Učinkovitost načina smještaja i indeksiranja podataka

Basioli, Karlo

Učinkovitost načina smještaja i indeksiranja podataka

Basioli, Karlo

SVEUČILIŠTE U ZAGREBU

PRIRODOSLOVNO–MATEMATIČKI FAKULTET

MATEMATIČKI ODSJEK

Karlo Basioli

UČINKOVITOST NAČINA SMJEŠTAJA I INDEKSIRANJA PODATAKA

Diplomski rad

Voditelj rada:
dr. sc. Ognjen Orel

Zagreb, rujan, 2022

Ovaj diplomski rad obranjen je dana _____ pred ispitnim povjerenstvom u sastavu:

1. _____, predsjednik
2. _____, član
3. _____, član

Povjerenstvo je rad ocijenilo ocjenom _____.

Potpisi članova povjerenstva:

1. _____
2. _____
3. _____

Sadržaj

Uvod

U današnjem svijetu, pogotovo širom upotrebom interneta, pojavljuje se ogroman broj aplikacija. Neke su veće, neke manje, neke su za osobnu upotrebu, a neke za poslovnu. Od mobilnih do računalnih aplikacija sve imaju jednu stvar zajedničku. Sve generiraju ili traže neke podatke. Oni mogu biti jednostavni, poput nekog korisničkog imena i datuma rođenja ili neka kompleksnija i specifičnija poput vremensko prostornih podataka kod GPS (Globalni položajni sustav) aplikacija.

Masovnije aplikacije generiraju ogroman broj podataka reda veličine nekoliko terabajta. Daljnjim napretkom očekuje se i porast ovih brojki. Da bi proizvođač ili korisnik imao koristi od ovoliko velikog broja podataka mora znati nešto računati s tim podacima. Postoje razni podatkovni inžinjeri i analitičari koji su zaduženi za održavanje i tumačenje akumuliranih podataka. Razvijaju se i nove grane računarstva kao što je *Big Data* koje razvijaju tehnike za rad s ovakvim skupovima podataka.
U nekim slučajevima podatke je potrebno distribuirati u *klastere* (skupine) od nekoliko računala kako bi ih uopće mogli pohraniti.

Kako uopće pohranjujemo sve ove podatke? Što kada njihov broj jako poraste? Kako efikasno raditi s njima?
Postoje razne tehnike zapisa podataka na disk. U ovom radu bavimo se pohranom podataka na jednom računalu te proučavamo kako se podaci zapisuju na disk. Svjesni činjenice da zapisani podaci na disku nisu pogodni za brzu interpretaciju prezentiramo i neke metode indeksiranja. Konkretno, bavimo se analiziranjem dva načina zapisa podataka na disk te dva različita tipa indeksiranja. Mjerimo njihove performanse, uspoređujemo ih i donosimo zaključke na temelju rezultata. Pri mjerenju nećemo koristiti gotove proizvode već ćemo samostalno implementirati mali sustav za upravljanje bazama podataka koji će omogućiti stvaranje tablice, indeksa nad njom te postavljanjem upita. Programski kod priložen je uz rad, a može se pronaći na [2]. Veći dio teorije korištene u ovom radu preuzet je iz knjige [3].

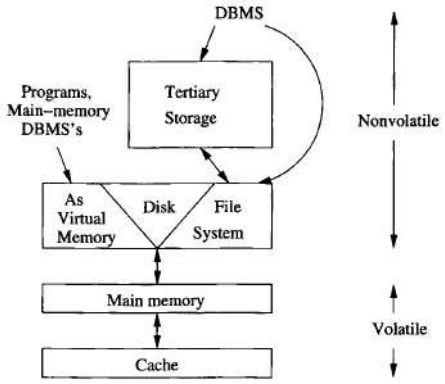
Poglavlje 1

Fizički smještaj podataka

U sljedećem poglavlju dajemo pregled fizičkog razmještaja podataka u nekom SUBP-u (Sustav za upravljanje bazama podataka). Za početak dajemo pregled memorijske hijerarhije računala i komentiramo brzinu određene memorije.

1.1 Memorijska hijerarhija

Tipično računalo može pohranjivati podatke na nekoliko različitih komponenti. Kapacitet i brzina pristupa podacima na tim komponentama jako je varijabilna. Komponente s bržim pristupom imaju puno veću cijenu po bajtu memorije.



Slika 1.1: Memorijska hijerarhija (slika preuzeta iz [3])

Ukratko opisujemo neke od razina u hijerarhiji koje će nam biti važne:

1. *Cache* (priručna memorija) je najmanja, ali memorija s najbržim vremenom pristupa u kojeg se učitavaju instrukcije i podaci potrebni procesoru

2. *RAM memorija* (Random Access Memory), na većini današnjih računala radi se o 4 do 8 GB RAM memorije. Memoriju još zovemo i radna zato što se sav rad računala odvija koristeći instrukcije i podatke koji se nalaze u RAM-u

3. *Sekundarna memorija* je tipično magnetski disk, ali u zadnje vrijeme pojavljuje se i nešto brži oblik sekundarne memorije - SSD (Solid-State Drive).

U tablici vidimo vremena pristupa određenim komponentama (radi se o vremenu potrebnom za učitavanje podataka iz nižeg sloja u viši pri čemu se memorija iz *Cache*-a učitava u registre procesora).

Komponenta	**Vrijeme pristupa**
Cache	1 ns
RAM	10-100 ns
Sekundarna	10 ms

1.2 Volatilna i nevolatilna memorija

Memorija se dijeli na volatilnu i nevolatilnu. Volatilna memorija je memorija koja nakon prekida rada računala *zaboravlja* što je pohranila. Tipično su takve RAM memorija i *Cache* memorija. Nevolatilna memorija je memorija čiji sadržaj ostane netaknut nakon prekida rada računala i njegovog ponovnog pokretanja.
Kako sustav za upravljanje bazom podataka mora zadržati podatke koje pohranjuje u bazu podataka jasno je da je upravljanje nevolatilnom memorijom od velike važnosti.
SUBP mora pohranjivati podatke u nekakvu nevolatilnu memoriju. U ovom radu fokusirat ćemo se na korištenje sekundarne memorije, odnosno zapisivanja podataka na disk.

Za potrebe ovog rada u implementaciji se nećemo brinuti o nižim razinama memorijske hijerarhije. Podatke ćemo zapisivati na disk te prepuštati datotečnom i operacijskom sustavu brigu o točnom zapisu na disk.

Poglavlje 2

Logički smještaj podataka

U ovom dijelu raspravljamo o logičkom načinu pohrane podataka na disk. Jedan podatak reprezentiran je kao *tuple* odnosno zapis koji se sastoji od niza uzastupnih bajtova na nekom disku.
Zbog jednostavnosti i potreba ovog rada ograničit ćemo se isključivo na zapisima fiksne veličine te ćemo pohranjivati 32-bitne i 64-bitne cijele i decimalne brojeve s i bez predznaka.

2.1 Zapis

Najjednostavnija vrsta zapisa sastoji se od nekoliko polja fiksne veličine. Ta polja mogu biti različitih tipova kao što su cijeli brojevi, nizovi znakova (eng. *string*), datumi, ... Tipovi koji se mogu nalaziti u zapisu ovise u SUBP-u. Zapise istog tipa, odnosno zapise koji sadrže ista polja često povezujemo u tablice. Navodimo primjer SQL (*Structured Query Language*) upita korištenog za stvaranje *MySQL* tablice:

```
CREATE TABLE primjer(
    cijeli_broj INT,
    datum DATE,
    decimalni FLOAT,
    fiksni_string CHAR(100)
);
```

Zapis često započinje sa zaglavljem (eng. *header*). U zaglavlju mogu stajati neke od sljedećih informacija:

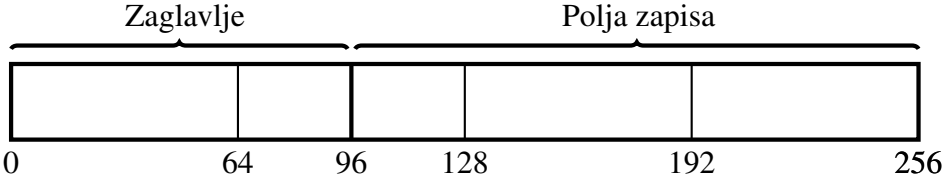
1. pokazivač na shemu zapisa

2. duljina zapisa
3. vremenski žig zadnjeg rada sa zapisom (posljednje čitanje, modifikacija, ...)
4. pokazivači na polja zapisa (važnije kod zapisa s varijabilnom duljinom)

Logički, zapise možemo organizirati horizontalno i vertikalno. Sada dajemo pregled tih razmještaja, pričamo o operacijama na bazi i napokon dajemo kratki pregled implementacije korištene u radu.

2.2 Horizontalni razmještaj podataka

Horizontalni razmještaj podataka organizira polja zapisa tako da, u skladu sa shemom, zapisuje vrijednosti polja jedna za drugim.
Na slici 2.1 vidimo kako bi izgledao zapis čije je zaglavlje sastavljeno od pokazivača na shemu i duljine zapisa, a ostala polja su 32-bitni cijeli broj, 64-bitni cijeli broj i 64-bitni cijeli broj.



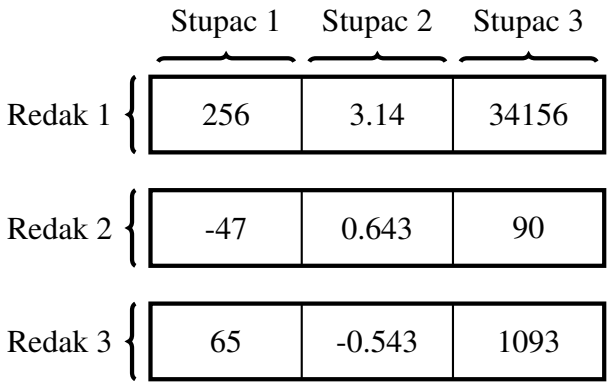
Slika 2.1: Horizontalni zapis

Na slici 2.2 vidimo primjer 3 zapisa bez zaglavlja u horizontalnoj tablici. Kao što vidimo zapisi su grupirani po recima.

Tablice u SUBP koji koriste horizontalni razmještaj podataka bit će niz uzastopnih zapisa. Nakon posljednjeg bajta jednog zapisa odmah počinje prvi bajt idućeg.
Popularni SUBP-ovi koji koriste horizontalni razmještaj su *Oracle*, *MariaDB*, *Postgres*, *IBM DB2* i *MySQL*.

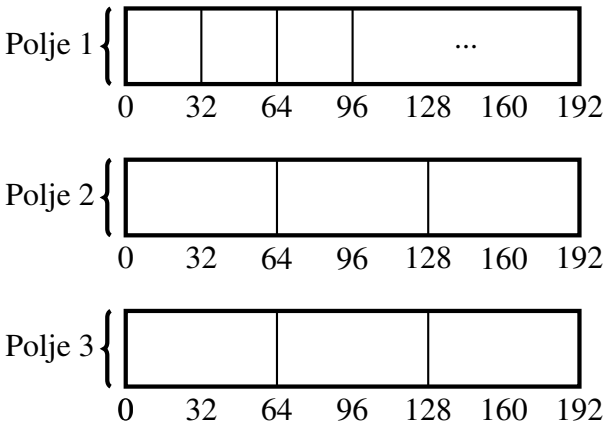
2.3 Vertikalni razmještaj podataka

Vertikalni razmještaj podataka radi suprotno. On organizira zapise po poljima, odnosno stupcima. Polja istog tipa se zapisuju jedna za drugim. Polja istog zapisa nalaze se na istom odmaku (eng. *offset*). U ovom kontekstu *offset* je redni broj vrijednosti u stupcu brojeći od nule (npr. ako se su u jednom stupcu zapisani brojevi $9, 5$ i 12, a u drugom



Slika 2.2: Primjer zapisa, horizontalna tablica

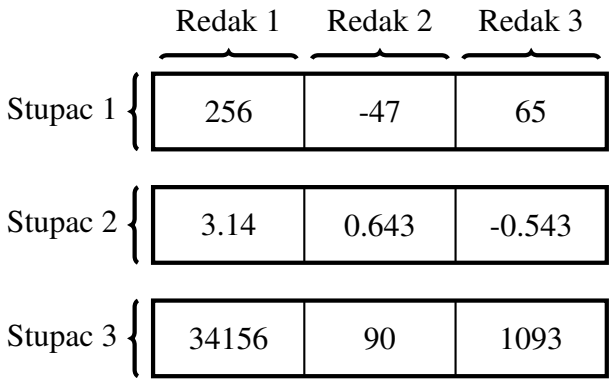
$-7.2, 0$ i -3.1 tada se 5 i 0 u svom stupcu nalaze na *offset*-u 1 i pripadaju istom zapisu). Ovakav smještaj možemo organizirati tako da zapišemo sve podatke uzastopno, odnosno prvo zapišemo sve vrijednosti prvog polja, pa sve vrijednosti drugog, ... Alternativno, možemo svaki stupac zapisivati nezavisno o ostalima.
Na slici 2.3 vidimo primjer polja zapisa sa slike 2.1 u vertikalnoj tablici.



Slika 2.3: Vertikalni zapis

Na slici 2.4 prikazani su isti zapisi kao na slici 2.2 u vertikalnom smještaju. Ovdje su zapisi grupirani po stupcima.

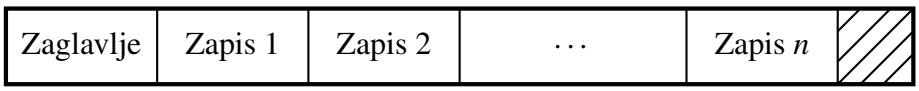
Ovakav razmještaj podataka koriste SUBP-ovi kao što su *Apache Cassandra*, *ScyllaDB*, *Redshift*, *BigQuery*, *Snowflake* ...



Slika 2.4: Primjer zapisa, vertikalna tablica

2.4 Pohrana zapisa u blokove

Zapisi se često dijele u blokove. Oni su logičke *n*-torke fiksne duljine. Duljina bloka izražena je u bajtovima i najčešće se radi o nekom višekratniku broja 512. Ovakva organizacija je česta zato što, kada od računala zatražimo da pročita nešto s diska, ono neće s diska pročitati samo 1, 2 ili 30 bajtova već će *zagrabiti* veći komad memorije. Računalo to radi zato što je pristup disku spor i ako mu već pristupamo ima smisla sa sobom povući nešto više memorije.
Na početku često može stajati i zaglavlje u kojem mogu stajati dodatne informacije o bloku. Ako zapis ne može stati u blok (recimo zauzeto je 504 bajta od 512, a zapis ima veličinu 16 bajtova) tada se taj zapis prebacuje u drugi blok, a preostale bajtove smatramo neiskorištenima.



Slika 2.5: Blok

Neke od informacija koje mogu pisati u zaglavlju su sljedeće:

1. informacije o tablici kojoj zapisi pripadaju
2. mapa *offseta* zapisa u bloku
3. vremenski žig posljednjeg rada s blokom

Promotrimo sljedeći primjer. Uzmimo blok veličine 512 bajtova. Pretpostavimo da nema zaglavlje i da u njega pohranjujemo zapise fiksne veličine od 24 bajta. Tada će u blok stati

$\lfloor 512/24 \rfloor = 21$ zapis. To znači da ćemo popuniti $21 \cdot 24 = 504$ bajtova dok će njih 8 preostati neiskorišteno.

2.5 Dodavanje, brisanje i promjena zapisa

Svaki sustav za upravljanje bazama podataka podržava operacije nad zapisima i tablicama u kojima se oni nalaze. Korisnik može dodavati zapise, brisati zapise i raditi promjene na zapisu od interesa.
U nastavku poglavlja ukratko opisujemo algoritme za izvođenje tih operacija te dajemo kôd u kojem algoritam implementiramo nad poljem cijelih brojeva u programskom jeziku C. Zbog sličnosti algoritama za horizontalne i vertikalne tablice opisujemo algoritam nad horizontalnim tablicama. Nakon svakog algoritma dajemo kratku napomenu o razlikama kod vertikalnih tablica.

Dodavanje zapisa

Da bismo dodali zapis u tablicu potrebno je napraviti sljedeće korake:

1. Pozicionirati se na kraj tablice
2. Zapisati novi zapis

```
#define MAX_SIZE 1000
bool insert(int arr[], int& size, int el) {
    if (size + 1 >= MAX_SIZE) return false;
    arr[size++] = el;
    return true;
}
```

Kod vertikalnih tablica, za svaki stupac se pozicioniramo na kraj niza vrijednosti stupca te zapišemo odgovarajuću vrijednost zapisa kojeg dodajemo.

Brisanje zapisa

Da bismo obrisali zapis iz tablice potrebno je napraviti sljedeće korake:

1. Pronaći zapis u tablici
2. Obrisati zapis
3. Pomaknuti sve zapise s desne strane jedno mjesto u lijevo

Napisana funkcija izbacuje najviše jedan element koji je jednak *el*:

```
void erase(int arr[], int& size, int el) {
    int i = 0;
    for(; i < size && arr[i] != el; ++i) {} // pronalazak
    if (i == size) return; // element nije u polju
    for(; i < size - 1; ++i)
        arr[i] = arr[i+1];
    --size;
}
```

Kod vertikalnih tablica nešto je kompleksnije pronaći zapis koji želimo obrisati. Potrebno je prolaziti kroz sve stupce te uspoređivati vrijednosti stupca na istom *offset*-u s odgovarajućim vrijednostima zapisa. Ako su sve vrijednosti jednake smatramo da smo pronašli zapis.
Sada za svaki stupac brišemo vrijednost zapisa te pomičemo sve vrijednosti s desne strane jedno mjesto u lijevo.

Promjena zapisa

Da bismo promijenili zapis u tablici potrebno je napraviti sljedeće korake.

1. Pronaći zapis koji trebamo promijeniti

2. Postaviti novu vrijednost odgovarajućeg/ih polja

U sljedećoj funkciji postavljamo vrijednost elementa jednakog *old* na *new*:

```
void update(int arr[], int& size, int old, int new_el) {
    int i = 0;
    for(; i < size && arr[i] != old; ++i) {} // pronalazak
    if (i == size) return; // element nije u polju
    arr[i] = new_el;
}
```

Kod vertikalnih tablica zapis pronalazimo kao što smo opisali kod brisanja zapisa. Sada postavljamo ažurirane vrijednosti u svaki stupac.

Kôd napisan u prethodnim funkcijama nećemo koristiti u implementaciji međutim, jasna je upotreba ovakvih algoritama.

2.6 O implementaciji SUBP

Kako je tema ovog rada usporedba učinkovitosti razmještaja podataka (ne analiziramo *join*-ove i druge operacije) reducirat ćemo se na jednu tablicu po testiranju.
Sheme našeg SUBP-a bit će tekstualne datoteke sljedećeg formata:

1. U prvom retku stoji ime tablice
2. U drugom retku stoji niz uređenih parova odvojenih znakom ; oni su odvojeni znakom : dok je format para *tip stupca : ime stupca*

Primjer sheme tablice *test*:

> *test*
> *int*32_*t* : *key*; *double* : *decimal_num*; *uint*32_*t* : *num*; *int*64_*t* : *big_num*

Zapis u implementiranom SUBP nema zaglavlje već samo polja. Zaglavlje nije bilo potrebno s obzirom da shemu držimo u odvojenoj datoteci koju SUBP učita pri početku izvođenja. Ostali metapodatci o zapisu nam neće trebati u svrhe testiranja koja ćemo raditi u ovom radu.

Tablice s horizontalno smještenim zapisima pohranjene su u binarnim datotekama koji nose ime tablice i imaju sufiks *.hor*.
Tablice s vertikalno smještenim zapisima pohranjene su u direktoriju koji nosi ime tablice. U direktoriju se nalaze binarne datoteke koje nose ime stupca i imaju sufiks *.ver*.

Tablica u ovom SUBP-u bit će jednaka direktoriju u kojem se nalazi shema tablice, jedan od prethodno opisanih smještaja podataka, binarna datoteka od 4 bajta sa sufiksom *.cnt* u kojima piše broj elemenata tablice i kasnije binarnog zapisa nekog indeksa nad tim podacima. O indeksima ćemo govoriti kasnije.

Poglavlje 3

Usporedba načina smještaja podataka

Nakon teoretskog uvoda u logički smještaj podataka želimo usporediti učinkovitost opisanih načina. Na početku, prezentiramo podatke koje ćemo generirati te ukratko opisujemo način generiranja. Zatim, dajemo uvid u značajne detalje implementacije te arhitekturu računala na kojem ćemo izvršavati testove. Napokon, motivirani s [1], pišemo testove iz kojih očekujemo uočiti razliku između logičkih smještaja. Iste testove izvršavamo te analiziramo rezultate i provjeravamo jesu li rezultati u skladu s očekivanima.

3.1 Generiranje podataka

Tablice generiramo *python* skriptom. Njena zadaća je učitati shemu formata opisanog u 2.6 i distribucije vrijednosti zadanih stupaca.
Opis distribucija nalazi se u tekstualnoj datoteci. U takvoj tekstualnoj datoteci nalazi se točno jedna linija u kojoj su znakom ; odvojene distribucije odgovarajućih stupaca. Broj distribucija treba biti jednak broju stupaca u shemi te *i*-ta distribucija odgovara *i*-tom stupcu tablice.
Svaka distribucija opisana je nizom odvojenim znakom :. Prvi element takvog niza opisuje tip distribucije, a preostali elementi su njeni parametri. Trenutačno su podržane sljedeće distribucije:

1. Gaussova (G), čiji su parametri redom očekivanje i standardna devijacija

2. Uniformna (U), čiji su parametri rubovi intervala

Distribucija za shemu iz 2.6 bi izgledala ovako:

$$U : 0 : 3000000; G : 0 : 1; G : 1500000 : 50000; U : -5000000 : 5000000$$

Prilikom pokretanja zadajemo jedan stupac čije vrijednosti će biti jedinstvene. Nazivat ćemo ga primarnim ključem tablice.

Rezultat ove skripte bit će dvije tablice koje sadrže iste podatke. Jedna će imati horizontalni razmještaj podataka, a druga vertikalni raspoređene u direktorije i binarne datoteke opisano kao u 2.6. Skripta također organizira zapise u blokove veličine 512 bajtova.
Generiramo milijun zapisa koristeći shemu iz 2.6 i distribucije iz ovog odjeljka te zadajemo prvi stupac kao primarni ključ tablice.

3.2 Napomene korištenoj arhitekturi i implementaciji

Arhitektura i korišteni alati

Svi testovi vezani uz ovaj rad izvršavat će se na *Dell*-ovom računalu sa sljedećim specifikacijama ([4]):

- **OS**: Windows 10 PRO (verzija 21H2)
- **Procesor**: Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz 1.80 GHz
- **Arhitektura**: 64-bitna (x64 procesor)
- **RAM**: 8 GB

Dajemo još i specifikacije diska korištenog za nevolatilnu memoriju:

- **Tip**: SSD
- **Model**: TOSHIBA KSG60ZMV256G M.2 2280 256GB
- **Veličina**: 256 GB

Sav kod korišten za testiranje razvijan je u programskom jeziku C++. Aplikacija je izgrađena korištenjem *CMake*-a (verzija 3.16) i korišten je prevoditelj *msvc* (Microsoft Visual Studio C++).

Sva testiranja odnosit će se na čitanje podataka s diska. Kao što već znamo čitanje s diska je *skupa* operacija. Poznato je da operacijski sustav ubrzava operacijama čitanja (i pisanja) *cache*-iranjem na raznim razinama memorijske hijerarhije ([7]). Neformalno opisano, svaki put kada aplikacija zatraži bajtove s određene lokacije od jezgre operacijskog sustava (eng. *kernel*) nekim sistemskim pozivom *kernel* će pročitati blok memorije gdje se ti bajtovi nalaze i spremiti taj blok u *kernel*-ov međuspremnik. Zatim prosljeđuje aplikaciji

točno bajtove koje je ona zatražila, ali zadržava prethodno učitani blok (ili blokove) memorije u svom međuspremniku. Ako aplikacija ponovo zatraži neke bajtove koji se nalaze na istom bloku kao prije *kernel* ovaj put neće morati raditi čitanje s diska. U ovom slučaju samo će pogledati svoj interni međuspremnik i puno brže vratiti aplikaciji zatražene bajtove.
Ovakva optimizacija sakriva pristupe disku od programera i aplikacije te uvelike ubrzava čitanje. Međutim, u ovom radu želimo imati veću kontrolu nad brojem čitanja s diska kako bismo jasnije vidjeli razlike između logičkih razmještaja, a kasnije i prednosti koje donosi indeks. U idućem odjeljku ćemo objasniti kako možemo imati bolju kontrolu nad čitanjem s diska.

Implementacija

Kao što je već spomenuto, od interesa će nam biti upiti na jednu tablicu. Implementirana je struktura opisana u 2.6.

Pristup disku

Objekti u programu koji reprezentiraju tablice čitaju s diska uz pomoć *io_handler* objekata. Poanta tih objekata je ta oni budu tanki omotači oko C biblioteke *fileapi* [6]. Ova biblioteka omogućuje interakciju s diskom, odnosno čitanje i pisanje.
Nećemo koristiti standardnu C++biblioteku za čitanje s diska zato što nam ova omogućuje zaobilaženje *cache*-iranja kao što je već opisano u ovom odjeljku.
Zaobilazak se postiže tako da se pri otvaranju odgovarajuće datoteke postavimo zastavice *FILE_FLAG_WRITE_THROUGH* i *FILE_FLAG_NO_BUFFERING* (napomena, slična stvar se na Linuxu postiže sa zastavicom *O_DIRECT*).

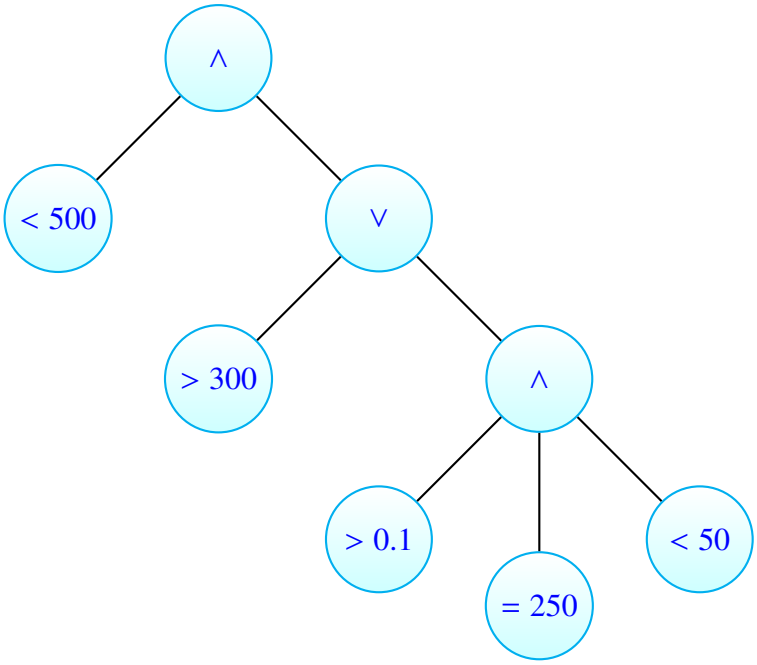
Upiti

Upiti na tablice bit će jednostavni. Upit može zatražiti sve zapise iz tablice koji zadovoljavaju određene uvjete ili zbroj podataka iz određenog stupca. Moguće je i postaviti limit na broj zapisa koje želimo dohvatiti.
Uz zbroj bi bilo trivijalno dodati druge agregacijske funkcije, ali za potrebe ovog rada bit će dovoljno samo zbroj.
Najznačajniji dio implementacije nalazi se u postavljanju uvjeta koje zapis mora zadovoljavati. Zapravo se radi o opcijama koje bi u tradicionalnom *SQL*-u došle nakon *WHERE* klauzule.
U ovoj implementaciji podržani su logički operatori $\wedge$ i $\vee$ te operatori uspoređivanja $=$, $<$

i >. U kontekstu programa objekt koji reprezentira uvjete nazivamo filter.
Svaki filter je implementiran kao stablo u kojem svaki vrh ima proizvoljan broj djece. Svaki list u ovom stablu označen je operatorom uspoređivanja, ostali vrhovi označeni su nekim od logičkih operatora. Listovi također drže vrijednost s kojom će raditi usporedbu te indeks stupca nad kojim rade usporedbu.
Promotrimo upit nad tablicom opisanom shemom iz 2.6 napisan u *SQL*-u:

```
id < 500 AND (num > 300 or (percentage > 0.1 and num = 250 and
   cnt < 50))
```

Na slici 3.1 vidimo reprezentaciju upita objektom tipa filter.



Slika 3.1: Reprezentacija upita

Na slici nisu napisani redni brojevi stupaca, ali jasni su iz promatranja upita i stabla.

Evaluacija upita

Kod **horizontalnih** tablica evaluacija je nešto jednostavnija.

1. za svaki blok memorije
   a) kreiramo retke

   b) evaluiramo upit ako postoji
   c) ako je uvjet zadovoljen radimo traženu operaciju (spremamo redak za povratnu vrijednost ili agregiramo zadani stupac)

2. završavamo ako smo prošli sve retke ili dostigli postavljeni limit

Važno je pripaziti da zanemarimo *višak* u blokovima ako on postoji.

Kod **vertikalnih** tablica evaluacija je malo složenija. Napomenimo da ako je povratna vrijednost upita agregacija po nekom stupcu smatramo da je ona spomenuta u upitu čak i ako nije navedena u uvjetima.

1. učitamo po jedan blok svakog stupca spomenutog u upitu
2. evaluiramo upit, ako je zadovoljen pamtimo *offset* stupca (time i zapisa) ili agregiramo vrijednost ovisno o tipu povratne vrijednosti
3. ako je povratna vrijednost upita niz zapisa
   a) pročitamo odgovarajuće blok(ove) preostalih stupaca
   b) iz zapamćenih *offset*-a pročitamo polja koja su dio zapisa
   c) spremamo redak
4. završavamo ako smo prošli sve elemente ili dostigli postavljeni limit

Primjetimo da je u koraku 3*a* napisano *blok(ove)*.

Ako radimo upit s uvjetom na nekoj 4 bajtnoj vrijednosti s blokom veličine 512 dobit ćemo točno 128 vrijednosti. Međutim, stupci koje bi trebali dohvatiti kasnije mogu imati 8 bajtova te se u bloku iste veličine nalazi samo 64 vrijednosti. Sada je jasno da ako želimo kreirati cijeli zapis trebamo pročitati 2 bloka 8 bajtnih vrijednosti.

Prednost ovog pristupa je što možemo izbjeći nepotrebno učitavanje idućeg bloka ako uvjeti upita nisu zadovoljeni. Također, u slučaju agregacijskih upita u potpunosti zanemarujemo vrijednosti stupaca koje ne agregiramo.

3.3 Testovi

Kao što smo već opisali možemo napraviti jako raznovrsne uvjete za naše upite. Inspirirani s [1] radit ćemo sljedeće upite:

1. dohvat svih zapisa
2. dohvat svih zapisa filtriranih po jednom ili više stupaca
3. suma svih vrijednosti stupca
4. suma svih vrijednosti stupca filtriranih po jednom ili više stupaca
5. dohvat zapisa po primarnom ključu

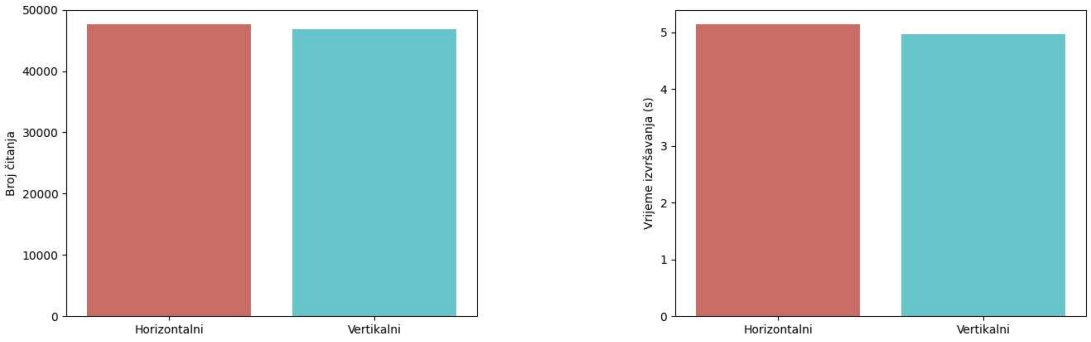
Testove ćemo raditi na podacima generiranim prethodno objašnjenim *python* skriptama.

3.4 Rezultati

Sada prezentiramo i tumačimo upite nad horizontalnim i vertikalnim tablicama redom kojim smo ih opisali u prethodnom odjeljku. Svako mjerenje predstavljamo s dva grafa. Na jednome će pisati broj čitanja s diska za oba tipa tablica, a na drugom vrijeme potrebno da bi se taj upit izvršio.

Dohvat zapisa

Na slici 3.2 vidimo da je za dohvat svih zapisa potrebno podjednako vremena i čitanja s diska. S obzirom da je potrebno dohvatiti jednak broj podataka ovakav rezultat ima smisla. Primjećujemo da je vertikalni zapis ipak nešto brži i koristi manje čitanja.
Ovo objašnjavamo činjenicom da kod horizontalnih tablica blokovi imaju neiskorištene bajtove dok kod vertikalnog smještaja u blokove spremamo vrijednosti čiji broj bajtova dijeli broj bajtova bloka (4 i 8 dijele 512).

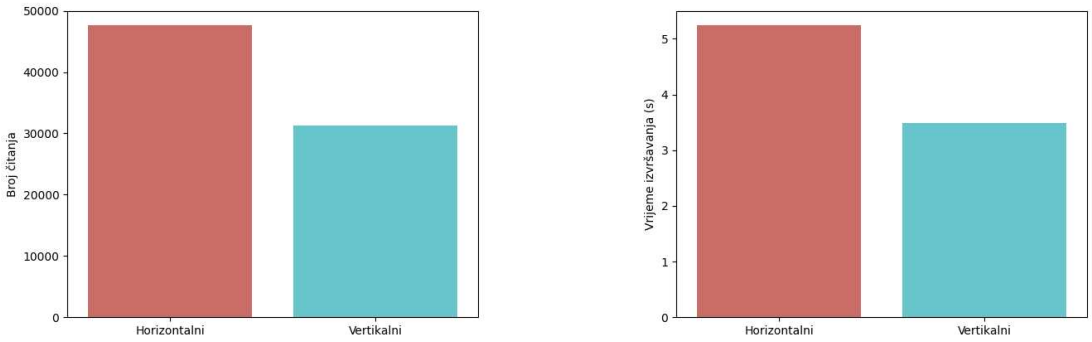


Slika 3.2: Dohvat svih zapisa

SQL upit ekvivalentan ovom testu bio bi:

```
SELECT * FROM test;
```

Na slici 3.3 vidimo rezultate dohvata svih zapisa sa zadanim uvjetom na jedan stupac. Konkretno, zadan je uvjet da se vrijednosti drugog stupca nalaze između 0.33 i 0.7. Već primjećujemo značajniju razliku u korist vertikalnog razmještaja. Do ove razlike dolazi zato što ne učitavamo dodatne stupce kada uvjet nije zadovoljen.



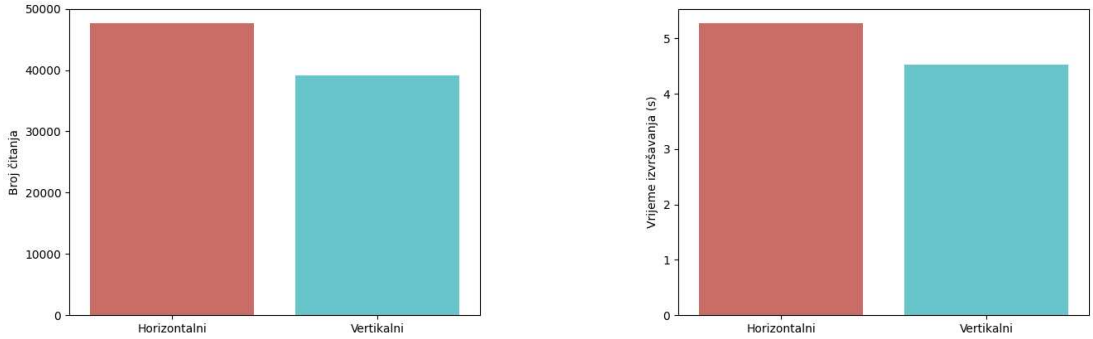
Slika 3.3: Dohvat svih zapisa s uvjetom na jedan stupac

SQL upit ekvivalentan ovom testu bio bi:

```
SELECT * FROM test WHERE decimal_num > 0.33 AND decimal_num < 0.7;
```

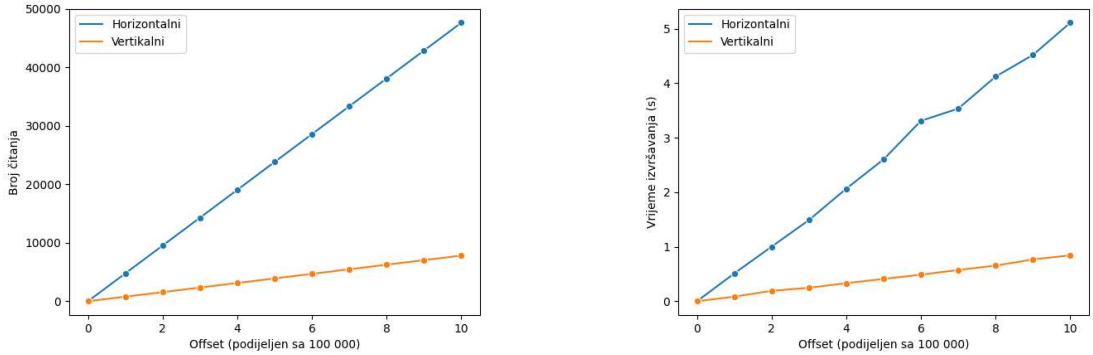
Na slici 3.4 vidimo rezulate dohvata svih zapisa sa zadanim uvjetom na dva stupca. Zadano je da vrijednosti prvog stupca trebaju biti veće od $1,700,000$ ili da vrijednosti četvrtog stupca trebaju biti manje od $-3,300,543$. Unatoč manjoj razlici u odnosu na prethodni test vertikalni razmještaj iz sličnih razloga ima bolje rezultate. Razlika je manja zato što je uvjet ipak puno slabiji te ga je više zapisa zadovoljilo što je rezultiralo dodatnim čitanjem.
SQL upit ekvivalentan ovom testu bio bi:

```
SELECT * FROM test WHERE key > 1700000 AND big_num < -3300543;
```



Slika 3.4: Dohvat svih zapisa s uvjetom na dva stupca

Na slici 3.5 prikazan je dohvat cijelog zapisa po zadanom primarnom ključu. Dohvaćali smo svaki sto tisućiti zapis počevši od prvog. Vidimo da vertikalni razmještaj treba značajno manje čitanja s obzirom da ne treba učitavati cijeli zapis već samo primarni ključ i tek kada pronađe zadanu vrijednost učitava vrijednosti preostalih stupaca. Na grafu malo jasnije vidimo i činjenicu da datoteku čitamo sekvencionalno zbog linearnog rasta broja čitanja.



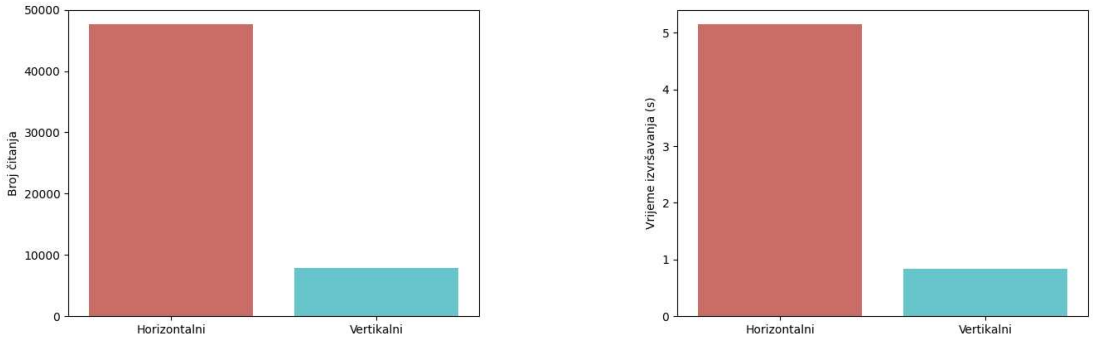
Slika 3.5: Dohvat zapisa po primarnom ključu

SQL upit (gdje je x vrijednost koju tražimo) ekvivalentan ovom testu bio bi:

```
SELECT * FROM test WHERE key = x;
```

Agregacija

Na slici 3.6 vidimo rezultate zbrajanja svih vrijednosti (konkretno trećeg stupca). Vidimo veliku razliku u korist vertikalnog razmještaja. Do nje dolazi zato što je potrebno učitati samo jedan stupac kako bismo izvršili agregaciju, dok kod horizontalnog moramo proći kroz cijelu tablicu i učitati cijeli zapis.



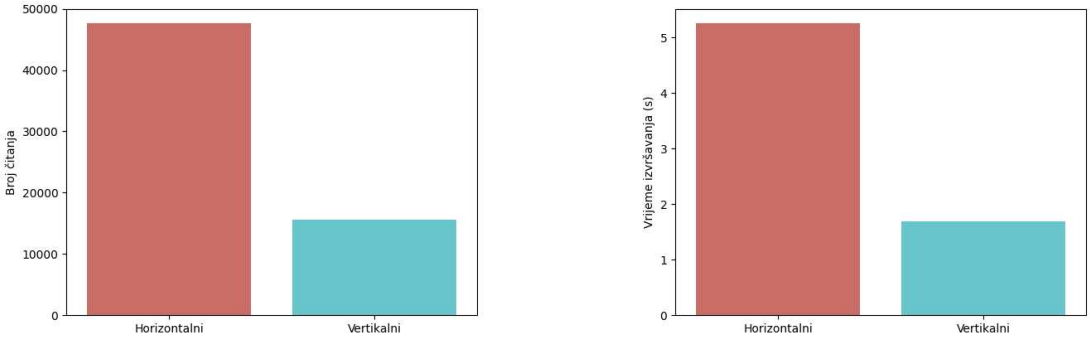
Slika 3.6: Suma svih zapisa

SQL upit ekvivalentan ovom testu bio bi:

```
SELECT sum(num) FROM test;
```

Na slici 3.7 vidimo rezultate zbrajanja svih vrijednosti (drugog stupca) s uvjetima da se vrijednost nalazi između 0.33 i 0.7 (kao u prethodnom pododjeljku). Ponovo vidimo da je vertikalni razmještaj puno efikasniji, ali visina stupca je otprilike duplo veća nego u ovom testu. Ako se vratimo na shemu iz 2.6 vidimo da je drugi stupac 8 bajtna vrijednost, a treći 4 bajtna što objašnjava ovakav rezultat.
SQL upit ekvivalentan ovom testu bio bi:

```
SELECT sum(decimal_num) FROM test WHERE decimal_num > 0.33 AND
   decimal_num < 0.7;
```



Slika 3.7: Suma svih zapisa s uvjetom na jedan stupac

3.5 Zaključak

Zaključno, vertikalni razmještaj pokazao se boljim u izvršavanju svih upita. Značajnu prednost pokazuje pogotovo u agregacijskim s obzirom da za njih treba učitati samo jedan stupac. Zaključujemo da vertikalne tablice želimo koristiti u slučajevima upotrebe kada će se na bazu postavljati puno upita.
Prednosti horizontalnog smještaja nisu vidljive u prethodnim testovima. Međutim, jasno je da porastom broja stupaca raste i broj zapisivanja na disk u slučaju ubacivanja novog podatka dok u horizontalnom smještaju ne dolazi do promjene i uvijek je potrebno jedno pisanje za novi zapis. Slično možemo zaključiti i za brisanje iz tablice ili modifikaciju zapisa. Vidimo da je horizontalni smještaj učinkovitiji u slučaju intenzivnijeg dodavanja i brisanja zapisa. Bliskost polja u horizontalnom smještaju također ima prednost pri čitanju zapisa ako znamo *offset* u tablici. Kada je on zadan za čitanje zapisa u horizontalnoj tablici potrebno je jedno čitanje, dok je u vertikalno potrebno onoliko čitanja koliko imamo stupaca. Realna u situacija u kojoj znamo *offset* zapisa je kada imamo indeks nad nekim stupcem (recimo primarnim ključem). Više o indeksima pričat ćemo u sljedećem poglavlju.

Poglavlje 4

Indeksi

Do sada smo pričali o običnom zapisivanju podataka na disk. Također, uvjerili smo se kako sekvencijalno čitanje nije upotrebljivo za bilo kakvu bazu podataka koja bi se koristila u produkciji.
Razlog ovome je naravno činjenica da je čitanje s diska sporo. Htjeli bismo na neki način odraditi što više posla u brzoj, glavnoj memoriji te što manje puta odlaziti na disk.
Tu u priču ulaze indeksi. Indeks je bilo koja struktura podataka koja prima vrijednost jednog ili više polja zapisa te pronalazi taj zapis *brzo*. Drugim riječima, indeks omogućuje pronalazak zapisa tako što nam smanjuje ukupan broj zapisa koje trebamo pregledati.
Polja na kojima se bazira indeks se zovu **ključevi pretraživanja** ili samo **ključevi**.

U ovom poglavlju dajemo pregled tipova indeksa i u početku prezentiramo nekoliko jednostavnijih primjera. Nakon toga definiramo najpoznatiji indeks, B-stablo te pričamo o korištenju *hash* tablice kao indeksa. Reduciramo se na jednodimenzionalne indekse (indekse po jednom stupcu).

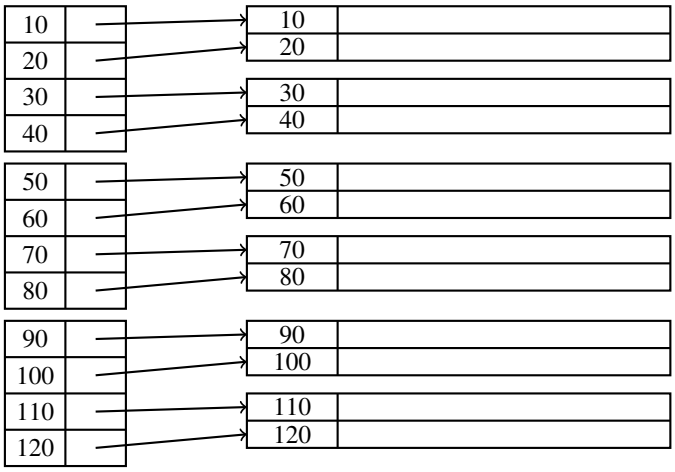
4.1 Osnovno o indeksima

U ovom odjeljku uvodimo osnovnu terminologiju te dajemo kratak pregled osnova o indeksima i uvodimo neke termine koje ćemo koristiti.
Pohrana podataka sastoji se od raznih datoteka. **Podatkovne datoteke** koriste se za spremanje, na primjer relacija. Za svaku podatkovnu datoteku može postojati neki broj **indeksnih datoteka**. Indeksne datotetke zadužene su za pohranu struktura podataka za indeksiranje po nekoj vrijednosti.

Indeksi mogu biti **gusti** (eng. *dense*) ili **raštrkani** (eng. *sparse*). Gusti indeks je onaj za koji u indeksnoj datoteci postoji zapis za svaki zapis u podatkovnoj datoteci. S druge strane raštrkani indeks je onaj koji u svojoj indeksnoj datoteci drži podatke o samo nekim zapisima. Na primjer, često imamo po jedan zapis u indeksnoj datoteci po bloku podatkovne datoteke (što naravno može povući zahtjeve na strukturu podatkovne datoteke).
Indekse također dijelimo i na **primarne** i **sekundarne**. Primarni indeksi[1] određuju lokaciju zapisa u podatkovnoj datoteci (preciznije njihova lokacija u indeksnoj datoteci jednoznačno određuje lokaciju u podatkovnoj), dok sekundarni ne rade to. Više o sekundarnim indeksima reći ćemo kasnije.

Sekvencionalne datoteke

Sekvencionalne datoteke su datoteke u kojima su zapisi sortirani po primarnom ključu. Zapisi su raspoređeni u blokove kao na slici 4.1 s desne strane. Primjetimo kako se radi o blokovima u koja stanu samo dva zapisa. Ovakva situacija se skoro nikada neće dogoditi u stvarnosti, ali olakšava crtanje primjera.



Slika 4.1: Sekvencionalna datoteka i gusti indeks

Gusti indeksi

Ako su zapisi u podatkovnoj datoteci sortirani nad njima možemo izgraditi gusti indeks. Radi se o nizu blokova koji pohranjuju samo ključeve zapisa (podsjećamo, ključ je vrijednost polja po kojem indeksiramo) i pokazivače na zapise.

[1]Primarni indeksi često se vežu uz primarne ključeve pa se pretpostavlja jedinstvenost vrijednosti nad kojima gradimo primarni ključ

Od gustih indeksa zahtijevamo da su ključevi pohranjeni u indeksnoj datoteci održani u istom sortiranom redosljedu kao i zapisi u podatkovnoj datoteci.
Indeksna datoteka je očito puno manja od podatkovne i ponekad ju možemo držati u radnoj memoriji.
Na slici 4.1 s lijeve strane vidimo primjer gustog indeksa.

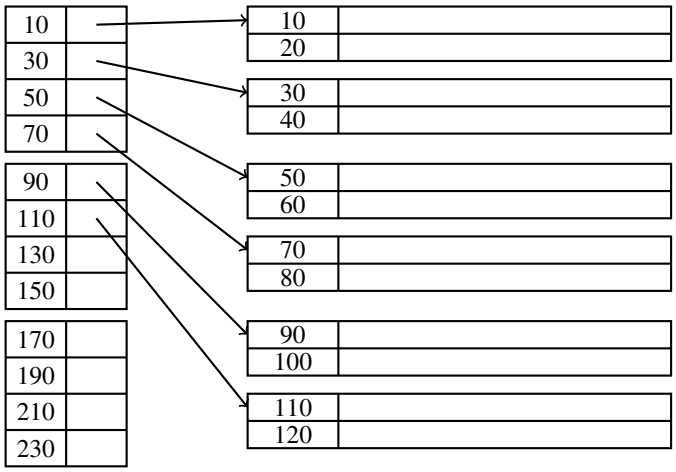
Faktori koji čine pretraživanje korištenjem gustog indeksa efikasnim su sljedeći:

1. Broj indeksnih blokova uglavnom je manji od broja podatkovnih blokova
2. Zbog sortiranosti ključeva možemo koristiti binarno traženje
3. Ponavljamo, ukoliko je datoteka dovoljno mala možemo ju držati u radnoj memoriji zbog čega neće doći do čitanja s diska

Raštrkani indeksi

Kao što smo prethodno spomenuli raštrkani indeksi često imaju jedan par ključ-vrijednost po bloku podatkovne datoteke. Zauzimaju manje prostora od gustih. Za razliku od gustog indeksa, kojeg možemo koristiti na bilo kakvim podatkovnim datotekama, u slučaju raštrkanog moramo imati podatkovnu datoteku sortiranu po ključu pretraživanja.

Na slici 4.2 vidimo primjer raštrkanog indeksa za koji vrijedi da svaki zapis pokazuje na prvi zapis bloka u podatkovnoj datoteci. Ključevi s vrijednosti većom od 110 u indeksu pokazuju na neke blokove koji nisu na slici pa ne crtamo njihov pokazivač.

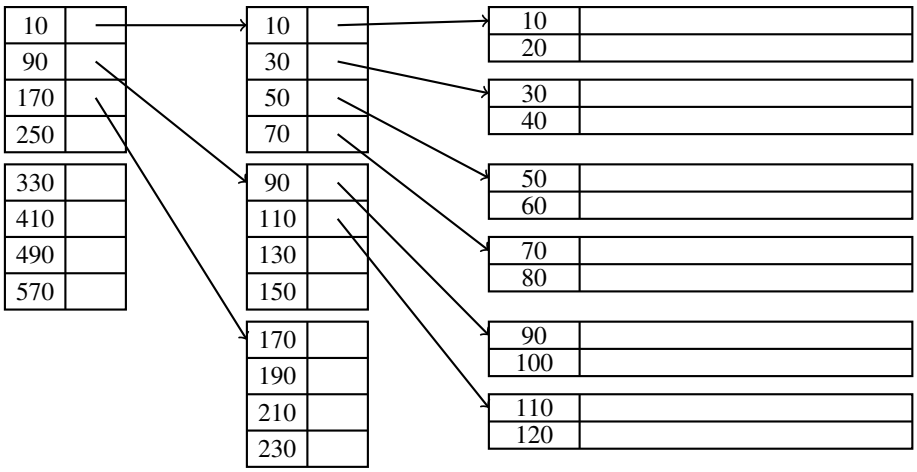


Slika 4.2: Raštrkani indeks

Kako bismo pronašli zadani ključ pretražujemo raštrkani indeks za najvećim ključem koji je manji ili jednak traženoj vrijednosti. Kako se radi o sortiranoj datoteci možemo koristiti binarno pretraživanje. Pokazivač do kojeg dolazimo binarnim pretraživanjem pokazuje na blok u kojem se potencijalno nalazi tražena vrijednost. Tada učitavamo blok u memoriju te ponovno binarnim traženjem utvrđujemo egzistenciju vrijednosti u učitanom bloku.

Višerazinski indeksi

U slučaju ogromnih podatkovnih datoteka moguće je da imamo i jako velike indeksne datoteke. Zapravo razmatramo slučaj u kojem je i sama indeksna datoteka sastavljena od velikog broja blokova. Tada možemo postaviti indeks na indeks.
Za dodani indeks kažemo da se nalazi na drugoj razini. Naravno, moguće je dodavati i dodatne razine ukoliko je to potrebno.
Na slici 4.3 vidimo primjer dvorazinskog indeksa.



Slika 4.3: Indeks s dvije razine

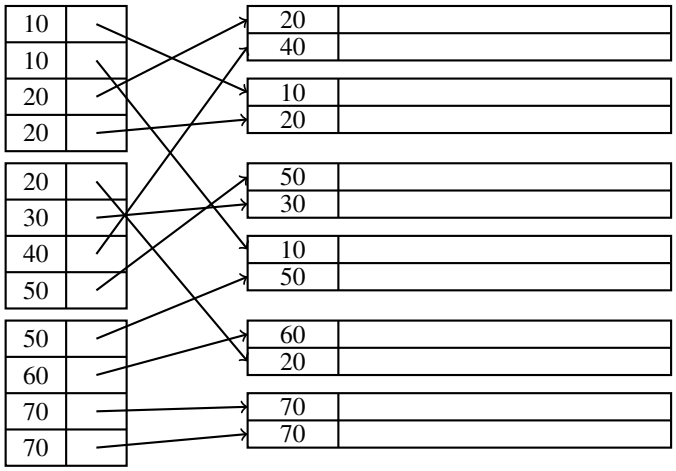
Indeks prve razine smije biti i raštrkan i gust, ali indeksi na višim razinama moraju biti isključivo raštrkani. Jasno je da ne bismo ništa dobili postavljanjem gustog indeksa na višu razinu osim nepotrebnog zauzimanja memorije.
Slično ovoj ideji i puno popularnije je B-stablo o kojem ćemo pričati nešto kasnije.

Sekundarni indeksi

Sekundarni indeks je, poput svakog drugog indeksa, struktura podataka koja služi pronalasku zapisa na temelju vrijednosti zadanog polja.

Za razliku od primarnih indeksa, sekundarni ne određuju lokaciju zapisa u podatkovnoj datoteci.

Važno je primjetiti kako su svi primjeri indeksa do sada imali sortirane podatkovne datoteke po polju po kojem su bile indeksirane. To ne mora biti u slučaju sekundarnih indeksa. Oni ne uvjetuju sortiranost podatkovne datoteke. Samim time oni ne smiju biti raštrkani već su uvijek gusti. Razlog tome je činjenica da oni sami nemaju nikakav utjecaj na lokaciju cijelog zapisa.
Naravno, zapisi u samom indeksu ostaju sortirani. Na slici 4.4 vidimo primjer sekundarnog indeksa.



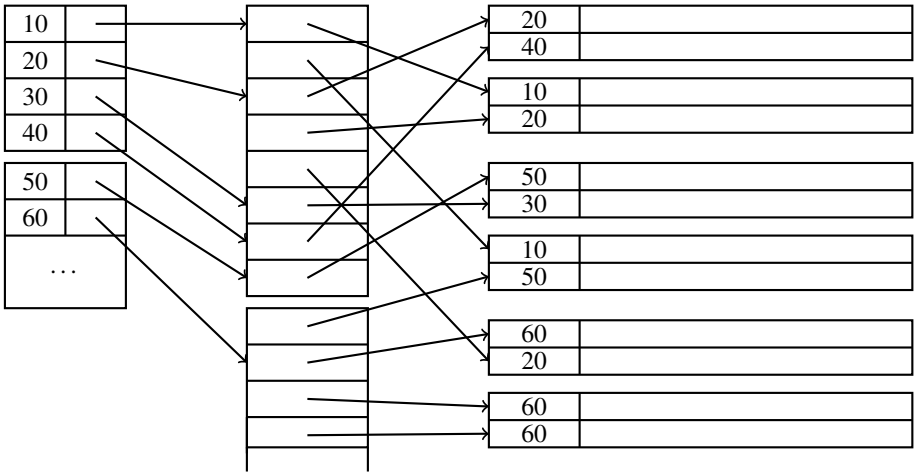
Slika 4.4: Sekundarni indeks

Primjetimo kako bismo za učitavanje zapisa s ključem 20 bilo potrebno učitati čak tri različita bloka. U slučaju primarnog indeksa podatkovna datoteka bila bi sortirana i trebali bismo imati samo dva čitanja. Jasno je da primarni indeks u ovom slučaju omogućuje manji broj čitanja. Međutim u slučaju sekundarnog indeksa ne trebamo održavati sortiranost podatkovne datoteke već samo indeksne. Također, moguće je izgraditi više sekundarnih indeksa nad jednom datotekom i njenim zapisima dok je moguće izgraditi samo jedan primarni indeks s obzirom da ne možemo održavati sortiranost zapisa po dva različita polja.

Ušteda prostora kod sekundarnih indeksa

Kod sekundarnih indeksa vidjeli smo i mogućnost postojanja duplikata. Moguće je imati velik broj ponavljajućih vrijednosti. To znači gubitak prostora za nepotrebno zapisivanje istih vrijednosti. Za rješavanje tog problema uvodimo koncept pretinca (eng. *bucket*).
Uvođenjem pretinaca u indeksnoj datoteci zabranjujemo pojavu duplikata. Svaki ključ u

indeksnoj datoteci sada uz sebe drži pokazivač na neku lokaciju u pretincu. U pretincu se nalaze pokazivači na zapise u podatkovnoj datoteci. Ako postoje duplikati ključa tada će pokazivači na njih unutar pretinca biti jedni kraj drugih. Ilustraciju vidimo na slici 4.5.



Slika 4.5: Sekundarni indeks s pretincima

Ukoliko želimo pronaći sve ključeve s vrijednosti 10 prvo pronalazimo odgovarajući ključ u sekundarnom indeksu. Njegov pokazivač nas usmjerava na pretinac. Zatim redom čitamo pokazivače u pretincu dok god nas oni usmjeravaju prema vrijednosti ključa. Kada učitamo vrijednost različitu od ključa znamo da trebamo stati.

Nakon općenite priče o indeksima i nekoliko primjera prelazimo na indekse koji se često koriste u modernim SUBP. Za početak proučavamo B-stabla, a zatim *hash* indekse.

4.2 B-stablo

B-stabla možemo smatrati poopćenjima višerazinskih indeksa. Ova struktura koristi se u većini komercijalnih SUBP-ova.
Glavne karakteristike B-stabla su to da one automatski održavaju koliko god razina indeksa je potrebno ovisno o veličini datoteke koju indeksiramo. Također, osiguravaju da je barem polovica svakog njihovog bloka u upotrebi iskorištena.
Preciznije, konkretna struktura o kojem pričamo naziva se B+ stablo, ali u ostatku tekstu na njega se referenciramo kao na B-stablo.

Struktura stabla

Blokovi koji čine B-stablo organizirani su u *balansirano* stablo. Stablo je balansirano ako su svi putevi od korijena do lista jednake duljine. B-stablo podijeljeno je u tri sloja. Korijen, srednji sloj i listovi. Dopušten je proizvoljan broj slojeva, konkretno proizvoljan broj srednjih slojeva.

Uz svaki indeks ovog tipa asociramo parametar n. To je parametar koji određuje strukturu blokova B-stabla. Svaki blok mora imati mjesta za n ključeva traženja i $n + 1$ pokazivača. Odabir ovog parametra ovisi o veličini bloka te tipovima pokazivača i ključeva. Želimo odabrati što veću vrijednost n kako bismo mogli pohraniti što više ključeva u jedan blok.

Uzmimo primjer bloka veličine 512 bajtova s ključem veličine 8 bajtova i pokazivačem veličine 4 bajta. Tražimo najveći n takav da:

$$8n + 4(n + 1) = 12n + 4 \leq 512$$

Zaključujemo da je n jednak 42 što znači da u blok mogu stati 42 ključa.

Važna pravila o podatcima u B-stablu:

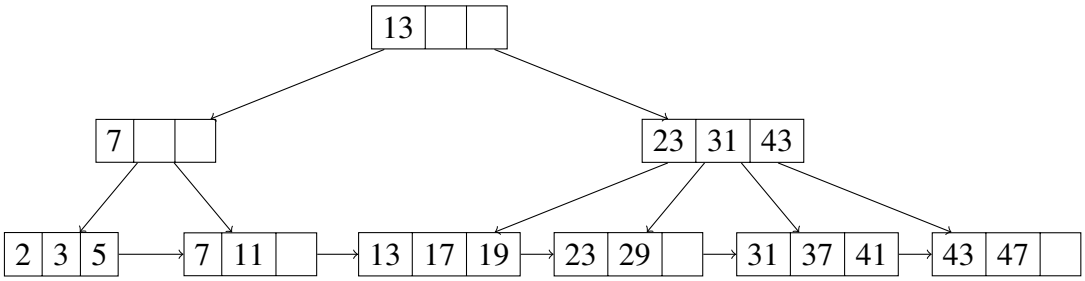
- Ključevi u listovima moraju biti jednaki ključevima iz podatkovne datoteke. Ključevi u listovima su sortirani uzlazno, gledano slijeva na desno.
- U korijenu stabla iskorištena su bar dva pokazivača[2].
- Posljednji pokazivač u listu pokazuje na blok s desne strane, odnosno na list koji sadrži idući po vrijednosti ključ. Zahtjevamo da je najmanje $\lfloor (n+1)/2 \rfloor$ od preostalih n pokazivača iskorišteno te pokazuju na neke zapise. Neiskorišteni pokazivači ne pokazuju nigdje, a i-ti pokazivač pokazuje na zapis s i-tim ključem.
- Pokazivači vrha srednjeg sloja pokazuju na blokove B-stabla na nižim razinama. Najmanje $\lceil (n + 1)/2 \rceil$ pokazivača treba biti u upotrebi (u slučaju korijena donja granica je 2). Ako je iskorišteno j pokazivača tada znamo da postoji $j - 1$ ključeva s vrijednostima $K_1, \ldots K_{j-1}$. Prvi pokazivač pokazuje na dio stabla gdje se nalaze blokovi u kojima se nalaze vrijednosti ključa manje od K_1. Za $i \in \{2, \ldots, j - 1\}$ i-ti pokazivač pokazuje na mjesto u stablu gdje možemo pronaći ključeve čija je vrijednost veća ili jednaka K_{i-1}, a manja od K_i. Posljednji pokazivač pokazuje na mjesto u stablu gdje možemo pronaći ključeve čija je vrijednost veća od K_{j-1}. Napomenimo

[2]U trivijalnom slučaju podatkovne datoteke s jednim zapisom korijen bi imao samo jedan pokazivač. Ovaj trivijalni slučaj je zanemaren u daljnjim razmatranjima.

da zapisi neće svi zapisi s ključem manjim od K_1 i većim od K_{j-1} biti dohvatljivi iz ovog bloka već će se do njih dolaziti preko nekog drugog bloka na istoj razini.

- Svi iskorišteni pokazivači i ključevi zapisani su na početku bloka, osim $(n + 1)$-og pokazivača u listu koji pokazuje na idući list.

Na slici 4.6 vidimo prikaz B-stabla s 3 razine i parametrom $n = 3$. Pretpostavili smo da se podatkovna datoteka sastoji od zapisa čiji su jedinstveni ključevi prosti brojevi od 2 do 47. Primjetimo kako je u ovom slučaju B-stablo primarni indeks. Također, radi se o gustom indeksu. Vrijednosti ključa u listovima su, kao što se vidi na slici, sortirani uzlazno slijeva na desno.



Slika 4.6: B-stablo

Primjene B-stabla

B-stablo može implementirati sve tipove indeksa iz odjeljka 4.1. Evo neki primjeri:

1. B-stablo može biti gusti primarni ključ za podatkovnu datoteku. Za svaki zapis u podatkovnoj datoteci postoji točno jedan par ključa i pokazivača u nekom listu stabla. Ne zahtjevamo sortiranost u podatkovnoj datoteci.

2. Ako je podatkovna datoteka sortirana B-stablo može implementirati raštrkani indeks. Na primjer jedan par ključa i pokazivača u listu po bloku podatkovne datoteke.

Postoje implementacije B-stabla koje ne zahtjevaju jedinstvenost ključa pretraživanja, ali o njima nećemo raspravljati u ovom radu.

Pretraživanje u B-stablu

Kao što smo rekli, pretpostavljamo da ne postoje duplikati u listovima stabla. Također, pretpostavljamo da je B-stablo gusti indeks.

Često pretražujemo neku tablicu za zapisom koji ima zadanu vrijednost ključa K. Zadani ključ tražimo u B-stablu rekurzivno počevši od korijena (blok koji sadrži korijen stabla često se nalazi u memoriji, dok se ostali blokovi, odnosno čvorovi stabla po potrebi učitaju s diska). Procedura pretraživanja je induktivna:

- **BAZA:** ako smo u listu pretražimo ključeve. Ako je i-ti ključ jednak K tada nas i-ti pokazivač vodi prema željenom zapisu. Ako nijedan ključ nema vrijednost K onda takav zapis ne postoji u bazi.
- **KORAK:** Ako nismo u listu tada vrh ima ključeve $K_1, \ldots K_n$. Po pravilima opisanim u prethodnom odjeljku određujemo koji pokazivač slijedmo za daljnje čitanje.

Promotrimo ponovo sliku 4.6. Pretpostavimo da tražimo vrijednost 40. Počevši od korijena te usporedbom s 13 zaključujemo da moramo slijediti drugi pokazivač. Sada vidimo kako je $31 \leq 40 < 43$ pa slijedimo treći pokazivač drugog čvora na drugoj razini stabla. Napokon dolazimo do lista čiji su ključevi $31, 37$ i 41. Zaključujemo da ta vrijednost ne postoji u stablu.
U slučaju da tražimo vrijednost 7 ponovno krećemo iz korijena. Ovaj put pratimo prvi pokazivač jer je $7 < 13$. Zatim u prvom čvoru druge razine pratimo drugi pokazivač zato što je $7 \leq 7$. Napokon, dolazimo do lista čiji elementi su 7 i 11 te pratimo prvi pokazivač do stvarnog zapisa u podatkovnoj datoteci.

Pretraživanje raspona

B-stabla ne koriste se samo za traženje jedne vrijednosti po ključu. Jako su efikasna u pretraživanjima raspona. Naravno, radi se o dohvatu svih zapisa čiji ključ se nalazi u nekom intervalu.
Na primjer, struktura može značajno ubrzati upite slične kao sljedeći SQL upit:

```
select * from t where col >= 100 and col <= 200;
```

Postupak pronalaska svih ključeva u intervalu $[a, b]$ je sljedeći. Kao u prethodnom odjeljku pronađemo ključ s vrijednosti a ako postoji. Ako ne postoji u listu do kojeg smo došli učitamo zapise na koje pokazuju ključevi čija je vrijednost veća od a, ali nije veća od b. Učitavamo sve zapise na koje pokazuju ključevi koji nisu strogo veći od b. Ako pročitamo sve ključeve lista, a još nismo došli do vrijednosti veće od b pratimo posljednji pokazivač u listu koji pokazuje na idući list u stablu. Ako ne postoji idući list onda stajemo s algoritmom.
Opisani algoritam je dobar i ako je $a = -\infty$ ili $b = +\infty$.

Promotrimo ponovno sliku 4.6. Pretpostavimo da je zadan interval $[30, 44]$. Prvo tražimo 30 što nas dovodi do lista gdje se nalaze 23 i 29. Kako nijedan broj u listu nije veći ili jednak 30 pratimo pokazivač do idućeg bloka. U idućem bloku nailazimo na 31, 37 i 41. Zapise na koje pokazuju ključevi s vrijednosti 31 i 37 učitamo, a kad dođemo do 41 završavamo algoritam bez učitavanja.

Ubacivanje novog ključa u B-stablo

Zapisi u bazama podataka često se mijenjaju. Moguće je brisati i dodavati neke podatke. Indeks mora biti konzistentan s podatkovnom datotekom s kojom je povezan. Ako promjenimo nesto u podatkovnoj datoteci ta promjena mora biti vidljiva u indeksu.
U ovom odjeljku pričamo o dodavanju ključa u B-stablo i zatim navodimo primjer. Proces ubacivanja je rekurzivan:

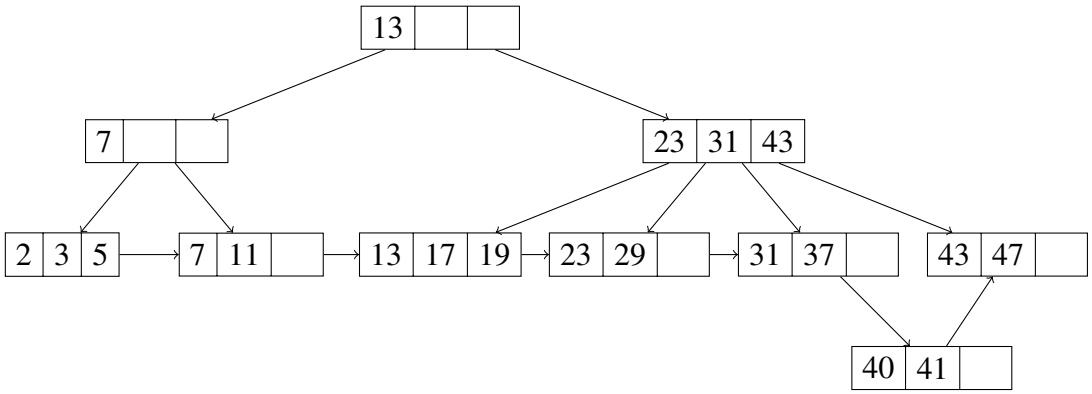
- pokušamo ubaciti ključ u odgovarajući list ako ima mjesta u tom listu
- Ako nema mjesta u listu, podijelimo ga na dva lista te podijelimo ključeve tako da je svaki list do pola pun.
- Podjela čvora na danoj razini utječe na razinu iznad. Na višoj razini se događa ubacivanje para ključ-pokazivač rekurzivno, na isti način kao u prethodnoj točki
- Ukoliko dođemo do korijena i nema mjesta za novi ključ radimo istu stvar kao prije. Jedina razlika je što sada postavljamo novi korijen koji ima pokazivače na točno dva čvora (prisjetimo se, čvor je jedini kojem je dozvoljeno imati samo dva pokazivača, svi drugi imaju donje granice ovisne o n)

Kada napravimo podjelu u nekom čvoru moramo pripaziti što se događa s ključevima. Neka je N list kapaciteta n u kojem je već spremljeno n ključeva. Pretpostavimo da pokušavamo ubaciti novi ključ zajedno s odgovarajućim pokazivačem. Stvaramo novi list M koji će biti *brat* lista N s desne strane. Prvih $\lceil (n+1)/2 \rceil$ parova ključ-pokazivač ostaje u N, dok preostale prebacujemo u M. Pokazivač na (n+1)-om mjestu u N će pokazivati na M dok će u M pokazivati na onaj list gdje je N pokazivao prije.
Neka sada čvor N nije list te ima kapacitet od n ključeva i $n+1$ pokazivača. Ako je u nižoj razini došlo do podjele i dodavanje novog pokazivača čvoru N uvećamo kapacitet tako da mora doći do podjele i na ovoj razini radimo sljedeći postupak:

1. Stvaramo čvor M, brat čvoru N neposredno desno
2. U N ostavimo prvih $\lceil (n+2)/2 \rceil$ pokazivača u N te preostalih $\lfloor (n+2)/2 \rfloor$ prebacimo u M

3. Prvih $\lceil n/2 \rceil$ ključeva ostalo je u N, dok je posljednjih $\lfloor n/2 \rfloor$ u M. Primjetimo da je preostao jedan ključ koji nije otišao ni u N ni u M. Označimo ključ s K. On predstavlja najmanji ključ dohvatljiv preko prvog djeteta od M. K ćemo ubaciti u roditelja od N i M te on služi za podjelu pretraživanja između N i M

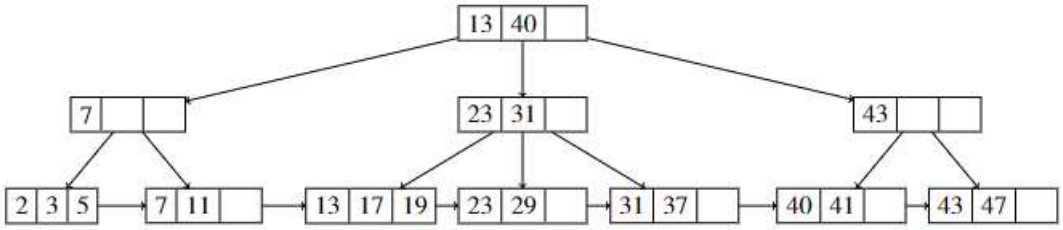
Pokažimo na stablu 4.6 ubacivanje čvora s ključem 40. Ispravan list za ubacivanje je peti list. Nakon ubacivanja on sadrži, redom, ključeve 31, 37, 40 i 41. Kako smo prešli kapacitet lista potrebno je stvoriti novi list. Stvaramo novi list te u njega prebacujemo 40 i 41. Postavljamo posljednji pokazivač originalnom lista na novi, te posljednji pokazivač novoga postavljamo na idući list (onaj na kojeg je prije pokazivao originalni). Rezultat vidimo na slici 4.7 (stablo i dalje ima 3 razine, vidimo novonastali list koji još nema roditelja).



Slika 4.7: Prvi korak ubacivanja ključa 40

Sada nam je potreban pokazivač na novi list. Dižemo se jednu razinu iznad (na roditelja od originalnog lista). Uz taj novi pokazivač asociramo ključ 40. Ovaj čvor (drugi po redu na drugoj razini) je punog kapaciteta. Ubacivanjem 40 u njega on sadrži ključeve 23, 31, 40 i 41 (pokazivač na list dobijen u prethodnom koraku također smatramo ubačenim, te je asociran uz vrijednost 40). Kako je pređen kapacitet čvora dolazi do podjele.
Prva tri pokazivača i prva dva ključa ostaju u originalnom čvoru dok posljednja dva pokazivača i posljednji ključ prelaze u novi čvor. Napokon, ključ 40 asociran s pokazivačem na novi čvor ubacujemo u gornju razinu, odnosno u korijen koji ima mjesta za novi čvor. Na slici 4.8 vidimo rezultat ubacivanja novog ključa u stablo.

Još jedna važna operacija je brisanje. U ovom radu se nećemo baviti tom operacijom pa nećemo prezentirati algoritam.



Slika 4.8: Kraj ubacivanja ključa 40

Učinkovitost B-stabla

B-stabla omogućuju pretraživanja, ubacivanje i brisanje zapisa s malo operacija s diskom. U praksi se pokazalo da uz pretpostavku dovoljno velikog parametra n do podjela (i spajanja u slučaju brisanja) blokova dolazi rijetko. Najčešće se ta operacija reducira samo na listove i time njihove roditelje.
Nas najviše interesira pretraživanje pa nas samim time zanima i broj čitanja s diska. Promotrimo pojednostavljeni primjer gdje pretražujemo broj čitanja s diska u slučaju traženja točno jednog zapisa po ključu. Tada je broj čitanja s diska jednak broju razina u stablu. Pitanje je koliko možemo očekivati da će stablo imati razina? Pokazat ćemo jedan primjer gdje je broj razina u stablu jako malen, a broj podataka koje indeksira jako velik.

Promotrimo primjer u kojem je veličina bloka 4096, veličina ključa je 4 bajta, a pokazivača 8 bajtova. Vrijednost parametra n tada je 340.
Sada pretpostavimo da prosječni blok ima 255 pokazivača (otprilike pola između minimalnog i maksimalnog broja pokazivača koje može imati). Ako svaki roditelj ima 255 djece i od te djece svako pokazuje na 255 listova. Ako svaki list pokazuje na 255 zapisa tada je ovo stablo od samo 3 razine dovoljno za podatkovnu datoteku s $255^3 \approx 16.6 \cdot 10^6$ zapisa.

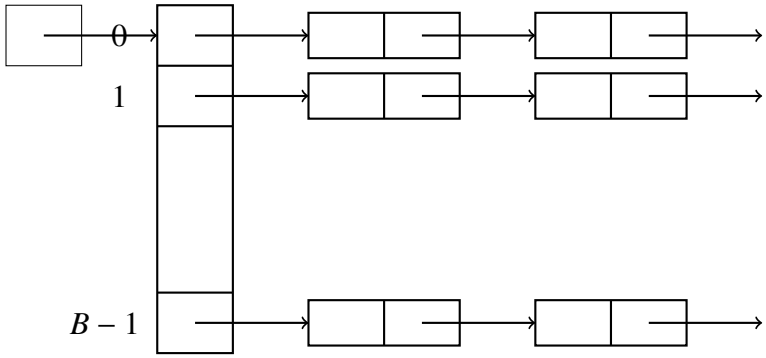
4.3 Hash tablice

U ovom poglavlju pričamo o korištenju *hash* tablica kao indeksa. U početku dajemo kratki podsjetnik što su to *hash* tablice te kako se implementiraju u glavnoj memoriji, zatim raspravljamo o općenitim *hash* tablicama s fiksnim brojem pretinaca i uvodimo *hash* tablice s dinamičkim brojem pretinaca.

Hash tablice u glavnoj memoriji

Hash tablice poznat su tehnika u programiranju. Poznatiji programski jezici imaju svoje implementacije *hash* tablica u svojem standardu. Na primjer u C++ imamo *unordered_map*,

u *Javi* imamo *HashTable*, u *pythonu dictionary* i tako dalje. Te *hash* tablice stoje u glavnoj memoriji i služe za brz dohvat vrijednosti po zadanom ključu. U ovakvim strukturama postoji *hash* funkcija h koja prima ključ K te računa vrijednost $h(K) \in \{0, \ldots, B-1\}$ gdje je B broj pretinaca u nekom polju. To polje pretinaca indeksirano je od 0 do $B-1$ te smatramo da vrijednost odgovarajućeg ključa po funkciji h predstavlja indeks pretinca u kojem se treba nalaziti ta vrijednost. Svaki element polja pretinaca je zaglavlje vezane liste odnosno pokazivač na njen prvi element. Ilustraciju *hash* tablice vidimo na slici 4.9. Za više o *hash* tablicama u glavnoj memoriji može se pronaći u [5].



Slika 4.9: Hash tablica

Hash tablice u sekundarnoj memoriji

Hash tablice koje pohranjuju jako veliki broj podataka koje je potrebno pohraniti u sekundarnoj memoriji nešto su drugačije od onih koje podatke drže u glavnoj memoriji.
Za početak, polje pretinaca sastavljeno je od blokova, ne od pokazivača na početak vezanih listi. Svaki zapis *hash*-iran po funkciji h u određeni pretinac dodan je u blok tog pretinca. Ako pretinac ima previše zapisa (zapise koji ne staju u njegove dosadašnje blokove) dodajemo još jedan blok u ovaj pretinac. Na ovaj način svaki pretinac sastavljen je od lanca blokova.

Uzmimo za primjer sljedeće implementacije *hash* tablice uz pretpostavku fiksnog broja pretinaca B. Na primjer možemo u glavnoj memoriji imati polje pokazivača duljine B. Svaki pokazivač pokazuje na prvi blok pretinca. Alternativno, možemo poslagati sve prve blokove pretinaca jedan do drugoga na disku. Na taj način možemo lako dohvatiti prvi blok i-tog pretinca.
Svaki blok pretinca može sadržavati dodatne informacije. Na primjer, može sadržavati pokazivač na idući blok u lancu.

Izbor hash funkcije

Hash funckija bi trebala dobro *raspršiti* vrijednosti dane uz zadani ključ. Rezultat funkcije trebao bi izgledati nasumično. Na taj način postižemo da svaki pretinac sadrži podjednak broj zapisa zbog čega ih možemo brže pronaći. Ove funkcije bi također trebale biti jednostavne i brze za izračun s obzirom da ćemo ih pozivati puno puta.

Ako *hash*-iramo po cijelim brojevima često za *hash* funkciju koristimo ostatak pri dijeljenju s B (brojem pretinaca). U slučaju da je ključ niz znakova možemo posumirati vrijednosti tih znakova (npr. ASCII) te pogledati ostatak dobivene sume pri dijeljenju s B. Česti izbor za B je neki prosti broj ili potencija broja 2.

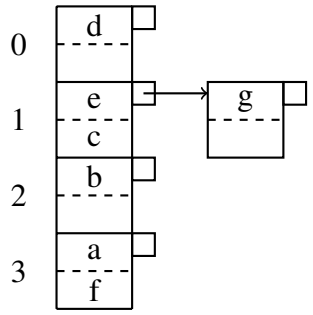
Ubacivanje u hash tablicu

Kada je potrebno ubaciti vrijednost s ključem K prvo računamo $h(K)$. Ako pretinac na mjestu $h(K)$ ima mjesta ubaujemo vrijednost u neki blok ovog pretinca. Ako nijedan blok u lancu nema mjesta tada dodajemo novi blok u lanac i pohranjujemo vrijednost tamo.

Uzmimo za primjer *hash* tablicu s 4 pretinca i blokovima koji mogu pohraniti dvije vrijednosti. Ako su ključevi iz skupa $\{a, b, c, d, e, f, g\}$ te sljedeća *hash* funkcija h:

$$\begin{array}{lll} h(a) = 3 & h(b) = 2 & h(c) = 1 \\ h(d) = 0 & h(e) = 1 & h(f) = 3 \\ & h(g) = 1 & \end{array}$$

Ako se u *hash* tablici već nalaze ključevi a, b, c, d, e, f i želimo dodati ključ g tada je potrebno stvoriti novi blok za pretinac pod rednim brojem 1 i tamo dodati ključ g. Vidimo ovo na slici 4.10.



Slika 4.10: Ubacivanje u drugi pretinac

Efikasnost hash tablica

Zanima nas koliko su *hash* tablice efikasne kada ih koristimo kao indeks.
U idealnom slučaju svaki pretinac sastavljen je od jednog bloka. U tom slučaju potrebno nam je samo jedno čitanje za dohvat podataka. Ovo je bolje od korištenja na primjer B-stabla, međutim, nisu podržana pretraživanja raspona.
Ipak, često se dogodi veći broj blokova u pretincu čime dolazi do povećanog broja čitanja s diska zato što trebamo pročitati svaki blok. Iz ovog razloga želimo da se u našim pretincima nalaze što kraći lanci blokova.

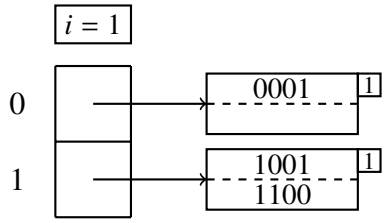
Do sada smo pričali o statičkim *hash* tablicama gdje se broj pretinaca nikad nije mijenjao. U idućim poglavljima raspravljamo o dinamičkim *hash* tablicama u kojima je dozvoljena promjena broja pretinaca. Promjena broja pretinaca nastoji omogućiti da svaki pretinac sadrži jedan blok.
Konkretno, raspravljat ćemo o proširivom i linearnom *hash*-iranju.

Proširivo hashiranje

Veći dodaci statičkim *hash* tablicama su sljedeći:

1. Polje pokazivača može rasti te mu je duljina uvijek potencija broja 2
2. Neki pretinci mogu dijeliti blok ako broj zapisa stane u blok
3. *hash* funkcija za svaki ključ računa niz od k bitova za neki k (npr. 32). Broj pretinaca će koristiti manji broj bitova, npr i bitova. Polje pretinaca imat će 2^i elemenata gdje je i broj korištenih bitova

Pojasnit ćemo ovo na primjeru. Na slici 4.11 vidimo primjer proširive hash tablice.



Slika 4.11: Proširiva hash tablica

Zbog jednostavnosti, pretpostavili smo da *hash* funkcija generira 4 bita. Trenutačno, tablica koristi samo jedan bit. Primjetimo da je tablica na slici dužna pamtiti koliko bitova koristi trenutačno.

Polje pretinaca drži pokazivače na dva bloka. Jedan od tih drži zapise za čiju evaluaciju ključa dobijamo niz bitova koji počinje s nulom, dok drugi drži podatke koji počinju s jedinicom.
U slučaju da je *hash* tablica koristila 2 bita tada bismo imali 4 pretinca i oni bi pokazivali na blokove u kojima bi bili ključevi koji bi počinjali sa, redom, $00, 01, 10$ i 11.

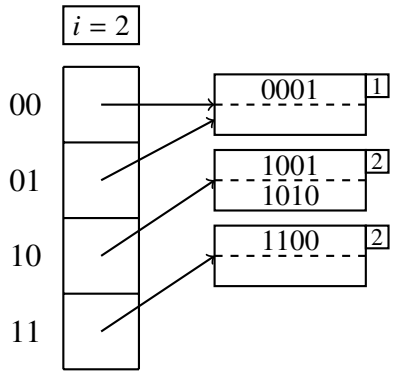
Primjetimo još brojku 1 u gornjem desnom kutu svakog bloka. Ona bi se inače nalazila u prije spomenutom zaglavlju bloka (prije smo tamo spremali pokazivač na idući blok u lancu kod statičkih tablica). Ova brojka ukazuje koliko bitova niza generiranog *hash* funkcijom se koristi za provjeru je li zapis s danim ključem član bloka. U danom primjeru taj broj je jednak za oba bloka, ali vidjet ćemo da to ne mora uvijek biti slučaj.

Ubacivanje u proširive hash tablice

Ubacivanje zapisa u proširivu *hash* tablicu kreće na isti način kao u statičkim *hash* tablicama. Prvo izračunamo vrijednost *hash* funkcije. Zatim uzmemo prvih i bitova generiranog niza i pogledamo pretinac koji se nalazi u polju pretinaca na tako definiranoj lokaciji. Pratimo pokazivač do bloka B. Ako ima mjestu u bloku tada ubacujemo novu vrijednost. Međutim, ako nema mjesta, postoje dvije mogućnosti ovisno o broju j u zaglavlju bloka (podsjetimo, radi se o broju bitova korištenim za provjeru je li zapis član zadanog bloka).

1. Ako je $j < i$
   a) Podijelimo blok B na dva dijela
   b) Podijelimo zapise iz bloka B na temelju $(j + 1)$-og bita - zapisi s nulom na tom mjestu ostaju u B, a oni s jedinicom idu u drugi blok
   c) U zaglavlju oba bloka zapišemo $j + 1$ (tamo gdje je prije pisalo j)
   d) Prilagodimo pokazivače u polju pretinaca tako da elementi koji su pokazivali na B sada pokazuju na B ili na novi blok ovisno o $(j + 1)$-om bitu njihovog indeksa u polju
   e) Ako postoji blok u koji možemo ubaciti zapis ubacujemo ga, inače ponavljamo postupak (npr. moglo se dogoditi da su svi podaci u bloku B imali istu vrijednost na $(j + 1)$-om bitu te da je novi zapis trebao završiti u tom bloku)
2. Ako je $j = i$ inkrementiramo i. Uduplamo veličinu polja pretinaca koje sada ima 2^{i+1} elemenata. Neka je w niz od i bitova koji je indeksirao neki pretinac u prethodnom polju pretinaca. U novom polju pretinaca sada imamo indekse $w0$ i $w1$, svaki od njih sada pokazuje na isto mjesto gdje je pokazivao pretinac s indeksom w. Oni dijele blok koji se sam po sebi ne mijenja. Sada radimo korak 1 zato što je $j < i$.

Pretpostavimo da u tablicu sa slike 4.11 ubacujemo zapis za čiji ključ je vrijednost *hash* funkcije jednaka 1010. Kako je prvi bit jednak 1 zapis pripada drugom bloku. No, taj blok je pun i treba doći do podjele bloka. Vidimo da je $j = i = 1$ pa za početak udvostručujemo veličinu polja pretinaca. Također postavljamo $i = 2$.
Drugi blok, čiji zapisi počinju s 1 potrebno je podijeliti kako bismo napravili mjesta za novi zapis. Uvećavamo vrijednost od j za 1. Sada particioniramo blok na dva bloka od kojih jedan počinje s 10, a drugi s 11. Sada možemo ubaciti naš ključ u drugi blok zato što smo prebacili ključ 1100 u drugi blok. Rezultat vidimo na slici 4.12.



Slika 4.12: Ubacivanje ključa 1010

Primjetimo kako dva pretinca čiji indeksi počinju s 0 pokazuju u isti blok za koji vrijedi da njegovi *hash*-irani ključevi počinju s nulom te mu u zaglavlju stoji 1 što indicira da za prepoznavanje pripadnosti zapisa u bloku koristimo samo 1 bit.

Linearno hashiranje

Najveća prednost proširivih *hash* tablica je to što za pronalazak jednog zapisa trebamo točno jedan pristup disku. Naravno, ovo vrijedi u slučaju da je polje pretinaca dovoljno maleno da se može nalaziti u glavnoj memoriji, ali to je razumna pretpostavka. Unatoč prednostima koje dobijamo postoje neke mane:

1. Kad dolazi do povećanja polja pretinaca potrebna je značajna količina posla (za velike i). Za vrijeme tog posla prekinut je pristup podatkovnoj datoteci i ubacivanja novih zapisa traju duže

2. Zbog eksponencijalnog rasta duljine polja pretinaca ona možda više neće stati u glavnu memoriju. Dolazi do povećanja broja pristupa disku jer polje pretinaca moramo držati u sekundarnoj memoriji.

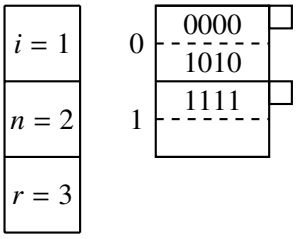
Glavni problem ovog *hash*-iranja je očito u prebrzom rastu broja pretinaca. Druga strategija, koja rješava taj problem, zove se linearno *hash*-iranje. U ovoj strategiji vrijede sljedeća pravila:

- Broj pretinaca n odabran je tako da je prosječan broj zapisa po pretincu uvijek fiksan postotak kapaciteta bloka tog pretinca
- Kako neće dolaziti do podjela blokova dozvoljeno je ulančavanje blokova. Međutim, prosječan broj dodatnih blokova po pretincu mora biti značajno manji od 1
- Broj bitova korišten za numeriranje elemenata polja pretinaca je $\lceil \log_2 n \rceil$. Iz vrijednosti koju dobijemo računanjem *hash* funkcije bitove uzimamo zdesna na lijevo (npr. ako je vrijednost funkcije 1011 te je $n = 4$ trebali bismo uzeti posljednja dva bita odnosno 11)
- Neka je i bitova *hash* funckije korišteno za numeriranje polja pretinaca i da je zapis s ključem K namjenjen pretincu $a_1 a_2 \ldots a_i$ odnosno, $a_1 a_2 \ldots a_i$ su posljednjih i bitova od $h(K)$. Neka je $a_1 a_2 \ldots a_i$ jednako nekom cijelom broju m. Ako je $m < n$ tada pretinac s brojem m postoji i zapise spremamo/pronalazimo u tom pretincu. Inače, ako je $n \leq m < 2^i$ tada pretinac m ne postoji i zapis stavljamo u pretinac $m - 2^{i-1}$ odnosno u pretinac označen nizom bitova $a_1 a_2 \ldots a_i$ s tim da promjenim a_1 iz jedinice u nulu.

Uzmimo za primjer linearno *hash*-iranu tablicu gdje je $n = 2$. Za određivanje pretinca danog zapisa koristimo samo $\lceil \log_2(2) \rceil = 1$ bitova vrijednosti *hash* funkcije. Pretpostavimo da *hash* funkcija generira 4 bita (ta 4 bita ćemo koristiti kao vrijednosti koje spremamo u tablicu zbog lakše ilustracije).
Dio strukture koja implementira linearno *hash*-iranje bit će parametar i (broj bitova *hash* funkcije trenutačno u upotrebi), n (broj pretinaca) i r (trenutačan broj vrijednosti u tablici. Omjer r/n želimo ograničiti tako da za tipični pretinac trebamo jedan blok memorije. Vrijednost n ćemo odabirati tako da $r \leq 1.7n$.
Na slici 4.13 vidimo linearnu *hash* tablicu na s dva pretinca. Primjetimo kako zadnji bit svake vrijednosti određuje pripadnost pretincu.

Ubacivanje u linearne hash tablice

Za dodavanje novog zapisa u tablicu računamo $h(K)$ gdje je K ključ zapisa koji ubacujemo kao do sada. Koristimo i bitova s kraja niza bitova dobijenog iz $h(K)$ kao broj m (ako je $i = 2$ te je hash funkcija generirala 1011 vrijedilo bi $m = 3$). Kao što je opisano u prethodnom odjeljku ako $m < n$ ubacujemo zapis u pretinac m, a ako je $m \geq n$ ubacujemo



Slika 4.13: Linearna *hash* tablica

zapis u pretinac $m - 2^{i-1}$. Ako nema mjesta u pretincu dodajemo novi blok u lanac i tamo ubacujemo zapis.

Svaki put kada ubacimo zapis provjeravamo gornju granicu na omjer r/n. Ako je omjer previsok dodajemo još jedan pretinac u tablicu. Ako je indeks pretinca kojeg smo dodali jednak $1a_2 \dots a_i$ tada iz pretinca indeksiranog s $0a_2 \dots a_i$ prebacujemo vrijednosti u novi pretinac ovisno o posljednjih i bitova njihovog ključa (zapravo prebacujemo vrijednosti koje su završile u ovom pretincu kada smo se nalazili u situaciji $n \leq m < 2^i$). Ako ovim prebacivanjem ispraznimo neki blok možemo ga obrisati.

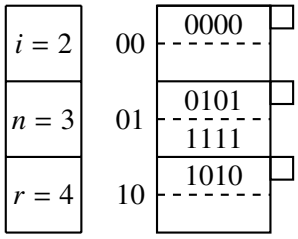
Ako broj pretinaca n pređe 2^i tada i uvećavamo za 1. Tehnički, svi indeksi pretinaca u polju pretinaca dobili su nulu s lijeve strane, ali vodeće nule ionako ne znače ništa pošto tumačimo indekse kao cijele brojeve.

Vratimo se sada na sliku 4.13 i ubacimo u tu tablicu zapis za čiji ključ je vrijednost *hash* funkcije jednaka 0101. Kako je posljednji bit jednak 1 on očito mora završiti u drugom pretincu. Tamo imamo mjesta i nije potreban novi blok pa ga samo ubacujemo u postojeći.

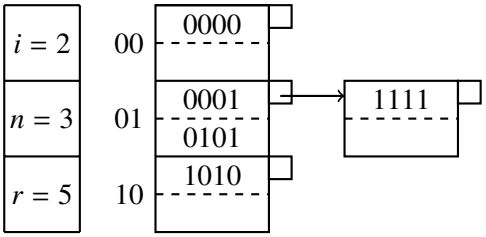
Sada imamo 4 zapisa u 2 pretinca. Vrijedi $r = 4 > 3.4 = 1.7n$ pa moramo dodati novi pretinac. Vrijednost od n postaje 3, a i inkrementiramo (ili računamo iz $i = \lceil \log_2(3) \rceil = 2$. Pretince 0 i 1 sada promatramo kao pretince 00 i 01 (naravno, ne radi nikakvu razliku, jer ih ionako tumačimo kao cijele brojeve). Idući pretinac indeksiran je brojem 10.
Nalazimo se u slučaju gdje je indeks novog pretinca oblika $1a_2 \dots a_i$ pa moramo prebaciti zapise iz pretinca indeksiranog s 00. Zapisi koji završavaju s 00 ostaju u tom pretincu, dok one koji završavaju s 10 prebacujemo u novi. Preciznije u pretincu 00 ostaje vrijednost 0000, a u novi ubacujemo 1010. Rezultat vidimo na slici 4.14.

Ako pokušamo ubaciti zapis za koji je vrijednost *hash* funkcije jednaka 0001 tada moramo pohraniti zapis u drugi pretinac (onaj indeksiran s 01). Pretinac je pun i potrebno je dodati novi blok. Nakon dodavanja novog bloka i ulančavanja zapis upisujemo u novi blok. Sada provjerimo $r = 5 < 5.1 = 1.7n$ što znači da ne trebamo dodavati novi pretinac. Novo stanje vidimo na slici 4.15.

Finalno, razmotrimo još dodavanje zapisa za koji je vrijednost *hash* funkcije jednaka 0111, primjetit ćemo da on pripada pretincu s rednim brojem 11. Međutim on ne postoji

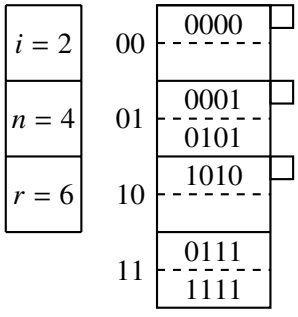


Slika 4.14: Dodavanje novog pretinca



Slika 4.15: Dodavanje novog bloka

pa ga ubacujemo u pretinac s indeksom 01. U njegovom bloku imamo mjesta pa ne treba dodavati novi blok.
U ovom slučaju imamo $r = 6 > 5.1 = 1.7n$ pa je potrebno dodati novi pretinac. Njegov indeks je 11. Indeks je formata $1a_2 \ldots a_n$ pa dijelimo zapise iz pretinca 01 između ova dva pretinca tako da 0111 i 1111 ubacimo u novi pretinac, a ostale ostavimo u starom. Također, sada pretinac 01 ima samo 2 zapisa pa možemo obrisati jedan blok. Vrijednost od i se ne mijenja. Rezultat vidimo na slici 4.16.



Slika 4.16: Dodavanje novog pretinca bez promjene parametra i

Poglavlje 5

Usporedba indeksa

U ovom poglavlju opisujemo način implementacije indeksa iz prethodnog poglavlja, prezentiramo testove i njihove rezultate. Testove izvršavamo na istim podacima koje smo koristili u poglavlju 3.

5.1 O implementaciji

Kod implementacije indeksa osigurano je da ključ pretraživanja bude 4-bajtni cijeli broj s predznakom. Indeks se izrađuje dohvatom svih podataka iz tablice te ubacivanjem jednog po jednog primarnog ključa u strukturu. Kada je indeks izgrađen možemo ga koristiti za izvršavanje upita. Indeksi za tražene vrijednosti vraćaju *offset* zapisa u podatkovnoj datoteci ako zapis s tom vrijednosti postoji, inače vraća -1. Oba indeksa pretpostavljaju jedinstvenost ključa.
Svaki blok indeksa nalazi se u zasebnoj datoteci. Ovo nije potrebno, ali jednostavnije je za pronalazak pogrešaka u razvoju.
U daljnjem dijelu opisujemo implementaciju struktura te računamo relevantne parametre za dane strukture.

B-stablo

Izgradnjom B-stabla nastaje direktorij *btree* koji sadrži točno jednu *.meta* binarnu datoteku i po jednu binarnu datoteku za svaki blok koji predstavlja jedan čvor u stablu. Svaki čvor ima jedinstveni identifikacijski broj.
Metapodatci koji se nalaze u *.meta* datoteci su identifikacijski broj korijena i najveći iskorišteni identifikacijski broj za imenovanje čvora. Prvi podatak koristimo za učitavanje stabla na početku rada, a drugi koristimo kako bismo znali pridružiti identifikacijski broj pri stvaranju novog čvora.

Datoteka s podacima o jednom čvoru imena je formata <*id_čvora*>.*btree*. Svaki blok ima zaglavlje od dva 4-bajtna cijela broja bez predznaka. Prvi označava broj ključeva u čvoru (označimo ga s c), a drugi označava je li čvor list. Idućih $4c$ bajtova predstavlja c 4-bajtnih cijelih brojeva bez predznaka koji predstavljaju vrijednosti ključa koji se nalazi u tom čvoru. Sljedećih $4(c + 1)$ bajtova predstavlja $c + 1$ 4-bajtnih cijelih brojeva koje interpretiramo kao pokazivače.
Ako se radi o čvoru koji nije list tada je svih $c + 1$ pokazivača identifikator čvora koji je njegovo dijete. U slučaju da se radi o listu tada je prvih c vrijednosti *offset* zapisa s odgovarajućim ključem dok je zadnji jednak identifikatoru idućeg lista u stablu. Ako ne postoji idući list u stablu tada će posljednja vrijednost biti jednaka -1.
Blokovi u ovoj implementaciji su veličine 512 bajtova. Ako uračunamo da zaglavlje zauzima 8 bajtova te da je veličina ključa 4 bajta i veličina pokazivača 4 bajta tada je vrijednost parametra n (odnosno maksimalan broj ključeva u jednom bloku) dobijen iz nejednadžbe:

$$\begin{aligned} 4n + 4(n + 1) &\leq 504 \\ 8n &\leq 500 \end{aligned}$$

Vrijednost paramtra n je jednaka 62.

Hash indeks

S obzirom da radimo s fiksnim brojem podataka u implementaciji se ne bavimo ni proširivim ni linearnim *hash*-iranjem. Fiksiramo broj pretinaca ovisno o broju stupaca koje imamo u tablici pri kreiranju *hash* tablice.
Pri kreiranju *hash* indeksa stvaramo direktorij *hash_ind* u kojem se nalaze *.meta* binarna datoteka i binarne datoteke koje predstavljaju blokove *hash* tablice. Svaki blok *hash* tablice ima jedinstveni identifikator.
U metapodacima stoji maksimalni iskorišteni identifikator bloka. Ovo koristimo u slučaju da dođe do prelijevanja u nekom bloku tablice te je potrebno ulančavati nove blokove.
Svaki blok počinje sa zaglavljem od dvije 4-bajtne cjelobrojne vrijednosti s predznakom. Prva predstavlja broj ključeva u bloku (označimo ga s c), a druga predstavlja identifikator idućeg bloka u lancu (-1 ako takav ne postoji).
Slijedi c 4-bajtnih cijelih brojeva koji predstavljaju vrijednosti ključeva te još c 4-bajtnih cijelih brojeva koji predstavljaju *offset*-e zapisa s danim ključem (i-ti ključ uparen je s i-tim *offset*-om).
Broj pretinaca odabiremo tako da u svakom bloku ima što manje prelijevanja pod pretpostavkom uniformno distribuiranih vrijednosti ključeva. Prvo računamo koliko vrijednosti

stane u blok (zaglavlje je veličine 8 bajtova te su i ključ i *offset* veličine 4 bajta):

$$4n + 4n + 8 \leq 512$$
$$8n \leq 504$$

U svaki blok stane 63 različita ključa s pripadnim *offset*-om.
Sada nas zanima koliko pretinaca nam je potrebno da bismo imali što manje prelijevanja. Ako želimo raspodjeliti milijun zapisa po pretincima tako da bude što manje prelijevanja potrebno nam je $\lceil \frac{1\ 000\ 000}{63} \rceil = 15874$ blokova.

Izvršavanje upita

Pri izvršavanju upita (neovisno o tipu indeksa) slijedimo iduće korake:

1. Za zadani ključ (ili ključeve) preko indeksa pronađemo *offset*-e zapisa
2. Sortiramo *offset*-e i pronađemo blokove kojima oni pripadaju
3. Učitamo blok po blok te iz njega izvučemo zapise zadane *offset*-om

5.2 Testovi

Kod indeksa ćemo testirati dohvat podataka u sljedećim slučajevima:

1. dohvat točno jednog zapisa po indeksiranom stupcu
2. dohvat svih zapisa po indeksiranom stupcu gdje je vrijednost indeksiranog stupca u zadanom rasponu

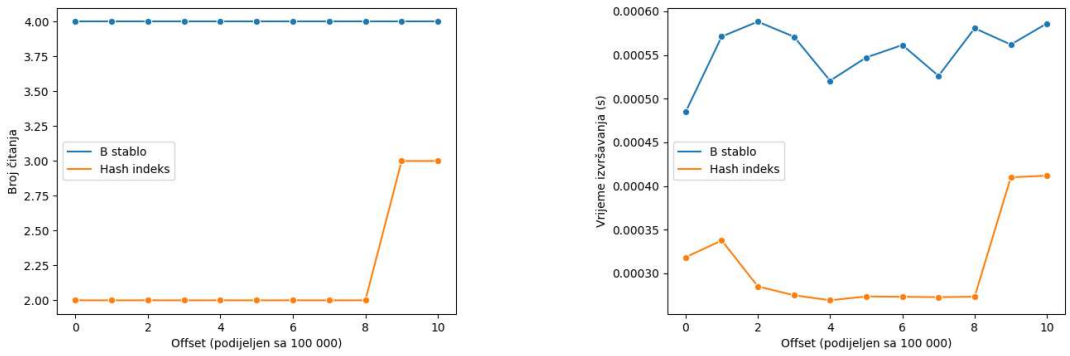
Prvi upit možemo izvršavati na oba indeksa dok upiti po zadanom rasponu za *hash* tablice nemaju smisla. Kada bismo radili takve upite nad *hash* indeksima trebali bismo za svaku vrijednost u rasponu računati vrijednost *hash* funkcije te čitati indeks što bi bilo jako neefikasno (na manjim cjelobrojnim rasponima bi bilo u redu, ali za na primjer decimalne bi bilo besmisleno).

5.3 Rezultati

U ovom odjeljku mjerimo rezultate prethodno navedenih upita. Prvo ćemo usporediti pretraživanje po ključu implementiranih indeksa, a zatim ćemo vidjeti efikasnost pretraživanja raspona kod B-stabla.
Napominjemo da u ovom odjeljku nećemo pisati ekvivalente SQL upite s obzirom da su dovoljno slični onima iz poglavlja 3.

Pretraživanje po ključu

Na slici 5.1 vidimo rezultate pretraživanja uz korištenje indeksa. Ključevi po kojima pretražujemo odabrani su na isti način kao u odjeljku 3.4. Vidimo da B-stablo ima konstantan broj čitanja koji odgovara njegovoj visini zbrojenoj s još jednim čitanjem s diska (3 + 1). Treba uzeti u obzir da je korijen stabla učitan u memoriju na početku izvršavanja i njegovo čitanje ne brojimo. Za *hash* indeks imamo gotovo konstantno vrijeme izvršavanja. Vidimo da većinom jedan put pročitamo indeks i zatim direktno čitamo s diska, ipak, u zadnja dva primjera ipak je došlo do ulančavanja te dva čitanja s diska pri korištenju indeksa. Ako usporedimo rezultate sa slikom 3.5 vidimo da oba indeksa donose veliko poboljšanje pri pretraživanju po ključu. Sekvencionalno pretraživanje koje je bilo potrebno koristiti i čija složenost je linearna u broju zapisa svedeno je na konstantnu. U konkretnom primjeru vrijeme izvršavanja upita s B-stablom kao indeksom bilo je 0.01 posto vremena izvršavanja upita bez indeksa, dok je vrijeme izvršavanja upita s *hash* indeksom bilo jednako 0.008 posto vremena izvršavanja upita bez indeksa.

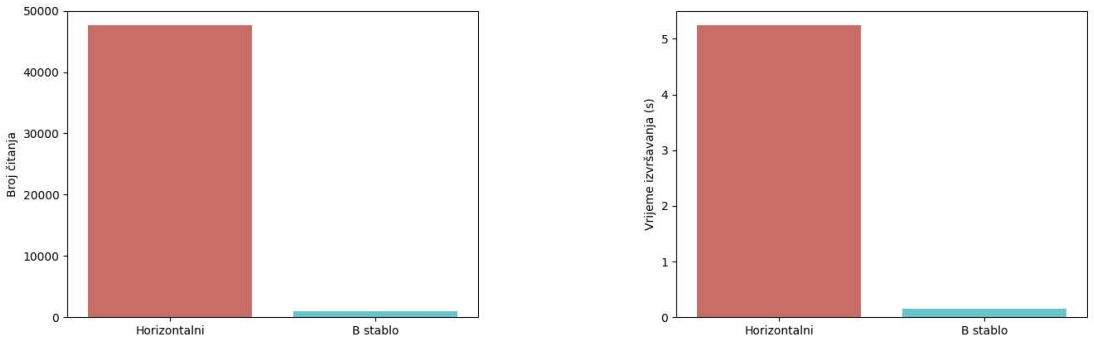


Slika 5.1: Pretraživanje korištenjem indeksa

Pretraživanje raspona

Na slici 5.2 vidimo rezultate pretraživanja raspona s i bez korištenja B-stabla kao indeksa. Zadan je raspon u kojem se primarni ključ mora nalaziti. Vrijednosti moraju biti veće od 8623 i manje od 11546. Na slici se jasno vidi da pretraživanje korištenjem B-stabla donosi značajnog ubrzanja i smanjenja broja čitanja. Ulančani listovi stabla nam omogućuju brz pronalazak svih vrijednosti u zadanom intervalu. Ipak, potrebno je biti pažljiv pri korištenju indeksa za ovakve upite. Ako je interval preširok može obuhvatiti jako puno zapisa. S obzirom da su zapisi nasumično razbacani po blokovima podatkovne datoteke

postoji mogućnost da unatoč korištenju indeksa moramo pročitati cijelu podatkovnu datoteku ili veliki dio nje.



Slika 5.2: Pretraživanje raspona

5.4 Zaključak

Zaključno, indeksi donose veliko ubrzanje i nameće se pitanje zašto ih ne bismo uvijek koristili. Unatoč njihovoj praktičnosti i ubrzanju upita indeksi su ipak zasebna struktura o kojoj se trebamo brinuti i koju je potrebno održavati skladno s promjenama u tablici. Svako brisanje i dodavanje u tablicu potrebno je ispratiti odgovarajućom promjenom u indeksu. Kada bismo imali indeks po svakom stupcu te operacije bi trajale nezanemarivo duže. Zato je potrebno pažljivo birati slučajeve kada koristiti indeks kako bismo dobili što efikasnije rezultate u produkcijskim bazama podataka.

Zaključak

Tehnologija se jako brzo razvija. Programeri su na svakom uglu suočeni s izborom kako najbolje riješiti neki problem iskorištavajući postojeće rješenje. Već sam izbor programskog jezika rijetko je trivijalan, a postoje i raznovrsni programski okviri (eng. *framework*) između kojih se teško odlučiti. Odabir pravog SUBP-a spada u jedne od težih izbora s kojim se moderni softverski inžinjeri susreću.

Svaki SUBP dolazi s prednostima i manama u određenim slučajevima upotrebe. Uz njegov izbor važno je uspješno administrirati i iskoristiti njegove prednosti na najbolji mogući način. U ovom radu analizirali smo dva razmještaja podataka i dva tipa indeksa koje moderni SUBP-ovi koriste.

Vidjeli smo da odabir razmještaja podataka ovisi o slučaju upotrebe. Ukoliko od SUBP zahtjevamo brže upite i efikasnije akumulacije podataka više će nam odgovarati vertikalni razmještaj podataka. U slučaju da će aplikacija koju gradimo intenzivno modificirati sadržaj tablica bolji izbor bio bi SUBP koji koristi horizontalni razmještaj podataka.

Upoznali smo važan alat u uspješnom korištenju SUBP-a — indekse. Obradili smo B-stablo i *hash* indeks te ih testirali. Zaključili smo da oba donose veliko ubrzanje upita, ali usporavaju dodavanje i brisanje iz tablice jer ih je potrebno držati ažuriranima. Zato je važno znati kakve upite najčešće možemo očekivati na tablicu i njene atribute te na temelju toga prosuditi nad kojim stupcima je najbolje izgraditi indeks. Ako očekujemo puno upita gdje zadajemo identifikator zapisa kojeg želimo dohvatiti vjerojatno trebamo *hash* indeks. Ukoliko upiti mogu biti nešto raznovrsniji vjerojatno želimo B-stablo koje podržava i upite po rasponu, ali i jako je efikasno pri dohvatu jedne vrijednosti.

Zaključno, u ovom radu predstavljene su osnove SUBP-ova uz konkretne testove koji pokazuju razlike između gradivnih blokova tog kompleksnog sustava. Razumijevanje tih razlika može pomoći programeru pri odabiru alata koji će mu najbolje koristiti za idući projekt.

Bibliografija

[1] Barnhill, Blake i Matt David: *Row vs Column Oriented Databases*. https://dataschool.com/data-modeling-101/row-vs-column-oriented-databases/.

[2] Basioli, Karlo: *Programski kod*. https://github.com/basioli-k/db-indexes.

[3] Garcia-Molina, Hector, Jeffrey D. Ullman i Jennifer Widom: *Database Systems, The Complete Book*. Pearson Education, 2009.

[4] Hugue., Dr. Michelle: *Measuring and Reporting Performance of Benchmarks*. https://www.cs.umd.edu/users/meesh/cmsc411/website/projects/morebenchmarks/measure.html.

[5] Manger, Robert: *Strukture podataka i algoritmi*. Element, 2015.

[6] Microsoft: *File API dokumentacija*. https://docs.microsoft.com/en-us/windows/win32/api/fileapi.

[7] O'Hallaron, David R. i Bryant Randal E.: *Computer Systems, A Programmer's Perspective*. Pearson Education, 2016.

Sažetak

Ukratko, u ovom radu objašnjeni načini smještaja podataka i tehnike indeksiranja.

U prva dva poglavlja dan je pregled fizičkog i logičkog smještaja podataka na disk. Uz memorijsku hijerarhiju računala obrazloženi su horizontalni i vertikalni način smještaj te njihove razlike.

U trećem poglavlju opisana je implementacija prethodno spomenutih načina pohrane. Također, objašnjen je način na koji su generirani testni podaci te su napokon napravljeni testovi na kojima se konkretnije vide razlike između dva načina smještaja.

U četvrtom poglavlju dan je opći pregled indeksa. Zatim su predstavljeni B-stablo i *hash* indeks, način dodavanja vrijednosti u te strukture, način pretraživanja i upotrebe tih struktura kao indeksa.

U petom poglavlju, slično kao u trećem, opisana je implementacija prethodno objašnjene dvije strukture te su napravljeni testovi i mjerenja koji pokazuju prednosti indeksiranja.

Summary

In short, this paper explores the types of data arrangement and techniques of indexing.

The first two chapters overview the physical and logical arrangement of data on a disk. We study the memory hierarchy and explain the horizontal and vertical data arrangements and their differences.

The third chapter describes the implementation of the previously mentioned types of data arrangement. It explains the data generation used for the tests and finally shows the test results which show the difference between the two arrangements.

The fourth chapter previews general information about indexes. Then we talk about the B-tree and *hash* index, insertion into said structures, value lookup, and the use of those structures as indexes.

The fifth chapter similar to the third describes the implementation of the previously mentioned structures. Tests that show the advantage of indexing are presented.

Životopis

Rođen sam 10.svibnja 1998. u Zadru gdje sam i živio cijelo djetinjstvo. Pohađao sam osnovnu školu Šimun Kožičić Benje te sam upisao prirodoslovno-matematičku gimnaziju Franje Petrića MIOC. Tamo sam razvio interes i ljubav prema matematici i programiranju.

Godine 2017. upisao sam Preddiplomski sveučilišni studij matematike na Matematičkom odsjeku Prirodoslovno-matematičkog fakulteta Sveučilišta u Zagrebu. Godine 2020 završio sam preddiplomski i postao sveučilišni prvostupnik matematike. Iste godine sam upisao Diplomski sveučilišni studij računarstva i matematike na istom fakultetu.