# 

**Univerzitet u Nišu, Elektronski fakultet**

logo_1960_4**Katedra za računarstvo**

**Optimizacija upita kod MariaDB**

Mila Rančić 1370

**Predmet: Sistemi za upravljanje bazama podataka**

Sadržaj

[Uvod 3](#_Toc101993275)

[Osnove obrade i optimizacije upita 4](#_Toc101993276)

[Šta je optimizacija upita? 5](#_Toc101993277)

[Kako radi optimizator? 6](#_Toc101993278)

[Optimizacija upita kod MariaDB 8](#_Toc101993279)

[Indeksi 10](#_Toc101993280)

[EXPLAIN naredba 11](#_Toc101993281)

[Zašto su upiti spori? 13](#_Toc101993282)

[Optimizacija pristupa podacima 13](#_Toc101993283)

[Da li se od baze podataka traže podaci koji nisu potrebni? 13](#_Toc101993284)

[Primeri čestih grešaka 14](#_Toc101993285)

[Preuzimanje više redova nego što je potrebno 14](#_Toc101993286)

[Preuzimanje svih kolona iz višestrukog join-a 14](#_Toc101993287)

[Preuzimanje svih kolona 14](#_Toc101993288)

[Preuzimanje istih podataka više puta 15](#_Toc101993289)

[Obrada prevelike količine podataka 15](#_Toc101993290)

[Vreme odgovora 15](#_Toc101993291)

[Ispitani redovi i vraćeni redovi 16](#_Toc101993292)

[Pregledani redovi i vsrste pristupa 16](#_Toc101993293)

[Evidencija sporih upita 17](#_Toc101993294)

[Restruktuiranje upita 19](#_Toc101993295)

[Složeni upiti naspram više upita 19](#_Toc101993296)

[Deljenje upita 19](#_Toc101993297)

[Dekompozicija (rastavljanje) join-a 20](#_Toc101993298)

[Tipovi optimizacije kod MariaDB 22](#_Toc101993299)

[Index Condition Pushdown 22](#_Toc101993300)

[Not null range scan optimizacija 25](#_Toc101993301)

[Rowid filtering optimizacija 26](#_Toc101993302)

[Optimizacija podupita 27](#_Toc101993303)

[Table pullout 28](#_Toc101993304)

[Semi-join materijalizacija 30](#_Toc101993305)

[First match 33](#_Toc101993306)

[Zaključak 36](#_Toc101993307)

[Reference 37](#_Toc101993308)

# Uvod

Svi sistemi baza podataka moraju biti u stanju da odgovore na zahtev za informacijama od korisnika - tj. upit. Upit predstavlja način zadavanja komandi sistemu za upravljanje bazom podataka. Ove komande mogu biti čitanje podataka iz memorije, upis novih podataka u memoriju, prepisivanje postojećih ili brisanje.

Svaki upit koji šalje korisnik može se izvršiti na više načina ako pogledamo relacionu algebru. Nekim načinima izvršenja može biti potrebno znatno više vremena da se izvrše, dok drugi mogu biti prilično skupi za sistem. Stoga, postoji potreba da se održi ravnoteža između ta dva parametra. Dakle, imajući u vidu oba faktora, bira se način izvršenja koji će raditi dovoljno brzo, ali i neće mnogo koštati. Upravo se biranje najboljeg načina za izvršenje jednog upita naziva optimizacija upita*.*

Kada je parsiranje završeno i relaciona algebra je obezbeđena, deo softvera koji se naziva *optimizator* poziva sve relevantne statistike o podacima iz baze podataka i donosi odluku nakon što izvede zaključak iz te statistike, odnosno nakon što odredi vreme koje bi bilo potrebno za izvršenje datog upita , i njegovu procenjenu cenu.

Optimizacija upita je u velikoj meri zadatak sistema za upravljanje bazom podataka, ali ipak deo tog zadatka je i na programerima. Usvajajući dobre prakse prilikom pisanja upita, oni sami direktno utiču na to kako će izgledati planovi izvršenja upita i pomažu optimizatoru da brzo i efikasno odabere dovoljno dobar plan. Optimizacija upita, optimizacija indeksa i optimizacija šeme baza podataka idu ruku pod ruku. Sticanjem iskustva u pisanju upita, uči se na to kako se dizajniraju tabele i indeksi za podršku efikasnim upitima. Slično tome, ono što se nauči o optimalnom dizajnu šeme će uticati na to kako se pišu upiti.

Kao sistem u kome će biti urađen praktični deo rada izabran je MariaDB. MariaDB je sistem otvorebog koda za upravljanje bazama podataka baziran na poznatijem MySQL sistemu. Nastao je 2008. godine kao rešenje koje treba da nadomesti nedostatke koje ima MySQL. Poznat je po dobrim performansama, jednostavnosti i pouzdanosti.

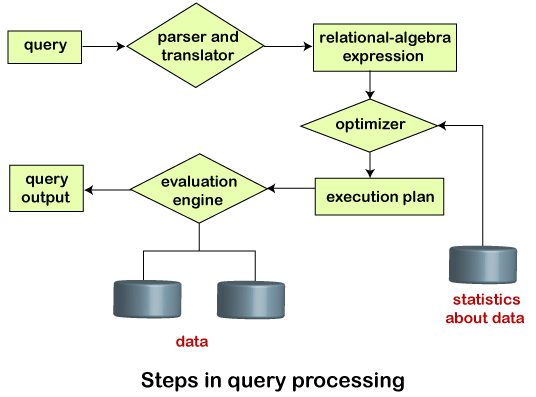
Rad je u globalu organizovan tako da objasni, i na primerima iz realnog sistema pokaže, i optimizaciju koju radi sam sistem za upravljanje bazom podataka, kao i deo koji je na programerima. Prvo poglavlje objašnjava optimizaciju upita u generalnom smislu, a potom, u naredno poglavlje, kako ona konkretno izgleda kod MariaDB-a. Slede dva poglavlja koja se bave “ljudskim” faktorom u optimizaciji upita. Prvo od njih odnosi se na problem sporih upita. U njemu ćemo videti neke od osnovnih problema i najčešćih grešaka koje prave programeri, a znatno utiču na performanse. U drugom ćemo, na praktičnim primerima,  videti neke od najboljih praksi za restruktuiranje upita u cilju optimizacije.  I na kraju, tu je poglavlje koje opisuje deo koji radi DBMS u procesu optimizacije upita. U njemu će biti objašnjene neke od najkorišćenijh strategija, odnosno vida optimizacije,  koje MariaDB koristi. Za svaki vid optimizacije koji se pominje objasnićemo osnovnu ideju i ,ono što je najvažnije, prikazati kako radi na praktičnim primerima.

# Osnove obrade i optimizacije upita

Obrada upita označava kompajliranje i izvršavanje specifikacije upita koja se obično izražava u deklarativnom jeziku upita baze podataka, kao što je SQL. Ona podrazumeva prevođenjee upita na visokom nivou u izraze koji se mogu razumeti na fizičkom nivou, optimizaciju upita i izvršenje upita da bi se dobili stvarni rezultati. Na najvišem nivou obrada upita sastoji se od faze kompajliranja, ili prevođenja, kako se često naziva, i faze izvršavanja. U fazi kompajliranja, prevodilac upita prevodi specifikaciju upita u izvršni program. Ovaj proces prevođenja sastoji se od leksičke, sintaksne i semantičke analize specifikacije upita, kao i faze optimizacije upita i generisanja koda. Generisani kod se obično sastoji od fizičkih operatora koje sistem za upravljanje bazom “zna” kako da izvrši. Ovi operatori implementiraju pristup podacima, spajanja,  selekciju, projekcije, grupisanje i agregaciju. Tokom faze izvršavanja, mehanizam baze podataka tumači i izvršava program koji implementira specifikaciju upita kako bi proizveo rezultat upita.

Najčeće se faza kompajliranja upita deli u fazu parsiranja i fazu optimizacije, tako da se  obrada upita po koracima prikazuje ovako :

1. Parsiranje i prevođenje
2. Optimizacija
3. Izvršenje



Slika - Faze u obradi upita

Dakle tokom prve faze vrši se provera sintakse, kao i semantička provera. Korisnički upit se prvo prevodi u jezik visokog nivoa baze podataka, npr SQL. Zatim, ukoliko je sintaksno ispravan, prevodi se u jezik niskog nivoa koji se može razumeti na fizičkom nivou, za šta se koriste izrazi relacione algebre.

U sledećem koraku se bira najefikasniji plan izvršenja za upit. Tokom analiziranja optimalnosti gleda se koliko je “skupa” svaka od operacija koja će se izvršiti da bi se dobili rezultati i na osnovu toga se formira put koji je najbolji po pitanju brzine, zauzeća memorije, iskorišćenja CPU-a itd. U ovaj fazi se pristupa statistici, odnosno katalozima sistema, kako bi planer imao sve potrebne informacije pre “donošenja” odluke koji je put najbolji. Obzirom da će planer ispitati više mogućih planova (nekad i sve), oni koji su obrađeni se čuvaju u katalozima da bi se na kraju izabrao jedan. Plan se dalje šalje izvršiocu, koji ga obrađuje, izvršava i vraća rezultat.

## Šta je optimizacija upita?

Kada budemo znali odgovor na to pitanje, znaćemo i kada je problem rešen i kada možemo  krenuti dalje, tražeći sledeći. Upiti se često mogu ubrzati, a postoje različiti načini da se to uradi, pri čemu je razlika među njima u utrošenom vremenu i resursima.

Programeri ne mogu uvek da potroše sve potrebne resurse da ubrzaju svoje upite i realno govoreći, ne bi trebalo da to žele. Zbog toga, počećemo sa definisanjem „optimalnog“ kao tačke u kojoj upit ima prihvatljive performanse i zadržava ga pristojan vremenski period.

Ako imamo beskonačne resurse – računarsku snagu, novac, vreme, itd. – možemo da uradimo skoro sve, ali malo nas ima taj luksuz i mora da se pozabavi problemom performansi.

Uz to, imamo nekoliko korisnih kontrolnih tačaka koje će osigurati da kontinuirano procenjujemo napredak tokom optimizacije. Te tačke bi bile:

1. Upit radi adekvatno.

2. Dalja optimizacija je preskupa u smislu resursa.

3. Došli smo do tačke da ako nastavimo sa ulaganjem vremena i resursa u optimizaciju to se neće ogledati u performansama

4. Otkrivamo još jedno rešenje koje čini našu trenutnu optimizaciju nepotrebnom

Mnogi ljudi misle da je preterana optimizacija bolja nego manja, ali to je gubitak što se tiče resursa. Recimo da imamo ogroman indeks pokrivanja koji zaista nije neophodan. Kad god se upisuje u tabelu tokom dužeg vremena, potrebni računarski resursi su ogromni. Ako moramo da prepišemo prethodni kod, to košta u smislu vremena razvoja i QA vremena. A ako pokušamo da prilagodimo upit koji već dobro funkcioniše, možda ćemo dobiti malu količinu optimizacije, ali cena resursa nije vredna toga.

Pre nego što se posvetimo optimizaciji, ovo su pitanja na koja treba odgovoriti:

* Koja je svrha upita?
* Kako bi trebalo da izgledaju rezultati?
* Koja vrsta korisničkog interfejsa, izveštaja ili koda generiše upit?

Uobičajeno je da želimo da se upustimo i borimo sa problemom što je pre moguće. Pokrećemo tragove, postavljamo planove izvršenja i prikupljamo statistiku vremena i IO pre nego što iznenada shvatimo da zapravo ne znamo šta se dešava u upitu. Zato je prvi korak  da razumemo upit. To možemo uraditi postavljenjem i ovih pitanja koja će pomoći u optimizaciji:

* Koja je veličina skupa rezultata? Da li treba da budemo spremni za nekoliko redovi ili milion njih?
* Da li neki parametri imaju ograničene vrednosti? Da li će neki parametar imati istu vrednost sve vreme? Postoje li ograničenja za bilo koju vrednost koje nam mogu pojednostaviti stvari i ukloniti jedno ili više istraživanja?
* Koliko često će se upit izvršavati? Ako radi jednom dnevno, neće biti tretiran isto kao onaj koji se izvršava svake sekunde
* Da li neke neuobičajene ili nevažeće vrednosti ukazuju na problem sa aplikacijom? Da li ste postavili unos na NULL, a ne bi trebalo? Da li postoje vrednosti koje nemaju smisla, kontradiktorne su ili se ne uklapaju u slučaj upotrebe upita?
* Možemo li videti bilo kakvu očiglednu optimizaciju, sintaksni ili logički problem? Možemo li da vidimo nešto što će izazvati a problem, bez obzira na promenljive, kao što su vrednosti parametara?
* Šta smatrate prihvatljivim performansama za upit? Koliko brz treba da bude? Koliko smanjenje upotrebe resursa je potrebno za poboljšanje performansi servera? Kako se upit trenutno ponaša?

Odgovori na ova pitanja često mogu pružiti bolja rešenja. Možda nam taj novi indeks ipak nije potreban ili bi se taj veliki upit mogao podeliti na nekoliko manjih, možda je jedna od vrednosti parametra pogrešna itd.

## Kako radi optimizator?

Poput računara sa ugrađenom veštačkom inteligencijom, koji “znaju” da igraju neke od najpopularnijih igara (šah, go itd) optimizator mora da razmotri ogroman broj poteza za vrlo kratko vreme, da eliminiše loš izbor, tako da ostane najbolji mogući. Može postojati milion kombinacija koje bi optimizator trebalo da razmotri u bilo kom trenutku, a samo će mali broj njih uspeti da prođe. Ako ste igrali šah protiv kompjutera, znaćete da je veća verovatnoća da će pogrešiti ako ima manje vremena za donešenje odluke.

Umesto šahovskih poteza, ovde se radi o planovima izvršenja koji su, kao što znamo, koraci koje mašina za izvršenje prati u obradi upita. Upit će imati više izbora da bi došao do tog plana i samo kratko vreme da to uradi.

Izbori će uključivati ova pitanja:

* Kojim redosledom treba spojiti tabele?
* Koji vrste spajanja su nam potrebni za tabele?
* Koji su indeksi potrebni?
* Da li data tabela zahteva traženje i skeniranje?
* Da li koristiti privremene tabele za rezultate?

Svi planovi izvršenja optimizatora moraju da daju iste rezultate, ali svaki se može razlikovati u performansama, prvenstveno zbog gore navedenih pitanja, zajedno sa ostalima.

Optimizacija upita je neverovatno zahtevna za CPU. Za provođenje planova potrebno je mnogo računarskih resursa, a pronalaženje pravog može potrajati više vremena nego što je dostupno. To znači da se mora postići i održavati ravnoteža između potrebnih resursa za optimizaciju, izvršenja i vremena potrebnog da se ceo proces završi. Stoga, optimizator nije dizajniran da odabere najbolji plan, već da pronađe najbolji nakon što prođe određeni vremenski period.

Metrika koja procenjuje planove i odlučuje šta treba uzeti u obzir naziva se *cena* upita. On nema jedinicu i predstavlja relativnu mera potrebnih resursa za svaki korak plana izvršenja. Ukupna cena je sumirana cena svakog koraka, i oni se mogu videti u bilo kom planu izvršenja. Oni uključuju procenjene troškove operatora, IO, troškove CPU-a i troškove podstabla.

Izračunavaju se troškovi podstabla svakog upita i oni se kasnije koriste za:

* Slična podstabla kako bi uklonili plan izvršenja uz priloženu visoku cenu, iz svih raspoloživih planova
* Rangiranje planova izvršenja koji su ostali u grupi na osnovu njihove cene.

S obzirom da je optimizacija veoma skupa, DB Server ima keš plana izvršenja. Ovde se čuvaju detalji o svim upitima koji se izvršavaju na serveru i koji plan je korišćen. Većina baza podataka ponavlja iste upite, kao što su postovi na društvenim mrežama, web pretrage i tako dalje. Ponovna upotreba plana olakšava i pojeftinjuje optimizaciju nekih upita jer je posao već obavljen.

Cena upita može biti korisna da pomogne da razumemo kako je upit optimizovan, ali moramo zapamtiti da je njegova glavna uloga da pomogne optimizatoru da izabere najbolje planove. Ona ne daje direktnu meru CPU-a, IO-a, trajanja, memorije ili bilo koje druge važne metrike korisniku koji čeka da se upit izvrši.

Kako optimizator prolazi kroz planove, on ih postavlja u red od najveće cene do najniže i na kraju će doći do jednog od ovih zaključaka:

* Svi planovi su sagledani, a najbolji izabrani
* Nema dovoljno vremena da se prođe kroz sve, i bira se najbolji do sada.

Kada je plan izabran, optimizator je završio i možemo preći na završni korak obrade - izvršenje.

# Optimizacija upita kod MariaDB

Kao što je objašnjeno u prethodnom poglavlju, optimizator upita počinje sa radom kad dobije ispravno generisano stablo od parsera.  Takvo stablo treba pretvoriti u plan izvršenja upita. Upit se često može izvršiti na mnogo različitih načina i dati isti rezultat. Posao optimizatora je da pronađe najbolju opciju.

MariaDB koristi optimizator zasnovan na troškovima (*cost based optimizer)*, što znači da pokušava da predvidi cenu različitih planova izvršenja i izabere “najjeftiniji”. Merna jedinica cene je prvobitno bila jedna pročitana stranica sa podacima od 4 KB, ali je postala sofisticiranija i sada uključuje faktore kao što je procenjena cena izvršenja poređenja WHERE uslova. Može se vidieti koliko je optimizator procenio da je upit skup pokretanjem upita, a zatim proverom promenljive sesije *Last\_query\_cost*:

*select count(\*) from api\_informationvalue where id > 5;*

*SHOW STATUS LIKE 'Last\_query\_cost';*



Slika - Rezultat i cena upita

Ovaj rezultat znači da je optimizator procenio da će morati da uradi oko 155 čitanja stranica sa podacima da bi izvršio upit. Procena je zasnovana na statistici poput:

* broju stranica po tabeli ili indeksu,
* kardinalnosti (broj različitih vrednosti) indeksa,
* dužini redova i ključeva i
* distribuciji ključeva.

Optimizator ne uključuje efekte bilo koje vrste keširanja u svoje procene — pretpostavlja da će svako čitanje rezultovati operacijom U/I diska. Pored toga, neće izabrati najbolji mogući plan izvršenja u svakoj situaciji. Razlozi za to su brojni, a neki od njih su:

• Statistika bi mogla biti pogrešna. Server se oslanja na mehanizme za skladištenje podataka da bi obezbedio statistiku, a oni mogu da variraju od potpuno tačnih do veoma netačnih. Na primer, InnoDB mehanizam za skladištenje ne održava tačne statistike o broju redova u tabeli zbog svoje MVCC arhitekture

• Plan koji čita više stranica može zapravo biti jeftiniji u nekim slučajevima, kao što je kada su čitanja uzastopna, ili kada su stranice već keširane u memoriji. MariaDB takođe “ne zna” koje su stranice u memoriji, a koje na disku

• MariaDB zapravo ne pokušava da ubrza upite; pokušava da minimizuje njihovu cenu, a kao što smo videli, određivanje troškova nije egzaktna nauka.

• Ne uzima u obzir druge upite koji se pokreću istovremeno, što može uticati na brzinu pokretanja upita.

• Ne stiže uvek da proceni svaki mogući plan izvršenja, tako da može da propusti najoptimalniji plan.

Optimizator upita je veoma složen deo softvera i koristi mnoge optimizacije da transformiše upit u plan izvršenja. Postoje dve osnovne vrste optimizacija, koje nazivamo *statičkim* i *dinamičkim*. *Statičke* optimizacije se mogu izvršiti jednostavnim pregledom stabla parsiranja. Na primer, optimizator može da transformiše WHERE u ekvivalentan oblik primenom pravila. Statičke optimizacije su nezavisne od vrednosti, kao što je vrednost konstante u WHERE uslovu. Mogu se izvršiti jednom i uvek će važiti, čak i kada se upit ponovo izvršava sa različitim vrednostima. Na njih se može gledati kao na  „optimizacije za vreme kompajliranja“.

Nasuprot tome, dinamičke optimizacije su zasnovane na kontekstu i mogu zavisiti od mnogih faktora, kao što je koja vrednost je u WHERE delu upita ili koliko redova ima u indeksu. Moraju se ponovo proceniti svaki put kada se upit izvrši. Njih zovu i  „optimizacijama za vreme izvršenja“.

Evo nekoliko tipova optimizacija koje MariaDB zna kako da uradi:

* Index Condition Pushdown
* Semi-Join Materialization
* First match
* Duplicate Weedout
* Rowid filtering
* Table pullout

Prethodna lista je nepotpuna, jer MariaDB vrši mnogo više optimizacija, ali bi trebalo da da predstavu o složenosti i inteligenciji optimizatora. Ono najvažnije što treba zaključiti iz ove liste, kao i iz celog ovog poglavlja, je da uvek treba pustiti optimizator da uradi svoj posao, jer će on to najbolje uraditi. Ne treba pokušavati sa nadmudrivanjem optimizatora jer ćemo završiti sa složenijim upitom, težim za održavanjem, bez ikakvog benefita. Mnogo bolji stepen optimizacije postiže se ako naučimo kako optimizator “razmišlja”, šta sve može da uradi i kako da mu u tome pomognemo.

Naravno, koliko god optimizator bio pametan, postoje situacije kada ne daje najbolje rezultate. To su obično slučajevi kada znate nešto o podacima koje optimizator ne zna, kao što su neke činjenice iz same logike aplikacije.

## Indeksi

Pre diskusije o EXPLAIN naredbi i o tome kako MariaDB optimizator bira plan izvršenja, važno je razumeti kako MariaDB koristi indekse. Indeks se može definisati na jednoj ili više kolona i njihov redosled je relevantan. Indeks koji uključuje string kolone može se definisati na njihovim prefiksima (krajnji levi deo podataka). Za kolone TEXT i BLOB indeks je obavezan. Postoje dva važna tipa indeksa: BTREE i HASH. Tu si i ostali tipovi, poput RTREE koji se koristi samo za geometrijske podatke. Indeksi FULLTEXT se koriste za izvršavanje upita za puni tekst.

*HASH* indeks se može koristiti samo za poređenja jednakosti: ona koja uključuju = ili <=> (*NULL-safe equal*) operatore. Ne može se koristiti za sortiranje ili grupisanje redova. Tip *BTREE* se može koristiti sa mnogim operatorima poređenja, kao što su <, <=, =, >=, >, LIKE, BETWEEN i IN. Može se koristiti za sortiranje i grupisanje redova.

Dakle, bilo koja naredba može iskoristiti BTREE indeks,dok se HASH obično može koristiti za upite koji traže specifičnu vrednost. Za takve upite, HASH indeksi mogu biti brži od BTREE indeksa. Ali ono što treba imati na umu je da će za mnogo tipove upita ovi indeksi jednostavno biti zanemareni.

MariaDB koristi krajnji levi deo indeksa. Na primer, ako je indeks definisan u koloni koja se zove col1, upit koji uključuje početni deo kolone obično će iskoristiti prednost indeksa:

*SELECT \* FROM t WHERE col1 LIKE 'begin%';*

Međutim, ako izvršimo upit koji čita samo krajnji desni ili centralni deo indeksa, indeks se neće koristiti. Na primer, sledeći upit neće koristiti indeks:

*SELECT \* FROM t WHERE col1 LIKE '%end';*

Iz istog razloga, ako indeks uključuje više kolona, može ubrzati upite koji koriste krajnje leve kolone u indeksu, ali ne i upite koji ne uključuju krajnji levi deo. Na primer, ako je indeks definisan u dve kolone koje se zovu col1 i col2, sledeći upit može da ga iskoristi ovim redosledom:

SELECT \* FROM t WHERE col1 = 10;

Sa drudge strane, ovaj upit ne može da koristi takav indeks:

SELECT \* FROM t WHERE col2 = 10;

Klauzule ORDER BY i GROUP BY mogu da koriste indeks čak i ako se redosled kolone ne podudara. Međutim, ovo zahteva sortiranje u dva prolaza; podaci će biti kopirani u privremenu tabelu ili datoteku, a zatim poređani. I/O koji zahtevaju takve operacije može znatno uticati na performanse i treba ga izbegavati kad god je to moguće. Na primer, sledeći upit može da koristi naš indeks i ne zahteva dodatne operacije sortiranja:

SELECT \* FROM t ORDER BY col1, col2;

Međutim, kada je ovako napisan upit on zahteva privremenu tabelu ili datoteku:

SELECT \* FROM t ORDER BY col2, col1;

Indeks je obično od velike pomoći ako može da izbegne bilo kakav pristup podacima za upit. Ovo se dešava ako naredba SELECT samo vraća kolone iz indeksa, a sve ostale klauzule mogu da iskoriste prednost iz indeksa. U ovom slučaju se zove indeks pokrivanja (covering indeks).

## EXPLAIN naredba

Naredba EXPLAIN je najbolji način za razumevanje kako se upit izvršava unutar servera. Počeći od MariaDB 10, ovo funkcioniše ne samo sa SELECT naredbama, već i sa naredbama UPDATE i DELETE. Sintaksa naredbe EXPLAIN je:

*EXPLAIN  [EXTENDED] <statement>;*

Nakon MariaDB 10, dodato je još jedno svojstvo ove komande:

*SHOW EXPLAIN FOR <thread\_id>;*

Ova komanda omogućava dobijanje plana izvršenja za naredbu koja se trenutno izvršava. Ovo je korisno kada upit oduzima mnogo vremena i želimo da znamo razlog. Za pregled pokrenutih upita i povezanih ID-eva niti u kojima se oni izvršavaju, može se koristiti SHOW PROCESSLIST.

Na primer, za dobijanje informacija o tome kako je upit izvršen, a zatim i na koji način je “prepisan” potrebno je uraditi sledeće:

EXPLAIN EXTENDED

SELECT tenant\_id, COUNT(\*)

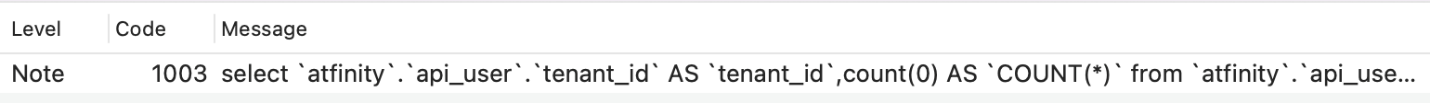
FROM `api\_user`

GROUP BY `tenant\_id`;

, a zatim:

SHOW WARNINGS;

, dobijamo:



Slika - Rezultat komande "Show warnings" koji pokazuje kako je upit prepisan

, gde je sadržaj “Message” kolone:

select `api\_user`.`tenant\_id` AS `tenant\_id`,count(0) AS `COUNT(\*)`

from .`api\_user`

group by `atfinity`.`api\_user`.`tenant\_id`

Jedan red se izlaza EXPLAIN naredbe sastoji se od sledećih kolona:

• *id*: Ova kolona definiše jedinstveni identifikator za svaki red

• *select\_type*: Ova kolona definiše tip komande SELECT

• *table*: Ova kolona definiše tabelu koju čita naredba SELECT

• *partitions*: Ova kolona definiše listu particija kojima će se pristupiti

• type: Ova kolona definiše tip JOIN

• possible\_key: Ova kolona definiše listu ključeva koji se mogu koristiti za izvršenje naredbe

• key: Ova kolona definiše ključ koji optimizator odluči da koristi. Ako je ova vrednost NULL, izvršeno je skeniranje cele tabele

• key\_len: Ova kolona definiše veličinu izabranog ključa, u bajtovima

• ref: Ova kolona definiše kolone koje se koriste za spajanje dve tabele

• rows: Ova kolona daje procenu koliko će redova biti ispitano

• filtered: Ova kolona prikazuje procenu koliko redova NEĆE bit vraćeno iskazano u procentima

• Extra: Ova kolona definiše neke dodatne informacije.

# Zašto su upiti spori?

Kada se o upitima govori iz ugla optimizacije, sve se svodi na vreme koje protekne od pokretanja upita do dobijanja rezultata nazad. Upiti su taskovi i sastoje se od podtaskova i oni imaju svoje vreme izvršenja. Da bismo optimizovali upit, moramo optimizovati i njegove podtaskove tako što ćemo neke od njih eliminisati, neke učiniti da se izvršavaju manje puta ili učiniti da se izvršavaju brže.

Generalno govoreći, na životni vek upita može se gledati prateći njegov dijagram sekvence od klijenta do servera, gde se analizira, planira i izvršava, a zatim ponovo vraća klijentu. Izvršenje je jedna od najvažnijih faza u životu upita. Uključuje mnogo poziva mehanizmu za skladištenje radi preuzimanja redova, kao i operacije koje slede nakon preuzimanja podataka kao što su grupisanje i sortiranje.

Dok izvršava sve ove zadatke, upit provodi vreme na mreži, u CPU-u, u operacijama kao što su statistika i planiranje, zaključavanje (mutex), a verovatno najviše u pozivima mehanizmu za skladištenje radi preuzimanja redova. Ovi pozivi troše vreme na memorijske operacije, CPU operacije, a posebno I/O operacije ako podaci nisu u memoriji.

U svakom slučaju, može se potrošiti previše vremena jer se operacije izvode nepotrebno, izvode previše puta ili su presporo. Cilj optimizacije je da se to izbegne, eliminisanjem ili smanjenjem broja operacija, ili njihovim ubrzanjem.

## Optimizacija pristupa podacima

Osnovni razlog zašto upit ne radi dobro je taj što radi sa previše podataka. Neki upiti jednostavno moraju da obrade mnogo podataka i ne može im se pomoći. To, ipak, nije toliko čest slučaj. Većina loših upita se može promeniti da bi se pristupilo manjem broju podataka. Upite koji imaju loš učinak korisno je analizirati u dva koraka:

1. Saznati da li aplikacija preuzima više podataka nego što je potrebno. To obično znači da pristupa velikom broju redova, ali takođe može da pristupa prevelikom broju kolona.

2. Saznati da li server analizira više redova nego što mu je potrebno.

## Da li se od baze podataka traže podaci koji nisu potrebni?

Neki upiti traže više podataka nego što im je potrebno, a zatim neke od njih odbacuju. Ovo zahteva dodatni rad servera, dodaje kašnjenje zbog mreže i troši memoriju i CPU resurse na serveru.

## Primeri čestih grešaka

### Preuzimanje više redova nego što je potrebno

Jedna od najčešćih grešaka je pretpostavka da MariaDB daje rezultate na zahtev. Ona zapravo izračunava i vraća ceo skup rezultata. Programeri naviknuti na neke druge sisteme baza podataka mogu očekivati drugačije ponašanje, kao što je recimo situacija kada je od N redova potrebno pribaviti samo prvih deset i prekinuti izršenje nakon što se oni pribave. Međutim, ono što će se zapravo desiti je da će klijentska biblioteka pribaviti sve redove i da će odbaciti veliku većinu. Najbolje rešenje za to je korišćenje LIMIT klauzule.

### Preuzimanje svih kolona iz višestrukog join-a

Ako želimo da preuzmemo sve glumce koji se pojavljuju u filmu Forrest Gump, ne treba napisati upit na ovaj način:

SELECT \* FROM actor

     INNER JOIN film\_actor USING(actor\_id)

     INNER JOIN film USING(film\_id)

     WHERE film.title = Forrest Gump';

To vraća sve kolone iz sve tri tabele. Umesto toga, ono što smo hteli možemo da dobijemo ako napišemo upit na sledeći način:

SELECT actor.\* FROM actor…. .;

### Preuzimanje svih kolona

Uvek treba da biti sumnjičav kada je tu SELECT \*. Da li su zaista potrebne sve kolone? Verovatno ne. Preuzimanje svih kolona može sprečiti optimizacije kao što su *covering* indeksiranje i izazvati dodavanje I/O operacija, memorije i dodatno opterećenje CPU-a. Naravno, nije uvek loše tražiti više podataka nego što je zaista potrebno. U mnogim slučajevima takav pristup pojednostavljuje razvoj, jer omogućava programeru da koristi isti deo koda na više od jednog mesta. To je razumno razmatranje, sve dok se vodi računa o tome koliko to ugrožava performanse. Takođe, moglo bi biti korisno preuzeti više podataka nego što je zaista potrebno ako se koristi neka vrsta keširanja u aplikaciji. Pribavljanje i keširanje celih objekata može biti poželjnije od pokretanja velikog broja pojedinačnih upita koji preuzimaju samo delove objekta.

### Preuzimanje istih podataka više puta

Čest je slučaj da se napiše kod koji više puta preuzima iste podatke sa servera baze podataka. Na primer, za dobijanje URL-a slike profila korisnika koja će se prikazati pored liste komentara, upit se može izvršavati više puta, za svaki komentar. S druge stranei, rezultat bi mogao biti keširan prvi put kada je dobijen i ponovo upotrebljen nakon toga. Jasno je da je drugi pristup mnogo efikasniji.

## Obrada prevelike količine podataka

Ukoliko upiti preuzimaju samo podatke koji su potrebni, naredni korak u cilju optimizacije je traženje upita koji ispituju previše podataka dok generišu rezultate. Kod MariaDB, najjednostavniji pokazatelji za to su:

* + Vreme odziva
  + Broj pregledanih redova
  + Broj vraćenih redova

Nijedna od ovih metrika nije savršen način za određivanje “cene” upita, ali oni otprilike odražavaju kojoj količini podataka server mora interno da pristupi da bi izvršio upit i približno oslikavaju koliko brzo se upit izvršava. Sve tri metrike se evidentiraju u evidenciji sporih upita, tako da je sam pregled evidencije sporih upita jedan od najboljih načina da se pronađu upiti koji ispituju previše podataka.

### Vreme odgovora

Vreme odgovora se sastoji od dve stvari: vreme usluge (*service time)* i vreme čekanja (*queue time*). Vreme usluge predstavlja koliko dugo je serveru potrebno da zaista obradi upit. Vreme čekanja je deo celokupnog vremena odgovora upita tokom kog server zapravo ne izvršava upit već čeka. Može čekati da se završi I/O operacija, zaključavanja reda i tako dalje. Problem je u tome što ne možete razbiti vreme odgovora na ove komponente osim ako ih ne možete izmeriti pojedinačno, što je obično teško uraditi. Uopšteno govoreći, najčešća i najvažnija čekanja su čekanja na I/O i zaključavanja.

Faktori, koji takođe mogu imati značajan uticaj na vreme odgovora su zaključavanja mehanizama za skladištenje (zaključavanja tabele i zaključavanja redova), konkurentnost i hardver. Kao rezultat toga, vreme odgovora nije dosledno pod različitim uslovima.

Kada se pogleda vreme odgovora na upit, trebalo bi se zapitati da li je vreme odgovora razumno za taj konkretni upit.

### Ispitani redovi i vraćeni redovi

Korisno je razmisliti o broju redova koji se ispituju kada se analiziraju upiti, jer se može videti koliko efikasno upiti pronalaze podatke koji su potrebni. Međutim, ovo nije savršena metrika za pronalaženje „loših“ upita. Nisu svi pristupi redovima jednaki. “Kraćim” redovima se brže pristupa, a čitanje redova iz memorije je mnogo brže od čitanja sa diska.

U idealnom slučaju, broj pregledanih redova bi bio isti kao i vraćeni broj, ali u praksi je to retko moguće. Na primer, kada konstruišete redove sa *join* naredbama, server mora da pristupi većem broju redova da bi generisao svaki red u skupu rezultata. Odnos pregledanih redova i vraćenih redova je obično mali – recimo, između 1:1 i 10:1 – ali ponekad može biti nekoliko redova veličine veći.

### Pregledani redovi i vsrste pristupa

Kada se razmišlja o ceni upita, u obzir treba uzeti cenu pronalaženja jednog reda u tabeli. MariaDB može da koristi nekoliko metoda pristupa da pronađe i vrati red. Neki zahtevaju ispitivanje mnogo redova, dok bi drugi mogli da generišu rezultat bez ispitivanja ijednog.

Metod(i) pristupa se pojavljuju u koloni *type* u izlazu EXPLAIN-a. Tipovi pristupa se kreću od potpunog pretraživanje tabele (*full table scan*) do pretraživanje indeksa (*index scan*), pretraživanje opsega (*range scan*), jedinstvenog pretraživanja indeksa (*unique index lookups*)  i konstanti (*constants*). Svaki metod je brži od prethodnog, jer zahteva čitanje manje podataka. Ne moraju se pamtiti svi tipovi pristupa, ali treba razumeti koncepte pretraživanje tabele, pretraživanje indeksa, opsega i pristup jednoj vrednosti.

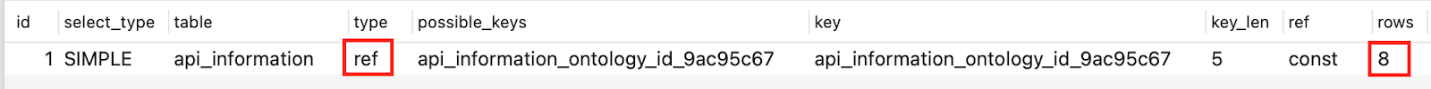
Ako rezultat ne prikaže željeni tip pristupa, najbolji način da se to reši je obično dodavanjem odgovarajućeg indeksa. Indeksi omogućavaju MariaDB da pronađe redove sa efikasnijim tipom pristupa koji ispituje manje podataka.

Na primer, jednostavan upit u bazi podataka:

SELECT \* from `api\_information` WHERE ontology\_id=2;

Ovaj upit vratiće 8 redova i EXPLAIN pokazuje da MariaDB koristi *ref* tip pristupa nad indeksom za izvršenje upita:

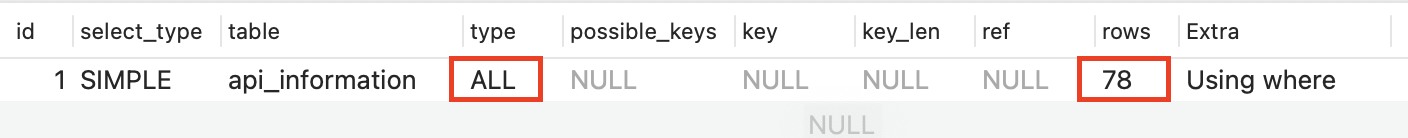
EXPLAIN SELECT \* from `api\_information` WHERE ontology\_id=2;



Slika - Izvršenje upita SA korišćenjem indeksa

EXPLAIN prikazuje da je MariaDB server procenio da je broj redova kojima mu je potrebno da pristupi jednak 8. Drugim rečima, optimizator je “znao” da odabrani tip pristupa može efikasno obezbediti rezultate. Šta bi se desilo da nije bilo indeksa? Bila bi korišćena manje optimalna strategija pristupa koja se može videti u sledećem primeru:

EXPLAIN SELECT \* from `api\_information` WHERE information\_type='string';



Slika - - Izvršenje upita BEZ korišćenjem indeksa

Kao što je očekivano, strategija pristupa je sada *full table scan* - *ALL* i sada je procenjeno da je broj redova kojima treba pristupiti 78.

## Evidencija sporih upita

Log sporih upita (Slow query log) skladišti SQL naredbe za koje je potrebno predugo da se izvrše. Da bi se on uključio potrebno je podesiti promenljivu slow\_query\_log na 1, ili opciju pri pokretanja --slow\_query\_log, takođe na 1. Suprotno, za potpuno isključicanje ovog vida evidencije, opcija --slow-query-log mora biti podešena na 0. Ukoliko nema prosleđenog argumenata podrazumeva se da je evidencija uključena. Pošto je promenljiva slow\_query\_log dinamička, evidencija sporog upita može biti omogućena ili onemogućena čak i u toku faze izvršavanja. Podrazumevano ime datoteke u koju se čuvaju podaci o sporim upitima je ime servera iza kojeg sledi sufiks *-slow.log*.

Baš kao i kod evidencije upita uopšte, spori upiti mogu se čuvati u vidu datoteke, tabele ili kombinovanjem oba. Promenljiva *log output* može se koristiti da odredi da li je to tabela ili datoteka. Dozvoljene vrednosti koje treba postaviti su FILE, TABLE i NONE (što onemogućava i spori dnevnik upita i opšti dnevnik upita). Dozvoljena je kombinacija ovih vrednosti (odvojenih zarezom).

Pravila koja određuju koji su upiti zapisani u evidenciji zavise od nekih sistemskih varijabli servera.  Prvo, možda želimo da evidentiramo upite koji uopšte ne koriste indekse. To se radi tako što se podesi promenljiva log\_queries\_not\_using\_indexes (ili --log\_queries\_not\_using\_indexes startup opcija) na 1 (podrazumevana vrednost je 0). Ova promenljiva je dinamička, ali postoji samo na globalnom nivou.

Čak i ako upit koristi indekse, možda ćemo želeti da ga evidentiramo ako pređe vremensko ograničenje. Ovo se može uraditi preko promenljive long\_query\_time ili opcije pokretanja --long\_query\_time. Ako je vrednost 0, nema vremenskog ograničenja i spori upiti se ne evidentiraju. Ova vrednost se izražava u sekundama. Decimalne vrednosti su dozvoljene sa preciznošću mikrosekundi (do šest decimalnih cifara), međutim, za dnevnike zasnovane na tabeli, decimalni deo se zanemaruje. Vreme izvršenja upita se računa od trenutka kada nit dobije sve neophodne lock-ove. To znači da, ako upit traje previše vremena samo zato što je blokiran sporim upitom, on se ne evidentira. Ova promenljiva je dinamička i postoji i na globalnom i na nivou sesije. Ovo omogućava konekcijama koje izvršavaju složenije upite da postave duže vremensko ograničenje.

Upit može imati i druge probleme koje bismo možda želeli da evidentiramo. Klase problema koji se evidentiraju navode su u promenljivoj log\_slow\_filter, odvojene zarezom. Dozvoljene vrednosti su opisane na sledeći način:

• *full\_scan*: Ova vrednost znači da je upit izvršio kompletno skeniranje tabele (isto kao *log\_queries\_not\_using\_indexes*)

• *full\_join*: Ova vrednost znači da je upit izvršio operaciju pridruživanja, ne koristeći indekse pri tome

• *filesort*: Ova vrednost znači da je upit izvršio operaciju sortiranja, što zahteva internu, privremenu tabelu u memorij

• *filesort\_on\_disk*: Ova vrednost znači da je upit izvršio sortiranje operacija, koja zahteva privremenu datoteku na disku

• *tmp\_table*: Ova vrednost znači da je upit kreirao privremenu tabelu

• *tmp\_table\_on\_disk*: Ova vrednost znači da je je implicitno kreirana privremena tabela zapisana na disku

• query\_cache\_miss: Ova vrednost znači da je izvršena pretraga upita, ali on nije pronađen u kešu

Čak i ako upit odgovara kriterijumima definisanim sistemskim varijablama *log\_queries\_not\_using\_indexes* i *long\_query\_time*, promenljiva *min\_examined\_row\_limit* može sprečiti njegovo evidentiranje. Upiti koji su ispitali manje od opisanog broja redova nikada se ne upisuju u log sporog upita. Vrednost 0 omogućava da se svi upiti upisuju. Maksimalna vrednost zavisi od platforme, ali je uvek veoma visoka. Ova promenljiva je dinamička i postoji i na globalnom i na nivou sesije.

Ovo podešavanje je obično korisno jer, ako upit ispituje samo nekoliko redova, ni u kom slučaju ne bi imao koristi od korišćenja indeksa (a optimizator to uzima u obzir). Ako upit prekorači vremensko ograničenje, još uvek je moguće da veliki broj redova nije razlog za to. Primer za to, ako upit zahteva složene operacije sortiranja ili grupisanja, ili ako poziva spore funkcije (uključujući store-ovane procedure).

# Restruktuiranje upita

Kod optimizacije problematičnih upita cilj bi trebalo da bude pronalaženje alternativnih rešenja za dobijanje željenih rezultata u smislu performansi. To ne mora nužno da znači vraćanje istih rezultata iz baze podataka. Ponekad je moguće transformisati upite u ekvivalentne forme koji daju iste rezultate i postižu bolje performanse. Međutim, trebalo bi razmisliti i o ponovnom pisanju upita ako se time dobija na efikasnosti, čak i ako bi se time dobili različiti rezultati. U ovom delu videćemo neke tehnike koje mogu pomoći u restruktuiranju širokog spektra upita.

## Složeni upiti naspram više upita

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

Međutim, ovaj savet se ne odnosi toliko na MariaDB sistem, jer je dizajniran da veoma efikasno upravlja povezivanjem i prekidanjem veze i da veoma brzo odgovara na male i jednostavne upite. Moderne mreže su takođe znatno brže nego što su bile, što smanjuje kašnjenje mreže.

Odgovor na vezu je i dalje spor u poređenju sa brojem redova koje MariaDB može interno da pređe u sekundi, što se broji u milionima u sekundi za podatke u memoriji. Sve ostalo je jednako, i dalje je dobra ideja da koristite što manje upita, ali ponekad možete učiniti upit efikasnijim tako što ćete ga razložiti i izvršiti nekoliko jednostavnih upita umesto jednog složenog. Nemojte se plašiti da to uradite; odmerite troškove i idite sa strategijom koja uzrokuje manje posla. Prikazaćemo neke primere ove tehnike malo kasnije u poglavlju.

Međutim, korišćenje previše upita je uobičajena greška u dizajnu aplikacije. Na primer, neke aplikacije izvode 10 upita koji vraćaju po jedan red da bi preuzele podatke iz tabele kada bi mogle da koriste jedan upit od 10 reda.

## Deljenje upita

Drugi način za raščlanivanje upita je po principu “podeli pa vladaj”, zadržavajući ga u suštini istim, ali se izvodi u manjim „komadima“ koji utiču na manje redova svaki put. Čišćenje starih podataka je odličan primer. Periodični poslovi čišćenja će možda morati da uklone dosta podataka, a to bi u jednom velikom upitu moglo da zaključa mnogo redova na duže vreme, popuni evidencije transakcija, resurse i blokira male upite koji ne bi trebalo da se prekidaju. Seckanje naredbe DELETE i korišćenje upita srednje veličine može značajno poboljšati performanse i smanjiti kašnjenje. Na primer, umesto pokretanja ovog monolitnog upita:

*DELETE FROM messages*

*WHERE created < DATE\_SUB(NOW(),INTERVAL 3 MONTH);*

možete da uradite nešto poput sledećeg pseudokoda:

*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*

Takođe bi mogla biti dobra ideja da dodate malo vremena (sleep) između naredbi DELETE da biste rasporedili opterećenje tokom vremena i smanjili količinu vremena zaključavanja.

## Dekompozicija (rastavljanje) join-a

Mnoge aplikacije koje imaju visoke performanse koriste strategiju razdvajanja join-ova u upitu. To se može postići pokretanjem više upita za jednu tabelu umesto spajanja sa više tabela, a zatim izvršiti spajanje u aplikaciji. Na primer, umesto ovog upita:

*SELECT \* FROM role*

*JOIN user\_role ON user\_role.role\_id=role.id*

*JOIN user ON user\_role.user\_id=user.id*

*WHERE user.username=mila;*

  Može se uraditi sledeće:

*SELECT \* FROM user WHERE usename='mila';*

*SELECT \* FROM user\_role WHERE role\_id=1234;*

*SELECT \* FROM role WHERE role.id in (1,2,3,4,5);*

Ovo na prvi pogled izgleda pogrešno, jer je povećan broj upita, a isti je efekat u smislu rezultata. Međutim, takvo restrukturiranje zapravo može dati značajne prednosti performansi:

* + Keširanje može biti efikasnije. Mnoge aplikacije keširaju „objekte“ koji se direktno mapiraju u tabele. U ovom primeru, ako je objekat sa username poljem “mila” već keširan, aplikacija može preskočiti prvi upit. Ako pronađete role sa ID-om 1, 2 ili 3 u kešu, možete ih ukloniti sa IN() liste. Keš upita bi takođe mogao imati koristi od ove strategije. Ako se samo jedna od tabela često menja, dekomponovanje join-a može smanjiti broj promašaja keša.
  + Pojedinačno izvršavanje upita ponekad može smanjiti sukob oko zaključavanja.
  + Izvođenje join-a u aplikaciji olakšava skaliranje baze podataka postavljanjem tabela na različite servere.
  + Sami upiti mogu biti efikasniji. U ovom primeru, korišćenje IN() liste umesto spajanja omogućava MariaDB-u da sortira ID-ove redova i preuzima redove optimalnije nego što bi to bilo moguće sa spajanjem. Kasnije ćemo ovo detaljnije objasnili.
  + Možete smanjiti suvišne pristupe redovima. Izvođenje spajanja u aplikaciji znači da preuzimate svaki red samo jednom, dok je spajanje u upitu u suštini strategija koja može više puta pristupati istim podacima. Iz istog razloga, takvo restrukturiranje takođe može smanjiti ukupan mrežni saobraćaj i upotrebu memorije.
  + U izvesnoj meri, ovu tehniku možete posmatrati kao ručnu implementaciju heš join-a umesto algoritma ugnežđenih petlji koji MariaSB koristi da izvrši spajanje.

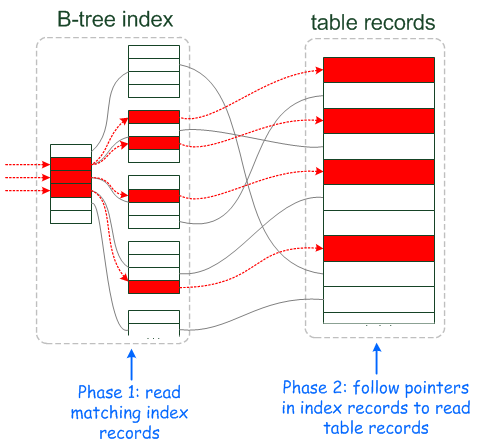
Kao rezultat toga, spajanje u aplikaciji može biti efikasnije kada keširate i ponovo koristite mnogo podataka iz ranijih upita, distribuirate podatke na više servera, zamenite spajanja sa IN() listama na velikim tabelama ili se spajanje odnosi na isti sto više puta.

# Tipovi optimizacije kod MariaDB

## Index Condition Pushdown

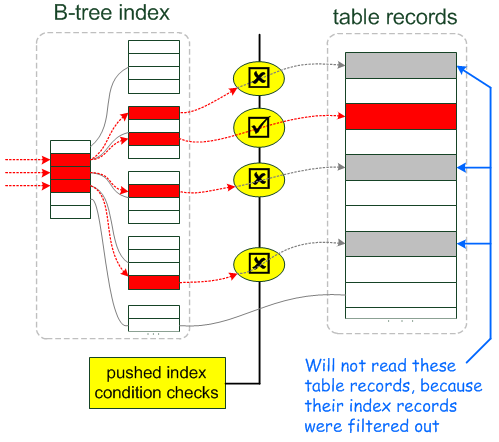
Index Condition Pushdown je tip optimizacije koji se primenjuje kada je tip za pristup jedan od onih koji se zasnivaju na korišćenju indeksa:  range, ref, eq\_ref, ref\_or\_null, and [Batched Key Access](https://mariadb.com/kb/en/block-based-join-algorithms/#batch-key-access-join). Ideja je proveriti deo WHERE uslova (*Pushed Index Condition*) koji se odnosi na kolone sa indeksom onog trenutka kada se pristupi samom indeksu. Ako provereni uslov nije zadovoljen, nema potreba za čitanjem tog celog reda iz tabele kasnije.

Mehanizmi za skladištenje podataka pretragu pomoću indeksa obavljaju iz dva koraka, kao što je prikazano na slici:



Slika - Korišćenje indeksa pri čitanju podataka

Cilj Index Condition Pushdown-a je da smanji broj pristupa celokupnim redovima proveravajući da li indeks redovi zadovoljavaju uslov iz WHERE klauzule koji se odnosni na njih:



Slika - Korišćenje indeksa pri čitanju podataka sa implementiranim ICP-om

Koliko se dobije na brzini zavisi od toga koliki je broj čitanja koje će ova strategija “odseći”, kao i od toga kolika je cena svakod od čitanja.

Sada ćemo ovaj vid optimizacije videti na delu, na primeru realnog upita. Pomoću EXPLAIN naredbe analiziraćemo izvršenje upita:

*SELECT \* FROM `api\_information`*

*WHERE `api\_information`.id IN*

*(SELECT information\_id*

*FROM `api\_informationvalue`*

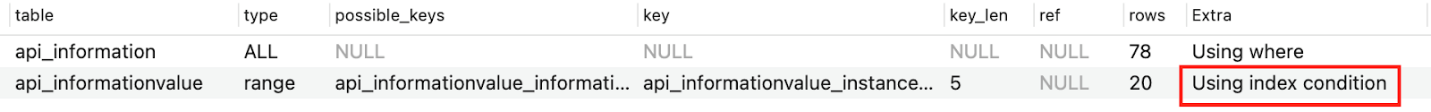
*WHERE instance\_id<3);*

Ono što će nam prvo biti od interesa je da vidimo da li je optimizator primenio *Index Condition Pushdown.* To se radi proverom *Extra* kolone izlaza, gde bi trebalo da stoji ***Using index condition.*** Nakon toga treba proveriti koliko smo zapravo dobili koriščenjem ove strategije. Jedan od načina za to je provera dve promenljive servera:

* [Handler\_icp\_attempts](https://mariadb.com/kb/en/server-status-variables/#handler_icp_attempts) : pokazuje koliko je puta pushed index condition proveren
* [Handler\_icp\_match](https://mariadb.com/kb/en/server-status-variables/#handler_icp_match): pokazuje koliko je puta uslov pushed index condition zadovoljen

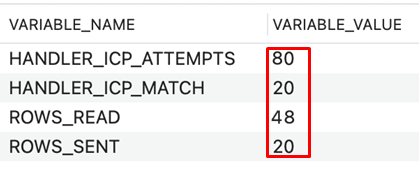
Drugim rečima , “attempts” je broj koliko puta je server poslao WHERE klauzulu na proveru da li može biti odsečena, a “match” broji koliko puta je taj pokušaj završen vraćenim redom. Veliki iznos za (attempts - match) znači da je optimizacija veća, odnosno da je više čitanja ušteđeno.

Dakle, za početak proveramo izlaz EXPLAIN-a:



Slika - Izlaz naredbe EXPLAIN za definisani upit

i potvrđujemo da je došlo do korišćenja ICP-a. Nakon toga proveravamo promenljive *Handler\_icp\_attempts* i *Handler\_icp\_match* :

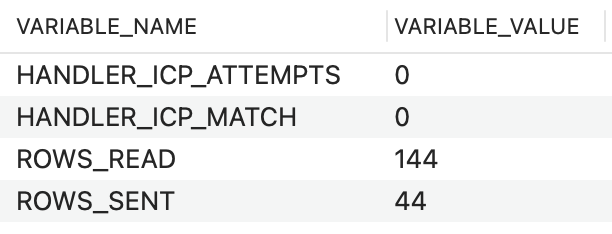


Slika - Status promenljivih

Da bismo videli koliki je stepen optimizacije, isključićemo ICP naredbom:

*SET optimizer\_switch='index\_condition\_pushdown=off'*

i pokrenuti prethodno definisani upit. Još jednom, proveravamo promenljive:



Slika - Status promenljivih bez korišćenja ICP-a

Na osnovu dobijenih rezultata vidljivo je da je korišćenjem ICP-a broj redova koji se čitaju smanjen u velikoj meri.

## Not null range scan optimizacija

Not null range scan omogućava optimizatoru da izvede skeniranje opsega pomoću NOT NULL uslova koji “izvuče” iz WHERE uslova. Ovaj vid optimizacije dostupan je od MariaDB verzije 10.5.0 i potrebno ga je uključiti (inicijalno je isključen). Za demonstraciju kako not\_null\_range\_scan radi pogledaćemo sledeći upit:

*EXPLAIN*

*SELECT api\_informationvalue.\**

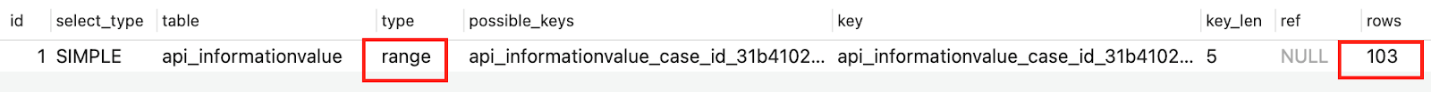
*FROM*

*api\_informationvalue*

*WHERE*

*`case\_id` < 2*

, kao i njegov izlaz:



Slika - Izlaz naredbe EXPLAIN za definisani upit

Iako je očekivano da se uradi pretraga cele tabele, odnosno da vrednost polja *type* bude *ALL,* možemo videti da je optimizator ipak bio u mogućnosti da odabere daleko optimalniji plan koji podrazumeva *range scan*. To je uradio tako što je iz uslova “*WHERE `case\_id` < 2”* izvukao dodatni uslov koji podrazumeva da *`case\_id`* polje mora da ima vrednost, odnosno NUT NULL uslov.

Ako za primer uzmemo upit za koji se ne koristi ova strategija, poput:

*EXPLAIN*

*SELECT api\_informationvalue.\**

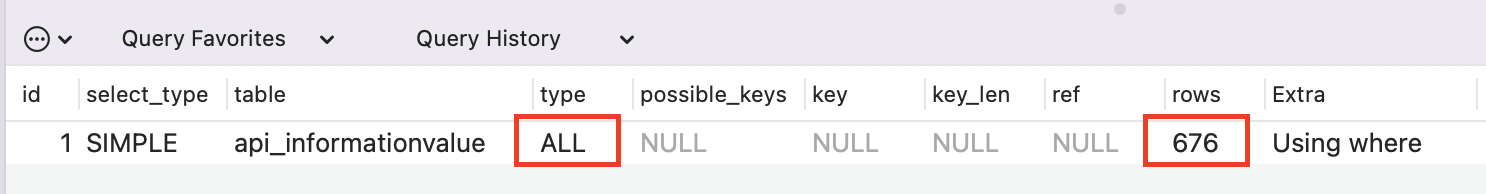
*FROM*

*api\_informationvalue*

*WHERE*

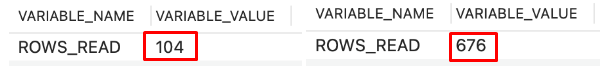
*`is\_touched`=1*

, dobijamo:



Slika - Izlaz EXPLAIN naredbe bez korišćenja NOT NULL SCAN optimizacije

, što nam govori da je procenjeno da će biti potrebno pročitati mnogo više redova. I zaista, ako proverimo realan broj pročitanih redova kod jednog, pa zatim kod drugog upita dobijamo:



Slika - Broj pročitanih redova za definisane upite

Broj pročitanih redova je višestruko manji, čime se direktno utiče na to da se upit izvrši na mnogo optimalniji način

## Rowid filtering optimizacija

U slučajevima kada postoji upit koji ima osobine:

* tabela koristi *ref* pristup nad indeksom IDX1,
* postoji restriktivni range predikat nad drugim indeksom IDX2,

dolazi do primene *rowid filtering* strategije. U takvim slučajevima preporučljivo je:

* uraditi indeks skeniranje nad indeksom IDX2 i prikupiti id-eve svih indeks rekorda u neku strukturu koju omogućava filtriranje,
* kada se radi *ref* pristup nad IDX1 proveriti strukturu pre čitanja celog rekorda

Ako za primer uzmemo upit:

*FROM api\_information JOIN api\_informationvalue*

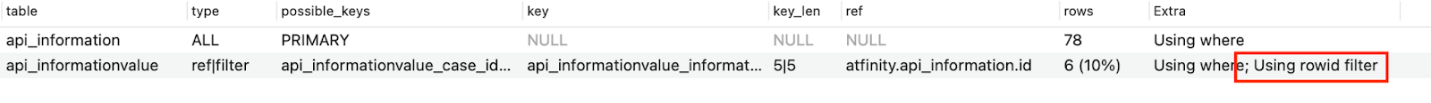
*ON api\_information.id = api\_informationvalue.information\_id*

*WHERE api\_information.is\_system\_information=1*

*AND api\_informationvalue.case\_id between 2 and 200;*

Recimo da je uslov nad tabelom *api\_information* veoma restriktivan (u našem primeru *api\_information.is\_system\_information=1)*, to bi značilo da tabela *api\_informationvalue* treba da bude prva u *join* delu. U tom slučaju bi optimizator mogao da koristi jednakost *api\_information.id = api\_informationvalue.information\_id* da uradi indeks pretragu kako bi došao do vrednosti *informationvalue za information.*

Kada takav upit analiziramo sa EXPLAIN dobijamo:



Slika - Izlaz naredbe EXPLAIN za definisani upit

Pored toga što se vidi da je ova strategija zaista iskorišćeno, ovde se može videti da je tabela api\_information zaista obrađena prva, kao i da kolona type ima deo "|filter" i da je očekivana selektivnost 10%. Ovde je odluka o korišćenju filtera donešena na osnovu cene izvršenja (*cost-based decision)*, na osnovu procenjene selektivnosti, koja je izračunata pomoću statistike o tabeli. Zbog toga što je procenio da će mali broj redova iz tabele učestvovati u krajnjem rezultatu, optimizator će ih izdvojiti u posebnu strukturu, u MariaDB to jeje sortirani niz ID-eva, i raditi sa njom umesto sa celom tabelom, što će višestruko popraviti performanse upita.

## Optimizacija podupita

MariaDB ima skup strategija, odnosno različite vidove optimizacije, koji se odnose na podupite. Dobar deo, tačnije njih 5, vezan je za *semi-join* upite.

Semi-join podupit ima oblik:

SELECT ... FROM outer\_tables

WHERE expr IN (SELECT ... FROM inner\_tables ...)

AND ...

Podupit je IN podupit i nalazi se u WHERE delu. Najvažnija stvar ovde je da smo kod semi-join podupita zainteresovani samo za u rekorde iz spoljne tabele koji imaju svoje redove sa kojima se poklapaju u podupitu. Za primer pogledaćemo sledeći upit.

SELECT \* FROM Country

WHERE Continent='Europe'

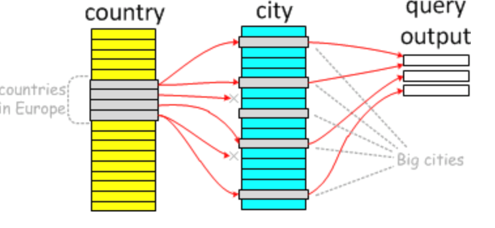
AND Country.Code IN

(SELECT City.country

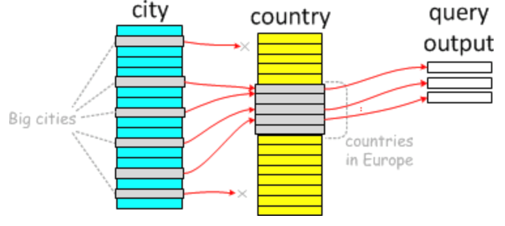
       FROM City

       WHERE City.Population>1\*1000\*1000);

Ovde je moguće krenuti od država (Country) iz Evrope i proveravati da li one imaju “velike” gradove. Semi-join omogućava i obruti pristup: krenuti od “velikih” gradova i videti kojim državama oni pripadaju.



Slika - Semi Join podupit



Slika - Isti upit izvršen u drugačijem reodsledu

MariaDB koristi 5 različitih strategija za izvršenje *semi-join* podupita:

* Table pullout
* FirstMatch execution strategy
* Semi-join Materialization execution strategy
* LooseScan execution strategy
* DuplicateWeedout execution strategy

### Table pullout

U teoriji relacionih upita, *semi-join* je SELECT naredba koja sadrži podupit u WHERE delu. U većini slučajeva takve podupite je bolje izvršiti pomoću JOIN naredbe koja izdvaja samo kolone iz tabele. Ako za primer pogledamo sledeći upit.:

*SELECT AVG(price) FROM product WHERE cat\_id =*

*(SELECT id FROM category WHERE id = 42);*

Isti rezultat možemo dobiti i pomoću JOIN upita:

*SELECT AVG(p.price)*

*FROM product p*

*LEFT JOIN category c*

*ON p.cat\_id = c.id*

*WHERE c.id = 42;*

MariaDB optimizacija poznata kao ***table pullout*** sastoji se od prevođenja *semi-join* upita u potpuni *JOIN*. Ako se ova optimizacija ne primeni, upit može biti veoma spor. *Table pullout* nije podržano u verzijama MariaDB starijih od 5.3. U modernim verzijama MariaDB-a, ako je podupit spor, trebalo bi da proverimo da li ga optimizator prepiše kao JOIN. Ako se to ne desi, treba da pokušamo da to uradimo sami. Kao demonstraciju kako ovo radi na konkretnom pimeru koji se izvršava na serveru, pogledaćemo upit:

*SELECT \**

*FROM api\_informationvalue*

*WHERE api\_informationvalue.information\_id in*

*(SELECT api\_information.id*

*FROM api\_information*

*WHERE api\_information.id < 10000);*

Ako se zna da postoji samo jedna informacija sa datim id-em (što se može proveriti time da li je kolona primarni ključ ili da li postoji indeks nad njom), upit se može napisati i u ovom obliku:

*SELECT api\_informationvalue.\**

*FROM*

*api\_informationvalue, api\_information*

*WHERE*

*api\_informationvalue.information\_id=api\_information.id*

*AND*

*api\_information.id < 10000;*

Ako izvršimo EXPLAIN za prethodno definisani upit u verzijama MariaDB pre 5.3, ili isključimo optimizator podupita pomoću naredbe

*SET @@optimizer\_switch='semijoin=off'*

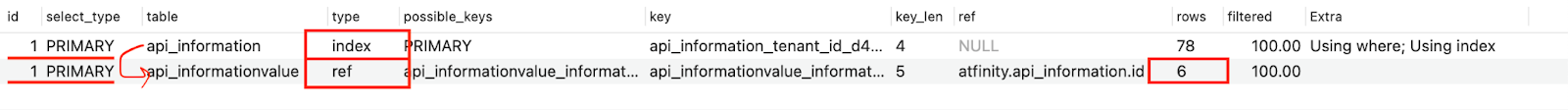
, dobićemo:

**

Slika - Izlaz naredbe EXPLAIN za definisani upit BEZ korišćenja TABLE PULLOUT-a

Izlaz naredbe pokazuje da urađena potpuna pretraga *informationvalue* tabele i za svaki red iz te table po jedna *lookup* pretraga nad tabelom *information*.

Sa druge strane, za verzije iznad 5.3 ili sa uključenim optimizatorom podupita dobijamo:

**

Slika - Izlaz naredbe EXPLAIN za definisani upit SA korišćenjEM TABLE PULLOUT-a

Sada vidimo da obe tabele sa *select\_type* imaju *PRIMARY*, kao i to da je redosled zamenjen, te da je prvo obrađena information, a posle nje informationvalue tabela. Ako se pokrene i EXPLAIN EXTENDED i SHOW WARNINGS, vidi se da je podupit zaista zamenjen JOIN-om:

select `api\_informationvalue`.`id` AS

,`api\_informationvalue`.`value` AS

`value`,`api\_informationvalue`.`secondary\_value` AS `secondary\_value`,`api\_informationvalue`.`deprecated` AS `deprecated`,`api\_informationvalue`.`case\_id` AS `case\_id`,`api\_informationvalue`.`information\_id` AS `information\_id`,`api\_informationvalue`.`instance\_id` AS `instance\_id`,`api\_informationvalue`.`tenant\_id` AS

`tenant\_id` from `api\_information` **join** `api\_informationvalue` where `api\_informationvalue`.`information\_id` = `api\_information`.`id`

and `api\_information`.`id` < 10000

Restruktuiranje upita na ovakav način, tj promena iz podupita u JOIN, omogućila je optimizatoru da bira između dve moguće JOIN strategije:

1. information -> informationvalue
2. informationvalue -> information

, nasuprot jedinom izboru informationvalue -> information. Zbog toga dolazi do odabira boljeg plana izvršenja. Bez *pullout* mehanizma podupit bi obradio (712 + 78\*712) = 56248 redova. Sa njim, to je (78 + 78\*6) = 546 redova. Iako nisu svi redovi isti, kao što nije isto čitanje različitih redova, razlika u broju pročitanih redova je prevelika.

### Semi-join materijalizacija

Ovaj vid optimizacija uključuje dve strategije:

* Materialization/lookup
* Materialization/scan

Da bismo videli koja je ideja pogledaćemo opet upit koji pronalazi evropske zemlje sa velikim gradovima:

SELECT \* from Country

WHERE Country.code IN

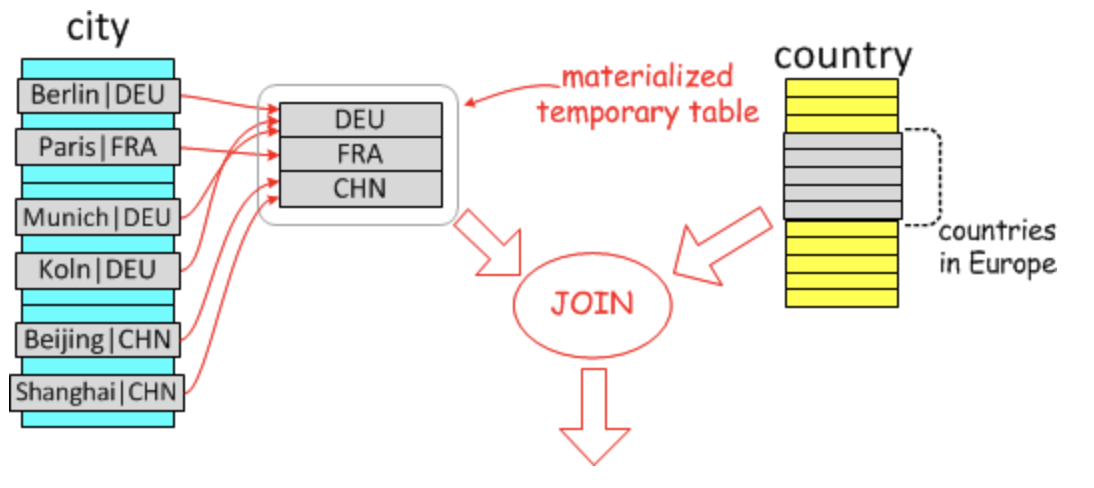
(SELECT City.Country

FROM City

WHERE City.Population > 7\*1000\*1000)

and Country.continent='Europe'

Ovde je podupit posebna celina i može se pokrenuti nezavisno od spoljašnjeg upita. Upravo je to polenta Semi-join Materialization strategije - pokrenuti podupit i popuniti privremenu, odnosno *materijalizovanu,* tabelu sa mogućim vrednostima City.country polja svih velikih gradova i nakon toga uraditi join sa državama iz Evrope.



Slika - Izvršenje JOIN-a kada je kreirana privremena tabela

Ovde JOIN može biti izveden u dva smera:

1. Od materijalizovane tabele ka tabeli “country”
2. Od tabele “country” ka materijalizovanoj tabeli

Prvi način podrazumeva potpunu pretragu privremene tabele, otud ime "Materialization-scan". Ako se JOIN izvodi u smeru od “country” najbolji način za to je kreirati *lookup* nad primarnim ključem, zbog čega se takav način zove "Materialization-lookup".

Sve do sada rečeno, potrebno je pokazati i na konkretnom primeru. Kada se upit:

*SELECT \* FROM `api\_information`*

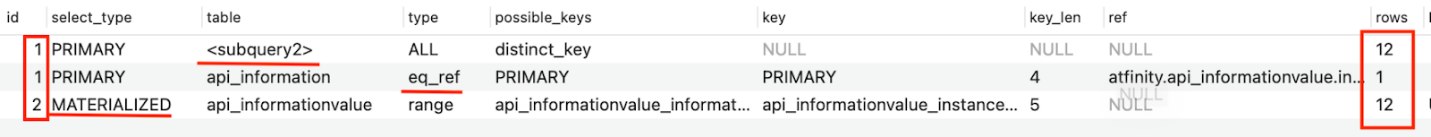
*WHERE `api\_information`.id IN*

*(SELECT information\_id*

*FROM `api\_informationvalue`*

*WHERE instance\_id<3);*

analizira sa EXPLAIN, dobija se:



Slika - Izlaz naredbe EXPLAIN za definisani upit

Ovde se može videti:

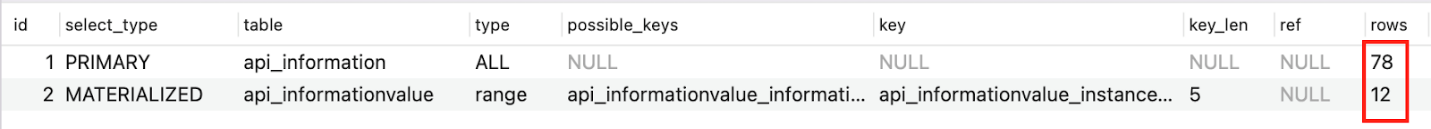
* Postoje dve SELECT dela (id-evi kolona su 1 i 2)
* Drugi SELECT (id=2) ima select\_type=MATERIALIZED. To znači da će se taj deo izvršiti i da će rezultati biti sačuvani u privremenoj tabeli sa primarnim ključem nad svim kolonama. Primarni ključ je kreiran na taj način da bi se sprečilo postojanje duplikata

Optimizator je izabrao da uradi potpunu pretragu nad materijalizovanom tabelom, pa je zbog toga ovo primer Materialization-Scan strategije.

Što se tiče cene, rade se 12 čitanja iz *api\_informationvalue* tabele, 12 upisa u materijalizovanu tabelu, ponovo njihova čitanja (pretpostavka je da neće biti duplikata), i zatim 12 *eq\_ref* pristupa tabeli *api\_information*. To je ukupno 36 čitanja i 12 upisa. Da bismo videli u kojoj meri je ovaj upi optimizovan, pokrenućemo ga još jednom, ali bez korišćenja strategije materalizacije. To se radi na sledeći način:

optimizer\_switch='semijoin=off,materialization=off'

Kao izlaz EXPLAIN naredbe sada se dobija:



Slika - Izlaz naredbe EXPLAIN za definisani upit bez korišćenja Semi Join Materialization-a

Cena je  sada (78 + 78\*12) = *1014* čitanja. Vidimo da je to poboljšanje od skoro 30 puta.

Ako sada prepravimo upit, i umesto redova koji pripadaju instancama sa id-evima 1 i 2, vratimo redova za sve instance sa id-em manjim od 5:

*SELECT \* FROM `api\_information`*

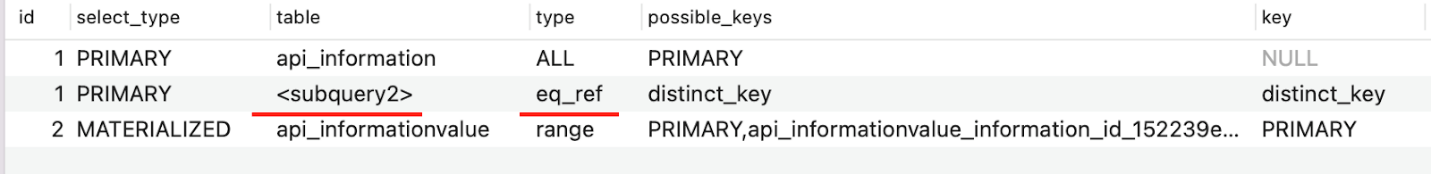
*WHERE `api\_information`.id IN*

*(SELECT information\_id*

*FROM `api\_informationvalue`*

*WHERE instance\_id<5);*

i analiziramo ga, dobijamo:



Slika - Izlaz naredbe EXPLAIN za definisani upit

Izlaz EXPLAIN naredbe je sličan kao u prethodnom primeru, osim što:

* <subquery2> tabeli se pristupa *eq\_ref* metodom
* Za to se koristi indeks po imenu *distinct\_key*

To znači da optimizator planira da kreira indeks nad materijalizovanom tabelom, tj da koristi Materialization-lookup strategiju.

### First match

Još jedan veoma važan vid optimizacije podupita kod MariaDB je FirstMatch. On se koristi u okviru IN podupita gde jedan takav podupit može prestati sad izvršenjem onog trenutka kada dođe do željenog reda. Znak da je korišćen ova strategija je kada u Extra koloni EXPLAIN naredbe vidimo FirstMatch(tableNumber). Recimo da imamo upit koji treba da pronađe sve države iz Evrope koje imaju bar jedan “veliki” grad:

SELECT \* from Country

WHERE Country.code IN

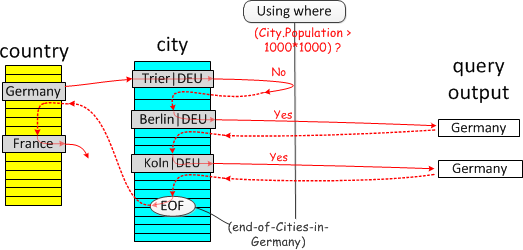
(select City.Country

            FROM City

            WHERE City.Population > 1\*1000\*1000)

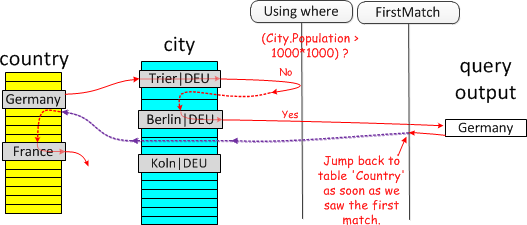
  AND Country.continent='Europe'

Plan izvršenja upita bi mogao da bude takav da se prvo pronađu sve evropske zemlje, a zatim da se  za svaku od proveri da li imaju grad sa populacijom većom od izabrane vrednosti. Izvršenje jednog takvog inner join-a izgledalo bi ovako:



Slika – Izvršenje upita bez FirstMatch optimizacije

Obzirom da “Germany” ima više od jednog velikog grada, odnsno na ovom primeru dva, ona će se više puta naću u izlazu upita. To nije dobro jer SELECT …… from COUNTRY ne bi trebalo da daje isti red više puta. *FirstMatch* strategija sprečava dupliranje redova u rezulatu tako što prekida izvršenje podupita čim pronađe jedan red koji zadovoljava uslov.



Slika - Izvršenje upita SA FirstMatch optimizacijom

Ono na šta treba obratiti pažnju je da je prekid izvršen tek nakon što je primenjen uslov iz “Using where” dela.

Sada ćemo videti kako *FirstMatch* izgleda na primeru upita koji radi nad bazom podataka koju koristimo kao deo praktičnog dela ovog rada. Definisaćemo upit koji vraća sve korisnike koji su meta admini i kreirali su bar jedno informaciju koja nije sistemska i pokrenuti ga pomoću explain extended naredbe.

*EXPLAIN EXTENDED SELECT \**

*FROM api\_user*

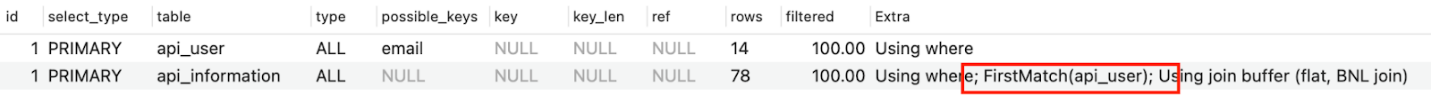
*WHERE api\_user.email in*

*(SELECT api\_information.\_created*

*FROM api\_information*

*WHERE api\_information.is\_system\_information=0)*

*AND api\_user.`access\_level`='META\_ADMIN';*



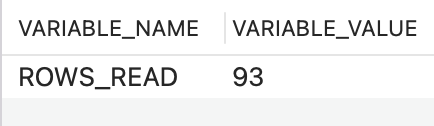
Slika - - Izlaz naredbe EXPLAIN za definisani upit za primer FirstMatch

U Extra koloni možemo videti da je za izvršenje ovog upita zaista korišćena *FirstMatch* strategija. Da bismo videli koliko smo ubrzanje postigli proverićemo koliki je broj pročitanoh redova. To radimo pomoću:

SELECT \* FROM information\_schema.SESSION\_STATUS

WHERE VARIABLE\_NAME='ROWS\_READ' ;

, a kao rezultat dobijamo:



Slika - Broj pročitanih redova za definisani upit

Bez korišćenja FirstMatch strategije ovaj broj bi bio (14 + 14\*78) = 1106, što znači da smo broj smanjili za više od 90%.

# Zaključak

Optimizacija upita predstavlja jedan od najvažnijih koncepata rada sa bazom podataka.  Ovaj proces je komplikovan, sastoji se iz faza, zavisi od mnogo faktora i nije ga lako u potpunosti razumeti. Podrazumeva i deo koji je na programerima, kao i na strategije i procese koji se izvršavaju u samom DBMS-u.

Osim što smo objasnili šta podrazumeva i kako funkcioniše, koliko je optimizacija važna videli smo i na primerima. Prikazano je nekoliko najboljih praksi za restruktuiranje upita u cilju postizanja boljih performansi, kao i nekoliko osnovih i najčešće korišćenih tipova optimizacije kod MariaDB. Njih je bitno znati jer na taj način znamo i šta možemo da očekujemo od baze i kako da joj pomognemo da izvrši upit na najoptimalniji način.

Postoji još mnogo faktora koji utiču na optimizaciju, koje nije bilo moguće obraditi u okviru ovog rada. To su strategije poput podešavanja maksimalnog broja konekcija, veličine pool bafera, veličine keša upita (query cache), maksimalne veličine tabele u memoriju (Max\_heap\_table\_size) itd. Svaki od ovih parametara može da utiče ,pre svega, na brzini kojom se upit izvršava, ali je manje zanimljivo demonstrirati ih na primerima, što je dodatni razlog zbog kog nisu obrađeni u ovom radu.

Dakle, objasniti sve aspekte optimizacije upita bi zahtevalo mnogo više od jednog rada. Ono što je bitno izvući kao pouku je da je ona veoma bitan, i neizbežan deo rada sa bazom, da postoji mnogo faktora koji utču na nju koji se odnose na sve faze upita, počeći od pisanja, do vraćanja rezultata, kao i to da Što bolje poznajemo kako optimizacija funckioniše to ćemo biti u boljoj poziciji za rad sa bazom.

# Reference

[1] Federico Razzoli , “Mastering MariaDB”, Packt Publishing, 2014

[2] Pierre Mavro , “MariaDB High Performance”, Packt Publishing, 2014

[3] IRussell J.T. Dyer, “Learning MySQL and MariaDB\_ Heading in the Right Direction with MySQL and MariaDB”, O'Reilly Media, 2015

[4] MariaDB Query Optimizations - <https://mariadb.com/kb/en/query-optimizations/>

[poslednji pristup 20.04.2022.]

[5] MariaDB Handler\_icp\_% Counters: What They Are, and How To Use Them   ​​       <https://www.percona.com/blog/2017/05/09/mariadb-handler_icp_-counters-what-they-are-and-how-to-use-them/#comment-10968069>

[poslednji pristup 20.04.2022.]

[6] Query Optimizer in MariaDB 10.4 <https://mariadb.org/wp-content/uploads/2019/03/optimizer-in-10.4-mariadb-unconf2018.pdf>

[poslednji pristup 23.04.2022.]

[7] 13 Tips for Tuning and Optimizing Mysql and Mariadb Databases

https://vitux.com/tune-and-optimize-mysql-mariadb

[poslednji pristup 24.04.2022.]