**Univerzitet u Nišu, Elektronski fakultet** &Bcy;&rcy;&ocy;&jsercy; &scy;&lcy;&ocy;&bcy;&ocy;&dcy;&ncy;&icy;&khcy; &mcy;&iecy;&scy;&tcy;&acy; &zcy;&acy; &scy;&iecy;&pcy;&tcy;&iecy;&mcy;&bcy;&acy;&rcy;&scy;&kcy;&icy; &ucy;&pcy;&icy;&scy;&ncy;&icy; &rcy;&ocy;&kcy; 2012

**Katedra za računarstvo**

**Obrada upita u MySQL-u**

**Seminarski rad iz Sistema za upravljanje bazama podataka**

**Student: Teodora Hafner 1237**

**Niš, 2020**

# Uvod

Obrada upita uključuje prevođenje upita visokog nivoa na izraze niskog nivoa koji se mogu iskoristiti na fizičkom nivou fajl sistema, optimizaciju upita i samog izvršenja upita za dobijanje potrebnih rezultata.

Osnovni dijagram obrade upita dat je na slici 1.

Diagram

Description automatically generated

*Slika 1- Blok dijagram obrade upita*

Kada razložimo ove osnovne komponente dobija se dijagram na slici 2.

Diagram

Description automatically generated

*Slika 2 Detaljni dijagram obrade upita*

Vodeći se dijagramom komponenti zaključujemo da su koraci u obradi upita sledeći:

* Korak 1:

**Parser**: u toku parsiranja, baza podataka vrši sledeće provere: proveru sintakse i semantike i proveru keša upita , nakon prevođenja upita u relacionu algebru.

1. Provera sintakse – zaključuje SQL sintaksnu ispravnost.
2. Provera semantike – određuje da li je upit smislen ili ne. Na primer, ako se u upitu pojavljuje ime tabele koja ne postoji u bazi, provera semantike određuje da li postoje greške ovog tipa i vraća grešku.
3. Provera postojanja upit au kešu (Shared pool check) - svaki upit poseduje svoj *hash* kod prilikom izvršenja. Ova provera određuje postojanje napisanog hash koda u zajedničkoj bazi i ako kod postoji onda baza ne preuzima dalje korake koji se tiču optimizacije i izvršenja upita.

**Hard parse** i **soft parse**:

Ako se pojavi novi upit za koji ne postoji heš kod u kešu, ovaj upit prolazi kroz dodatne korake koji podrazumevaju dodatno „tvrdo“ raščlanjivanje. Ako postoji, kao što smo naveli, on ne prolazi krz dodatne korake i direktno prelazi u izvršenje (kao što je prikazano na dijagramu). Ovo je poznato kao meko raščlanjivanje. „Tvrdo“ raščlanjivanje podrazumeva optimizovanje (Optimizer) i generisanje izvora reda (Row source generation).

* Korak 2

**Optimizacija upita**: tokom faze optimizacije, baza podataka mora da izvrši „tvrdo“ parsiranje bar jednog jedinstvenog DML izraza i izvrši optimizaciju u toku parsiranja. Ova baza nikad ne optimizuje DDL ukoliko ne sadrži DML komponente kao podupite koji zahtevaju optimizaciju. Ovo je proces koji u kome se plan izvršenja višestrukih upita koji zadovoljava krajnji upit ispituje i plan koji je najefikasniji i on se izvršava.

Katalog baze čuva planove za izvršenje a zatim optimizer parsuje plan koji je najjeftiniji za izvršenje.

**Generisanje izvora reda**: ovo je softver koji prima optimalan plan za izvršenje upita iz prethodnog koraka i proizvodi iterativni plan izvršenja koji koristi baza. Iterativni plan je binarni program koji kada se izvrši od strane sql engine-a proizvodi rezultat.

* Korak 3

**Mašina za izvršenje**: Upit se izvršava i prikazuje se dobijeni rezultat

U nastavku ovog rada, svakiod koraka biće dodatno obrađen, uz date primere.

# Keširanje upita

Omogućavanje keširanja upita postiže se postavljanjem promenjive **query\_cache\_type** na odgovarajuću vrednost ( u **my.cnf**)

**query\_cache\_type = 1**

Kada primi upit, MySQL najpre pokušava da locira rezultat bilo kog SELECT upita u kešu upita, da ga ne bi ponovo analizirao i izvršavao. Ovo je omogućeno tako što se svakom upitu dodeljuje heš vrednost i po toj vrednosti se proverava da li postoji takav upit u kešu. MySQL koristi tekst u originalnom obliku, pa je keš senzitivan na trivijalne varijacije.

Na primer, što se tiče keša, upit

**SELECT \* FROM table**

Se razlikuje od upita

**select \* FROM table**

Isto važi za varijacije u belinama. MySQL ne skraćuje dodatne beline na početku ili kraju upita. Ovo je retko problem jer se većina upita koji se ponavljaju generišu od strane aplikacije, nego od korisnika koji ga unosi.

MySQL keš pamti samo SELECT upite jer su oni jedini koje ima smisla keširati. Starije verzije MySQL-a, ne keširaju sve SELECT upite (4.0 i ranije). Logika za keširanje je jednostavna i proverava prva tri karaktera upita, tražeći poklapanje sa *SEL* sekvencom (ne razlikujući mala i velika slova). Međutim, ovo može predstavljati problem ako se nađe bilo koja belina ili komentar na početku upita, jer onda MySQL keš neće razmatrati ovaj upit. Ovo može predstavljati problem za neke aplikacije.

Poznajemo sisteme za procesiranje feed-ova u kojima programeri koriste komentare na početku upita da bi ugradili dodatne informacije o upitu. Kao na primer:

**/\* <b>GetLatestStuff</b> \*/ SELECT \* FROM sometable WHERE …**

Komentari na početku upita olakšavaju identifikaciju upita u administrativnim alatima koji primaju izlaz **SHOW PROCESSLIST** za prikaz na web stranici. Nažalost, ne postoji način da se MySQL-u naznači postojanje ovakvih upita i da je potrebna dublja obrada upita u potrazi za SELECT upitima koje treba keširati. Kao posledica, ovi upiti nikad neće biti keširani. Ovaj problem rešen je u verziji 5.0. MySQL-a.

Moguće je naznačiti MySQL-u da ne kešira određeni upit. Ovo se postiže dodavanjem naznake **SQL\_NO\_CACHE** u upitu iza ključne reči SELECT.

**SELECT SQL\_NO\_CACHE \* FROM mytable**

Ovo može biti korisno u kontroli opterećenja keša. Ako aplikacija poseduje set upita koji neće imati koristi od keširanja upita (npr. pokreću se jednom dnevno), nema smisla keširati ih. Takođe, ovim se ostavlja više prostora za keširanje upita koji se ponavljaju.

Ako se vrednost **query\_cache\_type** postavi na vrednost 2, tačnije MySQL keš radi u režimu na zahtev, MySQL radi na tome da nađe upit u kešu samo ako u upitu postoji naznaka **SQL\_CACHE**.

**SELECT SQL\_CACHE \* FROM mytable**

Ako se heš vrednost upita ne nalazi u kešu, MySQL šalje rezultate iz keša, zaobilazeći sve dodatne napore, kao što se i očekuje. Format rezultata u kešu upita identičan je kao format koji se koristi kada se rezultati šalju korisniku. Na ovaj način, mali su indirektni troškovi u preuzimanju rezultata iz keša i njihovo slanje klijentu. MySQL jednostavno šalje podatke preko mreže.

# Parsiranje, analiza i optimizacija upita

MySQL prima upite u SQL formatu. Kada je upit primljen, njega je prvo potrebno parsirati, što uključuje prevođenje iz suštinski tekstualnog formata u kombinaciju internih binarnih struktura kojima *optimizator* može lako baratati.

Optimizator, u ovom kontekstu, predstavlja modul odgovoran za kreiranje plana za izvršenje i izvršavanje istog sa ciljem da se dođe do rezultata koji su se upitom zahtevali. Optimizator bira redosled spojeva tabeli, metodu za čitanje redova u tabeli (npr. korišćenjem indeksa ili skeniranjem tabele), kao i koji ključevi će biti korišćeni. Cilj optimizatora jeste da dostavi rezultate upita u što kraćem vremenskom roku.

## Parser

Pre nego što MySQL uradi bilo šta sa upitom, on prvo mora da razloži dati upit na njegove komponente.

MySQL parser sastoji se iz dva dela:

* Leksičkog skenera
* Modul gramatičkih pravila

Leksički skener razlaže ceo upit na tokene (elemente koji su nedeljivi, npr. ime kolone), dok modul gramatičkih pravila traži kombinaciju SQL gramatičkih pravila koja generišu dobijenu sekvencu tokena, i izvršava kod koji se vezuje za izabrana SQL gramatička pravila. Ovim se kreira *stablo razlaganja* (parse tree), koje se prenosi u sledeću komponentu, optimizator.

Za razliku od parsera koji prevode tekstualnu reprezentaciju upita u bajt kod, MySQL parser je prevodi direktno u unutrašnje međusobno povezane C/C++ strukture u programskoj memoriji.

Na primer, kada se primi sledeći upit:

**SELECT count(\*) , name FROM product GROUP BY name**

Leksički skener ispituje tok karaktera upita, rastavlja ga u tokene i identifikuje svaki token. U konkretnom primeru, nalazi sledeće tokene:

* SELECT
* (
* \*
* )
* ,
* name
* FROM
* Product
* GROUP
* BY
* name

Svakom tokenu dodeljen je tip – ključna reč, string, broj, operator ili ime funkcije. Modul gramatičkih pravila traži poklapanja skupa tokena sa setovima pravila, i nalazi pravilo koje odgovara skupu tokena. U konkretnom primeru je *select* pravilo. Dalje se, po pronađenom pravilu, inicijalizuje stablo razlaganja, koje dalje dovodi do izvršenja funkcije mysql\_select().

Parser ima dva glavna cilja. Mora da bude jako brz (i po hiljade upita u sekundi, što bi bilo nemoguće i da se razlaganje obavlja jednu milisekundu), i generisano stablo razlaganja mora da obezbedi informacije optimizatoru na način koju mu omogućava brz pristup podacima (optimizatoru je potreban brz pristup različitim delovima WHERE klauzule, tabelama, poljima, listama ključeva, ORDER BY i GROUP BY izrazima…).

### Leksički skener

MySQL ima sopstveni leksički skener da bi dobio na performansama i fleksibilnosti. Ručno pisan identifikator tokena može se usavršiti različitim optimizacijama koje nisu moguće kod generisanog koda (kao kod korišćenja GNU Flex-a [[1]](#footnote-0) za generisanje leksičkih skenera). Dodatno, može se unaprediti tako da identifikuje tokene u zavisnosti od konteksta u kome se oni nalaze.

Jako efikasna pretraga ključnih reči uz pomoć heš vrednosti kreira se uz pomoć specijalnog uslužnog programa gen\_lex\_hash (unutar sql/gen\_lex\_hash.cc). Generisana heš funkcija je savršena, tačnije ne postoji kolizija u generisanim heš vrednostima ključnih reči. Skener označava svaki token kao ključnu reč, ime funkcije, brojnu vrednost određenog tipa ili nekim drugim simbolom koji ima značenje u gramatičkim pravilima.

Lista ključnih reči nalazi se unutar liste *symbols[]* unutar *sql/lex.h* a lista funkcija se nalazi u listi *sql\_functions[]* u istom fajlu (nakon verzije 5.1 prebačene u *native\_functions\_hash*, pa se ugrađene funkcije identifikuju od strane modula za gramatička pravila umesto u leksičkom skeneru).

Pristupna tačka leksičkom skeneru je funkcija ***yylex()*** unutar *sql/sql\_lex.cc*. Ime ove funkcije ima posebno značenje: mora da bude kompatibilna sa GNU Bison-om, generatorom modula gramatičkih pravila, koji obavlja preuzimanje tokena pozivom funkcije sa ovakvim imenom.

Modul gramatičkih pravila

Kao u mnogo drugih projekata otvorenog koda, modul gramatičkih pravila je generisan korišćenjem generatora uslužnog programa GNU Bison. Da bi se bolje razumeo proces parsiranja na bolji način potrebno je upoznati se sa Bison-om.

### Stablo razlaganja (Parse Tree)

Krajnji izlaz izvršenja parsera jeste stablo razlaganja. Jasno je da kompleksnost SQL sintakse zahteva podjednako kompleksne strukture koje bi efikasno čuvale podatke koji su potrebni za izvršenje bilo kog SQL izraza.

Stablo razlaganja predstavljeno je uz pomoć objekta tipa LEX. LEX ima mnogo atributa od kojih se kao važniji izdvajaju *sql\_command* i *select\_lex*. *sql\_command* prikazuje koji tip SQL upita se izvršava, da li je to select, update, insert, delete ili neki drugi tip upita. Ova vrednost se koristi kada se poziva mysql\_execute\_command() koja upućuje tok podataka funkciji koja je povezana sa tipom upita koji se u njoj nalazi. Select\_lex čuva informacije o upitu, kao što su WHERE klauzula, lista tabeli, informacije o naznakama optimizatoru, reference na druge instance objekata tipa SELECT\_LEX koji predstavljaju ugnježdene upite, ORDER BY, GROUP BY i HAVING izraze, i još puno drugih detalja vezanih za upit. Jako bitan atribut je **Item\* where** koji predstavlja root čvor stabla WHERE klauzule, zato što se većina informacija potrebnih optimizatoru izdvajaju upravo iz WHERE klauzule.

Klasa Item je roditeljska klasa za sve Item\_ klase i ove klase se koriste za predstavljanje čvorova unutar stabla izraza. Ove klase obuhvataju aritmetičke operacije (dodavanje, oduzimanje, množenje i deljenje), različite SQL funkcije, logičke operatore kao što su AND i OR, reference na polja u tabelama, i sve ostale elemente koji se mogu naći u izrazu WHERE, HAVING, GROUP BY, ORDER BY ili listi polja izabranog upita.

Klasa Item poseduje nekoliko metoda čije ime počinje sa val\_. Ostatak imena zavisi od tipa povratne vrednosti. Na primer, ako je povratna vrednost tipa integer, ime metode je *val\_int()*. Optimizator kasnije koristi objekat klase Item (prethodno pomenuti atribut **where**) da bi generisao izraz za filtriranje koji će koristiti da bi odredio koje redove će obrađivati. Izrazi za filtriranje se procenjuju pozivom Item::val\_int(). Ako poziv funkcije vrati vrednost 1, smatra se da je red zadovoljio ograničenja i da će biti uvrstan u skup rezultata. U suprotnom, odbacuje se. Izraz za filtriranje identičan je originalnoj WHERE klauzuli ako optimizator ne uspe da izvrši njeno unapređenje. Ako to nije slučaj, izraz za filtriranje biće ponovo napisan kako bi se eliminisala nepotrebna izračunavanja i dozvolila bolja upotreba ključeva. Takođe, može da sadrži delove HAVING klauzule.

Primer stabla izraza WHERE klauzule prikazan je na slici ispod i ono je rezultat sledećeg upita:

**SELECT count(\*) FROM customer WHERE name='Jones' AND age BETWEEN 25 AND 30**

Diagram

Description automatically generated

## Optimizator

Kada se upit razloži na više bazičnih komponenti, MySQL odlučuje šta da radi sa datim upitom. Ovo je korak u kome se koristi optimizator upita. Cilj optimizatora je da nađe najefikasniji način za izvršenje upita na osnovu informacija koje su mu date. U najvećem broju slučajeva, ovo znači da optimizator radi na tome da ograniči broj redova koje treba da obradi. Ovo radi zato što je vreme povezano sa U/I diskom često (ali ne uvek) faktor koji odrađuje vreme izvršavanja upita. Ova ista logika povezana je i sa dobrobitima korišćenja indeksa u bazi podataka.

Da bi objasnili kako radi ova komponenta MySQL-a, koristićemo sledeći upit:

**SELECT c.first name, c.last\_name ,c.phone,p .name ,p.price**

**FROM customer c, orders o, product p**

**WHERE c.id = o. customer\_id AND o.product\_id =p.id AND o.payment\_status = 'FAILED'**

**ORDER BY c.last\_name, c.first\_name**

Ovim upitom pokušavamo da dobijemo osnovne informacije klijenta (ime, prezime i broj telefona), kao i informacije o proizvodu čije je plaćanje iz nekog razloga nije uspelo.

Naivan način pribavljanja ovih podataka bio bi prolazak kroz sve redove tabele customer i za svaki od tih redova ponovno bi se prolazilo kroz sve redove tabele order a zatim za svaku kombinaciju redova iz ove dve tabele prolazilo bi se kroz red tabele product. Za svaku kombinaciju trojki ispituje se uslov dat u WHERE klauzuli i zadržavaju se samo redovi koji odgovaraju datom uslovu. Zatim bi se svi odgovarajući redovi sortirali i dostavili klijentu.

Ovaj pristup nije efikasan. Ako pretpostavimo da svaka tabela ima 10 000 redova, optimizator bi trebao da obradi 1 trilion kombinacija redova (10 000 10 000 10 000). Sa procesorom koji je u stanju da obradi 1 milion redova u sekundi, ova obrada bi trajala 1 milion sekundi, što je približino 11 dana.

Sa druge strane, ako pogledamo dati upit, ustanovićemo možemo kreirati indekse nad kolonama cutomer.id, product.id i orders.payment\_status i da su vrednosti za customer.id i product.id jedinstvene. Kako imamo restriktivno ograničenje nad kolonom orders.payment\_status koje bi eliminisalo veliki broj redova, ima smisla kretati od nalaženja redova u tabeli orders u kojima vrednost u koloni payment\_status odgovara vrednosti koja je preneta u upitu. Za svaki od redova koji zadovoljavaju ovaj uslov, nalaze se redovi iz tabele customer koji sadrže jedinstvenu prosleđenu vrednost vrednosti customer.id (customer. id orders.customer id) , kao i redovi iz tabele product koji sadrže jedinstvenu prosleđenu vrednost product.id (product.id = order.product\_id). Zaključujemo da je potrebno ispitati samo redove iz tabele orders koji vrednost kolone vrednost „FAILED". Pa čak da tabela ima 10 000 redova, optimizator u ovom slučaju obrađuje samo 10 000 redova.

lako je korišćenje indeksa povećalo vreme za koje se kreira jedna kombinacija koju treba obraditi, njihovo korišćenje je isplativo. Po standardnom modelu procene troškova MySQL optimizatora, pristup svakom indeksu traje tri puta više vremena od trajanja pristupa skeniranjem za istu tabelu. Time, naivnim pristupom se trojka redova formira za 1+1+1=3 jedinice vremena, dok se kod poboljšanog vremena ona kreira za 3+3+3=9 jedinica vremena. Poboljšanim pristupom, na istom procesoru, možemo obraditi 333 333 redova u sekundi (ako skeniranjem obradi 1 milion), ali je sada potrebno obraditi ne više od 10 000 redova, Eto bi značilo da obrada poboljšanim pristupom traje manje od 0,03 sekunde, u poređenju sa 11 dana.

Naravno, MySQL treba da donese odluku jako brzo i bez testiranja svih opcija. U suprotnom, može potrošiti više vremena na odluku kako izvršiti upit nego što će trajati samo njegovo izvršenje.

MySQL-ov optimizator ima nekoliko bitnih uloga:

* odlučuje koje indekse može da koristi da bi pribavio odgovarajuće redove iz tabela, i da izabere najbolji za svaku od tabela
* za svaku tabelu, treba da odluči da li je skeniranje tabele bolje nego korišćenje indeksa. Ako postoji puno redova u tabeli koji bi bili pribavljeni korišćenjem indeksa, njihovo korišćenje gubi na vrednosti i skeniranje tabele je bolji pristup
* da odredi redosled spojeva tabeli, ako se u upitu nalazi više od jedne tabele
* da ponovo napiše WHERE klauzulu kako bi eliminisao kod koji se ne koristi, eliminišući nepotrebna izračunavanja i menja ograničenja ako bi ona doprinela mogućnosti korišćenja nekih indeksa
* da eliminiše nekorišćene tabele iz spojeva
* da odluči da li se mogu koristiti indeksi za ORDER BY i GROUP BY
* da pokuša da zameni spoljašnje spojeve (outer join sa inner join)
* da pokuša da pojednostavi ugnježdene upite, kao i da utvrdi u kojoj meri se njegovi rezultati mogu keširati
* da objedini poglede (proširi referencu prikaza kao makro)

### Osnove algoritma optimizatora

Sledeće pitanje koje se postavlja jeste to kako optimizator donosi odluke o efikasnosti izvršenja upita. MySQL ima niz pravila i heuristike koje se razvijaju od ranih verzija. Ova pravila upravljaju procesom donošenja odluka. Ali kao i svaki kompjuterski program koji mora da obraduje beskonačne načine na koje ljudi mogu da sastavljaju podatke i traže informacije o njima, optimizator nije savršen. Pravila i heuristike koje se koriste uglavnom dobro funkcionišu, ali povremeno ne funkcionišu.

Sa gledišta MySQL optimizatora, svaki upit je set spojeva. Termin spoj koristi u širem kontekstu nego u SQL komandama. Upit nad samo jednom tabelom je degenerativni spoj. lako obično ne razmišljamo o čitanju redova jedne tabele kao spoju, iste strukture i algoritmi koji se koriste za konvencijalne spojeve savršeno funkcionišu da bi obradili upit sa jednom tabelom.

Jednostavni upiti, bez ugnježdenih upita ili unija, sastoje se iz samo jednog spoja. Upiti sa ugnježdenim upitima koji se ne mogu optimizovati, kao i upiti koji sadrže unije, zahtevaće više od jednog spoja. Neki ugnježdeni upiti zahtevaće spoj koji se može smatrati rekurzivnim: dok se izvodi jedan spoj, optimizator mora da izvrši ugnježdeni upit za svaki red spoja, što daje rezultat u svom sopstvenom spoju. Ipak, spoj je osnovna jedinica rada optimizatora. U izvornom kodu, spoj je povezan sa svojom opisnom klasom JOIN, a svaki spoj se kreira pozivom funkcijom.

Spoj je u suštini Dekartov proizvod podskupova tabela. Svaki podskup se dobija čitanjem redova iz tabele na osnovu vrednosti indeksa, opsega indeksa (ili skupa opsega indeksa), skeniranja celog indeksa ili skeniranja cele tabele. Redovi se zatim eliminišu, ako je potrebno, korišćenjem ograničenja WHERE klauzule.

Optimizator bira metode za pristup redovima tabele i stavlja tabele u redosled za koji veruje da je najefikasniji, tj. ima najmanju cenu koja je proporcionalna ukupnom broju redova koje optimizator treba da obradi. Glavne uloge optimizatora mogu se strogo podeliti na dva dela: da se za dati redosled spojeva nade najbolji redosled pristupa tabeli, i kada stekne mogućnost, nade najbolji redosled spojeva ili bar razumno dobar, u veoma kratkom vremenskom periodu.

Prvi problem rešava funkcija best\_access\_path(). Redosled pristupa definiše da li će optimizator čitati redove tabele na osnovu indeksa, skeniranjem cele tabele ili skeniranjem indeksa. Ako se koristi pristup redovima na osnovu indeka, dodatno se definiše kako e se ono izvoditi, na primer: čitanje jednog reda na osnovu jedne vrednosti, čitanje više od jednog reda na osnovu jedne vrednosti ili opsega vrednosti. Ova funkcija poziva se sa unapred smišljenim redosledom pristupa tabelama za delimičan redosled spojeva. Zbog toga je najbolji redosled pristupa već određen za stari delimičan plan, a optimizator samo treba da je izračuna za novu dodatu tabelu. Na primer, u jednom slučaju, stare tabele mogu da sadrže kolonu čija se vrednost može koristiti za indeksiranje, dok u drugom slučaju ta mogućnost ne postoji, Eto zahteva potpuno skeniranje nove tabele.

Problem koji se ostao jeste pronalaženje najboljeg redosleda spojeva. Za rešavanje ovog problema koriste se dva pristupa: iscrpna pretraga (pozivom funkcije find\_best()) i pohlepna „greedy" pretraga (pozivom funkcije greedy\_search()). Iscrpnom pretragom ispituju se sve moguće kombinacije tabela i pronalazi najbolji plan. Međutim, to može trajati jako dug vremenski period. Pohlepna pretraga funkcioniše tako što isproba sve kombinacije tabela promenljiva konfiguracije servera) od n tabela u upitu i pronalazi najbolju. Prva tabela iz rezultujućeg skupa stavlja se na prvo mesto u delimičnom redosledu spajanja, zatim se ispituju sve moguće kombinacije optimizer\_search\_depth tabela iz preostalih n-1 tabela. Svaka testirana kombinacija dodaje se postojećem delimičnom planu i procenjuje se trošak. Bira se kombinacija koja ima najmanji trošak i njena prva tabela stavlja se u delimičan redosled spajanja. Ovaj proces se ponavlja dok broj preostalih tabela ne bude jednak vrednosti optimizer\_search\_depth.

Oba pristupa su optimizovana na način da prekinu sa pretragom najboljeg redosleda spojeva ako trenutni redosled ima trošak koji prevazilazi najbolje troškove koji su do sada pronađeni. Stoga, u teoriji iscrpna pretraga može da ispita i n! kombinacija, a pohlepna pretraga optimizer\_search\_depth \* (n- optimizer\_search\_depth) kombinacija, ali u praksi su ti brojevi znatno manji. Stoga, iako pohlepna pretraga neće uvek pronaći najbolji plan, ona ima kontrolisanu kompleksnost i imaće prednost u performansama u odnosu na iscrpnu pretragu. Svakako, nije važno da li je pronađeni plan najbolji mogući ako dobitak u vremenu izvršenja nadoknađuje vreme potrošeno za pronalaženje najboljeg redosleda spojeva.

Programeri MySQL-a konstantno rade na poboljšanju optimizatora — pokušavajuéi da ga učine pametnijim i bržim sa svakom novom verzijom. U osnovu povratnih informacija krajnjih korisnika, oni traže načine da usavrše mogućnost MySQL-a da donosi pravu odluku. Zbog toga je bitno da krajnji korisnici prijavljuju upite za koje MySQL donosi loše odluke jer se ne prijavljeni problemi retko otklone.

Najveći deo MySQL-ovih napora za donošenje odluka odnose se na indekse i na redosled spojeva tabela. Ovo nisu jedini faktori, ali su jako bitni. Da bi se bolje razumeo rad MySQL-a kada mu se prosledi SELECT upit, koristi se izlaz EXPLAIN naredbe nad upitom.

## Raw source generator

Generator izvora reda je softver koji od dobijenog optimalnog plana izvršenje proizvodi plan upita, koji je upotrebljiv od strane baze podataka. Iterativni plan je binarni program koji, kada ga izvrši SQL virtuelna mašina, daje skup rezultata. Plan upita predstavlja kombinaciju koraka. Svaki korak vraća skup redova. Redovi u ovom skupu se koriste u sledećim korakom ili se u poslednjem koraku vraćaju aplikaciji. Izlaz iz generatora je plan izvršenja upita.

Nakon što je redosled pridruživanja utvrđen, optimizator počinje da izvršava spoj. Spoj se izvodi putem niza ugneždenih petlji, počevši od prve tabele. Za svaki zapis prve tabele, optimizator se ponavlja kroz drugu da bi kreirao kombinacije. Za svaki zapis u drugoj tabeli, zauzvrat, optimizator prolazi kroz svaki zapis treće i tako dalje i tako dalje, stvarajući kombinaciju zapisa za svaku iteraciju najdublje petlje.

Kombinacija se zatim poredi sa odredbom WHERE upita – ili preciznije optimizovanim izrazom filtera generisanog iz originalne odredbe WHERE. Na primer, ako je odredba WHERE name='Johnson' i age=31+1, izraz filtera postaje name='Johnson' i age=32.Možda se pitate zašto bi iko ikada napisao takvo ograničenje u svom ne optimizovanom obliku. U mnogim aplikacijama upiti se često generišu putem složenih algoritama poslovne logike, koji često proizvode upite koji nisu optimizovani i koje čovek nikada ne bi napisao. Pored toga, prepravljanje upita može da proizvede takav upit kada referenca kolone bude zamenjena konstantom. Stoga trivijalne optimizacije poput one o kojoj se ovde govori često dovode do značajnog dobitka brzine.

Vrednost izraza u WHERE procenjuje se što je ranije moguće. Na primer, ako se neki uslov u WHERE odnosi samo na prvu tabelu, on se procenjuje nakon čitanja reda iz prve tabele i pre nego što se spoji u drugu tabelu (uz pomoć make\_cond\_for\_table( )). Redovi koji odgovaraju ovim ograničenjima se prosleđuju send\_data ( ) objekta za obradu rezultata povezanog sa spojem. Rezultujući objekat obrade može poslati zapise klijentu, napisati ih u datoteku ili privremenu tabelu ili ih proslediti negde drugde radi dalje obrade. Tip objekta obrade rezultata je derivat klase select\_result.

## Izvršenje

MySQL prati plan izvršenja, pribavlja redove iz tabela koje su mu potrebne i po potrebi ih spaja na osnovu relevantnih redova (sa ili bez korišćenja indeksa). U toku izvršenja, može se javiti potreba da se kreira privremena tabela (u memoriji ili na disku), u kojoj će se čuvati rezultati. U trenutku kada su svi redovi obrađeni, MySQL ih šalje klijentu.

U toku izvršenja, MySQL takođe pribavlja informacije i statistiku o svakom izvršenom upitu, čemu spada:

* Ko je izdao dati upit
* Koliko dugo je trajao proces obrade i izvršavanja
* Koliko je redova vraćeno

Ove informacije pojavljuju se u evidenciji sporih upita ako vreme koje oduzima obrada i izvršenje upita prevazilazi prag vremena servera i ako je evidencija omogućena. Ako je upit izdat interaktivno, pojaviće se posle rezultata upita.

# Korišćenje explain naredbe nad upitom

Komanda MySQL EXPLAIN govori optimizatoru da prikaže svoj plan izvršenja upita. Plan izvršenja upita opisuje šta će optimizator uraditi da bi vratio rezultate upita. Na primer, počnite sa tabelom orders: čitanje redova na indeksom payment\_status, za svaki vraćen red iz tabele orders optimizator traži redove u tabeli customer po indeksu id, za svaku kombinaciju (porudžbina,kupac) traži odgovarajuće redove u tabeli product koristeći indeks za product.id, koristeći proizvedenu kombinaciju (order, customer, product) ažurirara privremenu tabelu, i kada završi prolazi kroz privremenu tabelu i preuzima rezultate GROUP BY.

Mnogo toga se može naučiti proučavanjem izlaza EXPLAIN naredbe nad upitom. Izlaz EXPLAIN-a prikazuje redosled tabela u spoju, koji indeksi se u teoriji mogu koristiti, koji indeksi se zapravo koriste i na koji način, da li su neki redovi isključeni ranije koristeći ograničenja odredbe WHERE, procenjenu veličinu svakog podskupa spoja, da li se koriste privremene tabele, da li su zapisi već pročitani u redosledu ključa ili je potrebno dodatno sortiranje za ORDER BY , kao i druge informacije relevantne za optimizaciju upita.

Primer izlaza EXPLAIN narebe:

Uzmimo za primer sledeći upit:

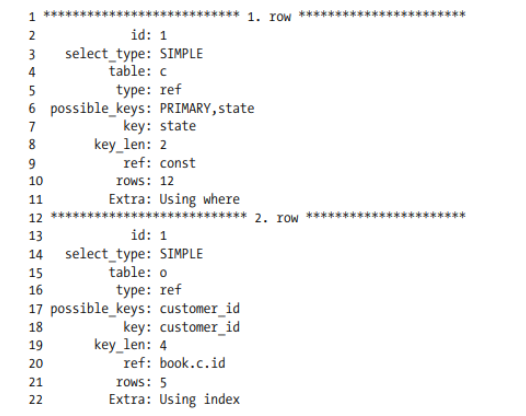
**SELECT count(\*) FROM orders o, customer c WHERE o.customer\_id = c.id AND c.state = 'UT'**

Da bi razumeli plan izvršenja upita, izvršavamo sledeći upit:

**EXPLAIN SELECT count(\*) FROM orders o, customer c WHERE o.customer\_id = c.id AND c.state = 'UT' \G**

Namena karaktera **\G** na kraju upita je da zatraži da lista rezultata bude prikazana vertikalno. Izlaz EXPLAIN naredbe sadrži puno kolona, zbog čega horizontalni prikaz rezultata nije čitljiv.

EXPLAIN naredba generiše sledeći rezultat koji je prikazan na slici ispod.



U rezultatu naredbe, linija broj 4 kazuje da će optimizator krenuti od obrade tabele *customer* (c je pseudonim za tabelu *customer*). Za ovu tabelu, optimizator ima mogućnost da koristi indekse *PRIMARY* i *state* (linija 6), i bira indeks *state* (linija 7). Indeks *state* koristiće se tako sto će dostaviti jednu vrednost a rezultat može sadržati više od jednog reda (linija 5). Optimizator će koristiti prva 2 bajta indeksa, što je u ovom slučaju ceo indeks (linija 8). Vrednost indeksa je konstanta koja je direktno pribavljena iz WHERE klauzule ili se dobija na neki drugi način za razliku od vrednosti neke druge kolone koja može da varira (linija 9). Optimizator procenjuje da će pretraga vratiti 12 redova koji odgovaraju ograničenjima (linija 10). Za redove koji su rezultat, proveravaće se da li odgovaraju ograničenjima definisanim WHERE klauzulom (linija 11).

Linija 15 ukazuje na to da je sledeća tabela za obradu tabela *orders.* Jedini indeks koji se može koristiti za ovu tabelu je customer\_id (linija 17) pa se on i koristi (linija 18). Dužina indeksa koja se koristi je 4 bajta (linija 19) i predstavlja celu dužinu indeksa. Slično kao i za tabelu *customer,* indeks će dostaviti jednu vrednost dok rezultat može biti više redova (linija 16). Međutim, vrednost indeksa više nije konstantan i pribavlja se iz kolone *id* trenutno obrađivane tabele *customer* (linija 20). Vrednost će varirati prema vrednosti kolone u redu koja se trenutno pribavlja iz tabele *customer*. Ova strategija optimizacije moguća je samo ako se tabela *customer* pozicionira pre tabele *orders* u planu obrade tabela. Zbog toga kažemo da tabela *order* zavisi od tabele *customer.*

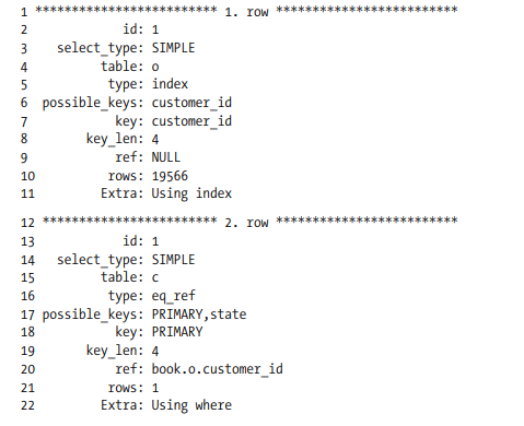
Optimizator procenjuje da, u proseku za svaku kombinaciju tabela koje prethode redosledu spajanja (u ovom slučaju to je samo tabela *customer*) morati da obradi još 5 redova u tabeli *orders*. Kako je optimizatoru dovoljna samo vrednost *customer\_id*, dovoljno je da pročita samo indeks i pri tome ne obradi celu tabelu (linija 22).

Da bi razumeli zašto je optimizator odabrao ovaj plan izvršenja upita, nateraćemo ga da izabere drugi plan izvršenja. Za to ćemo koristiti upit:

**EXPLAIN SELECT count(\*) FROM orders o STRAIGHT\_JOIN customer c**

**WHERE o.customer\_id = c.id AND c.state = 'UT' \G**

Direktiva STRAIGHT JOIN naznačava optimizatoru da tabela *orders* mora doći pre tabele *customer* u bilo kom planu redosleda spojeva koje on razmatra. Ovim se ostavlja jedan jedini redosled spoja tabela a to je: prvo *orders*, onda *customer*. Naredba EXPLAIN ima izlaz dat na sledećoj slici:



U prvom delu izlaza iz EXPLAIN naredbe, vidi se da optimizator skenira indeksom *customer\_id* tabelu *orders* (linije 5 i 7), i procenjuje da će to vratiti 19566 redova (linija 10). Za svaki red rezultat iz tabele *order*, traži se odgovarajući red iz tabele *customer* korišćenjem primarnog ključa (linija 18), i on će vratiti samo jednu vrednost (linija 16). Očekuje se samo jedan red rezultata zato što primarni ključ ima ograničenje da mora da bude jedinstven. Vrednost koja se koristi za pronalaženje reda rezultata *customer* tabele pribavlja se iz kolone *customer\_id* tabele *orders* (linija 20).

Zašto je prvi plan izvršenja bolji od drugog? Optimizator bira plan izvršenja koji ima manju cenu, koji se procenjuje u proporciji sa ukupnim brojem redova koje bi morao da obradi. Procena ukupnog broja redova računa se kao proizvod prosečnog broja redova koje treba obraditi iz svake tabele (vrednost koja se nalazi u *rows* polju izlaza EXPLAIN-a). Po tome, procena ukupnog broja redova koje treba obraditi za prvi plan izvršenja je 12\*4=48 kombinacija, dok se za drugi procenjuje 19566 \*1 =19566 kombinacija.

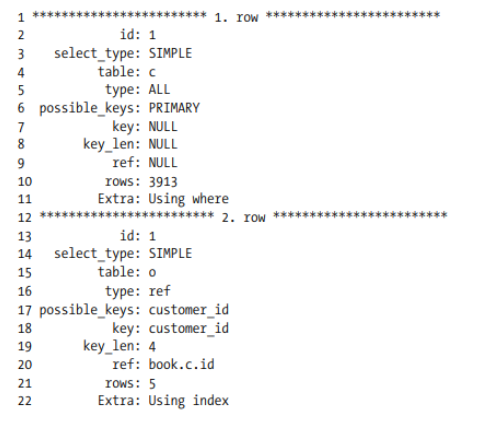
Da razmotrimo koje je još opcije imao optimizator. Zbog prirode WHERE klauzule, postoje dva moguća indeksa koja se mogu primeniti nad tabelom *customer*: oanj nad primarnim ključem i kolonom *state*. Ako pokušamo da optimizujemo malopređašnji JOIN korišćenjem ovog indeksa, upit koji bi koristili bio bi sledeći:

**EXPLAIN SELECT count(\*)**

**FROM customer c FORCE KEY(PRIMARY) STRAIGHT\_JOIN orders o**

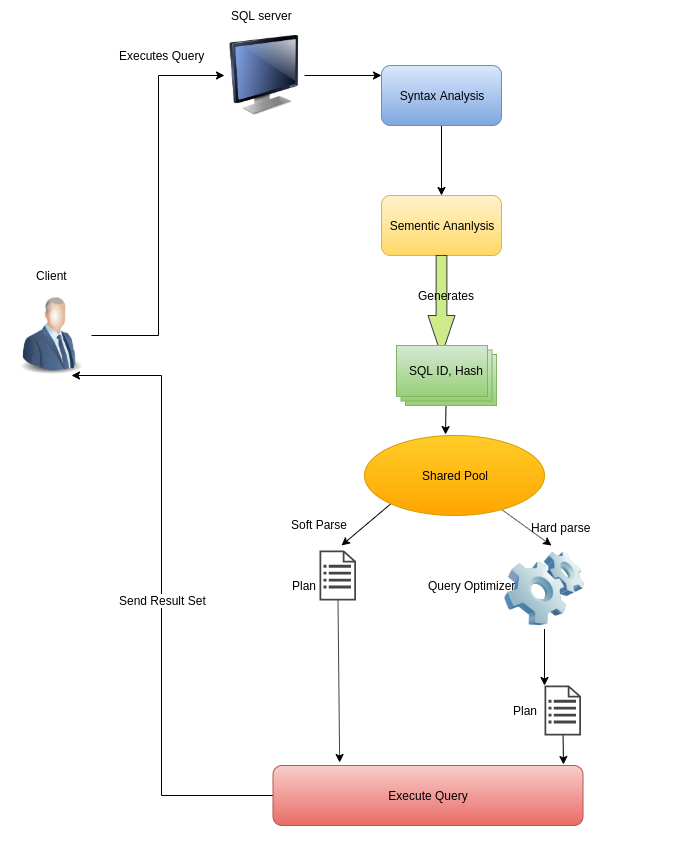
**WHERE o.customer\_id = c.id AND c.state = 'UT' \G**

I izlaz iz EXPLAIN naredbe bio kao na slici ispod.



Optimizator, kada se primora da koristi indeks nad kolonom primarnog ključa, zaključuje da je bolje da skenira celu tabelu *customer* (linija 5). Zaista, ne postojanje referentne vrednosti za indeks, zahtevalo bi obradu cele tabele. Da je bilo moguće da se dobije sve što je potrebno za spajanje i odredbu WHERE koja bi se podudarala sa ključem bez pristupa fajlu podataka, optimizator bi koristio ovaj ključ. Međutim, da bi se proverilo ograničenje za kolonu *state* (state = ‘UT’) optimizator treba da ispita vrednost ove kolone, što čini obradu znatno sporijom u odnosu na klasično skeniranje podataka. Svakako, ukupan broj kombinacija koji se treba obraditi ovim planom izvršenja upita je 3913\*5=19565, što je i dalje mnogo veće od 48 kombinacija u prvobitnom planu izvršenja.

# Zaključak



Osnovna funkcionalnost koju treba imati svaka baza podataka, pa time i MySQL jeste da izvršava upit za kratak (zadovoljavajući) vremenski period. Za koje vreme će se upit izvršiti u velikom delu zavisi od načina njegove obrade i optimizacije samog upita.

U ovom radu zaključili smo da se obrada i opitimizacija (jednim delom) upita oslanjaju na dva važna modula MySQL-a. Uz ova dva modula, keširanje samih upita jako doprinosi ubrzanju izvršenja upita, ako su se oni izvršavali veći broj puta u prošlosti.

Parsiranje upita prima tekstualni unos korisnika, razdvaja ga na tokene i proverava sintaksnu i semantiču ispravnost samog upita. Ukoliko se heš vrednost upita nalazi u kešu baze podataka on se odmah predaje na izvšenje, dok se u suprotnom prosleđuje optimizatoru.

Optimizator je modul koji je zadužen da generiše planove izvršenja upita, odredi trošak svakog od njih i izabere najoptimalniji, uglavnom se to svodi na nalaženje plana koji zahteva obradu najmanjeg broja redova. Plan izvršenja dalje se predaje modulu za izvršenje koje izvršava predati plan.

Dok je proces parsiranja upita jako razumljiv proces (zahteva poznavanje sintakse SQL-a i semantičkih pravila), formiranje plana izvršenja od strane optimizatora može biti jako kompleksno, te se korišćenjem EXPLAIN može pružiti dodatan uvid u logiku koju optimizator koristi kada odabira plan izvršenja.

# Reference

[1]Understanding MySQL internals, Sasha Pachev, glava 9.

[2] [Query Processing Basics http://books.gigatux.nl/mirror/highperfmysql/0596003064/hpmysql-CHP-5-SECT-1.html](http://books.gigatux.nl/mirror/highperfmysql/0596003064/hpmysql-CHP-5-SECT-1.html)

[3]<https://eng.uber.com/queryparser/>

[4]<https://www.exoscale.com/syslog/explaining-mysql-queries/>

[5]<https://logicalread.com/how-mysql-query-processed-mc13/#.YHcuM-gzZPZ>

[6]<https://medium.com/analytics-vidhya/explore-your-recursive-sql-query-f15b120e518e>

1. GNU Flex generiše C kod koji vrši skeniranje tokena koji se može integrisati u već postojeći kod. [↑](#footnote-ref-0)