Predavanje VII.

Evaluacija SQL upita

BAZE PODATAKA II

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

Sadržaj

- ponavljanje prethodnih predavanja
  - indeksi
  - implementacija indeksa
  - B+-Tree indeksi
  - EXPLAIN naredba
- evaluacija upita
- optimizacija upita
- operacija selekcije
- operacija projekcije
- join operacije
  - ugniježđena petlja
  - sort-mege
  - hash
  - outer join
- izvođenje upita primjer

Ponavljanje

- indeksi 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 puno manji od tablice
  - puno je brže izvoditi pretragu unutar indeksa
- predstavljaju vremensko i prostorno opterećenje
  - moraju biti ažurirani

Ponavljanje

- indeksi se u pravilu pohranjuju u datoteke koje su odvojene od datoteke tablice
  - čitaju se i zapisuju u blokovima
- 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
- različite vrste implementacije indeksa

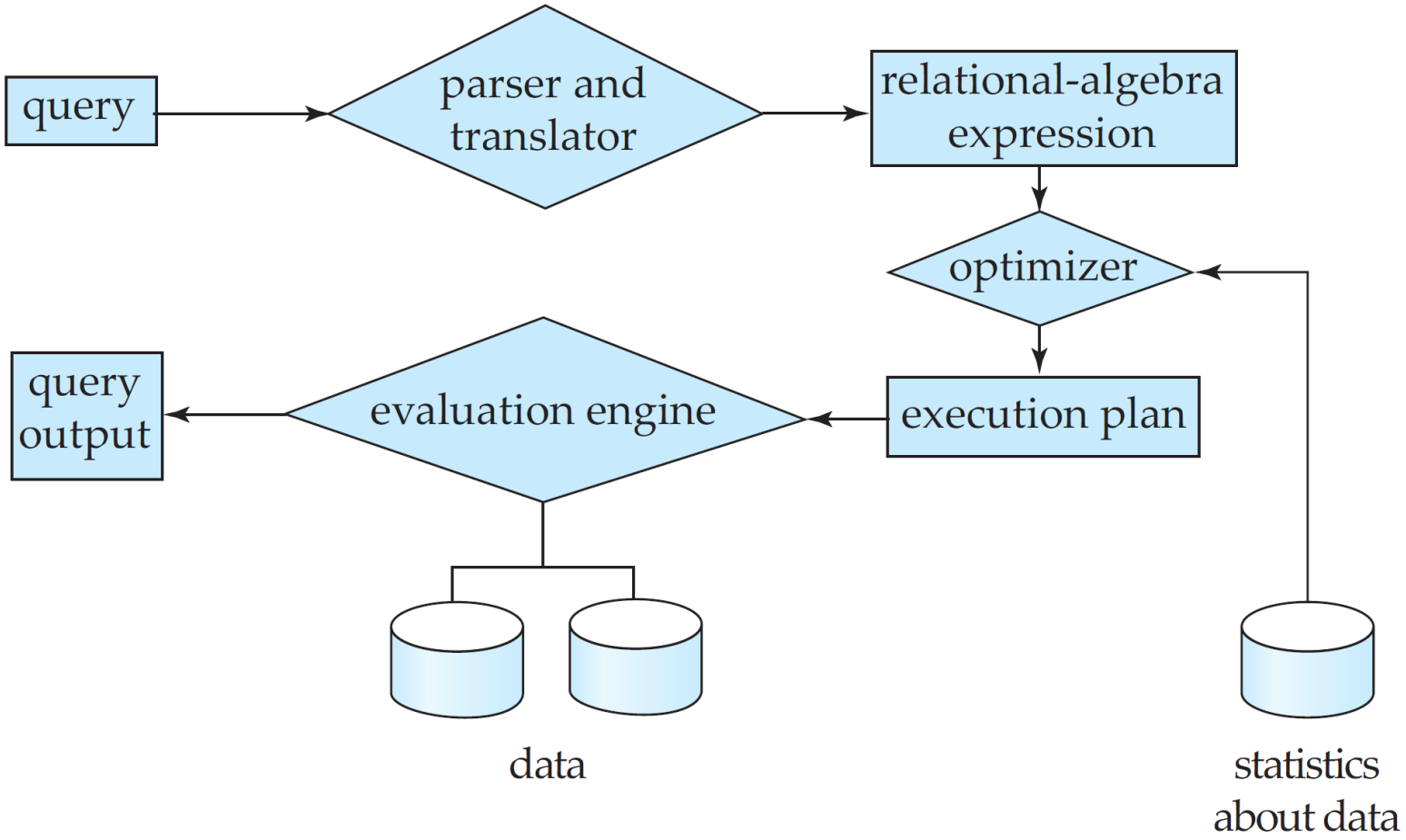
Ponavljanje

- B^+-Tree - 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 može biti sporo
  - potrebno je ažurirati stablo
- poboljšanje performansi značajno premašuje slabosti ovog formata

Evaluacija upita

- na zadnjem predavanju smo se bavili
  - detaljima implementacije baze podataka
  - kako se podaci pohranjuju te kako im baza podataka pristupa
  - kako korištenje indeksa može značajno ubrzati dohvat podataka za određene upite
- u nastavku…
  - što se događa kada zadamo upit?
    - te kako ga možemo ubrzati?
- da bi mogli optimizirati upita potrebno je razumjeti što baza podataka čini da bi došla do rezultata

Koraci obrade upita



Evaluacija upita

- potrebno je razumjeti:
  - detalje evaluacije upita
  - kako su implementirane operacije relacijske algebre
    - česti slučajevi korištene optimizacije u implementacijama
  - kako se koriste ti detalji od strane baze podataka za bolje planiranje i optimizaciju upita
- postoje izuzetci od pravila
  - npr. MySQL procesor join-a je drugačiji od ostalih sustava
  - svaki DBMS pruža dokumentaciju o načinu kako je implementirana evaluacija i optimizacija upita
    - postoje razlike od slučaja do slučaja, stoga je dokumentacija nužna

Procesiranje SQL upita

- baze podataka obrađuju SQL upit kroz tri osnovna koraka:
  - parsiranje SQL u internu reprezentaciju plana
  - transformiranje interne reprezentacije u optimiziran plan izvršavanja
  - evaluacije optimiziranog plana
- planovi izvršavanja se u pravilu temelje na proširenoj relacijskoj algebri
  - uključuju generalizirane projekcije, grupiranje i sl.
  - ali i neke druge značajke kao što su sortiranje rezultata, ugniježđeni podupiti, LIMIT/OFFSET…
    - LIMIT određuje broj dohvaćenih redova u rezultatu
    - OFFSET zamiče rezultate za određeni broj redova

Primjer evaluacije upita

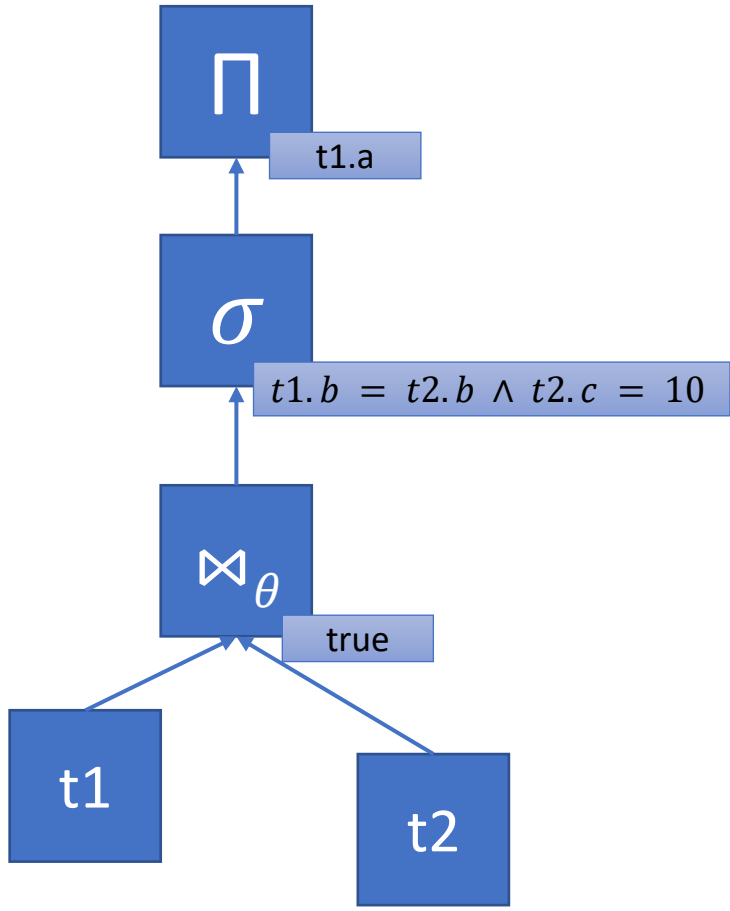
- pogledajmo jednostavan upit:

```
SELECT t1.a FROM t1, t2
   WHERE t1.b = t2.b AND t2.c = 10;
```

- upit izražen u relacijskoj algebri

$$\Pi_{t1.a}(\sigma_{t1.b = t2.b \wedge t2.c = 10}(t1 \times t2))$$

- baza podataka bi mogla stvoriti prikazanu strukturu
  - BP u pravilu implementira operaciju običnog join-a pomoću theta-join čvora
  - evaluacijska petlja prima rezultate iz gornjeg čvora Π



Optimizacija upita

- da li je moguće isti upit zapisati na drugi način?

$\Pi_{t1.a}(\sigma_{t1.b = t2.b \wedge t2.c = 10}(t1 \times t2))$

$\Pi_{t1.a}(t1 \bowtie_{t1.b = t2.b} (\sigma_{t2.c = 10}(t2))$

$\Pi_{t1.a}(\sigma_{t2.c = 10}(t1 \bowtie_{t1.b = t2.b} t2))$

  - koji je najbrži?
- optimizator upita generira mnogo ekvivalentnih planova koristeći skup pravila jednakosti
  - cost-based optimizatori svakom planu dodjeljuju trošak te potom izabiru onog s najmanjim troškom
  - heuristični optimizatori prate skup pravila za optimizaciju

Troškovi u evaluaciji upita

- postoji mnogo troškova koji se mogu promatrati prilikom evaluacije upita
- primarni trošak je čitanje podataka s diska
  - u pravilu podaci koji se obrađuju ne stanu u potpunosti u memoriju
  - potrebno je što više minimizirati pretragu po disku, pisanje i čitanje
- troškovi CPU-a i memorije su sekundarni
  - određeni načini računanja rezultata zahtijevaju više resursa CPU-a i memorije
  - trošak postaje jako bitan u slučaju paralelnog korištenja mnogo korisnika
- postoje i drugi troškovi
  - u distribuiranim sustavima, optimizator upita kontrolira mrežnu propusnost

Optimizacija upita

- pitanja koja optimizator upita mora razmotriti:
- kako su podaci relacije spremljeni na disku?
  - i koji su putevi pristupa dostupni do podataka?
- koje implementacije operacija relacijske algebre su dostupne za korištenje?
  - hoće li određena operacija biti značajnije bolja ili lošija od druge?
- kako baza podataka odlučuje koje je najbolji plan za izvršavanje upita?
- obzirom na odgovore zadanih pitanja, što mi možemo učiniti da baza podataka brže radi?
- *detaljnije o optimizaciji upita na slijedećem predavanju*

Operacija selekcije

- kako implementirati σ_P operaciju?
- jednostavno rješenje s prošlog predavanja: scan datoteke s podacima
  - file scan
  - testiranje predikata selekcije za svaku n-torku u datoteci s podacima
  - spora operacija jer se moraju učitati svi blokovi
- to je općenito rješenje ali nije brzo
- od čega se sastoji predikat selekcije P?
  - ovisno o karakteristikama P mogli bismo odabrati optimalniji plan evaluacije upita
  - ukoliko ne možemo, držimo se file scan-a

Operacija selekcije

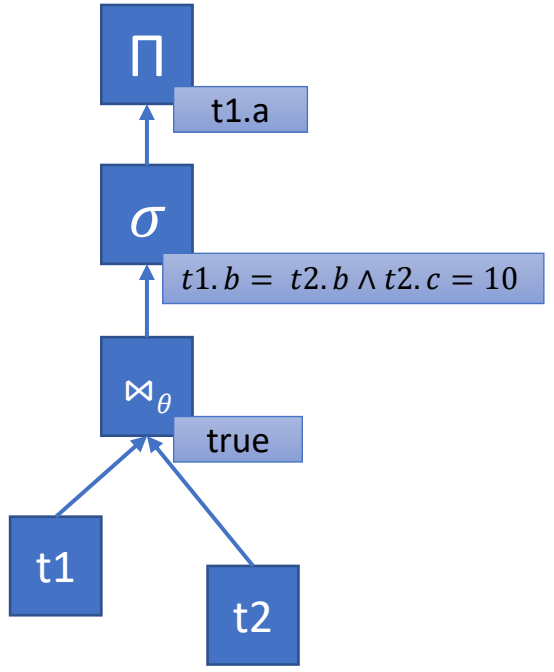
- većina predikata selekcije koristi binarnu usporedbu
  - da li je atribut jednak nekoj vrijednosti?
  - da li je atribut manji od neke vrijednosti?
- ukoliko je datoteka s podacima sortirana, možemo koristiti binarnu pretragu
  - značajno bi smanjili broj blokova koji trebamo čitati
  - održavanje logički redoslijed zapisa može biti skupo ukoliko se podaci često mijenjaju
- rješenje
  - koristimo heap file organizaciju datoteke s podacima
  - za važne atribute stvorimo indekse na datoteci s podacima

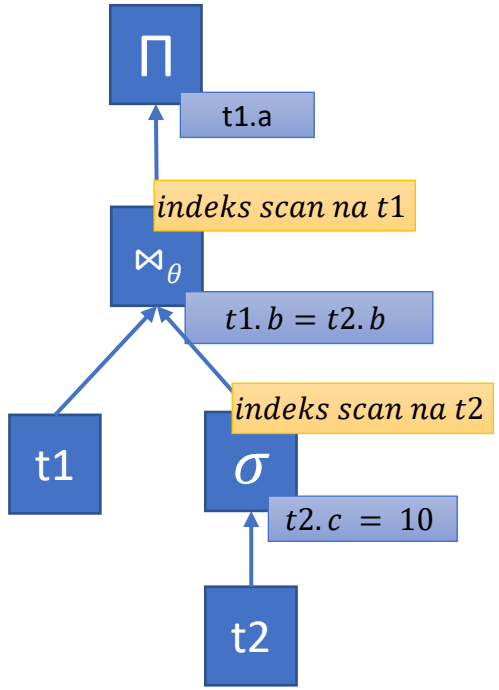
Operacije selekcije

- planer/optimizator upita će istražiti sve pristupne puteve za dani atribut
- za operaciju selekcije:
  - ukoliko je predikat selekcije test jednakosti i ukoliko je indeks dostupan za atribut tada se može koristiti indeks scan
  - indeks scan se može također koristiti za testove usporedbe/raspona ukoliko se radi o sortiranom indeksu
- za složenije testove ili ako ne postoji indeks za atribute korištene u upitu
  - koristi se file scan pristup

Optimizacija upita koristeći indekse

- optimizator upita pretražuje dostupne indekse
  - ukoliko select operacija može koristiti indeks plan je označen s tim detaljem
  - ukupni trošak se računa s obzirom na tu optimizaciju
- indeksi se mogu iskoristiti samo u određenim uvjetima
  - tipično samo od čvorova koji imaju direktan dostup tablici
  - prijašnji plan ne može iskoristiti indekse





Operacija projekcije

- operacija projekcije je jednostavna za implementaciju
  - za svaku ulaznu n-torku, stvori novu n-torku koja uključuje samo određene atribute
  - mogu uključivati i izračunate vrijednosti
- koji bi upit generalno bio brži?

  $\Pi_{stanje}(\sigma_{stanje<2500}(racun))$

  $\sigma_{stanje<2500}(\Pi_{stanje}(racun))$

  - želimo projicirati što manje redova da bi smanjili korištenje CPU-a i memorije
    - u skladu s tim prvi upit je bolji
  - dobar primjer heuristike: "Koristi projekciju što kasnije!"*
    - * u pravilu koristi selekciju prije projekcije, ali dobro je rano korištenje, radi međurezultata

Sortiranje

- SQL omogućava sortiranje redova rezultata
- baza podataka mora uključiti sortiranje u planove izvršavanja upita
  - podaci koji se sortiraju mogu biti značajnije veći od dostupne memorije
- za tablicu koja stane u memoriju koriste se klasični algoritmi za sortiranje (npr. quick-sort)
- za tablice koje ne stanu u memoriju mora se koristiti neka od tehnika za sortiranje u vanjskoj memoriji
  - tablica se dijeli u dijelove koji će biti sortirani u memoriji
  - svaki dio se zasebno sortira te se pohranjuje u privremenu datoteku
  - svi dijelovi se spajanju korištenjem n-way merge sort-a

Sortiranje

- u pravilu sortiranje treba biti primijenjeno što je kasnije moguće
  - u idealnom slučaju redovi koji se sortiraju će stati u memoriju
- druge operacije mogu koristiti rezultate sortiranja da bi poboljšale performanse
  - join operacije
  - grupiranje i agregacija
- sortiranje se može izvoditi uz pomoć sortiranog indeksa
  - radi se scan indeksa i dohvaćaju se svi zapisi tablice u sortiranom redoslijedu
  - s magnetnim diskovima, vrijeme traženja u pravilu čini ovaj pristup neisplativim
    - SSD nemaju taj problem

Join operacije

- join operacije su vrlo česte u SQL upitima
  - pogotovo kada se koriste normalizirane sheme
- potencijalno join operacija može biti vrlo skupa
  - r ⋈ s definirano kao $\sigma_{r.A=s.A}(r \times s)$
- jednostavna strategija za $\bowtie_\theta$

```
for each tuple t_r in r do begin
  for each tuple t_s in s do begin
    if t_r , t_s satisfy condition q then
    add t_r · t_s to result
  end
end
```

- $t_r \cdot t_s$ označava nadovezivanje (*engl. concatenation*) t_r s t_s

Join ugniježđenom petljom

- algoritam ugniježđene petlje

```
for each tuple t_r in r do begin
  for each tuple t_s in s do begin
    if t_r , t_s satisfy condition q then
    add t_r · t_s to result
  end
end
```

- vrlo spora implementacija join-a…
  - skenira *r* jednom, potom skenira *s* jednom za svaki red u *r*!
  - nije strašno ukoliko svi podaci stanu u memoriju
- … ali se može nositi s proizvoljnim uvjetima
  - za neke upite može biti jedino rješenje

Join ugniježđenom petljom

- većina join operacija uključuje uvjete jednakosti
  - nazivaju se equijoins
- indeksi mogu ubrzati traženje u tablicama…
- modifikacija join-a ugniježđenom petljom da bi unutarnja petlja koristila indeks

for each *tuple* t_r ***in*** *r* ***do begin***
use index on *s* to retrieve tuple t_s
if t_r *,* t_s *satisfy condition q* ***then***
    add $t_r \cdot t_s$ *to result*
end

- opcija samo za equijoin-ove, gdje indeks postoji na join atributima

MySQL join procesor

- MySQL join procesor se temelji na algoritmu za join ugniježđene petlje
  - umjesto dvije tablice, može spojiti N tablica odjednom

for each *tuple* t_r ***in*** *r* ***do begin***
  for each *tuple* t_s ***in*** *s* ***do begin***
    for each *tuple* t_t ***in*** *t* ***do begin***
      if t_r, t_s, t_t, *… satisfy condition q* ***then***
      add $t_r \cdot t_s \cdot t_t \cdot$ *… to result*
    end
  end
end

- koristi mnogo optimizacija
  - kad je moguće, vanjska tablica se obrađuje u blokovima, da bi se smanjio broj iteracija na unutarnjim tablicama
  - jako se iskorištavaju indeksi za pronalazak vrijednosti u unutarnjim tablicama
  - ukoliko se podupit može pretvoriti u konstantu, to se čini

MySQL join procesor

- obzirom da se MySQL procesor jako temelji na indeksima, za koje upite nije tako dobar?
  - upite na tablicama koje nemaju indeksa
  - upiti koji koriste join na izvedenim relacijama, starije MySQL verzije
  - MySQL 8.0 i novije - ako spremi izvedenu relaciju u privremenu tablicu, može napraviti indekse na tablici
    - česta tehnika za optimizaciju složenih upita u MySQL-u
- za složenije upite dobro bi došli sofisticiraniji algoritmi za spajanje tablica
  - većina baza podataka uključuje nekoliko snažnih join algoritama
  - i MySQL od 8.0.18

Sort-merge join

- ukoliko su tablice sortirane po join atributima može se koristiti merge-sort tehnika
  - mora biti equijoin
- jednostavan high-level opis:
  - dva pokazivača obilaze sortirane tablice
    - p_r počinje s prvom n-torkom u r
    - p_s počinje s prvom n-torkom u s
  - ukoliko n-torka nekog pokazivača ima manje vrijednosti od drugog pokazivača, povećaj taj pokazivač
  - ukoliko pokazivači pokazuju na iste vrijednosti u obje tablice, stvori join koristeći te redove

Sort-merge join

- mnogo bolje performanse nego join pomoću ugniježđenih petlji
  - značajno smanjuje pristupanje disku
  - nažalost tablice nisu uvjek sortirane
- korištenje merge sort join kada barem jedna relacija ima indeks na join atributima
  - npr. jedna relacija je sortirana a druga relacija ima definirane indekse na atributima koji se koriste u join-u
  - obilazak indeksa nesortirane relacije
  - kada se vrijednosti poklapaju, koristi se indeks da bi se povukli ti redovi
  - ukoliko se koristi ova tehnika mora se oprezno upravljati troškom pretrage po disku

Hash join

- još jedna join tehnika za equijoin
- za tablice *r* i *s*…
- koristi se hash funkcija na join atributima za podjelu redova iz *r* i *s* u particije
  - ista hash funkcija za obje tablice
  - particije se spremaju na disk kako se kreiraju
  - cilj je da svaka particija stane u memoriju
  - *r* particije: H_{r1}, H_{r2}, …, H_{rn}
  - *s* particije: H_{s1}, H_{s2}, …, H_{sn}
  - redovi u H_{ri} će se uparivati samo s redovima iz H_{si}

Hash join

- nakon stvaranja particija

for *i = 1* ***to*** *n* ***do***
  build a hash index on H_{si} *(koristi se druga hash funkcija!)*
  for each *row tr* ***in*** H_{ri}
    probe hash index for matching rows in H_{si}
    for each *matching tuple* t_s ***in*** H_{si}
      add $t_r \cdot t_s$ *to result*
    end
  end
end

- vrlo brza i učinkovita equijoin strategija
  - vrlo dobra za spajanje unutar izvedenih relacija
  - performanse mogu značajno opasti ukoliko se redovi ne mogu hash-irati u particije koje stanu u memoriju

Outer joins

- join algoritmi mogu biti modificirani tako da generiraju left outer join-ove razumno učinkovito
  - right outer join može biti transformiran kao left outer join
  - utjecati će na performanse ukoliko se generira veliki broj redova
- full outer join može biti značajnije teže implementirati
  - sort-merge join može izračunati full outer join relativno jednostavno
  - ugniježđeni join i hash join se puno teže primjenjuju
  - full outer join također može imati značajan utjecaj na performanse traženog upita

Druge operacije

- operacije na skupovima zahtijevaju eliminaciju duplikata
  - eliminacija duplikata se može izvesti pomoću sortiranja i hash-iranja
- operacije grupiranja i agregacije se mogu izvesti na više različitih načina
  - rezultati se mogu sortirati prema atributima grupiranja, te se potom računaju agregirane vrijednosti na sortiranim vrijednostima
    - svi redovi u danoj grupi su susjedni, stoga se memorija koristi vrlo učinkovito (nakon sortiranja)
    - MySQL koristi ovaj pristup kao zadani
  - može se koristiti i hash-iranje za izvođenje grupiranja i agregiranja
    - hash n-torki na atributima grupiranja, računanje agregiranih vrijednosti pojedinih grupa inkrementalno

Optimiziranje performansa upita

- ta bi poboljšali performanse upita potrebno je znati kako baza zapravo izvršava upit
- korištenje EXPLAIN izraza s prošlog predavanja
  - nakon izvođenja planera i optimizatora, prikazuje plan i procijenjeni trošak upita
- koristeći to informaciju možemo:
  - kada je potrebno, stvoriti indekse na tablici
  - izmijeniti upit na način da pomognemo bazi da odabere bolji plan
- teži slučajevi zahtijevaju više koraka:
  - pohrana i korištenje među-rezultata

Izvođenje upita - primjer

- za svaki predani zadatak na e-učenju, potrebno je pronaći prosječnu veličinu zadnje predaje za taj zadatak

```
SELECT kratki_naziv,
       AVG(zadnja_velicina_zadace) AS
      prosjecna_zadnja_velicina_zadace
FROM zadaca NATURAL JOIN
     predana_zadaca NATURAL JOIN
     (SELECT pre_zad_id,
             ukupna_velicina AS zadnja_velicina_zadace
      FROM datoteka NATURAL JOIN
           (SELECT pre_zad_id, MAX(datum_predaje) AS datum_predaje
            FROM datoteka GROUP BY pre_zad_id
            ) AS datum_zadnje_predaje
      ) AS velicina_zadnje_predaje
GROUP BY kratki_naziv;
```

Izvođenje upita - primjer

- za svaki predani zadatak na e-učenju, potrebno je pronaći prosječnu veličinu zadnje predaje za taj zadatak

```
SELECT kratki_naziv,
       AVG(zadnja_velicina_zadace) AS
      prosjecna_zadnja_velicina_zadace
FROM zadaca NATURAL JOIN
     predana_zadaca NATURAL JOIN
     (SELECT pre_zad_id,
             ukupna_velicina AS zadnja_velicina_zadace
      FROM datoteka NATURAL JOIN
           (SELECT pre_zad_id, MAX(datum_predaje) AS datum_predaje
            FROM datoteka GROUP BY pre_zad_id
            ) AS datum_zadnje_predaje
      ) AS velicina_zadnje_predaje
GROUP BY kratki_naziv;
```

Pronađi datum zadnje predaje za svaku predaju. Naziv stupaca mora biti odgovarajući za natural join.

Izvođenje upita - primjer

- za svaki predani zadatak na e-učenju, potrebno je pronaći prosječnu veličinu zadnje predaje za taj zadatak

```
SELECT kratki_naziv,
       AVG(zadnja_velicina_zadace) AS
       prosjecna_zadnja_velicina_zadace
FROM zadaca NATURAL JOIN
     predana_zadaca NATURAL JOIN
     (SELECT pre_zad_id,
             ukupna_velicina AS zadnja_velicina_zadace
      FROM datoteka NATURAL JOIN
           (SELECT pre_zad_id, MAX(datum_predaje) AS datum_predaje
            FROM datoteka GROUP BY pre_zad_id
            ) AS datum_zadnje_predaje
      ) AS velicina_zadnje_predaje
GROUP BY kratki_naziv;
```

Join izvedenih rezultata s fileset tablicom da bi došli do veličine predane zadaće.

Izvođenje upita - primjer

- za svaki predani zadatak na e-učenju, potrebno je pronaći prosječnu veličinu zadnje predaje za taj zadatak

```
SELECT kratki_naziv,
       AVG(zadnja_velicina_zadace) AS
      prosjecna_zadnja_velicina_zadace
FROM zadaca NATURAL JOIN
     predana_zadaca NATURAL JOIN
     (SELECT pre_zad_id,
             ukupna_velicina AS zadnja_velicina_zadace
      FROM datoteka NATURAL JOIN
           (SELECT pre_zad_id, MAX(datum_predaje) AS datum_predaje
            FROM datoteka GROUP BY pre_zad_id
            ) AS datum_zadnje_predaje
      ) AS velicina_zadnje_predaje
GROUP BY kratki_naziv;
```

Vanjski upit računa prosjeke zadnjih predaja i dodaje kratki naziv zadatka.

MySQL izvršavanje i analiza

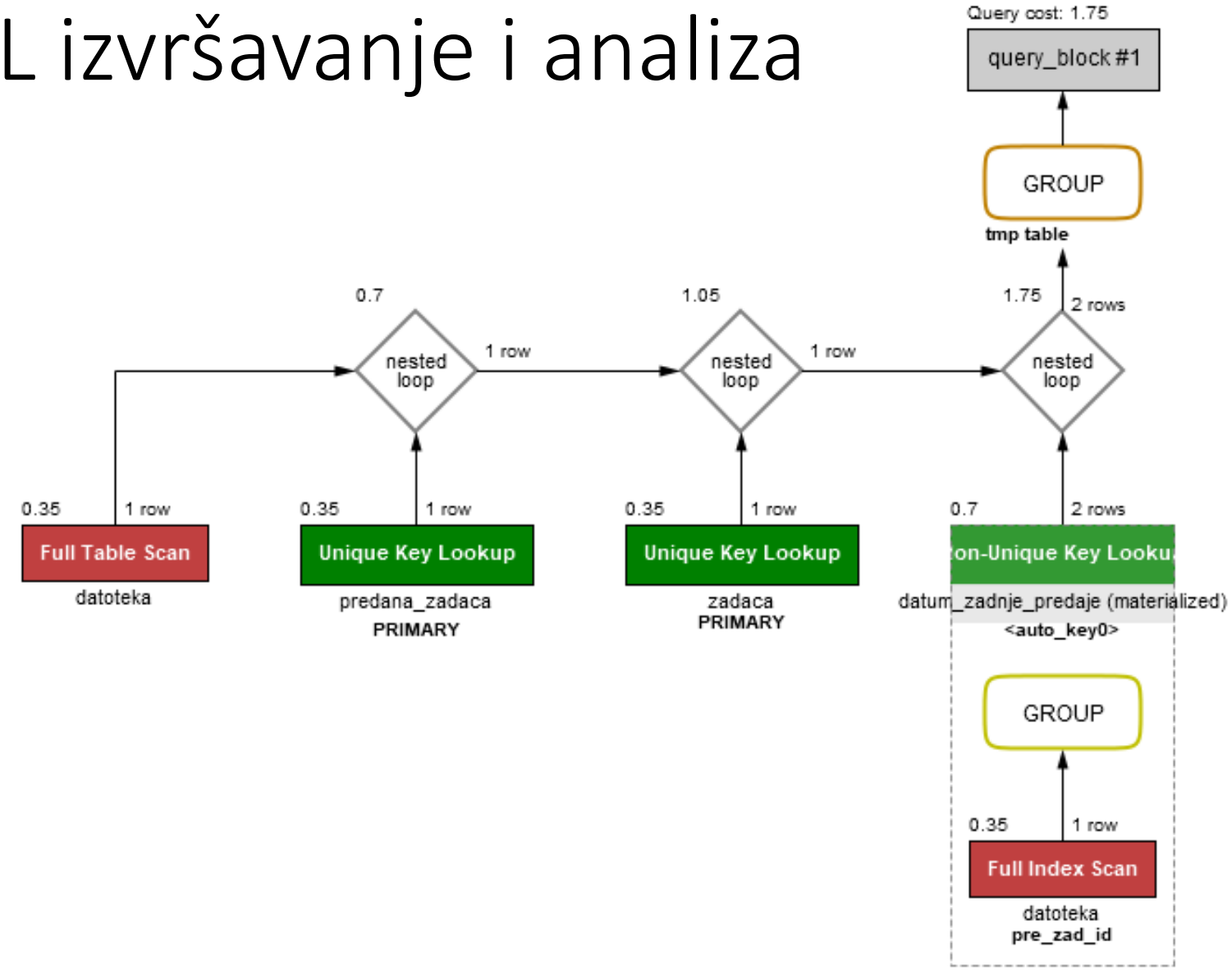
- MySQL izvršavanje upita

id	select_type	table	partitions	type	possible_keys	key	key_len
1	PRIMARY	datoteka	NULL	ALL	pre_zad_id	NULL	NULL
1	PRIMARY	zadaca	NULL	index	PRIMARY,zadaca_id,kratki_naziv	kratki_naziv	82
1	PRIMARY	predana_zadaca	NULL	eq_ref	PRIMARY,pre_zad_id,zadaca_id	PRIMARY	8
1	PRIMARY	<derived3>	NULL	ref	<auto_key0>	<auto_key0>	13
3	DERIVED	datoteka	NULL	index	pre_zad_id	pre_zad_id	8

ref	rows	filtered	Extra
NULL	1	100.00	Using where; Using temporary
NULL	7	100.00	Using index; Using join buffer (Block Nested Loop)
unipu.datoteka.pre_zad_id	1	5.00	Using where
unipu.datoteka.pre_zad_id,unipu.datoteka.datum_predaje	2	100.00	Using index
NULL	1	100.00	NULL

- FROMAT = JSON – detaljniji plan s troškom operacija

MySQL izvršavanje i analiza



Procjene upita

- planer/optimizator upita mora stvarati procjene troška izvršavanja za svaki upit
- baza podataka pohranjuje statistiku za svaku tablicu, da bi podržala planiranje i optimizaciju
- različiti detalji
  - neke baze podataka pohranjuju samo min/max/count vrijendosti svakog stupca, procjene su osnovne
  - druge generiraju i pohranjuju histograme vrijednosti za svaki bitni stupac, procjene su puno preciznije
- statistički podaci su vrlo skupi za održavanje, baza ih ne obnavlja prečesto
  - ukoliko se podaci često mijenjaju, statistički podaci se moraju osvježiti

Statistike tablica

- baze podataka pružaju način za računanje statistike tablica
- MySQL naredba

```
ANALYZE TABLE zadaca, predana_zadaca, datoteka;
```

- PostgreSql naredba

```
VACUUM ANALYZE;
```

  - za sve tablice u bazi podataka

```
VACUUM ANALYZE tablename;
```

  - za specifičnu tablicu
- prethodne naredbe su vrlo skupe
  - izvodi se puni table-scan i zaključavaju se tablice za ekskluzivni pristup

Zaključak

- način kako većina baza podataka vrednuje SQL upite
- neke operacije relacijske algebre mogu biti vrednovane na više različitih načina
  - optimizacije za neke od vrlo čestih slučajeva, npr. equijoin
- bazi podataka se mogu dati "upute"
  - staranje indeksa tamo gdje imaju smisla
  - formuliranje upita tako da budu učinkovitiji
  - osiguravanje da su statistike tablica što ažurnije, tako da planer upita ima najbolje šanse generirati dobar plan

Literatura

- Pročitati
  - [DSC] poglavlje 12.1. – 12.6.
  - Caltech CS121 - 12
- Slijedeće predavanje
  - [DSC] poglavlje 13.1. – 13.5.