**Univerzitet u Nišu, Elektronski fakultet**

**Katedra za računarstvo**

**SEMINARSKI RAD**

**Optimizacija upita kod MySQL**

Student:

Mladen Radenković 1083

Niš 2020. godine

Sadržaj

[Uvod 4](#_Toc37881022)

[Optimizacija pristupa podataka 4](#_Toc37881023)

[Načini restruktuiranja upita 4](#_Toc37881024)

[Složen upit ili više jednostavnih upita 4](#_Toc37881025)

[Seča upita 5](#_Toc37881026)

[Dekompozicija join-a 5](#_Toc37881027)

[Osnove izvršenja upita 5](#_Toc37881028)

[MySQL klijent/server protokol 6](#_Toc37881029)

[Stanja upita 7](#_Toc37881030)

[Keš upita 7](#_Toc37881031)

[Proces optimizacije upita 7](#_Toc37881032)

[Parser i preprocesor 8](#_Toc37881033)

[Optimizator upita 8](#_Toc37881034)

[MySQL-ova strategija izvršenja joina-a 11](#_Toc37881035)

[Plan izvršenja 13](#_Toc37881036)

[Optimizacija join-a 14](#_Toc37881037)

[Optimizacije sortiranja 15](#_Toc37881038)

[Mehanizam izvršenja upita 16](#_Toc37881039)

[Vraćanje rezultata klijentu 16](#_Toc37881040)

[Ograničenja MySQL optimizatora upita 16](#_Toc37881041)

[Korelisani podupiti 16](#_Toc37881042)

[Ograničenja UNION-a 18](#_Toc37881043)

[Optimizacija spajanja indeksa 19](#_Toc37881044)

[Paralelno izvršenje 19](#_Toc37881045)

[Labavo skeniranje indeksa 19](#_Toc37881046)

[MIN() I MAX() 21](#_Toc37881047)

[SELECT i UPDATE nad istom tabelom 21](#_Toc37881048)

[Optimizacija posebnih vrsta upita 22](#_Toc37881049)

[Optimizacija COUNT() upita 22](#_Toc37881050)

[Optimizacija JOIN upita 23](#_Toc37881051)

[Optimizacija GROUP BY i DISTINCT 23](#_Toc37881052)

[Optimizacija LIMIT i OFFSET 24](#_Toc37881053)

[Optimizacija SQL\_CALC\_FOUND\_ROWS 24](#_Toc37881054)

[Optimizacija UNION-a 25](#_Toc37881055)

[Opcije optimizatora upita 25](#_Toc37881056)

[Zaključak 27](#_Toc37881057)

[Literatura 27](#_Toc37881058)

# Uvod

Neophodni uslovi za postizanje visokih performansi prilikom izvršenja upita su optimizacija šeme baze podataka i dobro dizajniranje upita. Čak iako je šema baze podataka dobro dizajnirana, ako su upiti loši, performanse neće biti dobre.

U prvom delu ovog rada objasnićemo uopštena razmatranja dizajniranja upita – stvari koje prve treba da razmotrimo kada se upiti ne izvršavaju dobro. Zatim ćemo ući dublje u optimizaciju upita i internih podataka servera. Pokazaćemo kako se saznaje kako MySQL izvršava određeni upit, i kako da promenimo plan izvršenja upita. Takođe ćemo videti neka mesta na kojima MySQL ne vrši dobro optimizaciju upita i istražiti obrazce optimizacije upita koji pomažu MySQL-u da izvršava efikasnije upite.

Cilj ovog rada je objasniti kako MySQL stvarno izvršava upite. Na osnovu toga možemo razmišljati šta je efikasno, a šta neefikasno, i iskoristiti prednosti koje MySQL pruža, a izbegavati njegove slabosti.

# Optimizacija pristupa podataka

Osnovni razlog zašto se upiti ne izvršavaju dobro je zato što rade sa previše podataka. Većina loših upita mogu da se zamene da pristupaju manjoj količini podataka. Korisnik treba da analizira loše izvršenje upita u dva koraka:

1. Pronaći da li aplikacija pribavlja više podataka nego što je potrebno. Ovo obično znači da pristupa većem broju redova, ali takođe može da znači i da pristupa većem broju kolona.
2. Pronaći da li MySQL server analizira više redova nego što je potrebno.

# Načini restruktuiranja upita

Kako se bavimo opitimizacijom upita, cilj nam je da pronađemo alternativne načine za dobijanje željenih rezultata – ali to ne znači nužno dobijanje istih rezultata od MySQL-a. Ponekad možemo transformisati upite u ekvivalentne oblike i dobiti bolje performanse. Međutim, takođe možemo i prepisati upit tako da pribavimo drugačije rezultate, ako nam to pruža veću efikasnost. U ovom odeljku, objasnićemo tehnike koje mogu da nam pomognu za restruktuiranje velikog opsega upita.

## Složen upit ili više jednostavnih upita

Jedno bitno pitanje u dizaniranju upita je da li treba podeliti složen upit u nekoliko jednostavnijih upita. Tradicionalni pristup u dizajniranju baza podataka naglašava obavljanje što više posla sa što manje upita. Ovaj pristup je bio bolji zbog troškova mrežne komunikacije i preopterećenja parsiranja upita i faza optimizacije.

Međutim, ovaj savet nije mnogo primenljiv kod MySQL, zato što je on dizaniran da upravlja veoma efikasno konektovanjem i diskonektovanjem i da brzo odgovara malim i jednostavnim upitima. Savremene mreže se značajno brže nego one koje su se nekad koristile, tako da pokretanje većeg broja upita nije tako loša stvar. Dobra je ideja koristiti što je manje moguće upita, ali ponekad upit možemo učiniti efikasnijim njegovom dekompozicijom i izvršenjem nekoliko jednostavnih upita umesto jednog složenog.

Korišćenje previše upita je uobičajena greška u dizajnu aplikacije. Na primer, neke aplikacije izvršavaju 10 single-row upita za pribavljanje podataka iz tabele, a mogu da koriste jedan 10-row upit. Ponekad čak aplikacije pribavljaju svaku kolonu posebno, zahtevajući svaki red više puta.

## Seča upita

Drugi način za smanjenje upita je da podelimo upit tako da ga zadržimo u osnovi na isto ali da ga izvršavamo u manjim segmentima što utiče na manji broj redova svakog puta. Čišćenje starih podataka je dobar primer. Periodični poslovi čišćenja mogu da brišu sasvim malo podataka, a radeći to u jednom velikom upitu može da zaključa mnogo redova duže vreme, popuni transakcione logove i blokira male upite koji se ne bi trebalo prekidati. Iseći DELETE naredbu i koristiti upite srednje veličine može značajno da poboljša performanse.

## Dekompozicija join-a

Mnogi web sajtovi visokih performansi koriste dekompoziciju join-a. Join može da se razloži izvršenjem više upita nad jednom tabelom umesto join-a više tabela, i onda izvršenjem join-a u aplikaciji. Na prvi pogled ovo izgleda loše, jer povećavamo broj upita bez dobijanja ičega. Međutim, ovakvo restruktuiranje može da da značajne prednosti u preformansama:

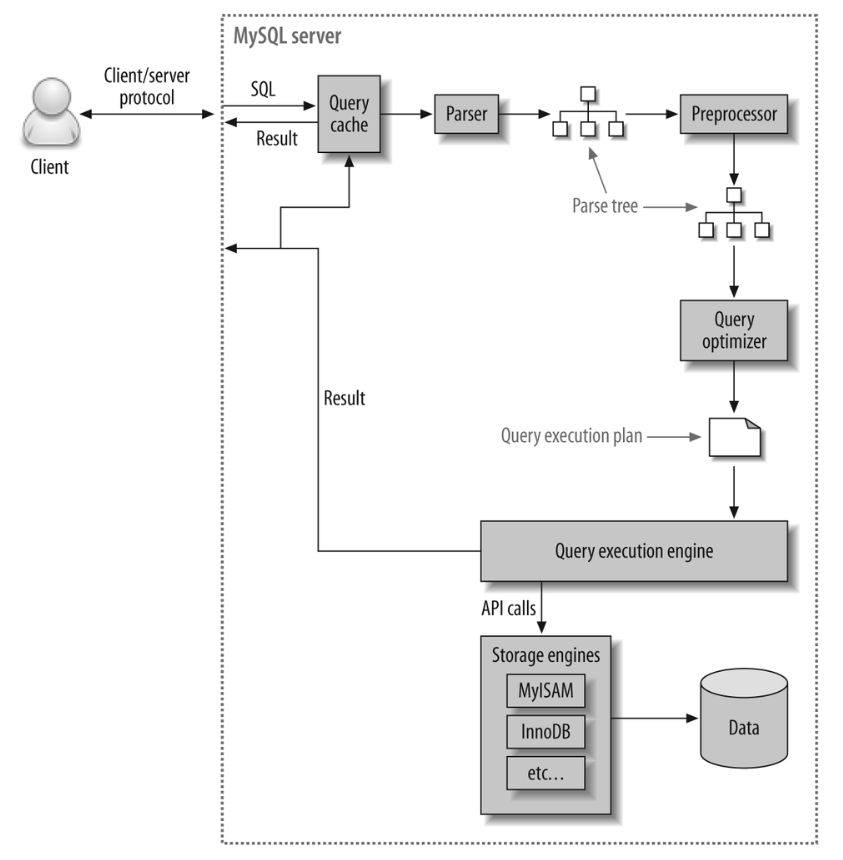
* Keširanje može da bude efikasnije. Mnoge aplikacije keširaju objekte koji se preslikavaju direktno u tabele.
* Za MyISAM tabele, izvršenje jednog upita po tabeli efikasnije koristi zaključavanje tabela: upiti zaključavaju tabele pojedinačno i relativno kratko, umesto zaključavanja svih njih za duže vreme.
* Izvršenje join-a u aplikaciji olakšava skaliranje baze podataka smeštanjem tabela na različitim serverima.
* Sami upiti mogu biti efikasniji.
* Možemo smanjiti reduntatne pristupe redovima. Izvršenje join-a u aplikaciji znači da pribavljamo svaki red samo jednom, dok join u upitu može da pristupa više puta istim podacima. Iz istog razloga, takvo restruktuiranje takođe može da smanji ukupni saobraćaj u mreži i upotrebu memorije.

# Osnove izvršenja upita

Ako želimo da dobijemo visoke performanse od MySQL servera, jedan od najboljih načina je da istražimo kako MySQL izvršava i optimizuje upite. Veliki deo optimizacije upita se sastoji iz razumevanja principa, nakon čega optimizacija postaje veoma logičan proces.

Slika 1 prikazuje kako MySQL izvršava upite. Sa slike možemo videti šta se dešava kada pošaljemo upit MySQL-u:

1. Klijent šalje SQL naredbu serveru.
2. Server proverava keš upita. Ako je pogodak, vraća sačuvane rezultate iz keša, inače, prosleđuje SQL naredbu u sledeći korak.
3. Server parsira, preprocesira i optimizuje SQL u plan izvršenja upita.
4. Mehanizam za izvršenje upita (*eng. query execution engine*) izvršava plan upućivanjem API poziva mehanizmu za skladištenje podataka (*eng. storage engine)*. Najčešći mehanizmni za skladištenje podataka su MyISAM i InnoDB.
5. Server šalje rezultat klijentu.



Slika 1

Svaki od ovih koraka ima dodatnu složenost, koju ćemo objasniti u sledećim odeljcima. Takođe ćemo objasniti i u kojim stanjima će upiti biti za vreme svakog koraka. Proces optimizacije upita je najsloženiji i njemu će biti posvećena najveća pažnja u ovom radu.

## MySQL klijent/server protokol

Klijent/server protokol je polu-dupleks, što znači da u bilo kom trenutku MySQL server može da ili šalje ili prima poruke, ali ne oba.

Ovaj protokol čini MySQL komunikaciju jednostavnom i brzom, ali takođe je i ograničava na neki način. Kao prvo znači da nema kontrole protoka, jednom kada jedna strana šalje poruku, druga strana mora da pribavi celu poruku pre nego što odgovori. Klijent šalje upit serveru kao jedan paket podataka. Jednom kada klijent pošalje upit, može samo da čeka za rezultate.

Suprotno, odgovor od servera se obično sastoji od više paketa podataka. Kada server odgovara, klijent mora da primi ceo skup rezultata. Ako su klijentu potrebna samo prva nekoliko redova koji se vraćaju, on mora da čeka da stignu svi paketi od servera i onda da odbaci one koji mu nisu potrebni. To nije dobra ideja, zbog toga su LIMIT klauzule važne.

Bibliotka koje se povezuju na MySQL pribavljaju ceo skup rezultata i čuvaju ga u memoriju ili da pribavljaju svaki red kako je potreban. Defaultno ponašanje je da pribave ceo rezultat i da ga čuvaju u memoriju. Ovo je važno zato što dok svi redovi budu pribavljeni, MySQL server neće otpustiti ključeve i druge resurse potrebne upitu. Upit će biti u Sending data stanju. Kada klijentska biblioteka pribavlja rezultate sve odjednom, to smanjuje količinu posla koje server radi: server može da završi i očisti upit čim je to moguće. Ovo uglavnom radi dobro, ali nije dobra ideja za ogromne skupove rezultata koji mogu da uzimaju mnogo vremena za pribavljanje i koriste dosta memorije.

## Stanja upita

Svaka MySQL konekcija, ili nit, ima stanje koje pokazuje šta se radi u datom trenutku. Kako upit napreduje kroz svoj životni ciklus, njegovo stanje se menja više puta. Mi ćemo ovde izlistati nekoliko stanja i objasniti šta ona znače:

* Sleep – Nit čeka za novi upit od klijenta.
* Query – Nit ili izvršava upit ili vraća rezultat nazad klijentu.
* Locked – Nit čeka da se ključ tabele odobri na nivou servera. Ključevi koji su implementirani od stane mehanizma za skadištenje, kao što su InnoDB ključevi redova, ne uzrokuju da nit pređe u Locked stanje.
* Analyzing and statistics – Nit proverava statistiku mehanizma za skladištenje i optimizuje upit.
* Copying to tmp table [on disk] – Nit obrađuje upit i kopira rezultate u privremenu tabelu, uglavnom za GROUP BY, za filesort ili za zadovoljenje UNION-a. Ako se stanje završava sa on disk, MySQL prebacuje tabelu iz memorije na disku.
* Sorting result – Nit sortira skup rezultata
* Sending data – Ovo može da označava nekoliko stvari: nit može da šalje podatke između faza upita, generiše skup rezultata ili vraća skup rezultata klijentu.

## Keš upita

Pre parsiranja upita, MySQL proverava keš upita, ako je keš omogućen. Ova operacija predstavlja case-sensitive pretragu keša. Ako se upit razlikuje od sličnog upita u kešu čak samo u jednom bajtu, on se neće podudarati, i obrada upita će preći u sledeću fazu. Ako MySQL pronađe podudaranje u kešu upita, on mora da proveri privilegije pre nego što vrati keširani upit. Ovo je moguće bez parsiranja upita, zato što MySQL čuva i informacije o tabeli sa keširanim upitom. Ako su privilegije uredu, MySQL pribavlja uskladišteni rezultat iz keša upita i šalje ga klijentu, izostavljanjem ostalih faza u izvršenju upita, upit se nikada ne parsira, optimizuje, niti izvršava.

# Proces optimizacije upita

Sledeća faza u životnom ciklusu upita pretvara SQL upit u plan izvršenja za mehanizam izvršenja upita. Ova faza ima nekoliko podfaza: parisranje, preprocesiranje i optimizaciju. Na dalje ćemo opisati svaku fazu posebno, iako se one često kombinuju upotpunosti ili delimično zbog efikasnosti.

## Parser i preprocesor

Na početku, MySQL parser deli upit na tokene i formira stablo parsiranja. Parser koristi MySQL-ovu SQL gramatiku za interpretiranje i validaciju upita. Na primer, on obezbeđuje da su tokeni u upitu validni i u ispravnom redosledu.

Pretprocesor onda proverava rezultujuće stablo parsiranja za dodatne semantike koje parser ne rešava. Na primer, on proverava da tabele i kolone postoje, i on rešava nazive i alijase kako bi obezbedio da reference kolona nisu dvosmislene. Zatim, pretprocesor proverava privilegije. Ovo je veoma brzo ako server nema ogroman broj privilegija.

## Optimizator upita

Slablo parsiranja je sada validno i spremno da ga optimizator pretvori u plan izvršenja upita. Upit može često da se izvršava na različite načine, a da proizvodi isti rezultat. Posao optimizatora je da pronađe najbolju opciju.

MySQL koristi optimizator zasnovan na ceni, što znači da pokušava da predvidi cenu različitih planova izvršenja i da izabere najjeftiniju. Jedinica cene je čitanje jedne nasumične stranice od četiri kilobajta podataka. Procena optimizatora se zasniva na statistici: broj stranica po tabeli ili indeksu, kardinalnosti (broj distinct vrednosti) indeksa, dužini redova ili ključeva i distribuciji ključa. Optimizator ne uključuje efekte bilo kog tipa keširanja u procenama – on procenjuje da će svako čitanje rezultirati I/O operacijama sa diska.

Arhitektura MySQL servera se sastoji iz različitih slojeva. Serverski sloj, koji sadrži optimizator upita, ne skadišti statistiku o podacima i indeksima. To je posao za mehanizme skadištenja i svaki mehanizam skadištenja čuva različite vrste statistike. Pošto server ne čuva statistiku, MySQL optimizator upita mora da traži statistiku o tabelama u upitima od mehnizma za skladištenje. Mehanizmi za skadištenje mogu da pruže optimizatoru statistiku, kao što je broj stanica po tabeli ili indeksu, kardinalnost tabela ili indeksa, dužina redova i ključeva. Ove informacije mogu da pomognu optimizatoru da odluči o najboljem planu izvršenja.

Optimizator može da ne izabere uvek najbolji plan iz mnogih razloga:

* Statistika može da bude pogrešna. Server se oslanja na mehanizme skadištenja podataka kako bi pružio statistiku, a ona može da budu u rasponu od potpuno tačne do potpuno netačne. Na primer, InnoDB mehanizam skladištenja ne održava tačnu statistiku o broju redova u tabeli.
* Metrika troškova nije upotpunosti ista stvarnim troškovima pokretanja upita, tako da čak i kada je statistika tačna, upit može da bude skuplji ili jeftiniji od MySQL-ove procene. Plan koji čita više stranica može da bude jeftiniji u nekim slučajevima, na primer kada su čitanja sekvencijalna tako da je I/O brži, ili kada su stranice već keširane u memoriji.
* Ideja MySQL-a o optimalnom može da se ne poklapa sa našom. Mi želimo najbrže izvršenje, ali MySQL ne razume „brzo“, on razume „cenu“, i kao što vidimo, određivanje cene nije egzaktna nauka.
* MySQL ne uzima u obzir druge upite koji se izvršavaju konkurentno, koji mogu da utiču na brzinu izvrešenja upita.
* MySQL ne radi uvek optimizaciju zasnovanu na troškovima. Ponekad on samo prati pravila, kao što su „ako postoji full-text MATCH() klauzula, koristi FULLTEXT indeks ako postoji“. On će to uraditi čak i kada bi bilo brže koristiti drugačiji indeks.
* Optimizator ne uzima u obzir cenu operacija koje nisu pod njegovom kontrolom, kao što je izvršenje stored funkcija ili korisnički definisanih funkcija.
* Kao što ćemo videti kasnije, optimizator ne može uvek da proceni svaki mogući plan izvršenja, tako da može da propusti optimalan plan.

MySQL optimizator upita je kompleksan deo softvera, i on koristi mnogo optimizacija za transformaciju upita u plan izvršenja. Postoje dva osnovna tipa optimizacija: statičke i dinamičke. Statičke optimizacije mogu da se izvršavaju jednostavnim razmatranjem stabla parsiranja. Na primer, optimizator može da transformiše WHERE klauzulu u ekvivalentan oblik primenom algebarskih pravila. Statičke optimizacije ne zavise od vrednosti, kao što su vrednosti konstante u WHERE klauzuli. One mogu da se izvrše jednom i biće uvek validne, čak i kada se upit ponovo izvrši sa drugačijim vrednostima. Njih možemo smatrati kao „compile-time“ optimizacije.

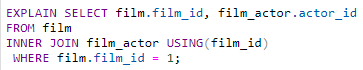
Nasuprot njima, dinamičke optimizacije se zasnivaju na kontekstu i mogu da zavise od mnogih faktora, kao što su vrednosti u WHERE klauzuli. One moraju ponovo da se evaluiraju svaki put kada se upit izvršava. Njih možemo smatrati kao „runtime“ optimizacije.

MySQL može da radi statičke optimizacije jednom, ali dinamičke optimizacije mora iznova da evaluira svaki put kada se izvršava upit. MySQL čak ponekad i ponovo optimizuje upit dok ga izvršava.

Ovo je nekoliko vrsta optimizacija koja MySQL zna kako da odradi:

* Redosled join-a: Tabele ne moraju uvek da se spajaju u redosledu koji je specificiran u upitu. Određivanje najboljeg redosleda spajanja je važna optimizacija, koja će biti kasnije u radu detaljnije objašnjena.
* Konverzija OUTER JOIN-a u INNER JOIN: OUTER JOIN ne mora da se izvršava kao OUTER JOIN. Neki faktori, kao što su WHERE klauzula i šema tabele, mogu da izazovu da OUTER JOIN bude ekvivalentan INNER JOIN-u. MySQL može da prepozna ovo i da ponovo napiše upit.
* Primena pravila algebarskih ekvivalencija: MySQL primenjuje alegebarske transformacije za pojednostavljenje i kanonizaciju izraza. On takođe može da vrši redukciju konstanti, eliminiše nemoguće konstante i konstantne uslove. Na primer, izraz (5=5 AND a>5) će se redukovati na samo a>5. Slično, (a<b AND b=c) AND a=5 postaje b>5 AND b=c AND a=5.
* Optimizacija COUNT(), MIN(), MAX() naredbi: Indeksi i kolone sa null vrednošću mogu često pomoći MySQL-u da optimizuje ove naredbe. Na primer, za pronalaženje minimalne vrednosti kolone koja je krajnje levo u B-Tree indeksu, MySQL može samo da zahteva prvi red u indeksu. On može da radi ovo čak i u fazi optimizacije upita, a da vrednost smatra kao konstantnu za ostatak upita. Slično, da bi pronašao maksimalnu vrednost u B-Tree indeksu, server čita poslednji red. Ako server koristi ovu optimizaciju, možemo videti „Select tables optimized away“ u EXPLAIN planu. Ovo znači da je optimizator uklonio tabelu iz plana upita i zamenio je sa konstantom. Isto tako, COUNT(\*) upiti bez WHERE klauzule često se mogu optimizovati na nekim mehanizmima za skadištenje podataka (kao što je MyISAM, koji čuva tačan broj redova u tabeli).
* Evaluacija i redukcija konstantnih izraza: Kada MySQL detektuje da se izraz može redukovati na konstantu, on će uraditi to za vreme optimizacije. Na primer, korisnički definisana promenljiva može da se konvertuje u konstantu ako se ona ne menja u upitu.

Ovo se može primeniti čak i na konstantno traženje po primarnom ključu ili jedinstvenom indeksu. Ako WHERE klauzula primenjuje kontantni uslov na takav indeks, optimizator zna da može potražiti vrednost na početku upita. Onda će on tu vrednost tretirati kao konstantu u ostatku upita. Takav primer je dat na slici 2. MySQL izvršava upit u dva koraka, što odgovara dvema vrstama u EXPLAIN izlazu prikazanom na slici 3. Prvi korak je pronalaženje željenog reda u film tabeli. MySQL-ov optimizator zna da je to samo jedan red, zato što postoji primarni ključ na kolonu film\_id, i on je već konsultovao indeks tokom faze optimizacije upita da vidi koliko redova će biti pronađeno. Budući da optimizator upita ima poznatu vrednost (vrednost u WHERE klauzuli) koju treba da koristi u pretrazi, ref u tabeli je tipa const. U drugom koraku, MySQL tretira film\_id kolonu iz reda pronađenog u prvom koraku kao poznatu vrednost. On može to da učini zato što optimizator zna da će do trenutka kada upit dođe do drugog koraka znati sve vrednosti iz prvog koraka. Vrednost kolone ref za tabelu film\_actor je tipa const, baš kao što je bilo i tabelu film.

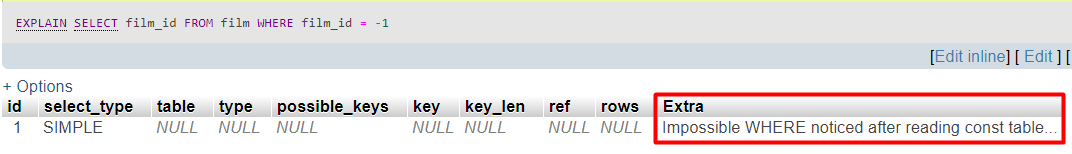


Slika 2



Slika 3

* Korišćenje indeksa – MySQL može ponekad da koristi indeks da izbegne čitanje sirovih podataka kada indeks sadrži sve kolone koje su potrebne upitu.
* Optimizacija podupita – MySQl može da konvertuje neke vrste podupita u efikasnije alternativne oblike, svodeći ih na pretrage indeksa umesto posebnih upita.
* Rani završetak – MySQL može da zaustavi obradu upita (ili koraka u upitu). Očigledan slučaj je LIMIT klauzula, ali postoje i nekoliko drugih tipova ranijeg završetka. Na primer, ako MySQL detektuje nemoguć uslov, može da prekine ceo upit, što možemo videti u sledećem primeru na slici 4:



Slika 4

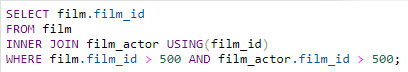
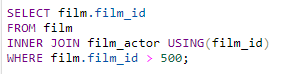
Upit se stopira tokom koraka optimizacije, ali MySQL može ranije da zaustavi izvršenje upita u nekim slučajevima. Server može da koristi ovu optimizaciju kada mehnizam za izvršenje upita prepozna potrebu za pribavljanjem posebnih vrednosti, ili za stopiranje kada vrednost ne postoji. Na primer, sledeći upit dat na slici 5, pronalazi sve filmove bez glumaca



Slika 5

Ovaj upit radi eliminaciju svih filmova koji imaju glumce. Svaki film može da ima više glumaca, ali čim se pronađe jedan glumac, zaustavlja se obrada tekućeg filma i prelazi se na sledećeg zato što se zna da WHERE klauzula zabranjuje prikazivanje tog filma.

* Propagacija jednakosti – MySQL prepoznaje kada upit sadrži dve kolone kao jednake – na primer, u JOIN uslovu – i propagira WHERE klauzule ekvivalentnim kolonama. Na pirmer, kao u upitu prikazanom na slici 6. MySQL zna da se WHERE klauzula primenjuje ne samo na *film* tabelu, nego i na *film\_actor* tabelu, zato što USING klauzula označava spoj te dve tabele po koloni *film\_id*. Ako koristimo drugi server baze podataka koji ne može ovo da prepozna, možemo ručno da specificiramo WHERE klauzulu za obe tabele, kao na slici 7, što čini upite težim za održavanje.



Slika 6 Slika 7

* IN() lista poređenja – U mnogim serverima baze podataka, IN() je samo sinonim za višestruku OR klauzulu, jer su one logički ekvivalentne. MySQL prvo sortira vrednosti u IN() listi, a onda koristi brzo binarno traženje da vidi da li je vrednost u listi. Složenost ovoga je O(logn), gde je n veličina liste, dok je složenost ekvivalentne serije OR klauzula O(n).

## MySQL-ova strategija izvršenja joina-a

MySQL koristi termin join za svaki upit, ne samo za upite koji spajaju redove iz dveju tabela, već za svaki upit (uključujući podupite, pa čak i SELECT za jednu tabelu). Na primer UNION upit MySQL izvršava se kao niz pojedinačnih upita čiji se rezultati spajaju u privremenu tabelu, a zatim ponovo čitaju. Svaki od pojedinačnih upita je join, u MySQL terminologiji.

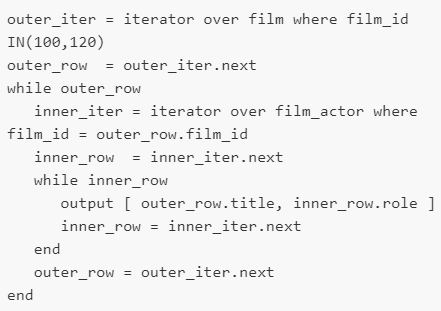
MySQL-ova strategija izvršenja join-a je jednostavna: on svaki join tretira kao “nested-loop join”. Ovo znači da MySQL pokreće petlju da pronađe red iz tabele, a onda pokreće ugnježdenu petlju da pronađe podudaran red u drugoj tabeli. To se nastavlja sve dok ne pronađe podudarni red u svakoj tabeli u join-u. Zatim se gradi i vraća red iz kolona imenovanih u SELECT listi. Onda MySQL pokušava da sagradi sledeći red tražeći više podudarnih redova u zadnjoj tabeli. Ako ih ne pronađe, vraća jednu tabelu unazad i tamo traži više redova. On nastavlja vraćanje sve dok ne pronađe drugi red u nekoj tabeli, u tom trenutku on traži podudarni red u sledećoj tabeli, i tako dalje.

Ovaj proces traženja redova, probijajući se u sledeću tabelu i onda vraćanje, može da bude napisan kao ugnježdena petlja u planu izvršenja – zato se naziva “nested-loop join”. Kao primer, razmotrimo upit na slici 8.



Slika 8

Pretpostavimo da je MySQL odlučio da spoji tabele u redosledu prikazanom u upitu, pseudokod na slici 9 prikazuje kako MySQL izvršava upit.

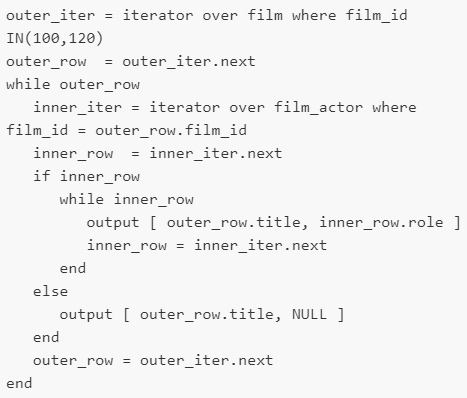


Slika 9

Ovaj plan izvršenja upita se lako primenjuje kako na upitu sa jednom tabelom, tako i na upitu sa više tabela, zbog čega se čak i upit sa jednom tabelom može smatrati join-om – join sa jednom tabelom je osnovna operacija od koje su složeniji join-i sastavljeni. Ovakav plan izvršenja se može primeniti i za OUTER JOIN-e. Ako prethodni primer upita promenimo u OUTER JOIN, kao na slici 10, odgovarajući pseudokod bi bio kao na slici 11.



Slika 10



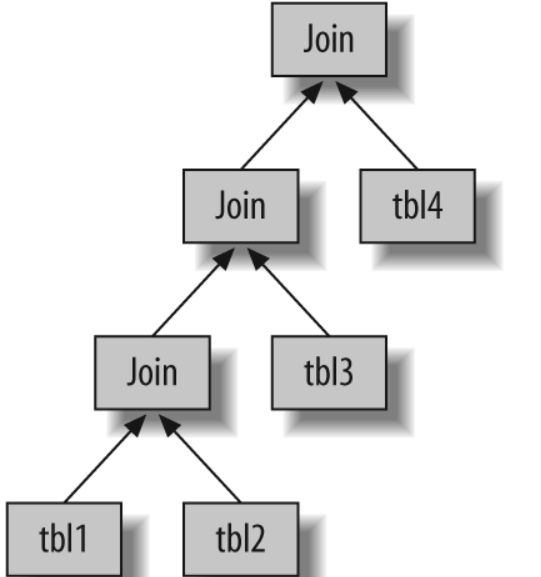
Slika 11

MySQL izvršava svaku vrstu upita na suštinski isti način. Na primer, on hendluje podupit u FROM klauzuli, prvo izvršavanjem, stavljanjem rezultata u privremenu tabelu, i onda tretira tu tabelu kao običnu tabelu, otuda se ona naziva izvedenom tabelom. MySQL izvršava UNION upite takođe sa privremenim tabelama, a sve RIGHT OUTER JOIN upite prepisuje u ekvivalentne LEFT OUTER JOIN upite. Ukratko, MySQL svaku vrstu upita pretvara u plan izvršenja.

Međutim, nije moguće izvršiti svaki SQL upit na ovaj način. Na primer, FULL OUTER JOIN ne može da se izvrši sa ugnježdenim petljama i vraćanjem unazad čim se nađe tabela bez podudarnih redova, zato što on može da počne sa tabelom koja nema podudarnih redova. Ovo objašnjava zašto MySQL ne podržava FULL OUTER JOIN.

## Plan izvršenja

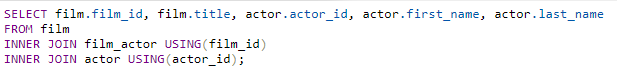
MySQL ne generiše bajt kod za izvršenje upita, kao što proizvode mnoge druge baze podataka. Umesto toga, plan izvršenja upita je zapravo stablo instrukcija koje mehanizam izvršenja upita prati da bi proizveo rezultate upita. Konačni plan sadrži dovoljno informacija za rekonstrukciju originalnog upita. Ako izvršimo EXPLAIN EXTENDED na upit, praćen sa SHOW WARNINGS, videćemo rekonstruisan upit. Bilo koji upit sa više tabela može konceptualno da se predstavi kao stablo, na primer join četiri tabela. Kao što smo objansili u prethodnom odeljku, MySQL uvek počinje sa jednom tabelom i pronalazi podudarne redove u sledećoj tabeli. Tako da MySQL-ovi planovi izvršenja uvek imaju oblik levo-dubokog stabla, a nikada nisu balansirana stabla, što je prikazano na slici 12.



Slika 12

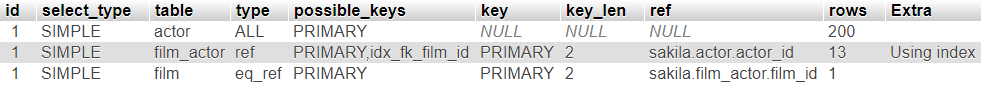
## Optimizacija join-a

Najvažniji deo MySQL optimizacije upita je optimizacija join-a, koja odlučuje o najboljem redosledu izvršenja join-a za upite sa većim brojem tabela. Često je moguće spojiti tabele u nekoliko različitih redosleda i dobiti iste rezultate. Optimizacija join-a procenjuje cene za različite planove i pokušava da izabere najjeftiniji koji daje iste rezultate. Kao primer uzećemo upit sa slike 13 , čije tabele mogu biti spojene u različitim redosledima bez promene rezultata.



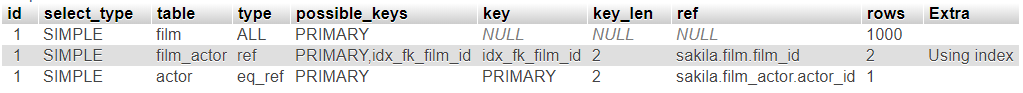
Slika 13

Na primer, MySQL može da počne sa *film* tabelom, da koristi indeks *film*\_*id* u *film*\_*actor* tabeli kako bi pronašao *actor\_id* vrednosti, i onda da traži redove po primarnom ključu tabele *actor*. Ako upotrebimo EXPLAIN možemo da vidimo kako MySQL stvarno izvršava upit, što je prikazano na slici 14. Ovo je sasvim drugačiji plan od onog koji smo predložili. MySQL želi da počne sa *actor* tabelom (mi to znamo zato što je ona izlistana prva u EXPLAIN izlazu) i ide u obrnutom redosledu.



Slika 14

STRAIGHT\_JOIN nalaže da se obrada join-a vrši u redosledu specificiranom u upitu. EXPLAIN izlaz za isti upit kao prethodni sa dodatim STRAIGHT\_JOIN-om, dat je na slici 15.



Slika 15

Ovo pokazuje zašto MySQL želi da okrene redosled join-a: na taj način omogućiće pregled manjeg broja redova u prvoj tabeli. U oba slučaja je moguće izvršiti brzu indeksnu pretragu u drugoj i trećoj tabeli. Razlika je u tome koliko će morati indeksnih pretraga da se uradi. Stavljajući *film* kao prvu tabelu, biće potrebno oko 1000 pretraga u *film*\_*actor* i *actor*, po jedna za svaki red u prvoj tabeli. Ako server skenira prvo *actor* tabelu, moraće da uradi samo 200 pretraga u kasnijim tabelama. Drugim rečima, obrnuti redosled join-a će zahtevati manje čitanja.

Ovo je jednostavan primer kako MySQL optimizator join-a može da preuredi upite da bi njihovo izvršenje učinio jeftinijim.Preuređenje join-a je obično veoma efikasna optimizacija. Postoje situacije kada ona neće rezultirati optimalnim planom, i za takve slučajeve možemo koristiti STRAIGHT\_JOIN i pisati upit u redosledu koji mislimo da je najbolji – ali takve situacije su retke.

Optimizator join-a pokušava da proizvede stablo plana izvršenja upita sa najnižom cenom. On procenjuje sve potencijalne kombinacije podstabla, počevši od svih planova za jednu tabelu. Nažalost, join sa n tabela će imati faktorijel od n kombinacija redosleda join-a za procenjivanje. Ovo se naziva prostor traženja svih mogućih planova upita, i on raste veoma brzo – 10 tabela u join-u mogu da rezultuju u 3628800 različitim načinima. Kada prostor traženja postane previše veliki, može da uzima previše vremena za optimizaciju upita, tako da server zaustavlja rad cele analize. Umesto toga, on prijavljuje skraćenice, kao što je “greedy” pretrage kada broj tabela prevazilazi optimizer\_search\_depth ograničenje.

MySQL ima mnogo heuristika koje se koriste za ubrzanje faze optimizacije, a koje su rezultat višegodišnjeg istraživanja i eksperimentisanja. Ovo može da bude prednost, ali takođe može i da MySQL promaši optimalni plan i izabere neki manje optimalan zato što pokušava da ne procenjuje svaki mogući plan upita.

## Optimizacije sortiranja

Sortiranje rezultata može da bude skupa operacija, tako da performanse možemo poboljšati izbegavanjem sortiranja ili izvršavanjem sortiranja na manjem broju redova.

Kada MySQL ne može da koristi indeks da bi proizveo sortiran rezultat, on mora sam da sortira redove. To može da radi u memoriju ili na disk, ali se ovaj proces uvek naziva filesort, čak iako se zapravo ne koristi fajl.

Ako će se vrednosti koje treba da se sortiraju smestiti u baferu sortiranja, MySQL može da izršava sortiranje upotpunosti u memoriju, koristeći quicksort. Ako MySQL ne može da radi soritranje u memoriju, on izvršava sortiranje na disk, sortirajući vrednosti u segmentima. On koristi quicksort za sortiranje svakog segmenta i onda spaja sortirane segmente u rezultat.

Postoje dva filesort algoritma:

* Dvoprolazi (stari)

Čita pokazivače na redove i ORDER BY kolone, sortira ih, i onda skenira sortiranu listu i ponovo čita redove za izlaz.

Dvoprolazni algoritam može da bude skup, zato što čita redove iz tabele dva puta, a drugo čitanje uzrokuje mnogo nasumičnih I/O. Ovo je posebno skupo za MyISAM, koji koristi sistemski poziv za pribavljanje svakog reda (zato što MyISAM se oslanja na keš operativnog sistema koji drži podatke). S druge strane, on čuva minimlanu količinu podataka tokom sortiranja, tako da ako su redovi koji treba da se sortiraju kompletno u memoriju, jeftinije je čuvati manje podataka i ponovo čitati redove da bi se generisao krajnji izlaz.

* Jednoprolazni (novi)

Čita sve kolone koje su potrebne za upit, sortira ih po ORDER BY kolonama, i onda skenira sortiranu listu i ispisuje specificirane kolone.

Ovaj algoritam je dostupan samo za MySQL 4.1 i novije verzije. On može da bude mnogo efikasniji, posebno za velike tabele, zato što on izbegava čitanje redova iz tabele dva puta. Međutim, on ima potencijal za korišćenje mnogo više prostora, zato što drži sve željene kolone za svaki red, ne samo kolone koje su potrebne za sortiranje redova.

Kada se koristi sortiranje sa JOIN-om, MySQL može da izvršava filesort u dve faze tokom izvršenja upita. Ako ORDER BY klauzula upućuje samo na kolone iz prve tabele u join redosledu, MySQL može da izvršava filesort te tabele, a onda da obrađuje join. Ako se ovo dešava, EXPLAIN pokazuje “Using filesort” u Extra koloni. Inače, MySQL mora da uskladišti rezultate upita u privremenu tabelu, a onda da odradi filesort privremene tabele. U ovom slučaju, EXPLAIN prikazuje “Using temporary Using filesort” u Extra koloni. Ako postoji LIMIT, on se primenljuje nakon filesorta, tako da privremena tabela i filesort mogu da budu veoma veliki.

## Mehanizam izvršenja upita

Parsiranje i faza optimizacije upita rezultiraju planom izvršenja upita, koji MySQL-ov mehanizam izvršenja upita koristi za obradu upita. Ovaj plan je struktura podataka, a ne izvršni bajt kod.

Nasuprot fazi optimizacije, faza izvršenja nije toliko složena. MySQL jednostavno prati instrukcije date u planu izvršenja upita. Mnoge operacije u planu pozivaju metode implementirane od strane interfejsa mehanizma za skladištenje, poznatog kao handler API. Svaka tabela u upitu je predstavljena instancom hendlera. Ako se tabela pojavljuje tri puta u upitu, server kreira tri hendler instance. MySQL kreira i hendler instance ranije u fazi optimizacije. Optimizator ih koristi za dobijanje informacija o tabelama, kao što su nazivi njihovih kolona i statistika indeksa.

## Vraćanje rezultata klijentu

Krajnji korak u izvršenju upita je odgovor klijentu. Čak i upiti koji ne vraćaju rezultat, odgovaraju klijentskoj konekciji sa informacijom o upitu, kao što je na koliko redova su uticali. Ako je upit keširan, MySQL će takođe smestiti rezultat u kešu upita u ovoj fazi. Server generiše i šalje rezultate inkrementalno. Čim MySQL obradi poslednju tabelu i generiše jedan red uspešno, on može da pošalje taj red klijentu. Ovo ima dve prednosti: omogućava serveru da izbegava držanje reda u memoriji i klijent počinje da dobija rezultate čim to bude moguće.

# Ograničenja MySQL optimizatora upita

MySQL-ov pristup za izvršenje upita „everything is a nested-loop join“ nije idealan za optimizaciju svih vrsti upita. Nasreću, postoji samo ograničen broj slučajeva gde MySQL optimizator upita radi loše i obično je moguće efikasnije napisati takve upite.

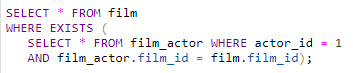
## Korelisani podupiti

MySQL ponekad optimizuje podupite veoma loše. Najgore greške se prave u IN() podupitima u WHERE klauzuli. Kao na primer, kada želimo da pronađemo sve filmove u bazi iz tabele film koji uključuju glumicu „Penelope Guiness“ (actor\_id=1). Prirodno je da se upit napiše sa podupitom, kao na slici 16. Prirodno je misliti da će MySQL izvršiti upit iznutra ka spolja, pronalaženjem liste film\_id vrednosti u IN() klauzuli.



Slika 16

Međutim, dešava se totalno suprotno. MySQL pokušava da pomogne podupitu potiskivajući korelaciju sa spoljnom tabelom u njega. On prepisuje upit tako da izgleda kao na slici 17. Sada, podupit zahteva film\_id iz spoljašnje tabele film i ne može da se izvrši prvi.



Slika 17

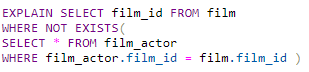
EXPLAIN za prethodni upit pokazuje rezultat kao DEPENDENT SUBQUERY. Prema EXPLAIN izlazu, MySQL će skenirati tabelu film i izvršiti podupit za svaki red koji pronađe. Ovo neće prouzrokovati loše performanse za male tabele, ali ako je spoljašnja tabela veoma velika, performanse će biti ekstremno loše. Ako su performanse loše, upit lako može da se prepiše kao JOIN upit na slici 18.



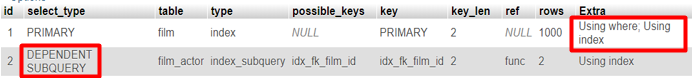
Slika 18

Druga dobra optimizacija je ručno generisanje IN() liste izvršenjem podupita kao posebnog upita sa GROUP\_CONCAT(). Ponekad ovo može biti brže od JOIN-a.

MySQL ne optimizuje uvek loše korelisane podupite. Ponekad je korelisani podupit potpuno odgovarajući, ili čak optimalan način za dobijanje rezultata. Primer koji ćemo razmotriti dat je na slici 19, a na slici 20 je prikazan plan izvršenja.

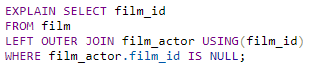


Slika 19



Slika 20

Standardni savet za ovaj upit je pisanje LEFT OUTER JOIN-a umesto korišćenja podupita, što je prikazano na slici 21, a na slici 22 je prikazan plan izvršenja za ovaj upit.



Slika 21



Slika 22

Teorijski, MySQL -ov plan izvršenja će biti suštinski isti u oba slučaja, ali postoje neke razlike:

* Tip SELECT-a nad tabelom film\_actor je DEPENDENT SUBQUERY u jednom upitu, a SIMPLE u drugom. Ova razlika jednostavno odražava sintaksu, zato što prvi upit koristi podupit, a drugi ne.
* Drugi upit nema „Using where“ u Extra koloni za tabelu film. To nije bitno, jer USING klauzula u drugom upitu znači isto kao WHERE.
* Drugi upit kaže „Not exists“ u Extra koloni film\_actor tabeli. Ovo je primer ranog završetka algoritma koji smo spominjali ranije. To znači da MySQL koristi not-exists optimizaciju za izbegavanje čitanja više od jednog reda u indeksu idx\_fk\_film\_id tabele film\_actor. Ovo je ekvivalentno NOT EXISTS() korelisanom podupitu, zato što zaustavlja obradu tekućeg reda čim pronađe poklapanje.

Teorijski, MySQL će izvršiti upite skoro identično. U stvarnosti, benčmarking je jedini način da vidimo koji pristup je stvarno brži.

## Ograničenja UNION-a

MySQL ponekad ne može da potiskuje uslove od spolja UNION-a ka unutrašnjosti, gde bi se oni mogli koristiti za ograničavanje rezultata ili omogućavanje dodatnih optimizacija. Ako želimo da svaki od pojedinačnih upita unutar UNION-a koristiti prednost LIMIT-a, treba da stavimo LIMIT klauzule unutar svakog dela UNION-a.Na primer, ako u UNION stavimo zajedno dve ogromne tabele i ograničimo rezultat na prva 20 reda, MySQL će obe ogromne tabele uskladištiti u privremenu tabelu i onda pribaviti samo 20 redova iz nje. Možemo izbeći ovo dodavanjem LIMIT 20 svakom upitu unutar UNION-a.

## Optimizacija spajanja indeksa

Algoritmi spajanja indeksa, uvedeni u MySQL 5.0, omogućavaju MySQL-u da koristi više od jednog indeksa po tabeli u upitu. Ranije verzije MySQL-a mogle su da koriste samo jedan indeks, pa kada ni jedan indeks nije dovoljno dobar da pomogne restrikcijama u WHERE klauzuli, MySQL je često birao skeniranje tabele. Na primer, film\_actor tabela ima indeks film\_id i indeks actor\_id i oba se koriste u WHERE uslovu u upitu na slici 23.



Slika 23

U starijim verzijama MySQL-a, ovaj upit bi proizveo skeniranje tabele ako se ne napiše kao UNION dva upita. U MySQL 5.0 i novijim verzijama, upit može da koristi oba indeksa, skenira ih istovremeno i spaja rezultate. Postoje tri varijacije algoritma: unija za OR uslove, presek za AND uslove, i unije preseka za kombinaciju prethodna dva. Prethodni upit koristi uniju dva skeniranja indeksa, što može da se vidi u Extra koloni na slici 24.



Slika 24

MySQL može da koristi ovu tehniku nad složenim WHERE klauzulama, tako da možemo videti ugnježdene operacije u Extra koloni za neke upite. Ovo često dobro radi, ali ponekad operacije baferovanja, sortiranja i spajanja koriste dosta CPU i memorijske resurse. Ovo se najčešće dešava kada nisu svi indeksi mnogo selektivni, tako da paralelna skeniranja vraćaju mnogo redova operaciji za spajanje.

## Paralelno izvršenje

MySQL ne može da izvršava jedan upit paralelno na više CPU-ova. Ovu karakteristiku nude drugi serveri baze podataka, ali ne i MySQL.

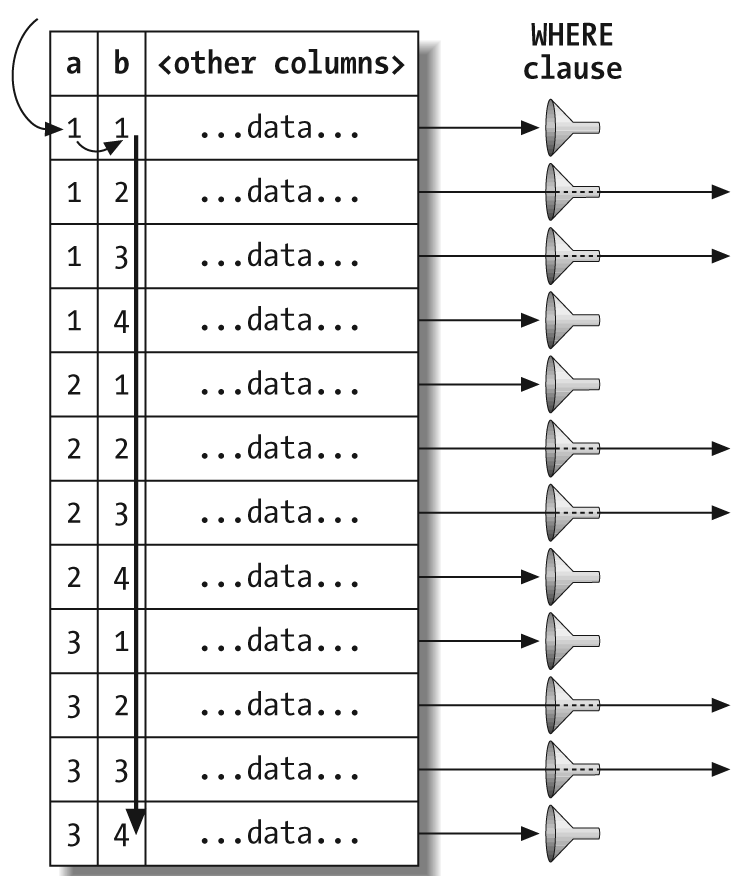
## Labavo skeniranje indeksa

MySQL-ova skeniranja indeksa uopšteno zahtevaju definisanu početnu tačku i definisanu krajnju tačku u indeksu, čak iako je samo nekoliko nesusednih redova u sredini potrebno za upit. MySQL će skenirati ceo opseg redova unutar tih krajnjih tačaka.

Pretpostavimo da imamo tabelu sa indeksom na kolone (a,b), i želimo da izvršimo upit sa slike 25. Pošto indeks počinje kolonom a, a WHERE klauzula u upitu ne specificira kolonu a, MySQL će uraditi skeniranje tabele i eliminisaće redove koji se ne podudaraju sa WHERE klauzulom, kao na slici 26.

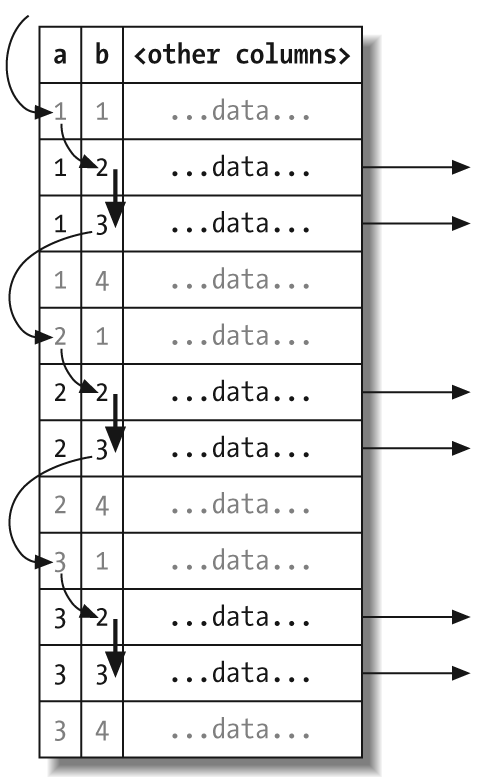


Slika 25



Slika 26

Postoji brži način za izvršenje ovog upita – labavo skeniranje indeksa. Indeksna struktura omogućava traženje od početka svakog opsega vrednosti, skeniranje sve do kraja opsega, i onda skakanje na početak sledećeg opsega. Na slici 27 je prikazano kako bi ova strategija izgledala kada bi MySQL mogao da radi to. Dakle, MySQL još uvek ne radi labavo skeniranje indeksa, koje bi bilo efikasnije, već skenira celu tabelu kako bi pronašao redove.



Slika 27

Na početku MySQL 5.0, labavo skeniranje indeksa je moguće u određenim ograničenim okolnostima, kao što su upiti koji pronalaze maksimalne ili minimalne vrednosti u GROUP BY, kao što je primer na slici 28. “Using index for group-by” informacija u EXPLAIN planu na slici 29 upućuje na labavo skeniranje indeksa. Ovo je dobra optimizacija za posebne namene, ali labavo skeniranje indeksa se idalje ne koristi za opšte namene.



Slika 28



Slika 29

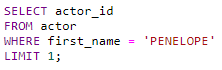
## MIN() I MAX()

MySQL ne optimizuje dobro određene MIN() I MAX() upite.

Kao primer uzmećemo upit kao na slici 30.Pošto ne postoji indeks na first\_name, upit izvršava skeniranje tabele. Ako MySQL skenira primarni ključ, on može teorijski da se zaustavi nakon čitanja prvog poklapanja reda, zato što je primarni ključ striktno rastući i svaki sledeći red će imati veću vrednost actor\_id. Međutim, u ovom slučaju, MySQL će skenirati celu tabelu. Rešenje je ukloniti MIN() i napisati upit sa LIMIT, tako da izgleda kao na slici 31.



Slika 30

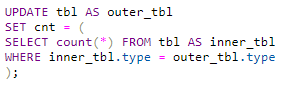


Slika 31

Svrha SQL-a je da mi treba da kažemo serveru šta želimo, a on treba da smisli kako da pribavi te podatke, dok u ovom slučaju, mi kažemo MySQL-u kako da izvrši upit i iz upita nije jasno da je ono što tražimo minimalna vrednost. Ovim se promašuje svrha SQL-a, ali se dobijaju visoke performanse.

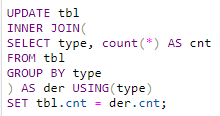
## SELECT i UPDATE nad istom tabelom

MySQL ne dozvoljava SELECT iz jedne tabele dok se istovremeno izvršava UPDATE na toj istoj. Na slici 32 je prikazan primer upita koji je onemogućen. Upit ažurira svaki red brojem sličnih redova u tabeli.



Slika 32

Da bi zaobišli ovo ograničenje, možemo koristiti izvedenu tabelu, jer je MySQL doživljava kao privremenu tabelu, što je prikazano na slici 33. Ovim se efikasno izvršavaju dva upita: jedan SELECT unutar podupita, i jedan višestruki UPDATE sa spojenim rezultatima iz podupita. Podupit otvara i zatvara tabelu pre nego što spoljašnji upit otvara tabelu, tako da će upit sada biti uspešan.



Slika 33

# Optimizacija posebnih vrsta upita

U ovom odeljku, daćemo savete kako optimizovati određene vrste upita.

## Optimizacija COUNT() upita

COUNT() je specijalna funkcija koja radi na dva veoma različita načina: prebrojava vrednosti i prebrojava redove. Vrednost je ne null izraz. Ako se specificira naziv kolone ili drugi izraz unutar zagrada, COUNT() broji koliko puta taj izraz ima vrednost. Drugi oblik COUNT() jednostavno broji redove u rezultatu. To radi i MySQL kada zna da izraz unutar zagrada nikada nije NULL. Najčešći primer je COUNT(\*), koji je posebni oblik COUNT() gde \* označava celu listu kolona u tabeli. Jedna od uobičajenih greški je specificiranje naziva kolona unutar zagrada kada želimo da prebrojimo redove. Kada želimo da znamo broj redova u rezultatu, treba uvek da koristimo COUNT(\*).

Uobičajena zabluda je da MyISAM ekstremno brzo izvršava COUNT() upite. On je brz, ali samo za poseban slučaj: COUNT(\*) bez WHERE klauzule, koji samo broji redove u celoj tabeli. MySQL može da optimizuje ovo zato što MyISAM mehanizam skladištenja uvek zna koliko je redova u tabeli. Ako MySQL zna da col ne može da bude NULL, on takođe može i da optimizuje COUNT(col) izraz tako što ga interno konvertuje u COUNT(\*).

Primer na slici 34, pokazuje kako možemo da pronađemo broj filmova čiji je film\_id veći od 5. Ako prikažemo ovaj upit sa SHOW STATUS, videćemo da je skenirano 1000 redova.



Slika 34

Ako poništimo uslov iz prethodnog primera i od ukupnog broja filmova oduzmemo broj filmova čiji je film\_id manji ili jednak od 5, možemo smanjiti broj redova koji se skeniraju na 5. Ova verzija upita je prikazana na slici 35. Ovaj upit čita manje redova zato što se podupit pretvara u konstantu tokom faze optimizacije upita.



Slika 35

## Optimizacija JOIN upita

Optimizacija JOIN upita je već objašnjena ranije u radu, ovde je dato samo par najvažnijih napomena kojih se treba pridržavati prilikom pisanja ovakvih upita kako bi dobili dobre performanse:

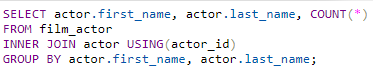
* Treba da budemo sigurni da postoje indeksi nad kolonama u ON ili USING klauzuli. Treba da razmotrimo redosled join-a kada dodajemo indekse, Ako spajamo tabele A i B po koloni c i optimizator upita odluči da spoji tabele u redosledu B,A, ne treba da indeksiramo kolonu iz tabele B. Nekoriščeni indeksi su dodatno opterećenje. Uopšteno, treba da dodamo indekse samo na drugu tabelu u redosledu spoja, ako nam nisu potrebni iz nekog drugog razloga.
* Treba da pokušamo da obezbedimo da izrazi GROUP BY ili ORDER BY upućuju samo na kolone iz iste tabele, tako da MySQL može da koristi indeks za ove operacije.
* Budimo pažljivi kada koristimo različite verzije MySQL, zato što su se sintkasa join-a i prioritet operatora menjali.

## Optimizacija GROUP BY i DISTINCT

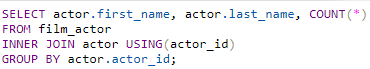
MySQL optimizuje ove dve vrste upita slično u mnogim slučajevima. Oba tipa upita koriste prednosti indeksa i to je najvažniji način njihove optimizacije.

MySQL ima dve vrste strategija za optimizaciju GROUP BY kada ne može da koristi indeks: može da koristi privremenu tabelu ili filesort za izvršenje grupisanja. Koja od ove dve strategije je efikasnija zavisi od samog upita. Možemo da nateramo optimizator da izabere jedan ili drugi metod sa SQL\_BIG\_RESULT I SQL\_SMALL\_RESULT.

Ako treba da grupišemo join po vrednosti koja dolazi iz pregledavajuće tabele, obično je efikasnije grupisati po identifikatoru pregledavajuće tabele nego po vrednosti. Na primer, upit na slici 36nije efikasan, dok upit napisan na slici 37 efikasniji. Ovaj upit koristi činjenicu da ime i prezime glumca zavisi od actor\_id, tako da će vratiti iste rezultate, ali to nije uvek slučaj da možemo odmah odabrati negrupisane kolone i dobiti isti rezultat.



Slika 36



Slika 37

Loša je ideja selektovati negrupisane kolone u upitu grupisanja, zato što će rezultati biti nedeterminisani i mogu lako da se promene ako promenimo indeks ili optimizator odluči da koristi drugu strategiju. Za ovakve upite server neće javiti grešku, a nisu dizajnirani na taj način u svrhu optimizacije. Bolje je da postavimo konfiguracionu promenljivu servera SQL\_MODE na ONLY\_FULL\_GROUP\_BY tako da javlja grešku umesto da dopušta pisanje loših upita.

## Optimizacija LIMIT i OFFSET

Upiti sa LIMIT i OFFSET su uobičajeni u sistemima koji rade straničenje i skoro uvek idu zajedno sa ORDER BY klauzulom. Korisno je imati indeks koji podržava sortiranje, inače će server morati da radi dosta filesorta.

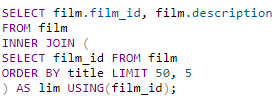
Učestali problem je visoka vrednost za pomeraj. Ako upit izgleda kao: LIMIT 10000, 20, biće generisano 10020 redova i odbaciće se prvih 10000 redova, što je veoma skupo.

Jedna jednostavna tehnika za poboljšanje efikasnosti je da primenimo pomeraj na indeks, a ne na celu tabelu, što je prikazano na slici 38.



Slika 38

Ako je tabela veoma velika, ovaj upit je bolje napisati kao na slici 39. Ovo omogućava serveru da pregleda što je moguće manje podataka u indeksu bez pristupa redovima, i zatim, nakon što nađe željene redove, spoji sa celom tabelom da bi se pribavile ostale kolone iz reda. Slična tehnika se primenjuje i u join-ima sa LIMIT klauzulama.



Slika 39

## Optimizacija SQL\_CALC\_FOUND\_ROWS

Druga uobičajena tehnika za prikazivanje straničenja je dodavanje SQL\_CALC\_FOUND\_ROWS upitu sa LIMIT, tako da možemo znati koliko redova bi bilo vraćeno bez LIMIT. Ova opcija samo kaže serveru da generiše i odbaci ostatak skupa rezultata, umesto da se zaustavlja tek kada dostigne željeni broj redova. Ovo je veoma skupo.

Boljie rešenje je konverzija straničenja u “next link”. Pod pretpostavkom da imamo 20 rezultata po stranici, upit bi trebalo da onda koristi LIMIT od 21 i da prikaže samo 20 redova. Ako 21. red postoji u rezultatu, on je na sledećoj stranici, a na njoj možemo da se prebacite pomoću “next” linka.

Druga mogućnost je da se pribavi i kešira mnogo više redova nego što nam treba – recimo 1000 – a da onda uzimamo iz keša za susedne stranice. Ova strategija omogućava aplikaciji da zna koliko je veliki ceo skup rezultata. Ako je manji od 1000 redova, aplikacija zna koliko linkova za stranice da renderuje. Ako je veći, aplikacija može samo da prikaže “više od 1000 rezultata je pronađeno”. Obe strategije su mnogo efikasnije od ponavljanja generisanja celog rezultata i odbacivanja većine njih.

## Optimizacija UNION-a

MySQL uvek izvršava UNION upite kreiranjem privremene tabele i njenim popunjavanjem rezultatima UNION-a. MySQL ne može da primeni mnogo optimizacija na UNION upite, kao na ostale. Možemo da pomognemo optimizatoru ručnim stavljanjem WHERE, LIMIT, ORDER BY i drugih klauzula u svakom SELECT-u u UNION-u.

Važno je uvek koristiti UNION ALL, ako ne treba da server eliminiše duple redove. Ako izostavimo ključnu reč ALL, MySQL dodaje distinct opciju privremenoj tabeli koja koristi ceo red za utvrđivanje jedinstvenosti. Ovo je skupo. Ključna reč ALL ne eliminiše privremenu tabelu. MySQL uvek stavlja rezultate u privremenu tabelu a zatim ih čita ponovo, čak i kada to nije stvarno potrebno.

# Opcije optimizatora upita

MySQL ima nekoliko opcija za optimizator koje možemo koristiti za upravljanje planom upita ako nismo zadovoljni optimizatorovim izborom. Odgovarajuća opcija se stavlja u upit čiji plan želimo da modifikujemo. U nastavku je data lista identifikatora opcija sa objašnjenjima kada ih je dobro koristiti. Opcije su:

* HIGH\_PRIORITY i LOW\_PRIORITY – Ove opcije govore MySQL-u kako da dodeli prioritet naredbi u odnosu na druge naredbe koje pokušavaju da pristupe istim tabelama.

HIGH\_PRIORITY govori MySQL-u da rasporedi SELECT naredbu pre drugih naredbi koje mogu da čekaju za ključeve, kako bi mogle da modifikuju podatke. Ona govori SELECT-u da izađe na početak reda, umesto da čeka red. Ovaj modifikator može da se primeni i za INSERT, čime jednostavno poništava efekat globalne postavke servera LOW\_PRIORITY.

LOW\_PRIORITY je obrnuta opcija: ona označava da naredba čeka na samom kraju reda ako postoje neke druge naredbe koje žele da pristupe tabelama. Ova opcija se može primeniti za SELECT, INSERT, UPDATE, REPLACE i DELETE naredbe.

Ove opcije su efikasne za mehanizme skladištenja sa zaključavanjem na nivou tabele i nikada ih ne treba primenjivati na InnoDB ili druge mehanizme skladištenja sa sitnim zaključavanjem i konkurentnim upravljanjem. Treba biti oprezan u upotrebi ovih opcija kod MyISAM-a, zato što mogu da onemoguće konkurente inserte i značajno da smanje performanse. Ove opcije samo utiču na to kako server stavlja u red naredbe koje čekaju da pristupe tabeli.

* DELAYED – Ova opcija se koristi za INSERT i REPLACE. Ona omogućuje da naredba, na koju se primenjuje, odmah vrati i postavi dodate redove u bafer koji će biti ubačen kada tabela bude slobodna. Ovo je najkorisnije za prijavljivanje i slične primene gde želimo da dodamo mnogo redova, a da klijent ne čeka, i bez prouzrokovanja I/O za svaku naredbu. Postoje mnoga ograničenja, kao što je da zakašnjena dodavanja nisu implementirana u svim mehanizmima za skladištenje.
* STRAIGHT\_JOIN – Ova opcija može da se javi ili nakon ključne reči SELECT u SELECT naredbi ili u bilo kojoj naredbi između dve spojene tabele. Prva upotreba forsira da se sve tabele u upitu spajaju u redosledu u kome su navedene u naredbi. Druga upotreba forsira spajanje dve tabele između kojih se pojavljuje opcija.

STRAIGHT\_JOIN opcija je korisna kada MySQL ne bira dobar redosled spajanja ili kada je optimizatoru potrebno dosta vremena da donese odluku o redosledu spajanja. U poslednjem slučaju, dodavanje ove opcije će smanjiti prostor traženja za optimizator.

* SQL\_SMALL\_RESULT i SQL\_BIG\_RESULT – Ove opcije su za SELECT naredbe. One kažu optimizatoru kako i kada se koriste privremene tabele i sortira u GROUP BY ili DISTINCT upitima. SQL\_SMALL\_RESULT govori optimizatoru da će rezultujući skup biti mali i može da se stavi u indeksirane privremene tabele da bi se izbeglo sortiranje za grupisanje, dok SQL\_BIG\_RESULT označava da će rezultat biti veliki i da će biti bolje koristiti privremene tabele na disku za sortiranje.
* SQL\_BUFFER\_RESULT – Ova opcija govori optimizatoru da stavi rezultate u privremenu tabelu i odključa tabele čim je to moguće. Ovo se razlikuje od baferovanja na klijentskoj strani. Baferovanje na serverskoj strani može da bude korisno kada se ne koristi baferovanje na klijentu, jer omogućava da se izbegne potrošnja memorije na klijentu i brzo otpuste ključevi.
* SQL\_CACHE i SQL\_NO\_CACHE – Ove opcije govore serveru da li upit jeste ili nije kandidat za keširanje u kešu upita.
* SQL\_CALC\_FOUND\_ROWS – Ova opcija govori MySQL-u da izračuna ceo skup rezultata kada postoji LIMIT klauzula, čak iako on vraća samo LIMIT redove.
* FOR UPDATE i LOCK IN SHARE MODE – Ove opcije kontrolišu zaključavanje za SELECT naredbe, ali samo za mehanizme skladištenja koji imaju zaključavanje na nivou reda. One omogućavaju zaključavanje povezanih redova. Može biti korisno kada želimo da zaključamo redove za koje znamo da će biti ažurirani kasnije ili kada želimo da izbegnemo eskalaciju zaključavanja i samo što pre dobijemo ekskluzivno zaključavanje.

Ove opcije nisu potrebne za INSERT ... SELECT upite, koji stavljaju ključeve za čitanje na izvornim redovima po defaultu.

Ove opcije podržava InnoDB i kada ih koristimo treba da budemo svesni da one mogu da onemoguće neke optimizacije, kao što je pokrivanje indeksa. InnoDB ne može da zaključa redove ekskluzivno bez pristupa primarnom ključu, gde se čuvaju informacije o verziji reda.

* USE INDEX, IGNORE INDEX i FORCE INDEX – Ove opcije govore optimizatoru koje indekse da koristi ili ignoriše za pronalaženje redova u tabeli.

FORCE INDEX je isti kao USE INDEX, ali on govori optimizatoru da je skeniranje tabele ekstremno skupo u poređenju sa indeksom. Možemo koristiti ove opcije kada mislimo da optimizator neće izabrati indeks ili kada želimo da dobijemo prednost indeksa iz nekog razloga.

U MySQL 5.0 i novijim verzijama, postoje neke sistemske promenljive koje utiču na optimizator:

* optimizer\_search\_depth – Ova promenljiva govori optimizatoru koliko iscrpno da ispituje parcijalne planove. Ako upiti uzimaju mnogo vremena u „Statistics“ stanju, možemo smanjiti ovu vrednost.
* opimizer\_prune\_level – Ova promenljiva, koja je po defaultu omogučena, omogućava da optimizator preskoči određene planove u zavisnosti od broja procenjenih redova.

Obe ove opcije upravljaju prečicama optimizatora. Ove prečice su značajne za dobre performanse složenih upita, ali mogu da prouzrokuju da server promaši optimalne planove.

# Zaključak

Cilj ovog rada je bio da pokažemo kako MySQL izvršava upite sa posebnim naglaskom na optimizatoru upita kao najsloženijem delu. Što se tiče optimizatora upita, razmotrili smo optimizaciju raznih vrsti upita, ograničenja optimizatora i dali primere dobre prase. Iz ovog rada možemo zaključiti da koliko god optimizator pametan bio, postoje slučajevi kada on ne daje najbolje rezultate. Ponekad možemo znati nešto o podacima što optimizator ne zna, kao što je činjenica da nešto mora biti tačno zbog aplikacione logike. Ako znamo da optimizator ne daje najbolje rezultate, a znamo zašto, možemo da mu pomognemo. Neki od načina da utičemo na optimizator su da dodamo nagoveštaj u upitu, da ponovo napišemo upit, redizajniramo šemu baze podataka ili da dodamo indekse.

# Literatura

<https://www.oreilly.com/library/view/high-performance-mysql/9780596101718/ch04.html>

<https://dev.mysql.com/doc/refman/8.0/en/optimization.html>