Univerzitet u Nišu

Elektronski fakultet



Sistemi za upravljanje bazama podataka

Optimizacija upita kod MySql-a

- Seminarski rad -

Mentor: Prof. Aleksandar Stanimirović Kandidat: Pavle Stojanović 1148

April 2021.

Sadržaj

[Životni ciklus SQL upita 2](#_Toc69032582)

[Da li od baze podataka tražite podatke koji vam nisu potrebni? 2](#_Toc69032583)

[Preuzimanje više redova nego što je potrebno 3](#_Toc69032584)

[Preuzimanje svih kolona iz spoja više tabela 3](#_Toc69032585)

[Preuzimanje svih kolona 3](#_Toc69032586)

[Da li MySQL obrađuje previše podataka? 4](#_Toc69032587)

[Vreme izvršenja 4](#_Toc69032588)

[Obrađeni i vraćeni redovi 4](#_Toc69032589)

[Pregledani redovi i tipovi pristupa 5](#_Toc69032590)

[Transformisanje upita 7](#_Toc69032591)

[Kompleksan upiti ili više jednostavnijih upita? 7](#_Toc69032592)

[Usitnjavanje upita 8](#_Toc69032593)

[Dekompozicija spojeva 8](#_Toc69032594)

[Osnove izvršenja upita 9](#_Toc69032595)

[MySQL Client/Server protocol 10](#_Toc69032596)

[Proces optimizacije upita 12](#_Toc69032597)

[Parser i pretprocesor 12](#_Toc69032598)

[Optimizator upita 12](#_Toc69032599)

[Preuređivanje spojeva 13](#_Toc69032600)

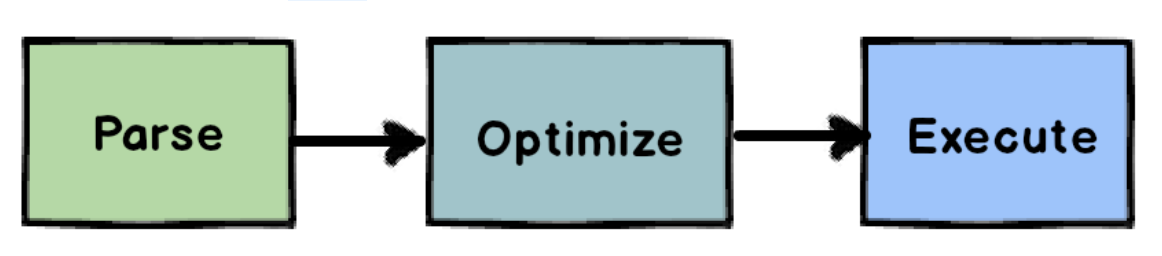
[Pretvaranje OUTER JOIN-ova u INNER JOIN-ove 13](#_Toc69032601)

[COUNT (), MIN () i MAX () optimizacija 13](#_Toc69032602)

[Evaluacija i smanjivanje konstantnih izraza 14](#_Toc69032603)

# Životni ciklus SQL upita

Sql upit prilikom izvršenja prolazi kroz sledeće korake:



* Analizator upita osigurava da je upit sintaksno i semantički ispravan i vraća greške ukoliko nije. Ukoliko je ispravan, pretvara ga u algebarski izraz i prosleđuje u sledeći korak.
* Planer i optimizator upita obavlja ključan posao. Prvo izvodi direktne optimizacije (poboljšanja koja uvek rezultiraju boljim performansama, poput pojednostavljivanja 5 \* 10 na 50). Zatim razmatra različite „planove upita“ koji mogu imati različite optimizacije, procenjuje troškove (CPU i vreme) svakog plana upita na osnovu broja redova u odgovarajućim tabelama, zatim bira optimalni plan i prenosi ga na sledeći korak.
* Izvršitelj upita uzima plan i pretvara ga u operacije za bazu podataka, vraćajući nam rezultate ako ih ima.

Najosnovniji razlog zbog kojeg upit ne funkcioniše dobro je taj što radi sa previše podataka. Neki upiti jednostavno moraju da prođu kroz puno podataka i ne može im se pomoći. To je retko. Većina loših upita se može promeniti kako bi se pristupilo manjoj količini podataka. Možemo kroz sledeća dva koraka da analiziramo upit sa lošim performansama:

1. Da li vaša aplikacija preuzima više podataka nego što je potrebno. To obično znači da pristupa prevelikom broju redova, ali možda i prevelikom broju kolona.
2. Saznajte da li Sql server analizira više redova nego što mu je potrebno.

# Da li od baze podataka tražite podatke koji vam nisu potrebni?

Neki upiti traže više podataka nego što im je potrebno, a zatim ih bacite. Ovo zahteva dodatni rad MySql servera, dodaje mrežne troškove i troši memoriju i CPU resurse na aplikacijskom serveru.

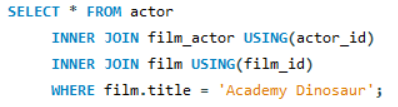
Evo nekoliko tipičnih grešaka:

## Preuzimanje više redova nego što je potrebno

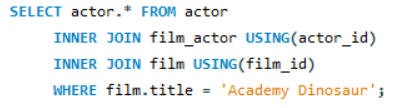
Jedna od uobičajenih grešaka je pretpostavka da MySql daje rezultate na zahtev, umesto da izračunava i vraća puni skup rezultata. To često vidimo u aplikacijama koje su dizajnirali ljudi upoznati sa drugim sistemima baza podataka. Ovi programeri su navikli na tehnike kao što su izdavanje SELECT izraza koji vraća mnogo redova, zatim pribavljanje prvih N redova i zatvaranje skupa rezultata (npr. Pribavljanje 100 najnovijih članaka za veb stranicu kada treba da prikažu samo 10 od njih na naslovnoj strani). Misle da će im MySQL pružiti ovih 10 redova i zaustaviti izvršavanje upita, ali ono što MySQL zaista radi je da generiše kompletan skup rezultata. Klijentska biblioteka tada preuzima sve podatke i odbacuje većinu. Najbolje rešenje je dodavanje ograničenja u upitu.

## Preuzimanje svih kolona iz spoja više tabela

Ako želite da preuzmete sve glumce iz filma 'Academy Dinosaur', ne pišite upit na sledeći način:



To vraća sve kolone iz sve tri tabele. Umesto toga, napišite upit na sledeći način:



Ovaj upit vraća samo one podatke koji Vam trebaju tj. podatke vezane za glumce.

## Preuzimanje svih kolona

Uvek treba biti sumnjičav kada vidite SELECT \*. Da li su Vam zaista potrebne sve kolone? Verovatno ne. Preuzimanje svih kolona može sprečiti optimizacije, kao i dodati I / O, memorijske i CPU troškove.

Neki DBA-ovi zabranjuju SELECT \* zbog ove činjenice i da bi smanjili rizik od problema kada neko promeni listu kolona tabele.

Naravno, traženje više podataka nego što vam zaista treba nije uvek loše. U mnogim slučajevima, ljudi kažu da rasipnički pristup pojednostavljuje razvoj, jer programeru omogućava da koristi isti deo koda na više mesta. To je razumno razmišljanje, ukoliko znate koliko košta u smislu performansi. Takođe, može biti korisno preuzeti više podataka nego što vam je stvarno potrebno ako u aplikaciji koristite neku vrstu keširanja ili ako imate na umu drugu korist. Preuzimanje i keširanje punih objekata može biti poželjnije od pokretanja mnogih odvojenih upita koji preuzimaju samo delove objekta.

# Da li MySQL obrađuje previše podataka?

Kada se uverite da vaši upiti preuzimaju samo podatke koji su vam potrebni, možete potražiti upite koji obrađuju previše podataka dok generišu rezultate.

U MySQL-u su najjednostavnije metrike troškova upita:

1. Vreme izvršenja
2. Broj pregledanih redova
3. Broj vraćenih redova

Nijedna od ovih metrika nije savršen način za merenje troškova upita, ali oni približno pokazuju količinu podataka kojoj MySQL mora interno da obradi da bi izvršio upit. Sve tri metrike su evidentirane u evidenciji sporih upita, pa je pregled ove evidencije jedan od najboljih načina za pronalaženje upita koji obrađuju previše podataka.

## Vreme izvršenja

Standardna funkcija evidentiranja sporog upita u MySQL-u ima ozbiljna ograničenja. Srećom, postoje zakrpe koje vam omogućavaju da evidentirate i merite spore upite u mikrosekundama. Oni su uključeni u MySQL 5.1 server, ali po potrebi možete i da zakrpite starije verzije. Pazite da previše ne naglašavate vreme izvršavanja upita. Lepo je pogledati jer je to objektivna metrika, ali nije konzistentna u različitim uslovima opterećenja. Ostali faktori - kao što je lock mehanizam(lock tabele i lock redova), velika konkurentnost i hardver, takođe mogu imati značajan uticaj na vreme izvršavanja upita. Ova metrika je korisna za pronalaženje upita koji najviše utiču na vreme odziva aplikacije ili najviše opterećuju server, ali vam ne govori da li je stvarno vreme izvršenja razumno za upit date složenosti.

## Obrađeni i vraćeni redovi

Korisno je razmisliti o broju obrađenih redova prilikom analiziranja upita, jer možete videti koliko efikasno upiti pronalaze potrebne podatke.

Međutim, kao i vreme izvršenja, to nije savršena metrika za pronalaženje loših upita. Nisu svi pristupi redovima jednaki. Pristup kraćim redovima je brže, a preuzimanje redova iz memorije je mnogo brže od čitanja sa diska.

U idealnom slučaju, broj pregledanih redova bio bi isti kao i broj vraćenih, ali u praksi je to retko moguće. Na primer, pri generisanju redova spojem, mora se pristupiti većem broju redova da bi se generisao svaki red u skupu rezultata. Odnos pregledanih i vraćenih redova je obično mali - recimo, između 1: 1 i 10: 1 - ali ponekad može biti i za red veličine veći.

## Pregledani redovi i tipovi pristupa

Kada razmišljate o ceni upita, razmislite o ceni pronalaska jednog reda u tabeli. MySQL može da koristi nekoliko metoda pristupa za pronalaženje i vraćanje reda. Neki zahtevaju ispitivanje mnogih redova, ali drugi mogu generisati rezultat bez ispitivanja redova.

Načini pristupa se prikazuje u koloni type u EKSPLAIN klauzuli. Tipovi pristupa se kreću od skeniranja pune tabele do skeniranja indeksa, skeniranja opsega, jedinstvenog pretraživanja indeksa i konstanti. Svaka od njih je brža od one pre nje, jer zahteva čitanje manje podataka. Ne treba da pamtite tipove pristupa, ali treba da razumete opšte koncepte skeniranja tabele, skeniranja indeksa, pristupa opsegu i pristupa sa jednom vrednošću.

Ako ne dobijate dobar tip pristupa, najbolji način za rešavanje problema je obično dodavanje odgovarajućeg indeksa. Indeksi omogućavaju MySQL-u da pronađe redove sa efikasnijim tipom pristupa koji ispituje manje podataka.

Na primer, pogledajmo jednostavan upit:

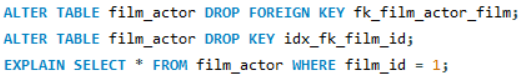


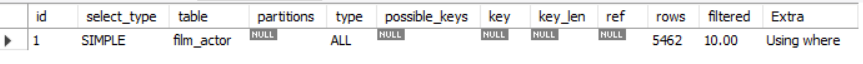
Ovaj upit će vratiti 10 redova, a EKSPLAIN pokazuje da MySQL koristi ref tip pristupa na indeksu idx\_fk\_film\_id za izvršavanje upita:





EKSPLAIN pokazuje da je MySQL procenio da mu je potrebano da pristupi samo 10 redova. Drugim rečima, optimizator upita je znao da izabrani tip pristupa može efikasno izvršiti upit. Šta bi se dogodilo da nema odgovarajućeg indeksa za upit? MySQL bi morao da koristi manje optimalan tip pristupa, kao što možemo videti ako izbrišemo indeks i ponovo pokrenemo upit:





Tip pristupa se promenio u skeniranje pune tabele (ALL), a MySQL sada procenjuje da će morati da ispita 5.073 reda da bi izvršio upit. „Using Where“ u koloni Ektra pokazuje da MySQL server koristi klauzulu WHERE za odbacivanje redova nakon što ih mašina za skladištenje pročita.

Generalno, MySQL može primeniti klauzulu WHERE na tri načina, od najboljeg do najgoreg:

* Primenite uslove na operaciju pretraživanja indeksa da biste eliminisali nepodudaranje redova.
* Koristite indeks pokrivanja („Using index“ u Extra koloni) da biste izbegli pristupe redovima i filtrirajte nepodudarajuće redove nakon pribavljanja svakog rezultata iz indeksa. To se dešava na serverskom sloju, ali ne zahteva čitanje redova iz tabele.
* Pribavite redove iz tabele, a zatim filtrirajte nepodudarajuće redove („Using where“ u Extra koloni). To se dešava na sloju servera i zahteva da server pročita redove iz tabele pre nego što ih može filtrirati.

Ovaj primer ilustruje koliko je važno imati dobre indekse. Dobri indeksi pomažu vašim upitima da dobiju dobar tip pristupa i ispituju samo redove koji su im potrebni. Međutim, dodavanje indeksa ne znači uvek da će MySQL pristupiti i vratiti isti broj redova. Na primer, evo upita koji koristi funkciju agregacije COUNT():



Ovaj upit vraća samo 200 redova, ali treba da pročita hiljade redova da bi se generisao skup rezultata. Indeks ne može da smanji broj redova koji se ispituju za upit poput ovog.

Nažalost, MySQL vam ne govori broj redova kojem je pristupio da bi generisao skup rezultata, govori vam samo ukupan broj redova kojima je pristupio. Mnogi od ovih redova mogu se eliminisati klauzulom WHERE i na kraju neće doprineti skupu rezultata. U prethodnom primeru, nakon uklanjanja indeksa na film\_actor, upit je pristupio svakom redu u tabeli, a klauzula WHERE odbacila je sve sem 10 redova. Samo preostalih 10 redova je korišćeno za generisanje skupa rezultata. Razumevanje kolikom broju redova server pristupa i koliko njih zaista koristi zahteva obrazloženje upita.

Ako otkrijete da je ispitan ogroman broj redova da bi se dobio relativno mali broj redova u rezultatu, možete isprobati neke sofisticiranije metode:

* Koristite indekse pokrivanja koji čuvaju podatke tako da mehanizam za skladištenje ne mora da preuzme kompletne redove.
* Promenite šemu.
* Transformišite složeni upit tako da MySQL optimizator može da ga izvrši optimalno.

# Transformisanje upita

Dok optimizujete problematične upite, vaš cilj bi trebalo da bude pronalaženje alternativnih načina za postizanje željenog rezultata, ali to ne znači nužno vraćanje istog rezultata iz MySQL-a. Upite možete ponekad transformisati u ekvivalentne forme i postići bolje performanse. Međutim, trebalo bi da razmislite i o ponovnom pisanju upita kako biste dobili različite rezultate, ako to donosi veću efikasnost. Na kraju ćete možda moći da izvršite isti posao promenom koda aplikacije kao i upita.

## Kompleksan upiti ili više jednostavnijih upita?

Jedno važno pitanje dizajna upita je da li je poželjno složeni upit podeliti na nekoliko jednostavnijih upita. Tradicionalni pristup dizajnu baze podataka naglašava obavljanje što više posla sa što manje upita. Ovaj pristup je bio istorijski bolji zbog troškova mrežne komunikacije i opštih troškova faza raščlanjivanja i optimizacije upita.

Međutim, ovaj savet se ne odnosi toliko na MySQL, jer je dizajniran da se vrlo efikasno nosi sa povezivanjem i prekidanjem veze i vrlo brzo odgovara na male i jednostavne upite. Savremene mreže su takođe znatno brže nego nekada, smanjujući kašnjenje mreže. MySQL može pokrenuti više od 50.000 jednostavnih upita u sekundi na serveru skromnog hardvera, tako da pokretanje višestrukih upita nije nužno tako loše.

Odgovor mreže je i dalje spor u poređenju sa brojem redova koje MySQL može interno obraditi u sekundi. I dalje je dobra ideja da koristite što manje upita, ali ponekad možete učiniti upit efikasnijim dekomponovanjem i izvršavanjem nekoliko jednostavnih upita umesto jednog složenog.

Međutim, korišćenje previše upita je uobičajena greška u dizajnu aplikacija. Na primer, neke aplikacije izvršavaju 10 upita sa jednim redom za preuzimanje podataka iz tabele a mogu da koriste jedan upit sa 10 redova.

## Usitnjavanje upita

Drugi način za uprošćavanje upita je podela i osvajanje, zadržavajući ga u suštini istim, ali ga izvršavati u manjim „komadima“ koji utiču na manje redova svaki put. Čišćenje starih podataka je sjajan primer. Posao periodičnog čišćenja možda će morati da ukloni popriličan broj podataka, a ako to uradite u jednom masivnom upitu, možete dugo vremena zaključati puno redova, popuniti evidencije transakcija, i blokirati male upite koji ne bi trebalo da se prekidaju. Seckanje klauzule DELETE i upotreba upita srednje veličine mogu znatno poboljšati performanse. Na primer, umesto da pokrenete ovaj monolitni upit:

mysql> **DELETE FROM messages WHERE created < DATE\_SUB(NOW(),INTERVAL 3 MONTH);**

možemo uraditi nešto slično ovom pseudokodu:

**rows\_affected = 0**

**do {**

**rows\_affected = do\_query(**

**"DELETE FROM messages WHERE created < DATE\_SUB(NOW(),INTERVAL 3 MONTH)**

**LIMIT 10000")**

**} while rows\_affected > 0**

Brisanje 10.000 redova odjednom je obično dovoljno velik zadatak da svaki upit učini efikasnim i dovoljno kratak zadatak da minimizira uticaj na server (mehanizmi za transakciono skladištenje mogu imati koristi od manjih transakcija). Takođe bi mogla biti dobra ideja da dodate malo vremena spavanja između naredbi DELETE kako biste vremenom rasporedili opterećenje i smanjili vreme zaključavanja.

## Dekompozicija spojeva

Mnoge web stranice visokih performansi koriste dekompoziciju spojeva. Možete da rastavite spoj tako što ćete pokrenuti više upita sa jednom tabelom umesto jednog upita kroz spoj, a zatim izvršiti spajanje u aplikaciji. Na primer, ovaj upit:

mysql> **SELECT \* FROM tag**

->  **JOIN tag\_post ON tag\_post.tag\_id = tag.id**

->  **JOIN post ON tag\_post.post\_id = post.id**

-> **WHERE tag.tag = 'mysql';**

Možete podeliti na više manjih:

mysql> **SELECT \* FROM tag WHERE tag='mysql';**

mysql> **SELECT \* FROM tag\_post WHERE tag\_id = 1234;**

mysql> **SELECT \* FROM post WHERE post.id in (123,456,567,9098,8904);**

Na prvi pogled ovo izgleda rasipno, jer ste povećali broj upita, a da ništa niste dobili zauzvrat. Međutim, takva transformacija zapravo može dati značajne prednosti u performansama:

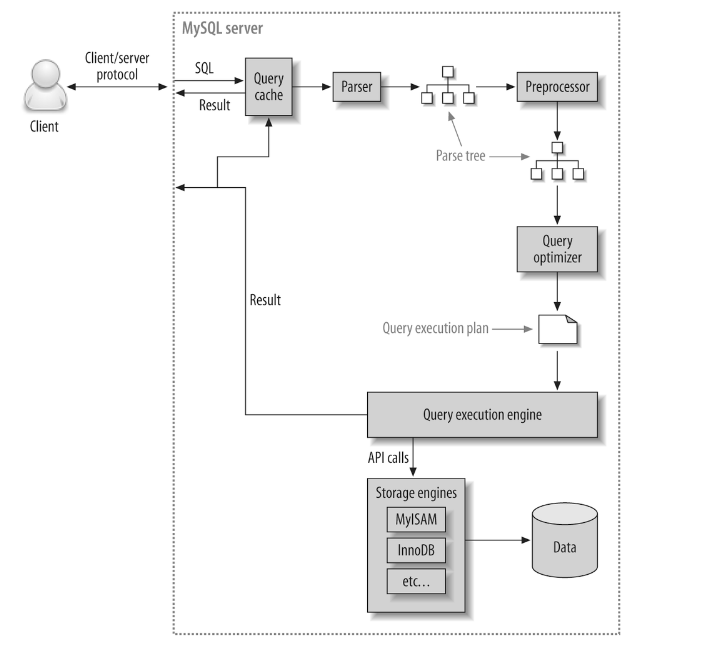
* Keširanje može biti efikasnije. Mnoge aplikacije keširaju „objekte“ koji se mapiraju direktno u tabele. U ovom primeru, ako je objekat sa oznakom mysql već keširan, aplikacija može preskočiti prvi upit. Ako u kešu pronađete postove sa ID-om 123, 567 ili 9098, možete ih ukloniti sa liste IN ().
* Izvršavanje jednog upita po tabeli efikasnije koristi lock-ove tabela: upiti će zaključati tabele pojedinačno i relativno kratko, umesto da ih sve zaključaju duže vreme.
* Spajanjem u aplikaciji olakšava se skaliranje baze podataka postavljanjem tabela na različite servere.
* Sami upiti mogu biti efikasniji. U ovom primeru, upotreba liste IN () umesto spoja omogućava MySQL-u da sortira ID-ove redova i preuzima redove optimalnije nego što bi to moglo biti moguće sa spajanjem.
* Možete smanjiti suvišni pristup redovima. Spajanje u aplikaciji znači da svaki red dohvaćate samo jednom, dok je spajanje u upitu u osnovi denormalizacija koja može u više navrata pristupiti istim podacima. Iz istog razloga, takvo restrukturiranje takođe može smanjiti ukupan mrežni saobraćaj i upotrebu memorije.

# Osnove izvršenja upita

Slika u nastavku prikazuje kako MySQL generalno izvršava upite.

* Klijent šalje SQL upit serveru.
* Server proverava keš memoriju upita. Ako postoji pogodak, vraća sačuvani rezultat iz keš memorije, u suprotnom, prosleđuje SQL naredbu na sledeći korak.
* Server analizira, pretprocesira i optimizira SQL u plan izvršenja upita.
* Mehanizam za izvršavanje upita izvršava plan upućivanjem poziva API-ju mehanizma za skladištenje.
* Server šalje rezultat klijentu.

Svaki od ovih koraka ima neku dodatnu složenost. Proces optimizacije upita je posebno složen i važan za razumevanje.



## MySQL Client/Server protocol

Iako ne morate da razumete unutrašnje detalje MySQL-ovog klijent / server protokola, morate da razumete kako to funkcioniše na visokom nivou. Protokol je half-duplex, što znači da u bilo kom trenutku MySQL server može da šalje ili prima poruke, ali ne u istom trenutku. Ovaj protokol čini MySQL komunikaciju jednostavnom i brzom, ali je i na neki način ograničava.

Prvo, to znači da nema kontrole protoka. Kada jedna strana pošalje poruku, druga strana mora da preuzme celu poruku pre nego što odgovori. To je poput igre bacanja lopte napred-nazad: samo jedna strana u svakom trenutku ima loptu i ne možete je baciti (poslati poruku) ako je nemate.

Klijent šalje upit serveru kao jedan paket podataka. Jednom kada klijent pošalje upit, više nema loptu, može samo da čeka rezultate. Suprotno tome, odgovor servera se obično sastoji od mnogih paketa podataka. Kada server odgovori, klijent mora da primi celokupan skup rezultata. Ne može jednostavno dohvatiti nekoliko redova, a zatim zatražiti od servera da se ne zamara slanjem ostatka. Ako klijentu trebaju samo prvih nekoliko redova koji se vraćaju, on mora ili da sačeka da stignu svi paketi sa servera, a zatim da odbaci one koji mu nisu potrebni ili da se nezahvalno prekine.

Nijedna nije dobra ideja, zato su odgovarajuće LIMIT klauzule toliko važne. Evo još jednog načina da razmišljamo o ovome: kada klijent preuzme redove sa servera, on misli da ih povlači. Ali istina je da MySQL server gura redove dok ih generiše. Klijent prima samo potisnute redove. Nema načina da kaže serveru da prestane da šalje redove. Većina biblioteka koje se povezuju na MySQL omogućavaju vam ili da preuzmete čitav skup rezultata i da ga baferujete u memoriju, ili da preuzmete svaki red po potrebi. Podrazumevano ponašanje je uglavnom preuzimanje celog rezultata i njegovo baferisanje u memoriji. Ovo je važno, jer dok se ne preuzmu svi redovi, MySQL server neće osloboditi lock-ove i druge resurse potrebne u upitu. Upit će biti u stanju „ Sending data “.

Kada klijentska biblioteka odjednom preuzme rezultate, to smanjuje količinu posla koji server treba da uradi. Većina klijentskih biblioteka vam omogućava da se prema rezultatu ponašate kao da ga dobijate sa servera, iako ga zapravo dobijate iz bafera u memoriji biblioteke. Ovo većinu vremena funkcioniše u redu, ali nije dobra ideja za ogromne skupove rezultata kojima treba dugo vremena da se preuzmu i koriste puno memorije. Možete koristiti manje memorije i započeti rad na rezultatu pre, ako uputite biblioteku da ne baferuje rezultat. Loša strana je ta što će lock-ovi i drugi resursi na serveru ostati otvoreni dok vaša aplikacija komunicira sa bibliotekom.

Pogledajmo primer korišćenja PHP-a. Ovako ćete obično pitati MySQL iz PHP-a:

**<?php**

**$link = mysql\_connect('localhost', 'user', 'p4ssword');**

**$result = mysql\_query('SELECT \* FROM HUGE\_TABLE', $link);**

**while ( $row = mysql\_fetch\_array($result) ) {**

**// Do something with result**

**}**

**?>**

Čini se da kod ukazuje na to da redove preuzimate samo kada vam trebaju, u while petlji. Međutim, kod zapravo preuzima čitav rezultat u bafer pomoću poziva funkcije mysql\_query(). While petlja jednostavno prolazi kroz bafer. Nasuprot tome, sledeći kod ne baferuje rezultate, jer koristi mysql\_unbuffered\_query() umesto mysql\_query():

**<?php**

**$link = mysql\_connect('localhost', 'user', 'p4ssword');**

**$result = mysql\_unbuffered\_query('SELECT \* FROM HUGE\_TABLE', $link);**

**while ( $row = mysql\_fetch\_array($result) ) {**

**// Do something with result**

**}**

**?>**

## Proces optimizacije upita

Sledeći korak u životnom ciklusu upita pretvara SQL upit u plan izvršenja za mehanizam izvršavanja upita. Ima nekoliko potkoraka: parsiranje, preprocesiranje i optimizacija. Greške (na primer, sintaksne greške) se mogu pojaviti u bilo kom trenutku procesa.

### Parser i pretprocesor

Za početak, MySQL-ov parser deli upit na tokene i od njih pravi „stablo za parsiranje“. Parser koristi MySQL-ovu gramatiku SQL-a za tumačenje i potvrđivanje upita. Na primer, osigurava da su tokeni u upitu važeći i u pravilnom redosledu i proverava.

Preprocesor zatim proverava rezultirajuće stablo parsiranja za dodatne semantike koje parser ne može da reši. Na primer, proverava da tabele i kolone postoje i rešava imena i pseudonime kako bi se osiguralo da reference kolona nisu dvosmislene. Dalje, pretprocesor proverava privilegije. To je obično vrlo brzo, osim ako vaš server nema veliki broj privilegija.

### Optimizator upita

Stablo parsiranja je sada validno i spremno da ga optimizator pretvori u plan izvršenja upita. Upit se često može izvršiti na više različitih načina i dati isti rezultat. Zadatak optimizatora je da pronađe najbolju opciju. MySQL koristi optimizator zasnovan na troškovima, što znači da pokušava da predvidi troškove različitih planova izvršenja i izabere najjeftiniji. Jedinica troškova je jedna slučajna pročitana stranica podataka od četiri kilobajta. Možete videti koliko je skupo optimizator procenio upit tako što ćete pokrenuti upit, a zatim pregledati promenljivu sesije Last\_query\_cost:









Ovaj rezultat znači da je optimizator procenio da bi za izvršenje upita trebalo da uradi oko 550 pročitanih slučajnih stranica podataka. Procenu zasniva na statistikama: broju stranica po tabeli ili indeksu, kardinalnosti (broju različitih vrednosti) indeksa, dužini redova i ključeva i distribuciji ključeva. Optimizator u svoje procene ne uključuje efekte bilo koje vrste keširanja - pretpostavlja da će svako čitanje rezultirati operacijom I / O diska.

MySQL-ov optimizator upita je izuzetno složen softver i koristi mnoštvo optimizacija da transformiše upit u plan izvršenja. Postoje dva osnovna tipa optimizacija, koje nazivamo statičkim i dinamičkim.

Statička optimizacija se može izvršiti jednostavnim pregledom stabla za parsiranje. Na primer, optimizator može transformisati klauzulu WHERE u ekvivalentni oblik primenom algebarskih pravila. Statička optimizacija je nezavisna od vrednosti, kao što je vrednost konstante u klauzuli WHERE. Mogu se izvršiti jednom i uvek će biti validni, čak i kada se upit ponovo izvršava sa različitim vrednostima. O njima možete razmišljati kao o „optimizaciji vremena kompajliranja“.

Suprotno tome, dinamičke optimizacije se zasnivaju na kontekstu i mogu zavisiti od mnogih faktora, kao što je vrednost u klauzuli WHERE ili broj redova u indeksu. Moraju se preispitati svaki put kada se izvrši upit. Možete ih smatrati „optimizacijom u trenutku izvršenja“. MySQL može jednom da izvrši statičku optimizaciju, ali mora da pokrene dinamičku optimizaciju svaki put kada izvrši upit.

U nastavku su opisane dodatne optimizacije koje MySql implementira.

#### Preuređivanje spojeva

Tabele ne moraju uvek da se spajaju redosledom koji navedete u upitu. Određivanje najboljeg redosleda spoja je važna optimizacija. Često je moguće spojiti tabele u nekoliko različitih redosleda i dobiti iste rezultate. Optimizator spojeva procenjuje troškove za različite planove i pokušava da izabere najoptimalniji.

#### Pretvaranje OUTER JOIN-ova u INNER JOIN-ove

OUTER JOIN ne mora nužno biti izvršeno kao OUTER JOIN. Neki faktori, poput klauzule WHERE i šeme tabele, zapravo mogu dovesti do toga da OUTER JOIN bude ekvivalentan INNER JOIN-u. MySQL to može prepoznati i preurediti spoj.

#### Primena pravila algebarske ekvivalencije

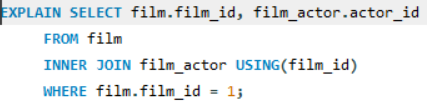
MySQL primenjuje algebarske transformacije za pojednostavljivanje i kanonizaciju izraza. Takođe može da savija i smanjuje konstante, eliminišući nemoguća ograničenja i stalne uslove. Na primer, izraz (5 = 5 i a> 5) smanjiće se na samo a> 5. Slično tome, (a <b AND b = c) AND a = 5 postaje b > 5 AND b = c AND a = 5. Ova pravila su vrlo korisna za pisanje uslovnih upita.

#### COUNT (), MIN () i MAX () optimizacija

Indeksi i null vrednosti u koloni često mogu pomoći MySQL-u da optimizuje upit. Na primer, da bi pronašao minimalnu vrednost kolone koja je krajnja leva u indeksu B-Stabla, MySQL može samo da zatraži prvi red u indeksu. To može čak i u fazi optimizacije upita i tretirati vrednost kao konstantu za ostatak upita. Slično tome, da bi pronašao maksimalnu vrednost u indeksu B-Stabla, server čita poslednji red. Ako server koristi ovu optimizaciju, videćete „ Select tables optimized away “ u EKSPLAIN klauzuli. To doslovno znači da je optimizator uklonio tabelu iz plana upita i zamenio je konstantom.

#### Evaluacija i smanjivanje konstantnih izraza

Kada MySQL otkrije da se izraz može svesti na konstantu, to će učiniti tokom optimizacije. Na primer, korisnički definisana promenljiva može da se pretvori u konstantu ako nije promenjena u upitu. Aritmetički izrazi su još jedan primer. Možda iznenađujuće, čak i nešto što biste mogli smatrati upitom, može se svesti na konstantu tokom faze optimizacije. Jedan primer je MIN () na indeksu. Ovo se čak može proširiti na konstantno traženje primarnog ključa ili jedinstvenog indeksa. Ako klauzula WHERE primenjuje konstantan uslov na takav indeks, optimizator zna da MySQL može potražiti vrednost na početku upita. Tada će vrednost tretirati kao konstantu u ostatku upita. Evo primera:



| id | select\_type | table | type | key | ref | rows |

+----+-------------+------------+-------+----------------+-------+------+

| 1 | SIMPLE | film | const | PRIMARY | const | 1 |

| 1 | SIMPLE | film\_actor | ref | idx\_fk\_film\_id | const | 10 |

+----+-------------+------------+-------+----------------+-------+------+

MySQL izvršava ovaj upit u dva koraka. Prvi korak je pronalaženje željenog reda u tabeli filmova. MySQL-ov optimizator zna da postoji samo jedan red, jer se u koloni film\_id nalazi primarni ključ, i već je konsultovao indeks tokom faze optimizacije upita da vidi koliko će redova naći. Budući da optimizator upita ima poznatu količinu (vrednost u klauzuli WHERE) za upotrebu u pretrazi, ref tip ove tabele je const.

U drugom koraku, MySQL kolonu film\_id iz reda koji je pronađen u prvom koraku tretira kao poznatu količinu. To može učiniti jer optimizator zna da će do trenutka kada upit stigne do drugog koraka znati sve vrednosti iz prvog koraka. Primetite da je ref tip tabele filma\_actor const, baš kao što je bio i kod film table.

Drugi način na koji ćete videti primenjene konstantne uslove je širenjem konstantnosti vrednosti sa jednog mesta na drugo ako postoji klauzula WHERE, USING ili ON koja ih ograničava na izjednačavanje. U ovom primeru, optimizator zna da klauzula USING prisiljava film\_id svuda u upitu da ima istu vrednost - mora biti jednaka konstantnoj vrednosti datoj u klauzuli WHERE.

### Pokrivajući indeksi

MySQL ponekad može da koristi indeks kako bi izbegao čitanje podataka iz redova, kada indeks sadrži sve kolone potrebne upitu.

#### Optimizacija podupita

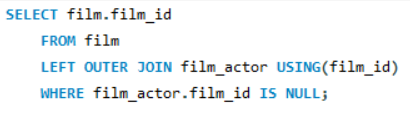
MySQL može pretvoriti neke vrste podupita u efikasnije alternativne oblike, smanjujući ih na indeksne pretrage umesto u zasebne upite.

#### Raniji završetak

MySQL može zaustaviti obradu upita čim ispuni upit. Očigledan slučaj je klauzula LIMIT, ali ne I jedini. Na primer, ako MySQL otkrije nemoguće stanje, može prekinuti čitav upit. To možete videti u sledećem primeru:

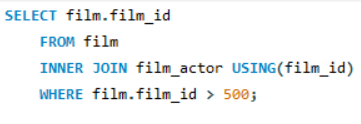


Upit u nastavku uklanja filmove koji imaju glumce. Svaki film može imati mnogo glumaca, ali čim pronađe jednog glumca, zaustavlja obradu trenutnog filma i prelazi na sledeći jer zna da klauzula WHERE zabranjuje prikazivanje tog filma.



#### Prostiranje jednakosti

MySQL prepoznaje kada upit koristi dve jednake kolone, na primer, u JOIN-u i propagira klauzulu WHERE na ekvivalentne kolone. Na primer, u sledećem upitu:



MySQL zna da se klauzula WHERE odnosi ne samo na tabelu film već i na tabelu film\_actor, jer klauzula USING garantuje da se dve kolone podudaraju.

#### IN () lista

Kod mnogih servera baza podataka IN () je samo sinonim za više OR klauzula, jer su logički ekvivalentni. U MySQL-u nije tako. On sortira vrednosti u IN() listi i koristi brzu binarnu pretragu da bi utvrdio da li je neka vrednost u listi. Složenost ovakvog pristupa je O (log n) umesto O (n).

Prethodna lista je nepotpuna, jer MySQL izvodi mnogo više optimizacija, ali daje predstavu o složenosti i inteligenciji optimizatora. Naravno, koliko god optimizator bio pametan, ima trenutaka kada ne daje najbolji rezultat. Ponekad možda znate nešto o podacima koje optimizator ne zna, na primer činjenica koja je zagarantovana tačna zbog logike aplikacije. Ako znate da optimizator ne daje dobar rezultat i znate zašto, možete mu pomoći. Neke od opcija su dodavanje nagoveštaja upitu, prepisivanje upita, re-dizajn šeme ili dodavanje indeksa.

# Literatura

* <https://www.khanacademy.org/computing/computer-programming/sql/relational-queries-in-sql/a/more-efficient-sql-with-query-planning-and-optimization>
* <https://cs.elfak.ni.ac.rs/nastava/pluginfile.php/17937/mod_resource/content/2/Optimizacija%20upita%202016.pdf>
* <https://www.section.io/engineering-education/mysql-query-performance-optimization-tips/>
* Korišćena baza:

