****

**Seminarski rad**

**Predmet:**

**Sistemi za upravljanje bazama podataka**

**Tema: Optimizacija MySQL upita**

**Doc. dr Aleksandar Stanimirović Student: Vesna Stojanović 1339**

**Sadržaj:**

1. Uvod 3
2. Saveti za optimizaciju upita 3
3. Odgovornosti Query Optimizer-a 3
4. Složen upit ili više jednostavnih upita 4

# Osnove izvršenja upita 5

* 1. MySQL klijent/server protokol 6

## [Stanja upita](#_2s8eyo1) 7

## [Keš upita](#_17dp8vu) 7

## Proces optimizacije upita 8

## Parser i preprocesor 8

## [Optimizator upita](#_lnxbz9) 8

1. Transformacije upita 9
2. Optimizacija zasnovana na troškovima 11
3. Osnove: Single Table SELECT 11
4. Table Join Order  12
5. Cena upita 13
6. Join Algoritmi 14
   1. Nested loop join 14
   2. Block nested join 16
   3. Hash join 18
7. Zaključak 21
8. [Literatura](#_41mghml) 22

## 

## 

## 

**1.Uvod**

Kada podnesete upit MySql-u za izvršenje, to nije tako jednostavno kao samo čitanje

podatka i vraćanje podataka. Za jednostavne upite koji zahtevaju sve podatke iz jedne tabele, nema mnogo opcija kako da preuzmete podatke. Međutim, većina upita je kompleksnija – neki su mnogo složeniji – i izvršavanje upita tačno onako kako je napisano nije najefikasniji način da se dođe do rezultata.

Glavni posao optimizatora je da pripremi upit za izvršenje i odredi optimalni plan upita. Prva faza uključuje transformaciju upita sa ciljem da se taj upit može izvršiti po nižoj ceni od originalnog upita. Druga faza se sastoji od izračunavanja cene na različite načine na koje upit može biti izvršen i određivanje najjeftinije opcije.Query Optimizer je zasnovan na potrosnji.

**2.Saveti za optimizaciju upita**

1. Indeksirati sve kolone koje se koriste u klauzulama Where“, „Order by“ i „Group by“
2. Optimizovati svoju šemu baze podataka
3. Normalizovati tabele
4. Izbegavati nulte vrednosti
5. Izbegavati previše kolona
6. Optimizovati join-e

Još neki od saveta su:

1. Izbegavati korišćenje funkcija u predikatima
2. Izbegavati korišćenje wildcard-a(%) na početku predikata
3. Izbegavati nepotrebne kolone u klauzuli SELECT
4. Koristiti unutrašnje spajanje umesto spoljašnjeg ako je moguće
5. Koristiti DISTINCT i UNION samo ako je neophodno
6. Klauzula ORDER BY je obavezna u Sql-u ako očekujete da ćete dobiti sortirani rezultat

**3.Odgovornosti Query Optimizer-a**

* Pretvaranje sub-optimalnih join tipova u efikasne celine
* Preuredjenje join tabela
* Redukcija constant expression-a
* Optimizacija algebarskih pravila
* Optimalno koriscenje indeksa
* Sortiranje optimizacija
* Optimizacija agregatskih funkcija

## **4.Složen upit ili više jednostavnih upita**

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

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

Vrste optimizacija koja MySQL zna kako da odradi:

* Redosled join-a: Tabele ne moraju uvek da se spajaju u redosledu koji je specificiran u upitu. Određivanje najboljeg redosleda spajanja je važna optimizacija, koja će biti kasnije u radu detaljnije objašnjena.
* Konverzija OUTER JOIN-a u INNER JOIN: OUTER JOIN ne mora da se izvršava kao OUTER JOIN. Neki faktori, kao što su WHERE klauzula i šema tabele, mogu da izazovu da OUTER JOIN bude ekvivalentan INNER JOIN-u. MySQL može da prepozna ovo i da ponovo napiše upit.
* Primena pravila algebarskih ekvivalencija: MySQL primenjuje alegebarske transformacije za pojednostavljenje i kanonizaciju izraza. On takođe može da vrši redukciju konstanti, eliminiše nemoguće konstante i konstantne uslove. Na primer, izraz (5=5 AND a>5) će se redukovati na samo a>5. Slično, (a<b AND b=c) AND a=5 postaje b>5 AND b=c AND a=5.
* Optimizacija COUNT(), MIN(), MAX() naredbi: Indeksi i kolone sa null vrednošću mogu često pomoći MySQL-u da optimizuje ove naredbe. Na primer, za pronalaženje minimalne vrednosti kolone koja je krajnje levo u B-Tree indeksu, MySQL može samo da zahteva prvi red u indeksu.

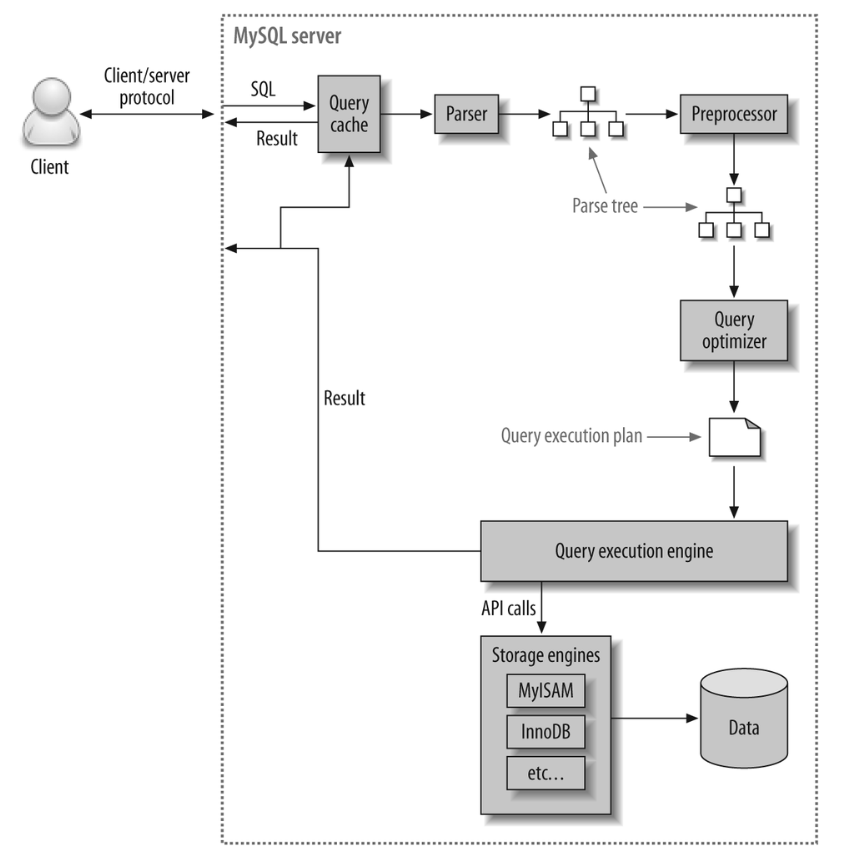
On može da radi ovo čak i u fazi optimizacije upita, a da vrednost smatra kao konstantnu za ostatak upita. Slično, da bi pronašao maksimalnu vrednost u B-Tree indeksu, server čita poslednji red. Ako server koristi ovu optimizaciju, možemo videti „Select tables optimized away“ u EXPLAIN planu. Ovo znači da je optimizator uklonio tabelu iz plana upita i zamenio je sa konstantom. Isto tako, COUNT(\*) upiti bez WHERE klauzule često se mogu optimizovati na nekim mehanizmima za skadištenje podataka.

# **5.Osnove izvršenja upita**

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

Slika 1 prikazuje kako MySQL izvršava upite.

1. Klijent šalje SQL naredbu serveru.
2. Server proverava keš upita. Ako je pogodak, vraća sačuvane rezultate iz keša, inače, prosleđuje SQL naredbu u sledeći korak.
3. Server parsira, preprocesira i optimizuje SQL u plan izvršenja upita.
4. Mehanizam za izvršenje upita izvršava plan upućivanjem API poziva mehanizmu za skladištenje podataka.
5. Server šalje rezultat klijentu.



Slika 1: Osnove izvrsenja upita

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

## **5.1.MySQL klijent/server protokol**

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

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

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

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

## **5.2.Stanja upita**

Svaka MySQL konekcija, ili nit, ima stanje koje pokazuje šta se radi u datom trenutku. Postoji nekoliko načina da vidite ova stanja, ali najlakši je da koristite komandu SHOW FULL PROCESSLIST (stanja se pojavljuju u koloni Command). Kako upit napreduje kroz svoj životni ciklus, njegovo stanje se menja više puta. Mi ćemo ovde izlistati nekoliko stanja i objasniti šta ona znače:

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

## **5.3.Keš upita**

## 

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

# **5.4.Proces optimizacije upita**

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

## **5.5.Parser i preprocesor**

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

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

## **5.6.Optimizator upita**

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

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

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

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

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

**6.Transformacije upita**

Način na koji čovek smatra da je prirodno da napiše upit možda nije isti kao optimalan način da se taj isti upit izvrši u MySql-u. Optimizator zna za nekoliko transformacija koje se mogu koristi za promenu upita dok se i dalje vraća isti rezultat, tako da upit postaje optimalniji za MySql.

Naravno, najvažnije je da originalni i prepisani upiti vraćaju isti rezultat. Relacione baze podataka su zasnovane na matematičkoj teoriji skupova, dakle

mnoge transformacije mogu koristiti standardna matematička pravila koja obezbeđuju to da dve verzije upita vraćaju isti rezultat. Jedna od najjednostavnijih vrsta transformacija koje optimizator može da obezbedi je propagacija(propagation).

Kao primer, pogledaćemo sledeći upit:

SELECT \*

FROM world.country

INNER JOIN world.city

ON city.CountryCode = country.Code

WHERE city.CountryCode = 'AUS';

Ovaj upit ima dva uslova: kolona city.CountryCode mora biti jednaka „AUS“, a kolona Country Code u tabeli city mora biti jednaka koloni Code. U country tabeli. Iz ta dva uslova može se izvesti zaključak da country. Code kolona takođe mora da bude jednaka „AUS“. Optimizator koristi ovo znanje za filtriranje country tabele direktno. Pošto je kolona Code primarni ključ tabele country, to znači da optimizator zna da će postojati samo jedan red koji odgovara uslovu, i optimizator može tretirati tabelu country kao konstantu. Ustvari, upit na kraju bude izvršen sa vrednostima kolona iz tabele country kao konstante na selektovanoj listi i skenira unos u tabeli city sa CountryCode = 'AUS':

SELECT 'AUS' AS `Code`,

'Australia' AS `Name`,

'Oceania' AS `Continent`,

'Australia and New Zealand' AS `Region`,

7741220.00 AS `SurfaceArea`,

1901 AS `IndepYear`,

18886000 AS `Population`,

79.8 AS `LifeExpectancy`,

351182.00 AS `GNP`,

392911.00 AS `GNPOld`,

'Australia' AS `LocalName`,

'Constitutional Monarchy, Federation' AS `GovernmentForm`,

'Elisabeth II' AS `HeadOfState`,

135 AS `Capital`,

'AU' AS `Code2`,

city.\*

FROM world.city

WHERE CountryCode = 'AUS';

Ovo je sigurna transformacija sa tačke gledišta performansi. Ostalo transformacije su komplikovanije i ne poboljšavaju uvek performanse. Iz tog razloga se može konfigurisati da li je optimizacija omogućena ili ne. Konfiguracija se vrši korišćenjem opcije optimizer\_switch i optimizer hints. Kada optimizator odluči koje transformacije da uradi, treba da odredi kako izvršiti prepisani upit.

**7.Optimizacija zasnovana na troškovima**

MySql koristi optimizaciju upita zasnovanu na troškovima. To znači da optimizator

izračunava cenu za različite operacije potrebne za izvršenje upita, a zatim kombinuje

ove delimične troškove za izračunavanje ukupne cene upita za moguće planove upita, i

bira najjeftiniji plan.

**8.Osnove: Single Table SELECT**

Principi izračunavanja troškova su isti bez obzira na upit, ali očigledno što je upit složeniji, procena troškova postaje složenija. Kao jednostavan primer, razmotricemo upit koji postavlja upite za jednu tabelu sa uključenom klauzulom WHERE i indeksiranom kolonom:

SELECT \*

FROM world.city

WHERE CountryCode = 'IND';

Tabela world.city ima sekundarni nonunique indeks na CountryCode-u

koloni kao što se može videti iz definicije tabele:

mysql> SHOW CREATE TABLE world.city\G

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. red\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Table: city

Create Table: CREATE TABLE `city` (

`ID` int(11) NOT NULL AUTO\_INCREMENT,

`Name` char(35) NOT NULL DEFAULT ",

`CountryCode` char(3) NOT NULL DEFAULT ",

`District` char(20) NOT NULL DEFAULT ",

`Population` int(11) NOT NULL DEFAULT '0',

PRIMARY KEY (`ID`),

**KEY `CountryCode` (`CountryCode`),**

CONSTRAINT `city\_ibfk\_1` FOREIGN KEY (`CountryCode`) REFERENCES `country`

(`Code`)

) ENGINE=InnoDB AUTO\_INCREMENT=4080 DEFAULT CHARSET=utf8mb4

COLLATE=utf8mb4\_0900\_ai\_ci

1 row in set (0.0008 sec)

Postoje dva načina na koje optimizator može izabrati da preuzme odgovarajuće redove. Jedan način je da se koristi indeks na CountryCode-u da pronađete odgovarajuće redove u indeksu, a zatim potražite tražene vrednosti reda.

Drugi način je da izvršite potpuno skeniranje tabele i proverite svaki red utvrdi da li ispunjava uslov filtera.

Koji od ovih metoda pristupa ima najnižu cenu (je najbrži) nije kao lako odrediti kako izgleda. To zavisi od nekoliko faktora:

**Koliko je selektivan indeks?**

Čitanje reda kroz sekundarno indeksiranje uključuje prvo pronalaženje reda u indeksu, a zatim eventualno traženje primarnog ključa da biste dobili red. Ovo

znači da je skuplje pregledati i dohvatiti red koristeći sekundarni indeks nego direktno čitanje reda i za indeks da bi pristup u celini postao jeftiniji od skeniranja tabele, indeks mora značajno smanjiti broj redova za ispitivanje. Selektivni indeks je relativno jeftinije koristiti.

**• Da li je indeks pokrivajući indeks?**

Ako indeks uključuje sve kolone potrebno za upit, moguće je preskočiti čitanje u stvarnom redu čineći korišćenje indeksa povoljnijim.

**• Koliko je skupo čitanje zapisa?**

Ovo opet zavisi od nekoliko faktora kao što su da li su podaci indeksa i reda već u bafer pool-u i, ako ne, koliko brzo se zapisi mogu čitati sa diska. Korišćenje indeksa će zahtevati više nasumičnih ulaza/izlaza s obzirom na prebacivanje između čitanja indeksa i čitanja grupisanog indeksa, tako da vreme traženja za lociranje zapisa postaje veoma važno.

**9.Table Join Order**

Za komplikovanije upite od jedne tabele SELECT izraza, optimizator ne samo da treba da uzme u obzir troškove pristupa svakoj tabeli već i da razmotri redosled u kome je svaka tabela uključena i koji indeks da koristi za svaku tabelu. Za outer i straight join-e, redosled spajanja je fiksan, ali za unutrašnje spojeve optimizator slobodno bira redosled, tako da optimizator mora izračunati cenu svake kombinacije. Broj mogućih kombinacija je N! koji se veoma slabo skalira. Ako ima pet tabela koje učestvuju u unutrašnjim spojevima, optimizator ima izbor od pet tabela kao prvo, zatim četiri tabele za drugi izbor, tri tabeleza treći izbor, dva tabele za četvrtu izbori na kraju jednu tabelu za poslednji izbor tabele:

Kombinacije = 5 \* 4 \* 3 \* 2 \* 1 = 5! = 120

MySql podržava spajanje do 61 tabele u kom slučaju postoji potencijalno 5.1E83

kombinacije za izračunavanje troškova koji su previsoki i verovatno će trajati duže

nego izvršavanje samog upita. Iz tog razloga, optimizator podrazumevano smanjuje

upitne planove na osnovu delimičnih procena troškova, dakle samo planove koji najviše obećavaju su u potpunosti ocenjeni. Takođe je moguće reći optimizatoru da nakon toga prestane sa procenom troškova uključujući zadati broj tabela.

**10.Cena upita**

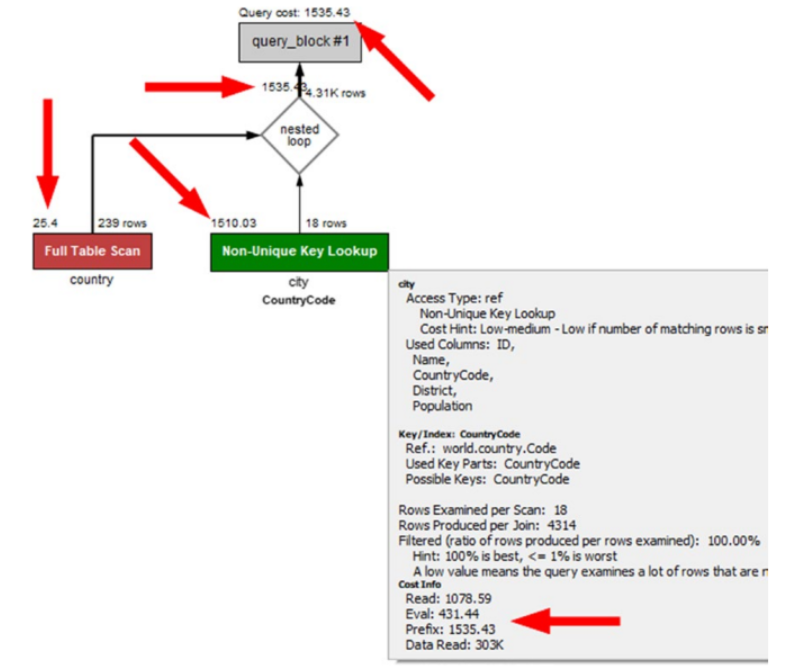
Ako želite da ispitate troškove koje je optimizator pronašao, moraćete da koristite stablo- (uključujući EXPLAIN ANALIZE) ili JSON formatiran EXPLAIN izlaz, MySql Workbench Visual Explain dijagram, ili trag optimizatora. Kao jednostavan primer, razmotricemo upit koji spaja tabele zemalja i gradova

SELECT \*

FROM world.country

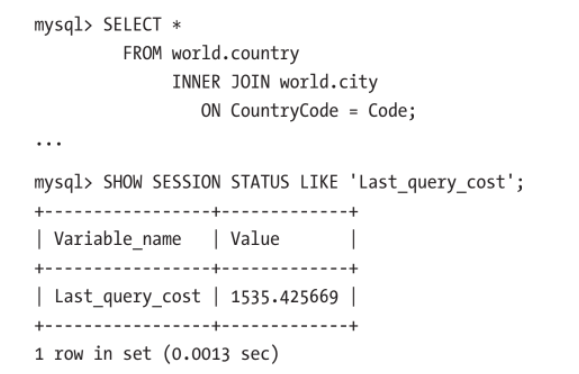
INNER JOIN world.city

ON CountryCode = Code;

****

Slika 2: Primer sa objasnjenjima i estimacijom cena

Slika pokazuje kako je optimizator odlučio da izvrši upit. Ovde su važan deo brojke na koje upućuju strelice. Ovo su procene troškova do kojih je došao optimizator za različite delove izvršenja upita sa nižom cenom, to bolje. Primer pokazuje da su procene troškova izračunate za vrlo specifične zadatke kao što je čitanje podatke, procena stanja filtera i tako dalje. Na vrhu dijagrama, ukupni upit cena se procenjuje na 1535,43.



Rezultat upita je uklonjen iz izlaza jer nije važan za ovu diskusiju. Važna stvar koju treba napomenuti u vezi sa Last\_query\_cost je da jeste procenjeni trošak zbog čega prikazuje istu vrednost kao i ukupni trošak u vizuelnom prikazu.

Ako želite informacije o stvarnoj ceni izvršenja upita, potrebno je da koristite EXPLAIN ANALYZE. Dijagram vizuelnog objašnjenja pominje da se upit izvršava pomoću ugnežđene petlje. To je samo jedan od algoritama za spajanje koje MySql podržava.

**11.Join Algoritmi**

Spajanje je veoma širok koncept u MySql -u – toliko da možete tvrditi da sve jeste

join. Čak se i ispitivanje jedne tabele smatra spajanjem. Najzanimljiviji su spojevi su oni između dve ili više tabela. U ovoj diskusiji tabela takođe može biti i izvedena tabela.

Kada se upit izvrši, a dve tabele treba da se spoje, MySql ima podršku za

tri različita algoritma. Algoritmi su

• **Nested loop (**Ugnežđena petlja)

• **Block nested loop (**Blok ugnežđenu petlja)

• **Hash join** (Heš pridruživanje)

**11.1.Nested loop**

Algoritam ugnežđene petlje je najjednostavniji od algoritama koji se koriste u MySql-u. Do MySql-a 5.6 to je bio i jedini algoritam koji je bio dostupan. Kao što ime kaže, funkcioniše ugnežđenjem petlje sa jednom petljom za svaku tabelu u spoju. Ne samo da je ugnežđeni algoritam pridruživanja veoma jednostavan; takođe dobro funkcioniše za pretraživanje indeksa.

SELECT CountryCode, country.Name AS Country,

city.Name AS City, city.District

FROM world.country

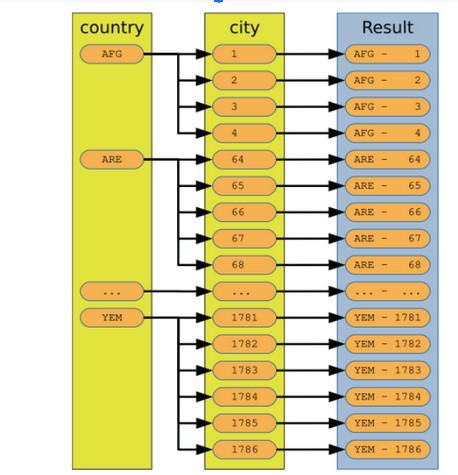
INNER JOIN world.city

ON city.CountryCode = country.Code

WHERE Continent = 'Asia';

Ovaj upit izvršiće se pomoću ugnežđene petlje sa skeniranjem tabele na tabeli country gde filter u klauzuli WHERE se primenjuje nakon čega sledi pretraživanje indeksa na tabeli city.

Ovo prikazuje isto spajanje ugnežđene petlje pomoću dijagrama. Vrednosti primarnog ključa odgovarajućih redova su uključene čak iako se svi redovi čitaju iz tabele country.



Slika 3: Primer nested loop join

Dijagram pokazuje da MySql skenira tabelu zemalja sve dok ne pronađe odgovarajući red klauzula WHERE. Na dijagramu, prvi odgovarajući red je AFG (za Avganistan). Onda pronađeni su svi redovi u tabeli gradova za Country Code = AFG (ID jednak 1, 2, 3 i 4), i svaka kombinacija se koristi za formiranje reda u rezultatu. Ovo se nastavlja sa zemljom kod jednak ARE (za Ujedinjene Arapske Emirate) i tako dalje do YEM (za Jemen). Tačan redosled skeniranja redova u tabeli zemalja i unutar Indeks CountryCode u tabeli gradova zavisi od definicija indeksa kao i od internih elemenata u optimizatoru, izvršiocu i mašini za skladištenje. Nikada se ne treba oslanjati na naređenje ostanite isti osim ako nemate izričitu klauzulu ORDER BY. Generalno, spajanje može biti složenije nego u ovom primeru zbog dodatnih filtera. Ipak, koncept ostaje isti.

Iako je jednostavnost obično dobra osobina, spajanje ugnežđene petlje ima neka ograničenja. Ne može se koristiti za izvršavanje full outer join-a jer spoj ugnežđene petlje zahteva prvu tabelu za vraćanje redova što nije uvek slučaj za full outer join. Zaobilazno rešenje je da se napiše full outer join kao unija levog i desnog spoljašnjeg spoja. Drugo ograničenje je da nested loop join nije veoma efikasan za spojeve koji ne mogu koristiti indekse jer to brzo postaje preskupo.

**11.2.Block Nested Loop**

Algoritam ugnežđene petlje u bloku je proširenje algoritma ugnežđene petlje. Takođe je poznat kao BNL algoritam. Umesto da submitt-ujete redove iz prve tabele u spoju jedan po jedan, bafer za spajanje se koristi da prikupi što više redova i uporedi ih sve u jednom skeniranju druge tabele. Ovo može značajno poboljšati performanse za neke upite preko algoritma ugnežđene petlje. Ako uzmete u obzir isti upit koji je korišćen kao primer za ugnežđenu petlju algoritam, ali onemogućavanje upotrebe indeksa (za simulaciju dve tabele bez indeksa) i ne dozvoljavajući heš pridruživanje (u 8.0.18 i novijim verzijama), imate upit koji može da iskoristi prednosti algoritam blok ugnežđene petlje. Upit je

SELECT CountryCode, country.Name AS Country,

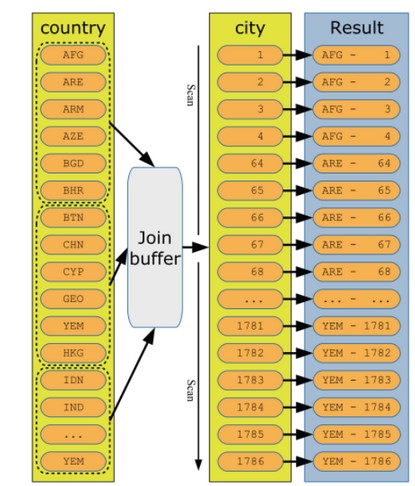
city.Name AS City, city.District

FROM world.country IGNORE INDEX (Primary)

INNER JOIN world.city IGNORE INDEX (CountryCode)

ON city.CountryCode = country.Code

WHERE Continent = 'Asia';



Slika 4: Primer blook nested loop join

Slika pokazuje kako se redovi iz tabele zemalja zajedno čitaju i čuvaju u join bafer-u. Svaki put kada je join bafer pun, vrši se skeniranje pune tabele za city tabele, a rezultat se gradi postepeno. Na slici se šest redova uklapa u bafer za spajanje. Pošto kolona Code zahteva samo 3 bajta po redu, u praksi će bafer za spajanje moći da zadrži sve pozivne brojeve zemalja osim kada se koriste najmanja moguća podešavanja od join\_buffer\_size. Što se tiče prethodnih primera, prvo se izvršit upit tražeći azijske gradove u jednoj konekciji:

SELECT CountryCode, country.Name AS Country,

city.Name AS City, city.District

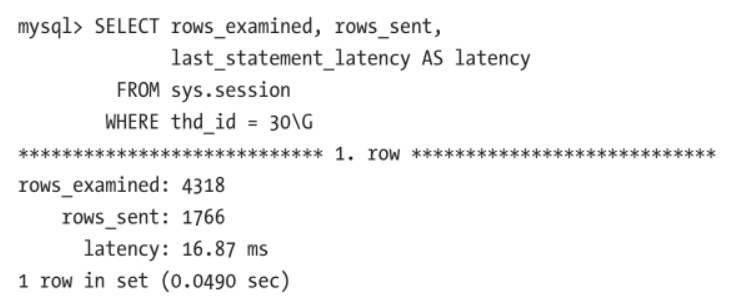
FROM world.country IGNORE INDEX (Primary)

INNER JOIN world.city IGNORE INDEX (CountryCode)

ON city.CountryCode = country.Code

WHERE Continent = 'Asia';

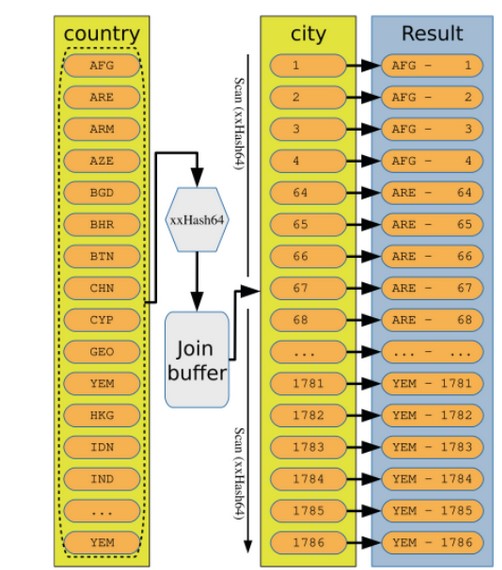
Rezultat koji se dobije:



Statistika pokazuje da blok ugnežđena petlja radi znatno bolje od algoritma ugnežđene petlje bez koristescenja indeksa. Poređenja radi, izvršavanje upita sa indeksom je ispitalo 2005 redova i trajalo je oko 4 ms, dok je korišćenje ugnežđenog spoja petlje bez indeksa ispitano 208268 redova i trajalo je oko 45 ms.

Za jedan upit, možete koristiti savete za optimizaciju BNL() i NO\_BNL() da omogući ili onemogući blok ugnežđenu petlju za određene spojeve. Iako je blok ugnežđena petlja veliko poboljšanje za neindeksirane spojeve, u većini slučajeva je moguće učiniti još bolje korišćenjem hash join-a.

**11.3.Hash join**

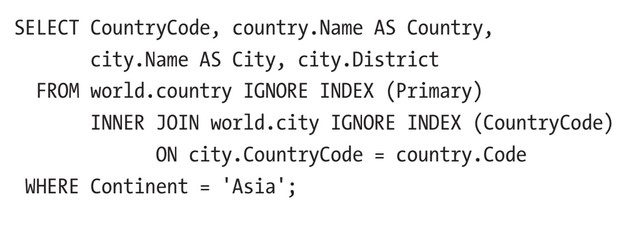


Slika 5: Primer in-memory hash join

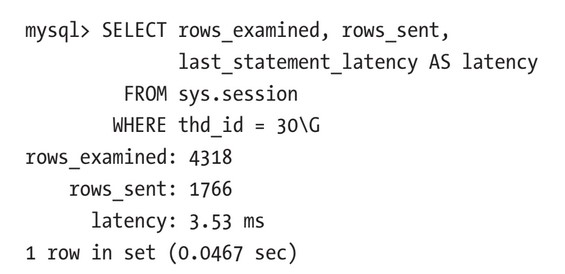
Algoritam heš pridruživanja je noviji dodatak MySql-u i podržan je u MySql-u 8.0.18 i u novijim verzijama. To označava značajan raskid sa tradicijom spajanja ugnežđenih petlji uključujući varijantu blok ugnežđene petlje.

Posebno je koristan za velike spojeve bez indekse, ali u nekim slučajevima može čak i nadmašiti spajanje indeksa. MySql implementira hibrid između klasičnog heš pridruživanja u memoriji i na disku GRACE algoritam heš pridruživanja. Ako je moguće uskladištiti sve hešove u memoriji, onda se koristi čista implementacija u memoriji. Bafer za spajanje se koristi za deo u memoriji, tako da je količina memorije koja se može koristiti za hešove ograničena join\_buffer\_ veličina. Kada se spajanje ne uklapa u memoriju, spajanje se preliva na disk, ali stvarno operacije spajanja se i dalje izvode u memoriji.

Slika prikazuje dijagram za heš algoritam spajanja u memoriji. Za jednostavnost, uključene su samo vrednosti primarnog ključa redova potrebnih za spajanje iako a vrši se kompletno skeniranje tabele za obe tabele. Vrednosti kolone Code za odgovarajuće redove iz tabele country su heširane i skladištene u join baferu. Zatim se izvrši skeniranje tabele za tabelu grada sa heš izračunat od CountryCode za svaki red, a rezultat se konstruiše iz odgovarajućih redova. Statistiku za upit možete proveriti na isti način kao i za prethodni algoritme tako što prvo izvršite upit u jednoj vezi:



Rezultat izvrsenog upita:



Možete videti da upit radi veoma dobro sa hash join-om ispitujući isti broj redova kao blok ugnežđena petlja, ali je brža od spajanja indeksa.

U nekim slučajevima, heš pridruživanje može nadmašiti čak i indeksno spajanje. Možete koristiti sledeća pravila za procenu kako će algoritam heš pridruživanja delovati u poređenju sa indeksom i blok ugnežđene petlje spoja:

• Za spajanje bez korišćenja indeksa, heš spajanje će obično biti mnogo brže od blok ugnežđenog pridruživanja osim ako nije dodata klauzula LIMIT. Uočena su poboljšanja za više od faktora 1000.4

• Za spajanje bez indeksa gde postoji klauzula LIMIT, blok ugnežđena petlja

može zavrsiti kada se pronađe dovoljno redova, dok heš spajanje će dovršiti celo spajanje (ali može preskočiti preuzimanje redovi). Ako je broj redova uključenih zbog klauzule LIMIT mali u poređenju sa ukupnim brojem redova pronađenih spajanjem, blokom ugnežđena petlja može biti brža.

• Za spajanja koja podržavaju indeks, algoritam heš pridruživanja može biti brži ako indeks ima nisku selektivnost.

Najveća prednost korišćenja heš spojeva je daleko za spojeve bez indeksa i bez klauzule LIMIT. Na kraju, samo testiranje može dokazati koja je strategija pridruživanja optimalno za vaše upite.

**12.Zaključak**

Kroz ovaj rad pokazano je kako MySql može optimizovati upite. Najveći deo ovog rada se bazira na MySql-u 8. Najveći akcenat u ovom radu stavljen je na optimizaciju join-a. Videli smo kada je najbolje koji join optimizator koristiti. Dati su saveti koji se mogu iskoristiti i koji znatno mogu pomoći pri optimizaciji upita. Primetili smo da je algoritam ugnežđene petlje je najjednostavniji od algoritama koji se koriste u MySql-u ali da spajanje ugnežđene petlje ima neka ograničenja. Ne može se koristiti za izvršavanje full outer join-a jer spoj ugnežđene petlje zahteva prvu tabelu za vraćanje redova što nije uvek slučaj za full outer join. Takodje smo videli i zaobilazna resenja tog problema. Drugo ograničenje je da nested loop join nije veoma efikasan za spojeve koji ne mogu koristiti indekse jer to brzo postaje preskupo. Primetili smo i da statistika pokazuje da blok ugnežđena petlja radi znatno bolje od algoritma ugnežđene petlje bez koriscenja indeksa. Možete videti da upit radi veoma dobro sa hash join-om ispitujući isti broj redova kao blok ugnežđena petlja, ali je brži od spajanja indeksa. Posebno je koristan za velike spojeve bez indekse, ali u nekim slučajevima može čak i nadmašiti spajanje indeksa.

# **13.Literatura**

1. MySql 8 Query Performance and Tuning -A Systematic Method For Improving Execution Speed by Jesper Wisborg Krogh
2. <https://www.oreilly.com/library/view/high-performance-mysql/9780596101718/ch04.html>
3. <https://dev.mysql.com/doc/refman/8.0/en/optimization.html>
4. <https://dzone.com/articles/how-to-optimize-mysql-queries-for-speed-and-perfor>
5. <https://www.oreilly.com/library/view/high-performance-mysql/9780596101718/ch04.html>
6. <https://www.section.io/engineering-education/mysql-query-performance-optimization-tips/>
7. <https://www.cloudways.com/blog/mysql-performance-tuning/>