# Predavanje VII.

*Performanse baze podataka i indeksi*

BAZE PODATAKA II

doc. dr. sc. Goran Oreški
*Fakultet informatike,*
*Sveučilište Jurja Dobrile, Pula*

# Sadržaj

- ponavljanje prethodnih predavanja
  - mediji za pohranu
  - RAID
  - organizacija podataka
  - organizacija datoteka
- performanse baze podataka
- pohrana tabličnih podataka
- indeksi
- implementacija indeksa
- $B^+$-Tree indeksi
- indeksi i upiti
- `EXPLAIN` naredba
- provjera korištenja indeksa

# Ponavljanje

- mediji se klasificiraju prema:
  - brzini kojom se može pristupiti podacima
  - cijenom medija po jedinici podatka
  - i pouzdanosti
- tipično su dostupni slijedeći mediji:
  - cache, glavna memorija, flash memorija, magnetni diskovi, optički mediji, trake za pohranu
- tipovi pohrane:
  - primarna
  - sekundarna
  - ternarna pohrana

# Ponavljanje

- RAID - tehnike organizacije diskova
- korištenje više diskova
  - zrcaljenje pruža veliku pouzdanost, ali je skupo
  - dijeljenje na razini bita ili bloka pruža brzi transfer podataka, ali ne poboljšava pouzdanost
- RAID razine:
  - RAID 0
  - RAID 1
  - RAID 5
  - RAID 6

# Ponavljanje

- trajni podaci su pohranjeni na medijima stalne pohrane
  - magnetni disk ili SSD koriste blokove kao jedinice pohrane, čitaju i pišu u blokovima
- blok može sadržavati nekoliko zapisa
- n-troke različitih relacija su u pravilu različite veličine
  - pohraniti zapise jednake veličine u jednu datoteku (jednostavnije rješenje)
  - ili omogućiti da datoteke pohranjuju zapise varijabilne veličine
- reprezentacija zapisa u strukturi datoteke
- organizacija zapisa u datoteci

# Performanse baze podataka

- u mnogo situacija potrebno je poboljšati performanse upita
  - kako veličina podataka raste, performanse upita se pogoršavaju te je potrebno napraviti ugađanje
  - ekstremni slučajevi: npr. skladišta podataka koja mogu sadržavati milijune ili milijarde zapisa za agregaciju i sumiranje
- za učinkovitu optimizaciju upita potrebno je razumjeti što baza podataka čini u pozadini
  - npr. *zašto su korelirani podupiti spori u izvršavanju*?
  - zato što se unutarnji upit mora izvršiti za svaki red koji je dohvaćen u vanjskom upitu

# Performanse baze podataka

- u nastavku predavanja ćemo se baviti proučavanjem kako većina baza podataka vrednuje upit
  - specifično, kako su implementirane operacije relacijske algebre i koje se optimizacijske tehnike koriste
  - kao i uvjek, i u ovom slučaju postoje iznimke
  - vrlo bitna tema za razumijevanje ograničenja pojedinog DBMS-a
- na ovom predavanju ćemo se više usredotočiti na pohranu podataka i metodologiju pristupa
- na slijedećem predavanju ćemo nastaviti s istraživanjem načina implementacije operacija relacijske algebre

# Pristup disku

- prvo pravilo performansa baze podataka:

  **Pristup disku je najskuplja stvar koju rade baze podataka!**

- pristupanje zapisima u memoriji se može izvesti od 10-100ns
- pristupanje zapisima na disku može trajati i 10-tke ms
  - to je otprilike veličina koja je otprilike 100000 puta veća (tj. vremenski duža)
  - čak i SSD diskovi trebaju desetke ili stotine µs (otprilike 1000x sporije)
- na žalost, disk IO nije moguće izbjeći
  - u pravilu količina podataka jednostavno ne može stati u memoriju
  - podaci moraju biti perzistentni i kada se baza podataka ugasi namjerno ili slučajno
- baze podataka nastoje minimizirati disk IO operacije

# Planiranje i optimizacija

- kada planer/optimizator upita zaprimi neki upit:
  - istraži mnogo ekvivalentnih planova u kojima procjenjuje njihov trošak (primarno IO trošak) i odabire onog s najmanjim troškom
  - prilikom evaluacije upita promatra mnogo opcija
    - koji su pristupni putevi do podataka koji se trebaju koristiti?
    - koje algoritme može koristiti za select, join, sortiranje i sl.?
    - koja je priroda samih podataka?
      - statistički podaci koje generira baza podataka na temelju unesenih podataka
- planer će učiniti najviše što može
  - ponekad ne može naći brzi način da izvede neki upit
  - zavisi o sofisticiranosti samog planera, implementaciji naprednih algoritama…

# Pohrana tabličnih podataka

- baze podataka u pravilu pohranjuju svaku tablicu u zasebnu datoteku
- datotečni IO se odvija u blokovima fiksne veličine zvanim *stranicama*
  - uobičajena veličina stranice je 4KB do 8KB, može se podesiti
  - disk može čitati/pohranjivati stranice mnogo brže u odnosu na manje količine bajtova ili pojedinačnih zapisa
  - također korištenjem stranica se olakšava bazi podataka upravljanje podacima unutar memorije
    - za taj zahtjevan zadatak se koristi buffer manager
- svaki blok unutar datoteke se sastoji od određenog broja zapisa
- često se pojedini zapisi mogu razlikovati po veličini

# Pohrana tabličnih podataka

- pojedini blokovi imaju unutarnju strukturu da bi upravljali:
  - zapisima koji mogu varirati po veličini
  - zapisima koji su obrisani
  - gdje i kako dodati novi zapis u blok, ukoliko ima mjesta za njega
- tablična datoteka sama ima svoju strukturu
  - da bi se uobičajene operacije što brže izvršavale
  - *Želim umetnuti novi red u bazu podataka. Koji blok ima prostor za umetanje ili je potrebno rezervirati novi blok na kraju datoteke?*

# Organizacija datoteke

- da li bi zapisi tablice u datoteci trebali biti organizirani na neki način?
- primjer: zapisi se pohranjuju u sortiranom redoslijedu, koristeći ključ
  - to se naziva sekvencijalna organizacija datoteke (*engl . sequential file organization*)
  - puno je jednostavnije i brže pronaći zapise na temelju ključa
  - puno je brže za korištenje upita koji koriste raspon
  - sigurno komplicira pohranu zapisa
    - nemoguće je predvidjeti redoslijed u kojem će zapisi biti dodani ili obrisani
    - često je potrebno provoditi reorganizaciju zapisa da bi se osiguralo da su zapisi pohranjeni u sortiranom redoslijedu
- zapisi bi se mogli i hash-irati po nekom ključu

# Organizacija datoteke

- takva organizacija se naziva hashing organizacija (*engl. hashing file organization*)
  - također ubrzava pristup preko specifičnih vrijednosti
  - s vremenom se pojavljuju slični izazovi organizacije
- najčešća organizacija je slučajnim odabirom (*engl. heap file organization*)
  - zapis može biti smješten bilo gdje u datoteci tablice, gdjegod ima prostora za njegovu pohranu
  - doslovno sve baze podataka pružaju heap organizaciju datoteke
  - u pravilu je savršeno dovoljno, izuzev za najzahtjevnije aplikacije

# Heap datoteka i upiti

- obzirom da baze podataka u pravilu koriste heap organizaciju datoteke, kako vrednuju slijedeći upit:

```
SELECT * FROM racun
  WHERE racun_id = '591';
```

- jednostavan pristup
  - pretraga kroz cijelu datoteku tablice, traže se zapisi koji imaju vrijednost 591 za atribut *racun_id*
  - takav pristup se naziva scan datoteke (*engl. file scan*)
- može biti spor, međutim za sada nam je jedina opcija…
- potreban je način za optimizaciju pristupa

# Indeksi tablice

- većina upita koristi mali broj redova tablice
  - potreban je način da brže dođemo do tih redova, u odnosu na pretragu cijele tablice
- rješenje: stvoriti indeks na tablici
  - svaki indeks je povezan s određenim stupcem ili skupom stupaca, koji se nazivaju ključem pretrage (*engl. search key*) za indeks
  - upiti koji koriste te stupce mogu biti puno brži ukoliko koriste indeks na tim stupcima
  - upiti koji ne koriste te stupce će i dalje koristiti scan datoteke
- indeks je uvjek strukturiran na neki način, za brži dohvat podataka
- indeks je puno manji od tablice
  - puno je brže izvoditi pretragu unutar indeksa

# Karakteristike indeksa

- postoji više varijanti indeksa ovisno o karakteristikama pristupa
  - koji način pretrage je najučinkovitiji za određeni tip indeksa?
  - koliko košta pronalazak određenog elementa ili skupa elemenata?
- indeksi predstavljaju vremensko i prostorno opterećenje
  - indeksi moraju biti ažurirani
    - često usporavaju update operaciju dok ubrzavaju select
- različiti indeksi podrazumijevaju različita opterećenja
  - koliko je vremena potrebno za dodavanje zapisa u indeks?
  - koliko je vremena potrebno za brisanje zapisa iz indeksa?
  - koliko dodatnog prostora zauzima indeks?

# Karakteristike indeksa

- dvije su glavne kategorije indeksa:
  - sortirani indeksi (*engl. ordered indexes*) - pohranjuju vrijednosti u sortiranom redoslijedu
  - hash indeksi - dijele vrijednosti u blokove, koristeći hash funkciju
- postoji mnogo varijanti unutar tih kategorija – npr. sortirani
  - primarni (*clustering index*) i sekundarni indeksi (*nonclustering index*), u ovisnosti o organizaciji datotečne tablice
  - *dense* vs. *sparse* indeksi (*hrv. gusti i rijetki!?*)
    - dense indeksi uključuju svaku pojedinu vrijednost iz izvornog stupca, brži su za pretragu ali zauzimaju više prostora
    - sparse indeksi uključuju neke vrijednosti iz izvornog stupca, pretraga zahtjeva više operacija ali je indeks manji

# Implementacija indeksa

- indeksi koje ćemo danas obraditi su dense indeksi
  - obzirom da proučavamo heap organizaciju datoteke, indeks koji ne obuhvaća sve vrijednosti kolone ne bi bio previše koristan
- indeksi se u pravilu pohranjuju u datoteke koje su odvojene od datoteke tablice
  - čitaju se i zapisuju u blokovima, iz istog razloga kao i prije
- koriste pokazivače (*engl. record pointers*) da bi referencirali specifične zapise u tablici
  - sadrži broj bloka i odmak na kojem se zapis nalazi
- zapisi indeksa sadrže vrijednosti (ili hash) i jedan ili više pokazivača na zapise tablice koje sadrže te vrijednosti

# Implementacija indeksa

- sve baze podataka pružaju sortirane indekse koji se temelje na nekoj vrsti strukture balansiranog stabla
  - $B^+$-tree i B-tree indeksi – tipično se nazivaju btree indeksi
- neke baze podataka pružaju i hash indekse
  - složeniji za upravljanje u odnosu na sortirane indekse, stoga nisu česti u open-source bazama podataka
- postoje i drugi tipovi indeksa
  - Bitmap indeksi – za ubrzanje upita na višestrukim kolonama
    - također ne pojavljuju se često u o-s bazama
  - R-tree indeksi – za brzu obradu upita koji koriste prostorne podatke

# $B^+$-Tree indeksi

- široko korišteni format za pohranu sortiranih indeksa
- upravlja strukturom balansiranog stabla
  - svaki put od korijena do lista ima jednak put
  - postaje učinkovit za upite, čak i uz umetanja i brisanja
- može zauzeti značajan dio prostora, obzirom da pojedini čvorovi mogu biti polu prazni
- ažuriranje indeksa za umetanje i brisanje ponekad može biti sporo
  - potrebno je ažurirati stablo
- poboljšanje performansi značajno premašuje slabosti ovog formata

# $B^+$-Tree indeksi

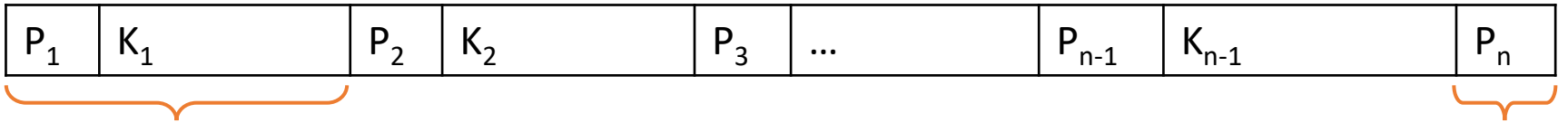
- svaki čvor stabla ima do *n* djece
  - broj n je fiksni za cijelo stablo
- svaki čvor pohranjuje *n* pokazivača i *n-1* vrijednosti

| $P_1$ | $K_1$ | $P_2$ | $K_2$ | $P_3$ | … | $P_{n-1}$ | $K_{n-1}$ | $P_n$ |
|---|---|---|---|---|---|---|---|---|

  - $K_i$ su vrijednosti ključa pretrage (*engl. search-key*), $P_i$ su pokazivači na zapise
  - vrijednosti se pohranjuju u sortiranom redoslijedu: ako $i < j$ tada $K_i < K_j$
  - svi čvorovi osim korijena moraju biti polu-puni
- veličina *n* ovisi o veličini bloka, veličini ključa pretrage i veličini pokazivača na zapis – u pravilu je *n* velik
  - btree indeksi su široke, plitke strukture stabla

# $B^+$-Tree čvorovi

- za čvorove lista vrijedi:



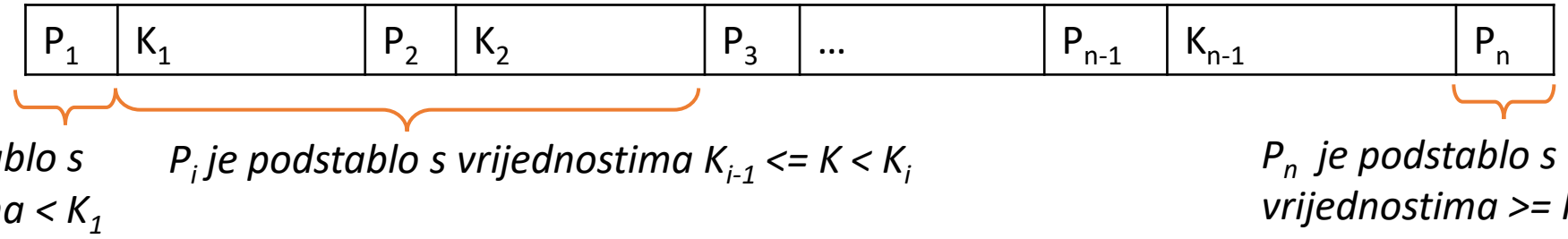

*$P_i$ pokazuje na zapis(e) koji sadrži vrijednost $K_i$*

*$P_n$ pokazuje na slijedeći list u nizu tj. na list koji počinje s vrijednosti $K_n$*

  - ukoliko je ključ pretrage ključ kandidat, tada $P_i$ pokazuje na zapis s vrijednosti $K_i$
  - ukoliko ključ pretrage nije ključ kandidat (u sekundarnom indeksu), tada $P_i$ pokazuje na kolekciju pokazivača koji pokazuju na sve zapise s vrijednosti $K_i$
- niti jedna dva lista se ne preklapaju u rasponu
  - zbog toga listovi mogu biti poredani u sekvencijalnom nizu

# $B^+$-Tree čvorovi

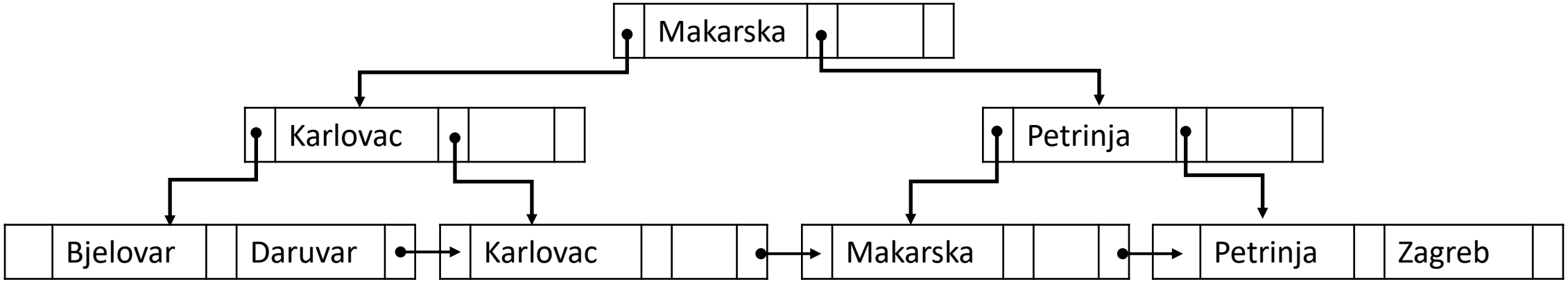
- za čvorove koji nisu list vrijedi:




  - svi pokazivači pokazuju na podstabla
- unutarnji pokazivači
  - pokazuju na podstabla koji imaju pokazivače na vrijednosti ključa pretrage barem $K_{i-1}$ i ne veće od $K_i$
- vanjski pokazivači
  - pokazuju na podstabla s većom ili manjom vrijednosti ključa pretrage

# Primjer $B^+$-Tree

- jednostavno $B^+$-Tree s $n$=3




- izvršavanje upita je jednostavno
- umetanje može zahtijevati da se jedan ili više čvorova podijele
- brisanje može zahtijevati da se jedan ili više redova spoje

# $B^+$-Tree i string ključevi

- string ključevi mogu biti problematični za indeksiranje
  - često su definirani da sadržavaju velike/varijabilne-veličine
  - veliki ključevi smanjuju faktor grananja za svaki čvor, povećavajući dubinu stabla i cijenu pristupa
  - veliki ključevi također mogu utjecati na restrukturiranje stabla
- rješenje je ne koristiti cijeli string
  - može se koristiti tehnika korištenja podstringa (*engl. prefix compression*)
  - čvorovi koji nisu listovi mogu pohraniti samo dio stringa
  - veličina prefiksa mora biti razumno velika da bi se dobro razdvajale vrijednosti između svakog podstabla

# Indeksi i upiti

- indeksi pružaju alternativni pristupni put do nekog zapisa u tablici
  - ukoliko se traži neka vrijednost ili njihov raspon indeks se koristi da bi se pronašla polazišna točka pretrage u tablici
- planer upita traži indekse na ključnim stupcima prilikom optimizacije upita
- primjer upita

```
SELECT * FROM racun WHERE racun_id = '591';
```

- ukoliko postoji indeks na *racun_id* stupcu, planer može koristiti taj indeks umjesto da radi scan datoteke
  - plan izvršavanja je anotiran s takvim detaljima

# Ključevi i indeksi

- baza podataka može automatski stvoriti indekse
  - DB će stvoriti indeks za kolone primarnog ključa, i ponekad na kolonama stranog ključa
  - to omogućava bazi puno brže provođenje ograničenja primarnog ključa i referencijalnog integriteta
- mnogi upiti koje smo koristili do sada koriste te indekse
  - npr. traženje po primanom ključu ili join primarnog i stranog ključa
- ponekad upiti koriste stupce koji nemaju definirane indekse

```
SELECT * FROM nastavnik WHERE primanja > 5000.00;
```

  - kako znamo koje indekse baza podataka koristi?
  - kako dodati nove indekse na tablice?

# EXPLAIN yourself

- većina baza podataka ima EXPLAIN tip naredbe
  - izvodi planiranje i optimizaciju upita te potom ispisuje detalje o planu izvođenja
  - ispisuje između ostalog i koji se indeksi koriste
- MySQL EXPLAIN naredba

```
EXPLAIN SELECT * FROM nastavnik WHERE id = 240;
```

```
+----+-------------+----------+------------+-------+---------------+---------+---------+-------+------+----------+-------+
| id | select_type | table    | partitions | type  | possible_keys | key     | key_len | ref   | rows | filtered | Extra |
+----+-------------+----------+------------+-------+---------------+---------+---------+-------+------+----------+-------+
|  1 | SIMPLE      | nastavnik | NULL       | const | PRIMARY       | PRIMARY | 4       | const |    1 |   100.00 | NULL  |
+----+-------------+----------+------------+-------+---------------+---------+---------+-------+------+----------+-------+
```

  - upit koristi indeks primarnog ključa za dohvat podataka
  - MySQL zna da će rezultat biti jedan red ili bez redova

# EXPLAIN yourself

- pogledajmo rezultate za drugi ID nastavnika

```
EXPLAIN SELECT * FROM nastavnik WHERE id = 1040;
```

```
+----+-------------+-------+-----+-------------------------------+
| id | select_type | table |     | Extra                         |
+----+-------------+-------+-----+-------------------------------+
|  1 | SIMPLE      | NULL  | ... | no matching row in const table |
+----+-------------+-------+-----+-------------------------------+
```

  - MySQL planer koristi indeks primarnog ključa da otkrije da korišteni ID ne postoji u tablici

- još jedan primjer:

```
EXPLAIN SELECT * FROM nastavnik WHERE primanja > 10000;
```

```
+----+-------------+----------+------------+------+---------------+------+---------+------+------+----------+-------------+
| id | select_type | table    | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra       |
+----+-------------+----------+------------+------+---------------+------+---------+------+------+----------+-------------+
|  1 | SIMPLE      | nastavnik | NULL       | ALL  | NULL          | NULL | NULL    | NULL |  250 |    33.33 | Using where |
+----+-------------+----------+------------+------+---------------+------+---------+------+------+----------+-------------+
```

# Dodavanje indeksa tablicama

- ukoliko veliki broj upita referencira stupce koji nemaju indeksa, performanse mogu postati problem
  - potrebno je stvoriti dodatne indekse kako bi se olakšao posao baze podataka
- indeks se stvara pomoću naredbe `CREATE INDEX`
- da bi se ubrzao upit koji koristi primanja nastavnika

```
CREATE INDEX idx_primanja ON nastavnik (primanja);
```

  - baza podatka će stvoriti novi indeks i popuniti ga sadržajem koji se trenutno nalazi u tablici
    - za velike tablice izvršavanje naredbe može potrajati
- mogu se definirati i indeksi na više stupaca

# Dodavanje indeksa tablicama

- prilikom stvaranja mogu se navesti razne opcije, kao na primjer tip indeksa
  - sve baze podataka će ukoliko se drugačije ne navede stvoriti btree indeks
- MySQL dopušta da se indeksi definiraju prilikom stvaranja tablice
  - ali mnoge druge baze ne
- postoji li negativna strana stavljanja indeksa na saldo računa u banci?
  - riječ je o banci, salda računa se stalno mijenjaju
  - definitivno će rezultirati s lošijim performansama za update
    - premda pogoršanje ne mora biti značajno

# Provjera korištenja indeksa

- vrlo je bitno provjeriti da li novo dodani indeks baza podataka zapravo koristi
  - ukoliko upiti ne koriste indeks najbolje ga se riješiti

```
EXPLAIN SELECT * FROM nastavnik WHERE primanja > 10000;
```

```
+----+-------------+----------+------------+------+---------------+------+---------+------+------+----------+-------------+
| id | select_type | table    | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra       |
+----+-------------+----------+------------+------+---------------+------+---------+------+------+----------+-------------+
|  1 | SIMPLE      | nastavnik | NULL       | ALL  | idx_primanja  | NULL | NULL    | NULL |  250 |    54.00 | Using where |
+----+-------------+----------+------------+------+---------------+------+---------+------+------+----------+-------------+
```

- u ovom slučaju baza ne koristi indeks
  - ukoliko drugi zahtjevni upiti koriste indeks, ima ga smisla zadržati
  - u suprotnom, najbolje ga je obrisati i ubrzati update operacije

# Indeksi na velikim vrijednostima

- veliki ključevi ozbiljno mogu utjecati na performanse indeksa
- primjer: $B^+$-Tree i B-Tree
  - najveći doprinos je veliki faktor grananja svakog čvora
  - velike ključne vrijednosti će dramatično smanjiti faktor, time produbiti stablo i povećati IO trošak
- indeks se može definirati samo na prvih *N* znakova ili bajtova stringa/LOB vrijednosti

```
CREATE INDEX idx_opis ON djelatnik (opis(5));
```

  - koristi se samo prvih 5 znakova opisa djelatnika za indeks
  - ukoliko se većina vrijednosti razlikuje u prvih 5 bajtova, indeks će biti manji i brži za upite i update
  - ukoliko se vrijednosti ne razlikuju, indeks neće donijeti mnogo razlike

# Indeksi i utjecaj na performanse

- dodavanje indeksa je uobičajen zadatak za većinu projekata na bazama podataka
- kao način utjecanja na performanse, koristi se kada baza podataka već ima podatke i upiti postaju spori
  - ranu optimizaciju uvjek treba izbjegavati
  - prilikom optimizacije mora se razumjeti što baza podataka zapravo radi
- indeksi predstavljaju opterećenje na vrijeme i prostor
  - ubrzavaju upite ali usporavaju sve modifikacije
- uvjek je dobro potvrditi da se novi indeks koristi, ukoliko to nije slučaj najbolje ga je obrisati

# Literatura

- Pročitati
  - [DSC] poglavlje 11.1. – 11.4
  - korisno ([DSC] poglavlje 10.5)
  - Caltech CS121 - 11
- Slijedeće predavanje
  - [DSC] poglavlje 12.1. – 12.6.
  - Caltech CS121 - 12