. To je prirodzene veľmi nepraktické.

Obyčajne sa preto všetky invertované zoznamy nachádzajú v spoločnom invertovanom súbore. Výsledkom nášho dotazu je však vždy celý invertovaný zoznam. Invertované zoznamy preto musia byť v invertovanom súbore umiestnené tak, že keď už patričný zoznam lokalizujeme, nesmie nám jeho celé prečítanie robiť vážnejšie časové komplikácie. Každý invertovaný zoznam by preto mal byť sústredený na jednom mieste invertovaného súboru, ideálne celý v tom istom bloku, aby sme ho mohli získať na čo najmenší počet prístupov.

Musíme ešte uviesť, ako v invertovanom súbore lokalizovať invertovaný zoznam, ktorý nás zaujíma. To už však nie je náročné. Môžeme použiť ľubovoľnú organizáciu známu z primárnych indexov (kľúčovo-sekvenčnú, indexovo-sekvenčnú, stromovú, či hašovanú), pričom v príslušných štruktúrach sa každá hodnota sekundárneho kľúča bude nachádzať len raz. Spolu s ňou sa v takejto štruktúre bude nachádzať smerník na začiatok invertovaného zoznamu. Ak potom chceme vyhľadávať podľa hodnoty sekundárneho kľúča, najprv v uvedenej štruktúre vyhľadáme sekundárny kľúč (tentoraz už bežnými algoritmami), tým získame smerník na začiatok invertovaného zoznamu. Invertovaný zoznam potom obsahuje smerníky na záznamy, ktoré nás zaujímajú. Samotné invertované zoznamy môžu byť uchovávané spolu s takouto vyhľadávacou štruktúrou alebo v osobitnom súbore. Ak je zaručené, že je maximálny počet prvkov invertovaného zoznamu zhora ohraničený nie príliš veľkým číslom, môže byť invertovaný zoznam implementovaný poľom.

Sekundárne indexy a smerníky

Súhrnne, k týmto dvom metódam vyslovme poznámku o implementácii smerníkov. Sekundárny index by podľa nich mal obsahovať smerníky na záznamy dátového súboru. Keby však skutočne šlo o smerníky, bolo by to veľmi nepraktické. To by totiž záznamy dátového súboru pripichlo. Musíme si tiež uvedomiť, že sekundárnych indexov môže byť v závislosti od typu dátového súboru aj pomerne mnoho. Každá zmena v dátovom súbore by potom vyžadoval príslušnú zmenu vo všetkých sekundárnych indexoch, teda na mnohých miestach. Výhodnejšie je preto implementovať smerníky zo sekundárneho indexu na záznamy dátového súboru len ako primárne kľúče. Pristúpiť zo sekundárneho indexu k záznamom potom značí vykonať vyhľadávanie v dátovom súbore podľa hodnoty primárneho kľúča. Tento postup má ešte jednu výhodu.

Ak napr. záznam z dátového súboru zmažeme, nevyhnutne sa to prejaví zmenou v primárnom indexe, nemusí to však vyžadovať žiadnu zmenu v sekundárnom indexe. Ak neskôr nejaký dotaz na čiastočnú zhodu vyvolá cez sekundárny index prístup k takto zmazanému záznamu, príslušné vyhľadávanie podľa primárneho kľúča bude v dátovom súbore neúspešné. Tak zistíme, že uvedený záznam bol zmazaný a túto položku sekundárneho indexu budeme ignorovať. Nie všetky zmeny dátového súboru samozrejme umožňujú neurobiť úpravu v sekundárnom indexe. Práve naopak. Vkladanie záznamu a jeho modifikácia si úpravy v sekundárnych indexoch vyžadujú.

Integrácia sekundárnych indexov do stromov

Pokúsme sa teraz zvážiť možnosť integrovať sekundárne indexy na viacero rôznych kľúčov do jedinej štruktúry — stromu. Potenciálne môže byť do tejto štruktúry zahrnutý aj primárny index. Samozrejme, algoritmy na údržbu takéhoto stromu a vyhľadávanie v ňom budú patrične zložitejšie.

Základná myšlienka tkvie v tom, že na jednotlivých úrovniach stromu sa ako rozlišovací atribút budú postupne cyklicky striedať jednotlivé vyhľadávacie kľúče. Napr. ak by mal strom podporovať súčasné vyhľadávanie podľa kľúča K_1 aj K_2, boli by záznamy na nultej úrovni rozlišované podľa kľúča K_1, na prvej podľa kľúča K_2, na druhej opäť podľa kľúča K_1, atď. Pozrime sa, čo tým pri dotaze na čiastočnú zhodu získame. Povedzme, že do súboru pristupujeme podľa vyhľadávacieho kľúča K_1. Na nultej úrovni sa dokážeme správne rozhodnúť, v ktorom podstrome pokračovať. Na prvej úrovni sú záznamy rozlišované podľa kľúča K_2, pre správny podstrom sa neviem rozhodnúť, a tak musíme postupne prezrieť všetky podstromy. Na týchto úrovniach teda používame prehľadávanie s návratom (backtrack). Keď pritom zostúpime až na druhú úroveň, opäť sa podľa kľúča K_1 vieme správne rozhodnúť, atď.

Značí to zozložitenie prístupu. Ak sa totiž strom vetví podľa kľúča, ktorý vo vyhodnocovanom dotaze na čiastočnú zhodu nie je špecifikovaný, backtrackujeme všetky podstromy.

Zamyslime sa ešte, či nám táto zmena prístupovej stromovej štruktúry neskomplikovala prístup podľa primárneho kľúča. Primárny kľúč nie je v tomto prístupovom mechanizme nijako odlíšený od vyhľadávacích kľúčov. Aj pri vyhľadávaní pomocou primárneho kľúča musíme na nešpecifikovaných úrovniach backtrackovať. Napriek tomu sa tým neskomplikovalo vyhodnocovanie dotazu na úplnú zhodu. V tomto prípade totiž dotaz na úplnú zhodu nezodpovedá prístupu podľa primárneho kľúča. Pri dotaze na úplnú zhodu máme špecifikované hodnoty všetkých polí, teda všetkých potenciálnych vyhľadávacích kľúčov a tak sa v každej úrovni stromu dokážeme rozhodnúť pre správny podstrom a nemusíme backtrackovať.

Uvedená myšlienka sa dá integrovať do rôznych stromových štruktúr. Jej integráciou do B-stromov vznikajú tzv. *viacrozmerné* B-stromy. Integráciou tejto myšlienky do binárnych vyhľadávacích stromov vznikajú *k–d stromy.* (Pri k–d stromoch si však musíme uvedomiť, že ako štruktúra odvodená od binárnych vyhľadávacích stromov — teda dátovej štruktúry operačnej pamäte — musí byť pre potreby súborovej organizácie najprv upravená vhodnou stratégiou pre rozmiestňovanie svojich uzlov do blokov.)

Integrácia sekundárnych indexov do hašovacích štruktúr

Možnosť realizovať dotazy na čiastočnú zhodu v hašovacích schémach dávajú *kompozitné* hašovacie funkcie. Pre každé z polí $F_1, \ldots, F_n$, podľa ktorého chceme vyhľadávať, zostrojíme hašovaciu funkciu $h_i(k_i)$. Kompozitnú hašovaciu funkciu zostrojíme pri špecifikovaní polí $F_1, \ldots, F_n$ na hodnotu $k = (k_1, \ldots, k_n)$ ako $h(k) = h_1(k_1) * h_2(k_2) * \cdots * h_n(k_n)$, kde operácia $*$ značí bitové zreťazenie (obr. 1.3). Teda, najprv určíme hodnoty dielčích hašovacích funkcií $h_i(k_i)$, prevedieme ich do dvojkovej sústavy, bitovo zreťazíme a získame tak skutočnú hašovanú adresu. Pritom ak sú dĺžky hodnôt dielčích hašovacích funkcií vopred stanovené na b_i bitov, dáva kompozitná hašovacia funkcia reťazec bitov dĺžky $b = \sum_{i=1}^{n} b_i$. Pri dotaze na úplnú zhodu máme špecifikované všetky hodnoty

100100	0010	$\cdots$	0010001
$h_1(k_1)$	$h_2(k_2)$	$\cdots$	$h_n(k_n)$

Obr. 1.3: Kompozitná hašovacia funkcia $h(k)$

k_i, a tak dokážeme plne vypočítať hodnotu hašovacej funkcie. Ak nie sú všetky polia $F_1, \ldots, F_n$ špecifikované, teda ak ide o dotaz na čiastočnú zhodu, dokážeme určiť aspoň niektoré $h_i(k_i)$ a máme tak detekované aspoň niektoré bity hašovanej adresy. Ostatné bity však už nezískame a musíme postupne preveriť všetky do úvahy pripadajúce adresy.

Bitové vektory

Skúmajme teraz situáciu, keď je invertovaných zoznamov len málo (a sú preto veľmi dlhé). Keďže ide o zoznamy, narába sa s nimi sekvenčne, a tak je nám ich priveľká dĺžka na obtiaž. Táto situácia nastáva, ak existuje len málo možných hodnôt sekundárneho kľúča. Ak je napr. typom sekundárneho kľúča pravdivostná hodnota (true, false), sú invertované zoznamy len dva a majú dĺžku asi ako polovica celého súboru. V tomto prípade by sme vlastne mali vystačiť len s jedným invertovaným zoznamom. Do druhého totiž patria práve tie záznamy, ktoré nepatria do prvého. Situáciu môže zachrániť *bitový vektor.* V ňom je každému záznamu dátového súboru priradený jeden bit, indikujúci hodnoty true, false. Ak je hodnôt sekundárneho kľúča viac ako dve, ale stále je ich málo, môžeme ich kódovať dvojicami, resp. trojicami bitov, najčastejšie sú však práve booleovské hodnoty. Načítanie bitového vektora je potom jednoduchšie než načítavanie dlhého invertovaného zoznamu. Čo je však na tomto prístupe problematické, je priradenie bitov bitového vektora záznamom dátového súboru. To totiž pre záznamy dátového súboru znamená existenciu istého zoradenia — totiž toho, v akom sú im prideľované bity bitového vektora. Keďže primárny dátový súbor zväčša podlieha netriviálnym dynamickým zmenám, je treba uvážiť možnosti údržby takéhoto bitového vektora na účely sekundárneho indexu.

1.9.3 Intervalové dotazy

Intervalovým dotazom je dotaz typu $a \leq X \leq b$, kde X je pole a a, b sú konštanty. Všimnime si najprv, ako sú intervalové dotazy podporované doteraz zmienenými štruktúrami.

Sekvenčná a stromová organizácia umožňujú vyhodnocovať intervalový dotaz tým spôsobom, že všetky záznamy uvedenú nerovnosť spĺňajúce sa nachádzajú medzi blokom, v ktorom sa má nachádzať záznam s hodnotou $X = a$ a blokom, v ktorom sa má nachádzať záznam s hodnotou $X = b$. Pri sekvenčnej organizácii je to dokonca súvislý úsek po sebe nasledujúcich blokov. Vyhodnotiť intervalový dotaz teraz môžeme prehľadaním všetkých takto identifikovaných blokov.

Úplne iná je situácia s hašovanými štruktúrami. Od hašovacích funkcií sme totiž požadovali „náhodné“ rovnomerné distribuovanie dát. Avšak na to, aby sme mohli efektívne nájsť záznamy, ktoré sú odpoveďou na intervalový dotaz, by sme potrebovali, aby boli záznamy zahašované usporiadane podľa hodnoty poľa X. Zdalo by sa, že podmienky náhodnosti a usporiadanosti si protirečia. Nie je to však pravda, ak poznáme (aspoň približne distribúciu hašovaných dát). Interpolačná hašovacia funkcia z časti 1.8.3 totiž usporiadanie zachovávala. Pri vyhodnocovaní dotazu $a \leq X \leq b$ potom stačí prezrieť všetky bloky s číslami medzi $h(a)$ a $h(b)$. Ak chceme intervalovo vyhľadávať podľa viacerých polí súčasne, aplikujeme kompozitnú hašovaciu funkciu.

1.9.4 Zložené dotazy

Zložené dotazy vyžadujú pred svojím vyhodnotením vyhodnotenie svojich zložiek, najčastejšie dotazov na čiastočnú zhodu a intervalových dotazov. Treba poznamenať, že pri zložitých dotazoch má význam dotaz pred prístupmi do súboru zoptimalizovať. Najčastejšie spájacie mechanizmy elementárnych dotazov sú logické spojky AND, OR zodpovedajúce prieniku a zjednoteniu. Najjednoduchší postup pre vyhodnotenie zložených dotazov $D_1 \wedge D_2$ a $D_1 \vee D_2$ je vyhodnotiť dotazy D_1 a D_2 a nad získanými výsledkami vykonať operácie prieniku a zjednotenia. To je v poriadku v prípade zjednotenia. V prípade prieniku však môže byť mohutnosť množiny záznamov vyhovujúcich dotazu $D_1 \wedge D_2$ podstatne menšia než mohutnosti množín záznamov vyhovujúcich dotazom D_1, D_2. Vyhodnocovať vopred dotazy D_1, D_2 teda môže byť neefektívne,

Iná možnosť je mať výsledky vyhodnotenia dotazov D_1, D_2 v tvare bitového vektora nad dátovým súborom. V tom prípade stačí na tieto vektory aplikovať operácie AND, OR a získame bitový vektor zloženého dotazu. Výhodou je vyššia rýchlosť vyhodnotenia zloženého dotazu (teda menší počet potrebných prístupov), nevýhody sú dané potrebou údržby bitových vektorov.

Treba ešte poznamenať, že na dotaz tvaru $X = x \wedge Y = y$, kde X a Y sú polia a x, y konštanty sa možno dívať buď ako na zložený dotaz, alebo na dotaz na čiastočnú zhodu so zloženým kľúčom (X, Y). Zložený dotaz vyžaduje vyhodnotenie prieniku elementárnych. To v prípade zloženého vyhľadávacieho kľúča nie je potrebné. Vtedy je však potrebné vytvorenie (sekundárneho) indexu pre zložený kľúč. V závislosti od aplikácie je potrebné zvážiť, pre ktoré zložené vyhľadávacie kľúče sa oplatí vytvoriť samostatný (sekundárny) index a ktoré radšej vyhodnocovať ako zložený dotaz. Samostatný index totiž zaberá priestor a vyžaduje údržbu.

1.10 Triedenie súborov

V tejto časti sa budeme venovať samostatnej oblasti súborových aplikácií — triedeniu súborov. Táto problematika v skutočnosti nie je priamou súčasťou databázovej organizácie — je súčasťou problematiky súborovej organizácie ako takej. Niekoľkokrát sme sa však aj pri databázových aspektoch zmienili o potrebe súbor utriediť. Pozrime sa preto na niektoré techniky triedenia.

V prvom rade si musíme uvedomiť, že triedenie v dvojúrovňovej pamäti sa svojou povahou odlišuje od triedenia vo vnútornej (operačnej) pamäti. Pretože práve externá pamäť je v rámci dvojúrovňovej pamäte tou významnou zložkou, hovorí sa o *metódach triedenia vo vonkajšej pamäti* alebo o *vonkajšom triedení*. Metódy triedenia v operačnej pamäti sú tu prakticky nepoužiteľné, pretože kladú dôraz na minimalizáciu počtu porovnaní, resp. presunov. Pri súborovom triedení však potrebujeme minimalizovať počet prístupov do vonkajšej pamäte.

Triedenie sa považuje za elementárny problém informatiky. Niet preto divu, že problematika vonkajšieho triedenia sa skúmala už od prvopočiatkov súborovej organizácie dát. Bolo preto vyvinutých mnoho metód triedenia sekvenčných súborov už na magnetických páskach — teda v sekvenčnej pamäti — bez využitia priamych prístupov. Budeme sa v ďalšom zaoberať týmito technikami — teda budeme triediť sekvenčné súbory neumožňujúce priamy prístup.

1.10.1 Kosekvenčné spracovanie súborov

Najjednoduchšou technikou súčasného spracovania súborov je vyvažované čítanie. Uvedenú schému demonštruje algoritmus na obr. 1.4. Kosekvenčne sa v nej spracovávajú dva vstupné súbory infile1 a infile2 do výstupného súboru outfile. Predpokladá sa, že kľúče (key) záznamov sú takého typu, že je na ňom zavadené usporiadanie V pamäti je v tomto algoritme zavedená bežná čítaná hodnota vstupných súborov in1 a in2. Booleovské indikátory valid1 a valid2 signalizujú, či sú údaje v premenných in1 a in2 platné. Premenná out potom drží hodnotu, ktorá sa práve bude spracovávať a zapisovať do výstupného súboru outfile.

Procedúry scan_infile1 a scan_infile2 slúžia na načítanie ďalšieho záznamu súborov do premenných in1, resp. in2 a nastavenie indikátorov valid1 a valid2. Procedúra initialize inicializuje celú súborovú schému. Jadrom algoritmu je funkcia get_next. Tá na požiadanie vráti hodnotu záznamu, ktorý sa bude spracovávať. Hlavný program potom po inicializácii spracováva záznamy postupne vracané funkciou get_next, až kým sa oba vstupné súbory neminú. Každý takýto záznam je najprv transformovaný bližšie nešpecifikovanou procedúrou process a potom zapísaný do výstupného súboru outfile. Procedúra get_next pritom vyberá spomedzi záznamov v premenných in1 a in2, ak sú obe platné, tú s menšou hodnotou kľúča. Keď sa niektorý zo súborov minie, procedúra get_next dočíta druhú z nich. Procedúra get_next teda číta vstupné súbory „vyvažovane“ vzhľadom k údajom, ktoré nesú.

Uvedená schéma je pomerne užitočná na rôzne účely, v skutočnosti nie je podstatné, že sú spracovávané súbory — rovnako dobre by sa dala použiť aj na spracovávanie (usporiadaných) spájaných zoznamov. Napr. miernou modifikáciou procedúry get_next možno získať algoritmus na nájdenie prieniku dvoch usporiadaných spájaných zoznamov. Tým sa však nebudeme zaoberať a všimneme si ju z hľadiska triedenia súborov.

Pri praktickej realizácii sa samozrejme nebudeme obracať na súbor pre každý záznam osobitne. Jedno načítanie prevedie do operačnej pamäti (minimálne) jeden blok. Potenciálne tak vo vyrovnávacej pamäti môže byť aj niekoľko blokov z toho istého súboru. Procedúra get_next v tomto prípade vráti záznam s najmenšou hodnotou kľúča spomedzi všetkých aktuálnych záznamov všetkých súborových blokov umiestnených vo vyrovnávacích pamätiach. Na to je teda

```
procedure balanced_reading;
var infile1,infile2,outfile:file of T;
    valid1,valid2:boolean;
    in1,in2,out:T;

procedure scan_infile1;
begin
  if not eof(infile1) then begin
                             read(infile1,in1);
                             valid1:=true;
                           end
  else valid1:=false;
end;

procedure scan_infile2;
begin
  if not eof(infile2) then begin
                             read(infile2,in2);
                             valid2:=true;
                           end
  else valid2:=false;
end;

procedure initialize;
begin
  open(infile1) for reading;
  open(infile2) for reading;
  open(outfile) for writing;
  scan_infile1; scan_infile2;
end;
                                  {pokračovanie}
```

Obr. 1.4: Algoritmus vyvažovaného čítania

```
                                   {pokračovanie}
function get_next:T;
begin
  if valid1 and valid2 then
    begin
      if in1.key<=in2.key then begin
                                 get_next:=in1;
                                 scan_infile1;
                               end
      else begin
             get_next:=in2;
             scan_infile2;
           end
  else if valid1 then begin
                        get_next:=in1;
                        scan_infile1;
                      end
       else begin
              get_next:=in2;
              scan_infile2;
            end;
end;

begin {balanced_reading}
  initialize;
  while valid1 or valid2 do
    begin
      out:=get_next;
      process(out);
      write(outfile,out);
    end;
  close(infile1);
  close(infile2);
  close(outfile);
end;
```

Obr. 1.4 (pokračovanie): Algoritmus vyvažovaného čítania

1	2	5	7	9	14	3	6	4	8	11	10	13	12

Obr. 1.5: Príklad rozdelenia súboru na behy

vhodné záznamy v operačnej pamäti utriediť niektorou z metód vnútorného triedenia. Ďalej, pretože pri takomto spracovávaní dochádza ku fyzickému kopírovaniu záznamov, pričom podstatné sú len hodnoty kľúčov, aj tu, ako už toľkokrát predtým, je užitočné oddeliť kľúče záznamov a dodatočnú informáciu záznamov (a prepojiť ich smerníkom od kľúča k záznamu — prakticky formou hustého indexu). Triediť sa potom budú len kľúče záznamov.

Schéma vyvažovaného čítania, za predpokladu, že procedúra process nevykonáva žiadnu činnosť vedie k tomu, že ak boli vstupné súbory utriedené, zlúčia sa do nového spoločného utriedeného súboru (získame teda utriedené zjednotenie utriedených vstupných súborov). Metódam triedenia založených na tejto myšlienke sa hovorí *zlučovanie* (angl. merge). Ak totiž aj vstupné súbory neboli utiedené, súbor vzniknutý zlúčením bude „utriedenejší". Ako charakteristiku utriedenosti možno zobrať počet *behov* (angl. runs) súboru. Behom nazveme každú maximálnu utriedenú súvislú podpostupnosť súboru. Na obrázku 1.5 je príklad určenia behov v súbore s celočíselnými kľúčmi. Utriedený súbor obsahuje jediný beh, kým n-prvkový súbor utriedený v opačnom poradí, ako je žiadúce, obsahuje n jedenprvkových behov. Budeme sa zaoberať rôznymi druhmi zlučovania a tam popíšeme príslušné detaily. V skratke však možno povedať, že súbor sa najprv rozdelí — *rozdistribuuje* — na menšie súbory, ktoré sa potom zlúčia a tento proces sa iteruje, až kým nie je súbor utriedený. Pre jednotlivé iterácie je preto dôležité zmenšovanie počtu behov súboru. Každá iterácia sa skladá z niekoľkých čítaní súboru nazývaných *fázy*. Práve opísaný postup by mal napr. dve fázy: distribučnú a zlučovaciu. Pri charakteristike zlučovania sa zvykne hovoriť aj, o koľkofázové zlučovanie ide. Druhým dôležitým parametrom je počet vstupných súborov. Schému kosekvenčného spracovania totiž možno bez ťažkostí upraviť na viacero vstupných súborov. Zlučovanie s k vstupnými súbormi sa nazýva *k-cestné.*

1.10.2 Priame zlučovanie

Priame zlučovanie ignoruje existenciu behov v súbore, teda ignoruje fakt, že vstupný súbor môže byť už z časti usporiadaný. Predpokladajme, že triedený súbor obsahuje 2^n kľúčov. Triedenie priamym zlučovaním potom prebehne v n iteráciách. Každá z iterácií sa skladá z dvoch fáz — distribučnej a zlučovacej. Zlučovacia fáza presne zodpovedá technike kosekvenčného dvojcestného spracovania z predchádzajúcej časti. Distribučná fáza v k-tej iterácii rovnomerne rozdelí triedený súbor na 2^{n-k+1} úsekov dĺžky 2^{k-1}, ktoré sú striedavo distribuované do dvoch súborov infile1 a infile2. Teda, prvých 2^{k-1} kľúčov triedeného súboru sa umiestni do súboru infile1, druhých 2^{k-1} kľúčov do súboru infile2, atď. cyklicky. Zlučovacia fáza potom zlúči súbory infile1 a infile2 do jedného súboru.

Indukciou teraz ľahko vidíme, že pred vykonaním k-tej iterácie sú úseky triedeného súboru dĺžky 2^{k-1} zaručene utriedené. Skutočne, pred vykonaním prvej iterácie ide o úseky dĺžky 1 a taký úsek je automaticky utriedený. A ak je pred vykonaním k-tej iterácie skutočne uvedený stav, distribučná fáza rozmiestni tieto úseky striedavo do súborov infile1 a infile2. Zlučovacia fáza potom zlúči vždy dva utriedené úseky dĺžky 2^{k-1} do jedného utriedeného úseku dĺžky 2^k. V konečnom dôsledku teda po vykonaní n-tej iterácie (teda pred vykonaním $(n+1)$-vej iterácie, ktorá už ale nie je potrebná) obsahuje súbor jeden utriedený úsek dĺžky 2^n, t.j. súbor je utriedený.

Priame zlučovanie teda celkovo utriedi súbor dĺžky M po $\lceil \log_2(M) \rceil$ iteráciách, bez ohľadu na stav súboru. Toľkoto iterácií sa teda vykoná aj v prípade, že bol vstupný súbor náhodou utriedený. Každá iterácia pritom vyžadovala dve čítania súboru[16] — išlo teda o dvojfázové dvojcestné zlučovanie.

1.10.3 Prirodzené zlučovanie

Prirodzené zlučovanie sa líši od priameho zlučovania tým, že sa berú do úvahy behy. V distribučnej fáze sa triedený súbor nerozdeľuje na úseky pevnej dĺžky, ale na úseky zodpovedajúce jeho behom. Prvý beh sa umiestni do súboru infile1, druhý do súboru infile2, atď. cyklicky. Ostatné časti algoritmu zostávajú nezmenené. Okrem toho, že takáto distribúcia behov berie do úvahy počiatočný stav súboru (napr. utriedený súbor sa triediť už nebude, pretože obsahuje len jediný beh), prichádza do úvahy, že len obyčajným umiestnením dvoch behov za seba do infile-súboru v distribučnej fáze sa tieto spoja do jedného.

Preskúmajme, že ide naozaj o výhodu. Ak triedený súbor obsahuje t behov, budú rozdistribuované do súborov infile1, infile2 po $t/2$ behov. Niektoré behy sa však týmto rozmiestnením samovoľne zlúčia do väčších. V prípade súboru infile1 nech takýmto zlúčením ubudne u_1 behov a v prípade súboru infile2 u_2 behov a nech $u_1 \geq u_2$. Súbory infile1, resp. infile2 teraz obsahujú $t/2 - u_1$, resp. $t/2 - u_2$ behov. Keďže $t/2 - u_1 \leq t/2 - u_2$, v zlučovacej fáze vznikne najprv $t/2 - u_1$ zlúčených behov, čím sa súbor infile1 vyčerpá. Súbor infile2 bude v tomto okamihu obsahovať ešte $t/2 - u_2 - (t/2 - u_1) = u_1 - u_2$ behov, ktoré sa do výstupného súboru len prekopírujú. Po dokončení zlučovacej fázy bude teda výstupný súbor outfile obsahovať $t/2 - u_1 + (u_1 - u_2) = t/2 - u_2 \leq t/2$ behov. Jednou iteráciou teda počet behov klesne na polovicu alebo menej a samovoľné zlučovanie behov predstavuje výhodu.

Celkový počet iterácií je potom najviac ak logaritmický od počtu behov súboru. V najhoršom prípade je potom logaritmický od dĺžky súboru (tento prípad sa dosahuje napr. ak je súbor utriedený v obrátenom poradí, než je želané). Opäť ide o dvojfázové dvojcestné zlučovanie.

1.10.4 Vylepšenia prirodzeného zlučovania

Zvážime teraz niekoľko možností, ako pomerne jednoduchú a efektnú schému prirodzeného zlučovania v rôznych smeroch vylepšiť.

Vyvažované viaccestné zlučovanie

Už sme spomínali, že kosekvenčne možno po patričnej úprave algoritmu zlučovať aj viacero súborov naraz. To je pomerne jednoduché vylepšenie. Nepríjemné však je, že algoritmus vyžaduje dve fázy, teda dve čítania súboru. Jedna je aktívna — zlučovacia. Druhá — distribučná — akoby k triedeniu žiaden prínos nemala. Nie je však algoritmický dôvod odkladať distribúciu až do nasledujúcej

[16] Neskôr sa budeme zaoberať aj možnosťou eleminácie distribučnej fázy.

fázy. Len čo začne výstupný súbor outfile vznikať, ihneď by ho bolo možné distribuovať. Pravda, do vstupných súborov to nejde, pretože tie ešte nie sú dočítané. Môžeme však na to použiť nové súbory, čo je podstatta *viaccestného vyvažovaného zlučovania.*

Viaccestné vyvažované zlučovanie bude teda len jednofázové. Budeme naň potrebovať n vstupných a n výstupných súborov. Triedený súbor sa v rámci inicilizácie schémy rozdistribuuje do n vstupných súborov. Tie sa potom začnú zlučovať a dáta, ktoré tak vzniknú sa hneď budú distribuovať na n výstupných súborov. Následne si vstupné a výstupné súbory vymenia úlohy a celý proces sa iteruje.

Ušetrili sme jednu fázu v každej iterácii, teda teoretický čas triedenia by mal klesnúť zhruba na polovicu (to je nezanedbateľná úspora). Zvážme však, čo strácame a či skutočne s takouto úsporou počítať môžeme. V prvom rade, počet súborov, s ktorými pracujeme sa vzrástol na takmer dvojnásobok. Keby sme používali na uchovávanie súborov magnetické pásky, značilo by to dvojnásobný počet pások a čítaco-zapisovacích zariadení. Pravdepodobnejšie budeme používať disk. Ten môže obsahovať aj viacero súborov, a tak sa zdá, že nebudeme potrebovať viac diskov. V skutočnosti však viac súborov na disku, s ktorými sa súčasne pracuje, značí, že hlavičky disku sa musia medzi týmito súbormi prepínať, čo je značné zdržanie. Buď teda musíme predsa len zvýšiť počet diskov alebo sa zmieriť s tým, že zlepšenie nebude dvojnásobné, ako by sa teoreticky dalo očakávať.

Pri kosekvenčnom spracovaní súborov na diskoch pritom aj vo všeobecnosti stojí za úvahu, aby sa jednostlivé pracovné súbory nachádzali na rôznych diskoch, ak je to možné.

Optimalizácia distribučnej fázy

Vyvažované viaccestné zlučovanie nám ponúklo možnosť integrácie distribučnej fázy do zlučovacej. Distribučná fáza ako taká v ňom teda neexistuje. Ničmenej, distribúciu behov na pásky je tak či tak potrebné vykonať. Nie je pritom podstatné, či je distribučná fáza samostatná alebo integrovaná do fázy zlučovacej.

Videli sme, že počet behov sa pri ditribúcii do vstupných súborov môže znížiť samovoľným spojením za sebou umiestňovaných behov. Aj keď neberieme ohľad na túto skutočnosť, dosiahneme logaritmický počet iterácií. Videli sme, že počet behov klesne v jednej iterácii z t na $t/2 - \min\{u_1, u_2\}$, kde u_1, u_2 sú počty samovoľne zlúčených behov (pri dvojcestnom zlučovaní). Vzniká otázka, či nemožno dosiahnuť ešte lepšie časy, ak si počas distribúcie budeme všímať, ako sa behy samovoľne zlučujú a podľa toho distribúciu aktuálne pozmeníme. Pravda, ak k žiadnemu samovoľnému zlúčeniu nedôjde, nič nezískame a pôjde len o obyčajné prirodzené zlučovanie.

Všimnime si to na príklade. Nech má napr. vstupný súbor 64 behov, ktoré rozdelíme do súborov infile1 a infile2 po 32 behov, pričom v súbore infile1 sa všetky samovoľne spoja do jediného behu a v súbore infile2 k žiadnemu samovoľnému spájaniu nedôjde. Zlučovanie teraz zlúči jediný beh súboru infile1 s prvým behom súboru infile2 a zvyšných 31 behov súboru infile2 sa len prekopíruje. Výstupný súbor outfile má teraz 32 behov, čo je zníženie na polovicu, ako sme aj teoreticky odvodili. Avšak, keby sme si samovoľné zlučovanie všimli a k jedinému veľkému behu súboru infile1 umiestnili aj 15 behov súboru infile2 a povedzme, že by už k ďalším samovoľným zlúčeniam nedošlo, obsahoval by

Krok	1	2	3	4	5	6	7
Počet behov v 1. súbore	21	13	–	3	1	–	1
Počet behov v 2. súbore	13	–	5	2	–	1	–
Počet behov v 3. súbore	–	8	3	–	1	1	–

Tabuľka 1.2: Dvojcestné polyfázové zlučovanie

súbor infile1 16 behov a infile2 17 behov. Zlúčenie vedie k výstupnému súboru so 17 behmi, čo je ale zmenšenie takmer na štvrtinu!

Optimalizácia distribučnej fázy je preto založená na sledovaní počtu vznikajúcich behov tak, aby po skončení bol počet behov vstupných súborov zhruba rovnaký, aj keď dôjde k samovoľnému spájaniu.

1.10.5 Polyfázové zlučovanie

Aj polyfázové zlučovanie je vylepšením prirodzeného zlučovania, avšak vylepšením veľmi významným. Vylepšuje viaccestné zlučovanie. Keby sa totiž vo viaccestnom zlučovaní behy do vstupných súborov nedistribuovali rovnomerne, ale podľa iného kľúča a zlučovalo by sa iba do jedného výstupného súboru, niektorý zo vstupných súborov by sa prirodzene minul skôr než ostatné. Keby sme v zlučovaní pokračovali, bolo by to neefektívne, pretože jeden vstupný súbor by sa teraz nevyužíval. V polyfázovom zlučovaní si v tomto okamihu výstupný súbor a práve vyprázdnený súbor vymenia úlohy. Zlučovanie pokračuje s tým, že výstup sa začne zapisovať do uprázdneného súboru. Prakticky sa teda začína stierať viditeľné oddelenie jednotlivých iterácií. Distribučná fáza sa nielen spojila so zlučovacou, ako je to pri viaccestnom vyvažovanom zlučovaní, ale dokonca sa s ňou prepojila a poprepletala. Fázy sa tak stali neoodeliteľné, odkiaľ pochádza názov — *polyfázové zlučovanie*. Polyfázové zlučovanie pritom ako k-cestné potrebuje len $(k+1)$ súborov, ktoré sa podľa potreby vymieňajú úlohy.

Priebeh zlučovania si demonštrujeme na príklade uvedenom v tabuľke 1.2. Budeme pritom ignorovať problém samovoľného zlučovania behov pri distribúcii. Uvažujeme dvojcestné zlučovanie, teda v každom okamihu máme dva vstupné a jeden výstupný súbor. Čísla udávajú počty behov v jednotlivých súboroch. Schéma je automaticky určená počiatočnou distribúciou behov. Pre uvedených 42 behov a dvojcestné zlučovanie sú počty $(21, 13)$ optimálne. Nie je ťažké všimnúť si, že počty behov sú Fibonacciho čísla. Dá sa tiež ukázať, že pri p-cestnom zlučovaní treba behy počiatočne distribuovať ako Fibonacciho čísla p-teho rádu definované rekurentne

$$F_i^{(p)} = \begin{cases} 0 & \text{ak } i < p \\ 1 & \text{ak } i = p \\ F_{i-1}^{(p)} + F_{i-1}^{(p)} + \cdots + F_{i-p-1}^{(p)} & \text{inak} \end{cases}$$

Bežné Fibonacciho čísla sú pritom Fibonacciho čísla druhého rádu. Indukciou ľahko vidieť, že ak pri p-cestnom zlučovaní distribuujeme behy počiatočne v počtoch $(F_{k+p+1}^{(p)}, F_{k+p}^{(p)}, \ldots, F_{k+1}^{(p)})$, budú v nasledujúcom kroku rozdelené do súborov v počtoch $(F_{k+p}^{(p)}, F_{k+p-1}^{(p)}, \ldots, F_k^{(p)})$ v nejakom poradí. Utriedený súbor dostaneme po k krokoch.

Z hľadiska efektivity je polyfázové triedenie jedno z najkvalitnejších — je rýchle a pritom pri p-cestnosti potrebuje len $(p + 1)$ súborov. Samozrejme, musíme uvážiť, že začiatočná distribúcia nie vždy zodpovedá ideálnemu prípadu — počet triedených kľúčov asi nebude rovný Fibonacciho číslam. Navyše môže dochádzať k samovoľnému zlučovaniu behov, čo v tomto prípade môže narúšať pomerne krehkú vyváženosť zlučovania. Najčastejšie sa preto pri nevhodnom počte behov uvažuje s prázdnymi — *fiktívnymi* behmi.

Problematika zlučovania bola preskúmaná z rôznych hľadísk. Na magnetických páskových pamätiach sa napr. uvažovali vylepšenia umožňujúce čítanie pásky aj odzadu (aby ju medzi iteráciami nebolo potrebné prevíjať) — tými sa však už nebudeme zaoberať.

1.11 Fyzická optimalizácia operácií relačnej algebry

Zhrnieme teraz, aké možnosti nám poskytujú štruktúry, ktoré sme zaviedli pre urýchlenie prístupu k súborom, na rýchle vykonávanie operácií relačnej algebry. Predpokladáme pritom, že primárny dátový súbor nesie ako záznamy riadky (veľmi rozsiahlej) databázovej relácie.

Projekcia

K optimalizácii projekcie niet čo dodať. Je to jednoduchá operácia, ktorá svojou podstatou vlastne nemá s organizáciou súborov nič spoločné. Jediný problém, ktorý nám hrozí, je, že vykonaním selekcie môžu vyniknúť duplicitné riadky. Ich odstránenie si v princípe vyžaduje utriedenie. Treba preto zvoliť vhodnú stratégiu — kedy triediť. Dotazovacie jazyky, ako napr. SQL obsahujú príkazy explicitne takéto triedenie ovplyvňujúce.

Selekcia

Selekcia svojou podstatou zasa predstavuje dotaz na čiastočnú (prípadne úplnú) zhodu, resp. intervalovú zhodu. Optimalizácii selekcií sme sa teda už vlastne venovali v rámci časti o dotazoch zavedením rôznych sekundárnych indexov v indexovo-sekvenčnej, stromovej alebo aj hašovanej organizácii súborov.

Zjednotenie

Zjednotenie relácií v súboroch možno bez väčších ťažkostí realizovať technikou vyvažovaného čítania podľa algoritmu na obr. 1.4. Ak zjednocujeme neutriedené súbory, opäť môžu vznikať duplicitné riadky a treba preto vhodne stanoviť, kedy sa ich oplatí triedením odstrániť. Ak zjednocujeme utriedené súbory, technika vyvažovaného čítania nám umožňuje duplicitu riadkov detekovať pri zjednocovaní. Okrem toho, automoaticky získavame utriedený súbor.

Rozdiel

Rozdiel $S \setminus T$ relácií S a T sa najjednoduchšie realizuje prechodom cez súbor relácie S a postuné vyhľadávanie jednotlivých záznamov v súbore relácie T.

```
for each tuple r in R do
  for each tuple s in S do
    output tuple (rs);
```

Obr. 1.6: Elementárny výpočet kartézskeho súčinu

```
for each segment E of relation S do
  for each blok B of relation R do
     for each tuple s in segment E do
       for each tuple r of blok B do
         output tuple (rs);
```

Obr. 1.7: Kartézsky súčin veľkých relácií

Ako prístupová štruktúra súboru relácie T sa preto hodí štruktúra podporujúca dotaz na úplnú zhodu.

1.11.1 Výpočet kartézskeho súčinu

Najväčším problémom výpočtu kartézskeho súčinu je veľkosť výsledku. Operovanie na výsledkom (ako jeho zápis do súboru apod.) je potom najvýznamnejšou zložkou náročnosti výsledku. Zjednosušene môžeme povedať, že pri výpočte kartézskeho súčinu $S \times T$ relácií S a T vykonáme zhruba rádovo $B_S.B_T$ súborových operácií, ak samé relácie S, resp. T zaberajú B_S, resp. B_T blokov algoritmom na obr. 1.6. Tento postup však možno ešte za istých okolností trochu vylepšiť. V situácii, keď je niektorá z relácií (napr. S) dostatočne malá na to, aby sa zmestila celá do operačne pamäte a pritom ešte zostal aspoň jeden blok operačnej pamäte voľný, môžeme umiestniť celú reláciu S do pamäte, postupne blok po bloku čítať reláciu R a vytvárať kartézsky súčin.

Aj keď ani jedna z relácií na umiestneni do operačnej pamäte nie je dostatočne malá, môžeme menšiu z nich (povedzme opäť S) rozdeliť na úseky, ktoré sa do pamäte vojdú. Najprv potom vytvoríme kartézsky súčin prvého úseku relácie S s celou reláciou R (čo si vyžiada čítanie celého súboru relácie R), potom druhého úseku relácie S s celou reláciou R, atď. Postupujeme teda algoritmom na obr. 1.7.

1.11.2 Výpočet prirodzených spojení

Prirodzené spojenia majú blízko ku kartézskym súčinom. Kartézsky súčin je totiž špeciálnym prípadom prirodzeného spojenia. Všetky ďalej skúmané metódy preto nemôžu byť v najhoršom prípade lepšie, než metódy výpočtu kartézskeho súčinu.

Výber z kartézskeho súčinu

Triviálnym spôsobom výpočtu prirodzeného spojenia $S \bowtie T$ relácií S a T je najprv vytvoriť kartézsky súčin a potom v ňom realizovať selekciu prirodzeného spojenia, teda postupovať podľa $S \bowtie T = \sigma_F(S \times T)$, kde F je formula rovnosti

```
for each tuple r in R do
  for each tuple s in S do
    if r.X = s.X then output tuple (rs);
```

Obr. 1.8: Výpočet prirodzeného spojenia metódou vloženého cyklu

```
for each tuple r in R do
  begin
     take K;
     for each tuple s in S(K,*) do
     if r.X = s.X then output tuple (rs);
  end
```

Obr. 1.9: Využitie indexu v metóde vloženého cyklu

spoločných atribútov relácií. Metóda má teda zložitosť vždy rádovo veľkosti kartézskeho súčinu.

Priamočiarou realizáciou tohto algoritmu je metóda vloženého cyklu (angl. nested loop). Pri triviálnej realizácii je tento algoritmus rovnako náročný ako výpočet kartézského súčinu. Ak však relácia S má vybudovaný index pomocou klúča K $\subseteq$ X, môžeme tento efektívne využiť.

V bežnom prípade však prirodzené spojenie nevracia výsledok veľký až tak ako kartézsky súčin, je preto namieste snaha tento postup vylepšiť

Výpočet prirodzeného spojenia triedním

Obyčajne je výsledok prirodzeného spojenia omnoho menší než by bol príslušný kartézsky súčin, a tak je vhodné vyhnúť sa konštrukcii kartézskeho súčinu, ak je to možné. Túto možnosť nám opäť poskytuje schéma vyvažovaného čítania. Musíme len urobiť miernu modifikáciu funkcie get_next, ktorá je na obr. 1.10. Predpokladáme pritom, že common je spoločným atribútom spájaných relácií. Správna funkčnosť algoritmu pritom vyžaduje, aby boli súbory relácií S a T vopred utriedené. Algoritmus potom vyvažovane číta oba súbory a vyhľadáva záznamy ktoré sa nelíšia hodnotou atribútu common. Ak už také nemožno nájsť, funkcia get_next vráti hodnotu nil. Funkcia make_tuple vyrobí z prvkov relácií S

```
function get_next:T;
begin
  while valid1 and valid2
           and (in1.common<>in2.common) do
      if in1.key<in2.key then scan_infile1
      else scan_infile2;
  if not valid1 or not valid2 then get_next:=nil
  else get_next:=make_tuple(in1,in2);
end;
```

Obr. 1.10: Upravená funkcia get_next

```
for each blok B in T do
  for each tuple t in B do
    join t with tuples in S having S.common=t.common;
```

Obr. 1.11: Použitie indexu v jednej relácii

```
for each value of common do
  join tuples in S having S.common=common
  with tuples in T having T.common=common;
```

Obr. 1.12: Použitie indexu v oboch reláciách

a T nelíšiacich sa hodnotou atribútu common prvok relácie $S \bowtie T$.

Čo sa týka zložitosti, čítanie vyžaduje načítanie súborov relácií S a T, každého raz a zápis zapísanie výsledku, ktorý bude mať veľkosť B_R. Ide teda o čas $O(B_S + B_T + B_R)$. Ak však súbory relácií S a T ešte neboli utriedené, musíme ich najprv utriediť, čo dokážeme v čase $O(B_S \log B_S + B_T \log B_T)$. Celkovo teda vypočítame prirodzené spojenie na $O(B_S \log B_S + B_T \log B_T + B_S + B_T + B_R) = O(B_S \log B_S + B_T \log B_T + B_R)$ prístupov. Výhodnosť postupu teraz záleží od veľkosti výsledku B_R. Ak výsledok zodpovedá veľkosti kartézskeho súčinu, teda $B_R = B_S.B_T$, nezískali sme nič. Ak je však výsledok menší (a to sa dá očakávať), teda $B_R = O(B_S \log B_S + B_R \log B_R)$, je najvýznamnejšou zložkou postupu triedenie a dosiahli sme celkové asymptotické zlepšenie na počet prístupov $O(B_S \log B_S + B_R \log B_R)$.

Výpočet prirodzených spojení pomocou indexov

Predpokladajme, že na spoločný atribút common relácií S a T, ktorých prirodzené spojenie chceme vypočítať, je v jednej z týchto relácií (napr. S) vytvorený sekundárny index. Sekundárny index je typicky podstatne menší než dátový súbor alebo aj primárny index, ak je implementovaný tak, že každú hodnotu sekundárneho kľúča obsahuje len raz. Vtedy totiž jednému sekundárnemu kľúču sekundárneho indexu zodpovedá viac dátových záznamov. Tento index teraz môžeme použiť na pristúpenie k záznamom relácie S, ktoré majú požadovanú hodnotu atribútu common podľa algoritmu na obr .1.11. Jedným súborom teda prechádzame a v druhom vyhľadávame. Ideálne je, ak dokážeme vyhľadanie urobiť na jeden prístup do vonkajšej pamäte, teda ideálne je ako index použiť hašovaciu schému.

Myšlienku možno rozšíriť zavedením indexu na atribút common aj v prípade súboru s druhou reláciou T. Pre každú možnú hodnotu atribútu common potom pristúpime naraz k záznamom s touto hodnotou atribútu aj v jednom aj v druhom súbore. Opäť je ideálne, ak prístup vieme urobiť rýchlo, teda napr. v prípade indexu realizovaného hašovacou schémou. Postupujeme teda podľa algoritmu na obr. 1.12. Problémom ale je ako určiť všetky hodnoty atribútu common, ktoré treba v indexoch vyhľadávať. Zrejme pritom ide o hodnoty, ktoré sa vyskztujú aspoň v jednej z relácií S, T ako hodnoty atribútu common.

Jedna možnosť je, že ide o atribút, ktorého všetky možné hodnoty poznáme — ide najmä o prípad, keď je týchto hodnôt málo. V tom prípade môžeme preskúšať postupne všetky. Druhá možnosť je tieto hodnoty zistiť z požitých indexov, využívajúc, že tieto indexy sú menšie než pôvodné dátové súbory. Nemusíme však pritom prezerať oba indexy, zaujímavé sú totiž len hodnoty, ktoré sú obsiahnuté v oboch. postačuje preto použiť len hodnoty atribútu common z menšieho z týchto indexov. Problémom však teraz je, že zistenie všetkých hodnôt sekundárneho kľúča nám hašovaný index neumožňuje. Musíme preto použiť iný druh indexu, čo môže zhoršiť počet prístupov, pretože sa zväčší počet prístupov do externej pamäte pri vyhľadávaní pomocou sekundárneho indexu.

Ak náhodou máme hustý index vybudovaný takým spôsobom, že vieme nájsť bloky obsahujúce smerníky na všetky vety s danou hodnotou indexu. Je výhodné vypočítať spojenie ako prienik množín indexov. Prevaha indexovacích a hašovacích metód tak veľká, že v niektorých prípadoch sa vyplatí vybudovať index alebo zahašovať relácie len za účelom výpočtu konkrétneho spojenia.

Obsah

1 Fyzická organizácia dát **1**

1.1 Fyzické parametre vonkajších pamäťových médií 1
1.1.1 Magnetické páskové pamäte 1
1.1.2 Rotačné pamäte 2
1.2 Súbory a ich organizácia 2
1.2.1 Údajová štruktúra súboru 3
1.2.2 Údajová štruktúra záznamu 4
1.2.3 Údajová štruktúra bloku 5
1.2.4 Fragmentácia 7
1.3 Schémy organizácie súborov 9
1.4 Sekvenčné súbory a haldy 11
1.5 Indexovo-sekvenčné súbory 13
1.6 Stromy ako dátové štruktúry 15
1.6.1 B-stromy 17
1.6.2 B^*-stromy 20
1.6.3 B^+-stromy 20
1.7 Stromové súbory 20
1.7.1 B-stromová organizácia súboru 21
1.7.2 B^+-stromová organizácia súboru 22
1.7.3 Virtuálne B-stromy 23
1.8 Hašované súbory 23
1.8.1 Vlastnosti hašovacích funkcií 24
1.8.2 Distribučne nezávislé hašovacie funkcie 25
1.8.3 Distribučne závislé hašovacie funkcie 27
1.8.4 Riešenie kolízií oddeleným reťazením 28
1.8.5 Riešenie kolízií spoločným reťazením 28
1.8.6 Riešenie kolízií otvorenou adresáciou 29
1.8.7 Rozšíriteľné hašovanie 32
1.9 Dotazy 34
1.9.1 Dotazy na úplnú zhodu 34
1.9.2 Dotazy na čiastočnú zhodu 35
1.9.3 Intervalové dotazy 38
1.9.4 Zložené dotazy 39
1.10 Triedenie súborov 39
1.10.1 Kosekvenčné spracovanie súborov 40
1.10.2 Priame zlučovanie 43
1.10.3 Prirodzené zlučovanie 44
1.10.4 Vylepšenia prirodzeného zlučovania 44

1.10.5 Polyfázové zlučovanie 46
1.11 Fyzická optimalizácia operáciírelačnej algebry 47
1.11.1 Výpočet kartézskeho súčinu 48
1.11.2 Výpočet prirodzených spojení 48