



UNIVERZITET U NIŠU
ELEKTRONSKI FAKULTET



OPTIMIZACIJA UPITA KOD MySQL BAZE PODATAKA

Seminarski rad

Predmet: Sistemi za upravljanje bazama podataka

Student:

Marko Kocić, br. ind. 1478

Mentor:

Doc. dr Aleksandar Stanimirović

Niš, april 2023. godina

Sadržaj

UVOD.....	3
PREGLED.....	4
Optimizacija na nivou baze podataka.....	4
Optimizacija na hardverskom nivou.....	5
OPTIMIZACIJA SQL NAREDBI.....	6
Optimizacija <i>SELECT</i> naredbi.....	6
<i>WHERE</i> klauzula.....	7
<i>IS NULL</i> optimizacija.....	8
<i>ORDER BY</i> optimizacija.....	8
<i>GROUP BY</i> optimizacija.....	10
<i>DISTINCT</i> optimizacija.....	12
<i>LIMIT</i> optimizacija.....	13
Range optimizacija.....	13
Index Merge optimizacija.....	14
Nested-Loop Join algoritam.....	15
Block Nested-Loop Join algoritam.....	15
Hash Join optimizacija.....	15
Engine Condition Pushdown optimizacija.....	16
Index Condition Pushdown optimizacija.....	16
Multi-Range Read optimizacija.....	17
Filtriranje uslova.....	17
Constant-Folding optimizacija.....	18
Function Call optimizacija.....	18
Izbegavanje potpunog skeniranja tabela.....	19
Optimizacija ugnježenih upita.....	19
<i>INSERT</i> , <i>UPDATE</i> i <i>DELETE</i> optimizacija.....	20
Ostali optimizacioni saveti.....	20
InnoDB – optimizacija upita.....	20
OPTIMIZATOR I PLAN IZVRŠENJA UPITA.....	22
<i>EXPLAIN</i> naredba.....	22
Kontrolisanje optimizatora.....	24
Statistika optimizatora.....	28
ZAKLJUČAK.....	30
LITERATURA.....	31

1. UVOD

Optimizacija radi poboljšanja performasi je veoma bitan faktor kod realnih aplikacija. Tokom vremena baza podataka postaje sve veća i veća i ukoliko se ne vodi računa o optimizaciji u vidu konfiguracije, štimovanja (*tuning*) i merenja performasi na više nivoa, može se desiti da sam sistem radi isuviše sporo i da vreme čekanja postane neprihvatljivo za krajnjeg korisnika. Najverovatnije, ako se ništa ne uradi na poboljšanju, aplikacija neće doživeti slavan kraj. Optimizacija se treba vršiti na svim područjima, od individualnih *SQL* naredbi, čitave aplikacije, pojedinačnih servera baza podataka, pa sve do višestruko umreženih servera baza podataka. Trebalo bi već u samom razvoju voditi računa o performansama kako bi problemi u produkciji bili redukovani. Poboljšanje upotrebe resursa (centralnog procesora i memorije) može unaprediti skalabilnost, omogućavajući bazi podataka da podnese veće opterećenje bez usporavanja.

U ovom seminarskom radu sam se bavio optimizacijom upita kod *MySQL* baze podataka.

2. PREGLED

Performanse baze podataka zavise od nekoliko faktora kao što su tabele, upiti i podešavanja konfiguracije. Prethodno pomenuti softverski faktori utiču na pojedinosti sa hardverskog nivoa, kao što su instrukcije centralnog procesora i ulazno-izlazne operacije. Na samom početku, inženjer uči pravila i smernice visokog nivoa i meri performanse koristeći časovnik (štopericu), dok kasnije, kada postane stručnjak, uči šta se dešava „pod haubom” i meri stvari kao što su broj *CPU* ciklusa i *I/O* operacija. Prosečan korisnik ima za cilj dobijanje najboljih performansi baze podataka iz svojim postojećih softverskih i hardverskih konfiguracija. Napredni korisnici traže priliku za poboljšanje samog *MySQL* softvera ili pak razvijaju sopstvena rešenja koja se baziraju na proširivanje *MySQL* sistema.

2.1. Optimizacija na nivou baze podataka

Na nivou baze podataka, optimizacija se može vršiti praćenjem sledećih preporuka:

- Trebalo bi da tabele budu pravilno struktuirane, tačnije u zavisnosti od svrhe da sadrže prikladan broj kolona određene namene i ispravno definisanog tipa podataka. Aplikacija koja često vrši ažuriranja obično ima dosta tabela sa malim brojem kolona, dok aplikacija koja analizira velike količine podataka ima manji broj tabela sa velikim brojem kolona.
- Potrebno je imati prave indekse u cilju efikasnog izvršenja upita.
- Koristiti odgovarajući mehanizam za skladištenje (*storage engine*) podataka za svaku tabelu ponaosob zajedno sa svim prednostima istog. Izbor mehanizma je veoma važan za performanse i skalabilnost.
- Svaka tabela u zavisnosti od mehanizma za skladištenje treba koristiti prikladan format reda (*row format*). Konkretno, kompresovane tabele zauzimaju manje prostora na disku i samim tim zahtevaju manje *I/O* operacija za čitanje i pisanje. Što se tiče *InnoDB* tabela, kompresija je omogućena kod svih tipova, dok je ista dozvoljena samo kod *read-only MyISAM* tabela.
- Aplikacija treba koristiti odgovarajuću strategiju zaključavanja (*locking strategy*). Dozvoljavanje deljenog pristupa radi konkurentnog izvršavanja operacija i zahtevanje ekskluzivnog pristupa kada je to neophodno (kritične operacija imaju najviši prioritet). I ovde je izbor mehanizma za skladištenje od presudnog značaja. *InnoDB* rešava većinu problema sa zaključavanjem bez uključenosti *developer-a*, omogućavajući bolju istovremenost u bazi podataka i smanjujući količinu eksperimentisanja i podešavanja u samom kodu.
- Memorijska oblast koja se koristi za keširanje treba biti ispravne veličine, tačnije treba biti dovoljno velika za smeštanje podataka kojima se često pristupa, ali ne toliko da preopterećuje fizičku memoriju i izazove straničenje.

2.2. Optimizacija na hardverskom nivou

Kako sama baza podataka postaje sve zauzetija, tako aplikacija baze podataka dostiže hardverska ograničenja. Administrator baze podataka mora u ovim trenucima da proceni da li je moguće dodatno podesiti aplikaciju ili pak rekonfigurisati server radi izbegavanja „uskih grla”, ili je, šta više, potrebno dodavanje hardverskih resursa. Sistemska „uska grla” obično nastaju iz sledećih izvora:

- Traženje podataka. Potrebno je vreme da disk pronađe podatak – obično je srednje vreme manje od 10ms (što dalje implicira da u teoriji može izvršiti 100 traženja u sekundi). Ovo vreme se polako poboljšava sa novim diskovima i veoma ga je teško optimizovati za pojedinačnu tabelu. Način optimizacije vremena traženja jeste distribucija podataka na više od jednog diska.
- Čitanje i pisanje. Kada je disk na ispravnoj poziciji možemo pročitati ili upisati podatke. Disk obezbeđuje protok od najmanje 10 do 20MB/s. Ove operacije je lakše optimizovati u poređenju sa operacijom traženja jer primera radi čitati se može izvoditi paralelno sa više diskova.
- CPU ciklusi. Kada su podaci u glavnoj memoriji, obrađujemo ih radi dobijanja rezultata. Ograničavajući faktor je imati velike tabele u poređenju sa količinom memorije koja je na raspolaganju. Brzina nije problem kod malih tabela.
- Opseg memorije. Kada je procesoru potrebno više podataka nego što može smestiti u keš procesora, propusni opseg glavne memorije postaje „usko grlo”. Za većinu sistema ovo je nemoguće da se desi, ali ipak treba biti svestan.

3. OPTIMIZACIJA SQL NAREDBI

Osnovna logika aplikacije baze podataka se izvodi preko *SQL* naredbi, bilo direktno koristeći interpretator ili pak indirektno preko *API*-a. Cilj je ubrzati *MySQL* aplikaciju. Smernice pokrivaju *SQL* operacije čitanja i pisanja podataka, ali i operacije koje se koriste u specifičnim okolnostima (na primer nadgledanje baze podataka).

3.1. Optimizacija *SELECT* naredbi

Upiti u vidu *SELECT* naredbi izvode sve operacije pretraživanja u bazi podataka i njihovo podešavanje predstavlja glavni prioritet.

Glavna razmatranja u okviru optimizacije upita su sledeća:

- Napraviti *SELECT ... WHERE* upit bržim dodavanjem indeksa. Treba podesiti indekse na kolonama koje se javljaju u *WHERE* delu radi brže evaluacije, filtriranja i preuzimanja rezultata. Sa željom izbegavanja gubitaka prostora na disku, kreirati mali skup indeksa koji ubrzavaju mnoge povezane upite u aplikaciji. Indeksi su posebno važni za upite koji uključuju više tabela koristeći spojeve i strane ključeve.
- Izolovati i podešavati delove upita koji zahtevaju previše vremena za izvršenje.
- Smanjiti broj skeniranja tabele u celini – potpuno skeniranje tabele (posebno važi za velike tabele).
- Održavati statistiku tabele ažurnom koristeći periodično naredbu *ANALYZE TABLE*, tako da optimizator ima informacije potrebne za izradu efikasnog plana izvršenja.
- Naučiti tehnike podešavanja i indeksiranja, kao i konfiguracione parametre koji su specifični za svaku tabelu u odnosu na prethodno izabran mehanizam za skladištenje. Kako kod *InnoDB*-a, tako i kod *MyISAM*-a postoji niz smernica za omogućavanje i održavanje visokih performansi u upitima.
- Izbegavati transformacije upita na način koji otežava razumevanje, posebno ako optimizator automatski izvrši neke od istih transformacija.
- Ako se problem u performansama ne reši praćenjem osnovnih smernica, istražiti unutrašnje detalje specifičnog upita čitanjem *EXPLAIN* plana (prilagoditi indekse, *WHERE* klauzule,...).
- Podesiti veličinu i svojstva memorije koju *MySQL* koristi za keširanje – ponovljeni upiti se rešavaju brže jer se sami rezultati preuzimaju iz memorije.
- Čak i za upit koji se izvršava prihvatljivo brzo (korišćenje keša), možda se može dalje optimizovati i zahtevati recimo manje keš memorije, čineći aplikaciju skalabilnijom.
- Pozabaviti se problemima zaključavanja jer na brzinu izvršenja upita mogu uticati i druge sesije koje istovremeno pristupaju tabelama.

3.1.1. *WHERE* klauzula

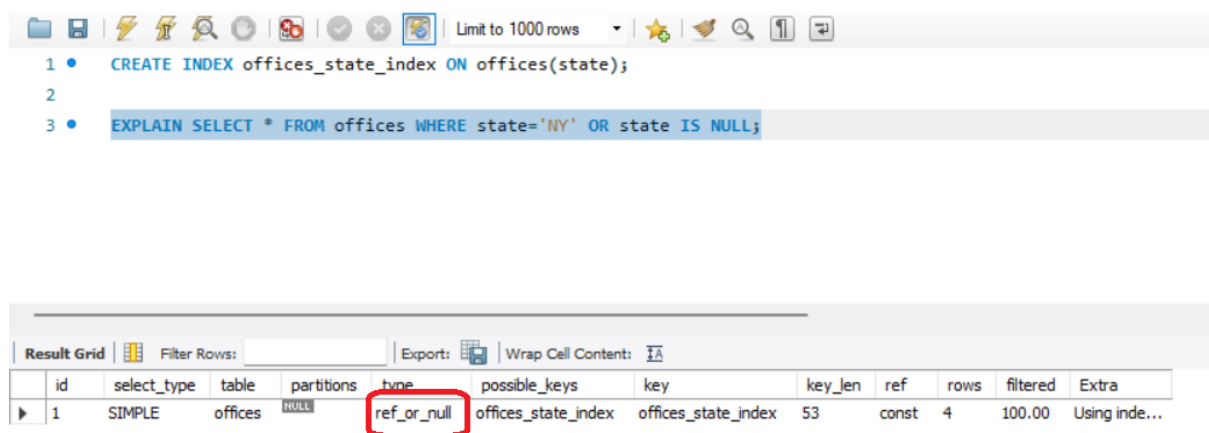
Saveti za optimizaciju se odnose na *WHERE* deo kako kod *SELECT*, tako i kod *UPDATE* i *DELETE* naredbi.

Ne savetuje se žrtvovanje čitljivosti radi činjenja aritmetičkih operacija bržim jer se takav upit teško može razumeti i kasnije održavati, a i sam *MySQL* radi automatski optimizacije u fazi pripreme i to:

- uklanjanje suvišnih zagrada;
- zamena konstanti;
- izbacivanje nepotrebnih (suvišnih) delova upita;
- konstantni izrazi koje koriste indeksi se evaluiraju samo jednom;
- poređenje kolona numeričkih tipova sa konstantnim vrednostima se proveravaju i uklanjaju za nevažeće ili pak vrednosti van opsega;
- *COUNT(*)* bez *WHERE* dela vezane za jednu tabelu se kod *MyISAM* direktno čita iz informacija o tabeli;
- brzo otkrivanje da su pojedine *SELECT* naredbe nemoguće i ne vraćaju nikakve redove (loš izraz konstanti);
- *HAVING* se spaja sa *WHERE* ukoliko se ne koristi *GROUP BY* ili funkcije agregacije (*COUNT()*, *MIN()*, *MAX()*,...);
- za svaku tabelu u spoju konstruisan je jednostavniji *WHERE* radi dobijanja brze procene *WHERE* za tabelu i što bržeg preskakanja redova;
- u upitu se najpre čitaju konstantne tabele (prazna tabela, tabela sa jednim redom ili pak tabela čije se kolone primarnog ključa ili jedinstvenog indeksa javljaju u *WHERE* delu i gde se svi delovi indeksa koju su definisani kao *NOT NULL* upoređuju sa konstantnim izrazima);
- najbolja kombinacija spajanja tabela dobija se isprobavanjem svih mogućnosti. Ako sve kolone u klauzulama *ORDER BY* i *GROUP BY* potiču iz iste tabele, ta tabela je prva pri spajanju.
- ako postoji *ORDER BY* klauzula i druga *GROUP BY* klauzula, ili ako *ORDER BY* ili *GROUP BY* sadrže kolone iz tabela koje nisu prve tabele u redu za spajanje, kreira se privremena tabela;
- ako se koristi modifikator *SQL_SMALL_RESULT*, privremena tabela se nalazi u memoriji;
- svaki indeks tabele se ispituje i koristi se najbolji osim ukoliko optimizator veruje da je efikasnije koristiti skeniranje tabele. Jedno vreme je korišćeno skeniranje na osnovu toga da li je najbolji indeks obuhvatao više od 30% tabele, ali fiksni procenat više ne određuje izbor. Optimizator je sada složeniji i zasniva svoju procenu na dodatnim faktorima kao što su veličina tabele, broj redova i veličina ulazno-izlaznog bloka;
- u nekim slučajevima, *MySQL* može čitati redove iz indeksa čak i bez konsultovanja datoteke sa podacima. Ako su sve kolone koje se koriste iz indeksa numeričke, samo stablo indeksa se koristi za rešavanje upita.

3.1.2. IS NULL optimizacija

MySQL može izvršiti istu optimizaciju za vrednost kolone kako za konstantne vrednosti, tako i za *IS NULL* ispitivanje. Ukoliko se u *WHERE* klauzuli nalazi uslov da je vrednost neke kolone *NULL* za kolonu koja je definisana kao *NOT NULL*, taj izraz je svakako optimizovan. MySQL takođe optimizuje i kombinaciju *col_name = expr OR col_name IS NULL*, a *EXPLAIN* u tom slučaju prikazuje *ref_or_null* koji funkcioniše tako što najpre čita *expr* vrednosti, a zatim zasebno traži redove čija je vrednost *col_name NULL*. Primer jednog takvog upita je dat na slici ispod. Prethodno je kreiran indeks nad kolonom *state* u tabeli *offices*. Važno je napomenuti da optimizacija može obraditi samo jedan *IS NULL* za bilo koji deo ključa.



The screenshot shows a MySQL command prompt with the following commands:

```
1 • CREATE INDEX offices_state_index ON offices(state);
2
3 • EXPLAIN SELECT * FROM offices WHERE state='NY' OR state IS NULL;
```

Below the commands is the output of the *EXPLAIN* statement, showing a table with columns: id, select_type, table, partitions, type, possible_keys, key, key_len, ref, rows, filtered, and Extra. The row for id=1 shows the type as *ref_or_null*, which is highlighted with a red box.

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	offices	NULL	ref_or_null	offices_state_index	offices_state_index	53	const	4	100.00	Using inde...

Slika 3.1. *IS NULL* optimizacija – *ref_or_null*

3.1.3. ORDER BY optimizacija

MySQL može koristiti indekse radi zadovoljavanja *ORDER BY* klauzule ili pak *filesort* operacije (ukoliko se ne može koristiti indeks).

Korišćenjem indeksa izbegava se dodatno sortiranje koje je uključeno prilikom izvođenja *filesort* operacija. Optimizator uvek procenjuje da li će skenirati celu tabelu ili će pak koristiti indekse. MySQL najčešće odlučuje da koristi indekse ukoliko se u *ORDER BY* klauzuli nalaze indeksirane kolone koje se nalaze i u *SELECT* delu i to ne kao alias. Ukoliko upit počinje sa *SELECT **, najverovatnije se neće koristiti indeks.

MySQL 5.7 i ranije verzije su implicitno vršile sortiranje prilikom korišćenja *GROUP BY* pod određenim uslovima i zahtevalo se od *developer*-a da u želji za sprečavanjem prethodno navedenog doda *ORDER BY NULL*. U trenutnoj verziji MySQL 8.0 to se ne dešava.

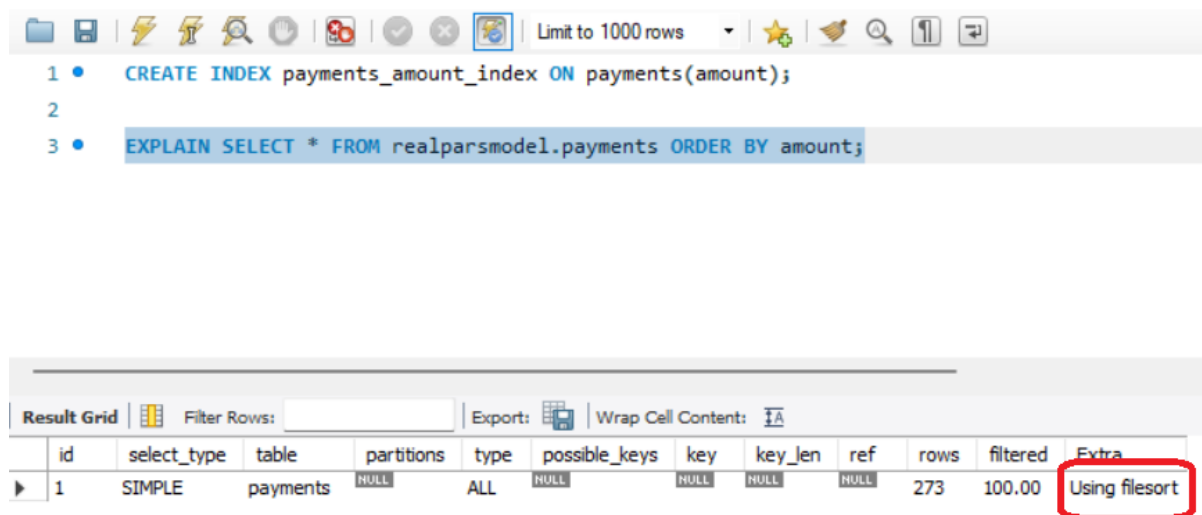
Ukoliko se ne mogu koristiti indeksi, izvršavaju se *filesort* operacije koje čitaju redove tabela i sortiraju ih. *MySQL 8.0.12* verzija vrši inkrementalno alociranje memorijskih *buffer*-a ukoliko je neophodno (sistemska promenljiva *sort_buffer_size*), za razliku od ranijih verzija kod kojih je bilo zastupljeno fiksno alociranje memorije. *Filesort* operacije mogu koristiti i privremene datoteke ukoliko rezultat ne može stati u operativnu memoriju.

U cilju povećanja brzine *ORDER BY* soritrnja (pokušali smo da nateramo *MySQL* da korisiti indekse i nismo uspeali), predlaže se sledeće:

- Povećati vrednost sistemske promeljive *sort_buffer_size* (idealno vrednost treba biti toliko velika da ceo rezultat može stati u *buffer* za sortiranje).
- Povećati vrednost *read_rnd_buffer_size* promenljive (čitanje više redova istovremeno).
- Postaviti da sistemska promenljiva *tmpdir* ukazuje na file system sa velikom količinom slobodnog prostora.

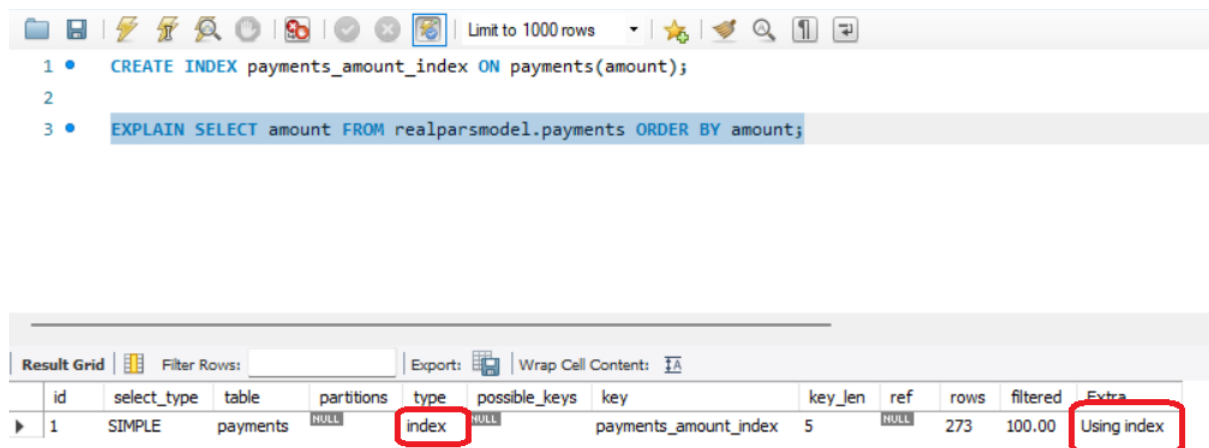
Ukoliko dodatna (*extra*) kolona izlaza *EXPLAIN* naredbe sadrži *Using filesort*, to znači da indeksi nisu korišćeni već su se izvodile *filesort* operacije.

Na slici ispod može se zapaziti da bez obzira na to što postoji indeks nad kolonom *amount*, s obzirom da je *SELECT **, a ne *SELECT amount*, umesto indeksa koristi se *filesort* (vrši se potpuno skeniranje tabele).



Slika 3.2. Bez obzira na postojanje indeksa koristi se još uvek *filesort*

Konačno kada pribavljamo između ostalog i kolonu na osnovu koje sortiramo i pritom je indeksirana, koristi se indeks (mogli smo da u *SELECT* delu imamo još kolona, koristio bi se još uvek indeks).



Slika 3.3. Korišćenje indeksa kod *ORDER BY* optimizacije

3.1.4. *GROUP BY* optimizacija

Najopštiji način da se zadovolji *GROUP BY* klauzula je skeniranje cele tabele i kreiranje privremene u kojoj su svi redovi iz svake grupe uzastopni i nakon toga izvršenje funkcija agregacije (ukoliko postoje) nad privremenom tabelom. *MySQL* ima bolji način u cilju zadovoljavanja *GROUP BY* klauzule (pokušava da ne kreira privremene tabele).

Najvažniji preduslovi za korišćenje indeksa za *GROUP BY* su da kolone koje se javljaju u *GROUP BY* klauzuli ukazuju na atribut iz istog indeksa i da taj indeks skladišti svoje ključeve po redosledu (što je tačno, na primer, za *B* stabla, ali ne i za *hash* indekse). Bitno je koji se delovi indeksa koriste u upitu, koji su uslovi specificirani za ove delove, kao i koje agregatne funkcije su izabrane i sve to u cilju korišćenja indeksa umesto pravljenja privremenih tabela.

Postoje dva načina korišćenja indeksa, a to su:

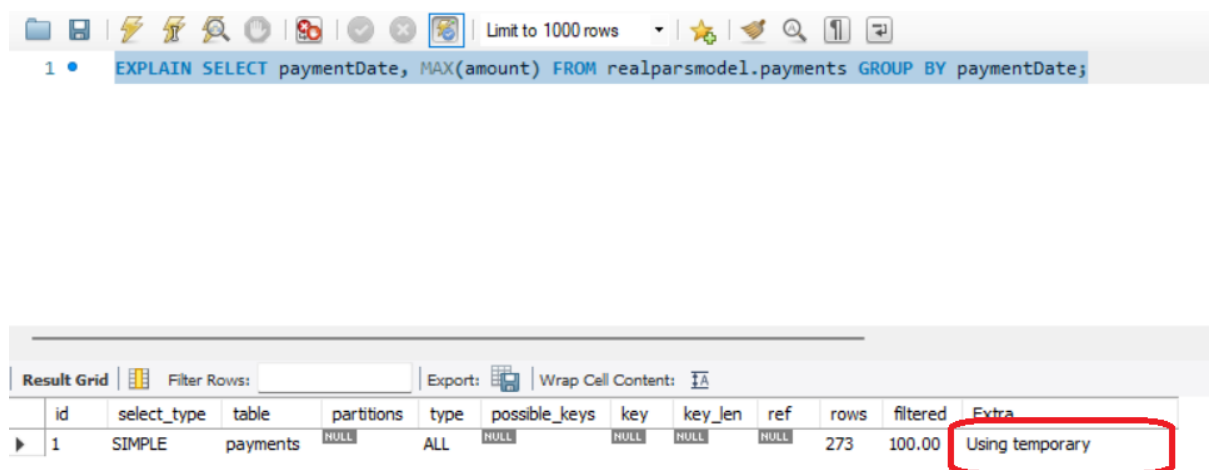
- *Loose Index Scan* - labavo skeniranje indeksa
- *Tight Index Scan* - čvrsto skeniranje indeksa

Najefikasniji način za obradu *GROUP BY* je kada se indeks koristi za direktno preuzimanje kolona za grupisanje. *MySQL* ovde koristi svojstvo nekih tipova indeksa prema kojima su ključevi poređani (*B* stabla). Ovo svojstvo omogućava korišćenje grupa za pretraživanje u indeksu bez potrebe da se uzmu u obzir svi ključevi u indeksu koji zadovoljavaju sve uslove u *WHERE* klauzuli. Kada ne postoji klauzula *WHERE*, skeniranje

labavog indeksa čita onoliko ključeva koliki je broj grupa, što može biti mnogo manje od broja svih ključeva. Upit se treba odnositi na jednu tabelu. Funkcije agregacije koje se mogu koristiti su *MIN()* i *MAX()*.

U zavisnosti od uslova upita, skeniranje čvrstog indeksa može biti potpuno skeniranje indeksa ili samo skeniranje opsega indeksa. Kada nisu zadovoljeni uslovi za skeniranje labavog indeksa još uvek se ne moraju praviti privremene tabele već se može pokušati sa skeniranjem čvrstog indeksa. Operacija grupisanja se izvodi tek nakon pronalaska svih ključeva koji zadovoljavaju uslov.

Primeru radi, želimo da vidimo maksimalni iznos transakcije po datumu. Inicijalno nemamo kreirani indeks nad kolonom koja se javlja u GROUP BY delu. Vidimo na slici ispod da se ne koristi indeks, već se rezultat dobija kreiranjem privremene tabele. Po pitanju performansi, ovo nije najsjajnije.



```
1 • EXPLAIN SELECT paymentDate, MAX(amount) FROM realparsmodel.payments GROUP BY paymentDate;
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	payments	NULL	ALL	NULL	NULL	NULL	NULL	273	100.00	Using temporary

Slika 3.4. Kreiranje privremene tabele prilikom izvršenja upita

Nakon kreiranja indeksa nad kolonom *paymentDate* i ponovnog izvršenja istog upita, koristi se indeks, što je sa stanovišta performansi daleko bolji pristup (učinjena je optimizacija).

1 • `CREATE INDEX payments_paymentDate_index ON payments(paymentDate);`
 2
 3 • `EXPLAIN SELECT paymentDate, MAX(amount) FROM realparsmodel.payments GROUP BY paymentDate;`

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	payments	HULL	index	payments_paymentDate_index	payments_paymentDate_index	3	HULL	273	100.00	HULL

Slika 3.5. Korišćenje indeksa prilikom *GROUP BY* umesto kreiranja privremene tabele

3.1.5. *DISTINCT* optimizacija

U mnogim slučajevima *DISTINCT* u kombinaciji sa *ORDER BY* zahteva privremenu tabelu. *DISTINCT* se može posmatrati kao specijalni slučaj *GROUP BY*. Sve one optimizacije koje se primenjuju nad *GROUP BY*, odnose se i na *DISTINCT*. U kombinaciji sa *LIMIT n*, *MySQL* se zaustavlja kada pronade *n* jedinstvenih redova.

Na slici ispod je prikazano korišćenje indeksa nad kolonom *courseVendor* radi efikasnijeg izvršenja upita (različiti prodavci kurseva čija je cena kursa veća od 100) – bez korišćenja privremene tabele koja se upotrebljavala bez novokreiranog indeksa.

1 • `CREATE INDEX courses_courseVendor_index ON courses(courseVendor);`
 2
 3 • `EXPLAIN SELECT DISTINCT courseVendor FROM courses WHERE buyPrice > 100;`

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	courses	HULL	index	courses_courseVendor_index	courses_courseVendor_index	52	HULL	110	33.33	Using where

Slika 3.6. Korišćenje indeksa prilikom *DISTINCT* operacije

3.1.6. *LIMIT* optimizacija

Ukoliko nam je potreban samo određen broj torki, ne treba pribavljivati ceo rezultat i ne koristiti višak podataka, već treba koristiti *LIMIT* klauzulu u upitu. Nema potrebe da kroz mrežu idu bespotrebni podaci. *MySQL* ponekad optimizuje upit sa *LIMIT* klauzulom bez *HAVING* kluzule:

- Ako imamo *LIMIT*, *MySQL* čak i ako bi bez istog zaključio da mu se više isplati da izvrši potpuno skeniranje tabele, ipak će u nekim slučajevima koristiti indekse.
- U kombinaciji *ORDER BY* i *LIMIT n*, neće se vršiti celokupno sortiranje rezultata, već sortiranje prestaje po pronalasku prvih *n* torki.
- U kombinaciji *DISTINCT* i *LIMIT n*, *MySQL* se zaustavlja kada pronađe *n* jedinstvenih redova.
- Kod *GROUP BY*, grupisanje prestaje po pronalasku prvih *n* redova.
- *LIMIT 0* nije beznačajan (bez obzira što vraća prazan prazan skup), već može biti koristan za proveru validnosti upita, kao i za pribavljanje tipova podataka kolona.

Primer radi, ukoliko su nam potrebna 3 najveća iznosa plaćanja izvršićemo sledeću naredbu:
`SELECT amount FROM payments ORDER BY amount DESC LIMIT 3;`
a ne da vratimo sve iznose (bez *LIMIT* opcije), pa onda u aplikaciji uzimamo prva 3.

Faktor na koji utiče izvršenje plana svakako jeste i *LIMIT*, tako da *ORDER BY* sa i bez *LIMIT*-a može vratiti različiti redosled torki.

3.1.7. *Range* optimizacija

Metod pristupa opsegu (*range access*) radi preuzimanja podskupa redova tabele koji se nalaze unutar jednog ili više intervala vrednosti indeksa koristi se za jednodelni ili višedelni indeks.

Definicija uslova opsega za jednodelni indeks:

- Kako za indekse strukture *B* stabla, tako i za *hash* indekse, poređenje dela ključa sa konstantnom vrednošću je uslov opsega kada se koriste `=`, `<=>`, `IN()`, `IS NULL`, kao i `IS NOT NULL` operatori.
- Dodatno za indekse strukture *B* stabla, poređenje dela ključa sa konstantnom vrednošću je uslov opsega kada se koriste `>`, `<`, `>=`, `<=`, `BETWEEN`, `!=` ili pak `<>`. Takođe može se koristiti i *LIKE* poređenje ukoliko je argument konstantni string koji ne počinje sa `%`. Generalno treba izbegavati kod *LIKE*-a da konstantni string počinje sa `%` – najbolje je da se ovaj specijalni karakter nalazi na kraju.
- Za sve tipove indeksa, višestruki uslovi opsega u kombinaciji sa *OR* ili *AND* formiraju uslov opsega.

Na slici ispod želimo da pribavimo sve datume plaćanja u određenom periodu. Kao što vidimo vrednost kolone *type* je *range*, što implicira korišćenje optimizacije opsega.

Zapravo, preuzimaju se samo redovi koji su u datom opsegu koristeći indeks *payments_paymentDate_index* koji je u našem slučaju *B* stablo.

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	payments	NULL	range	payments_paymentDate_index	payments_paymentDate_index	3	NULL	234	100.00	Using where; Using index

Slika 3.7. Range optimizacija

MySQL pokušava da izdvoji uslove opsega iz *WHERE* klauzule za svaki od mogućih indeksa. Tokom procesa ekstrakcije, uslovi koji se ne mogu koristiti za konstruisanje uslova opsega se odbacuju, uslovi koji proizvode opsege koji se preklapaju se kombinuju, a uslovi koji proizvode prazne opsege se uklanjaju.

Uslovi opsega kod višedelnih indeksa su proširenje uslova opsega za indeks sa jednim delom. Uslov opsega na višedelnom indeksu ograničava redove indeksa da leže unutar jednog ili više intervala ključeva. Intervali ključeva su definisani preko skupa ključeva, korišćenjem redosleda iz indeksa.

3.1.8. Index Merge optimizacija

Metod pristupa spajanja indeksa (*Index Merge*) preuzima redove sa više skeniranih opsega i spaja njihove rezultate u jedan. Ovaj metod pristupa objedinjuje skeniranja indeksa samo iz jedne tabele, a ne iz više tabela. Spajanje može da proizvede unije, preseke ili unije preseka osnovnih skeniranja.

Algoritam ima poznata ograničenja:

- Teško se bori sa kompleksnom *WHERE* klauzulom sa duboko ugnježenim *AND* i *OR* operatorima. Savetuje se da sam *developer* izvede sledeće transformacije:
 $(x \text{ AND } y) \text{ OR } z \Rightarrow (x \text{ OR } z) \text{ AND } (y \text{ OR } z)$
 $(x \text{ OR } y) \text{ AND } z \Rightarrow (x \text{ AND } z) \text{ OR } (y \text{ AND } z).$
- Nije primenjiv na *full-text* indekse.

U *EXPLAIN* izlazu ovaj metod se navodi kao *index_merge* u *type* koloni, dok u *extra* delu stoji o kom algoritmu je reč.

3.1.9. *Nested-Loop Join* algoritam

MySQL spaja tabele koristeći algoritam ugnježdene petlje ili pak neku njegovu varijaciju. Jednostavan algoritam ugnježdene petlje čita redove iz prve tabele u petlji (spoljnoj) jedan po jedan, prosleđujući svaki red u ugnježdenu (unutrašnju) petlju koja obrađuje sledeću tabelu u spoju. Ovaj proces se ponavlja onoliko puta koliko ima tabela za spajanje. Obično čita tabele u unutrašnjim petljama mnogo puta. Uz korišćenje indeksa nad odgovarajućom kolonom u unutrašnjoj tabeli, ne mora čitati celu tabelu.

3.1.10. *Block Nested-Loop Join* algoritam

Umesto slanja jednog reda unutrašnjoj petlji, kod ovog algoritma imamo slanje *buffer*-a, odnosno niza redova. Umesto poređenja cele tabele u unutrašnjoj petlji sa samo jednim redom iz spoljašnje, ista se upoređuje sa celokupnim sadržajem *buffer*-a i na taj način smanjujemo broj čitanja unutrašnje tabele i dobijamo bolje performanse.

3.1.11. *Hash Join* optimizacija

Podrazumevano, *MySQL* (8.0.18 i novije verzije) koristi *hash join* kad god je to moguće. Inače, *hash join* se koristi za velike tabele i sastoji se iz dva koraka: *build phase* i *probe phase*. U prvoj fazi se prolazi kroz celu prvu tabelu i izračunava se *hash* funkcija. Zatim, u drugoj fazi se prolazi kroz celu drugu tabelu i takođe se izračunava ista *hash* funkcija. Poklapanje ovih vrednosti predstavlja spoj. Moguće je kontrolisati da li se *hash join*-ovi koriste korišćenjem jednog od *hint*-a za optimizaciju *BNL* i *NO_BNL* ili postavljanjem *block_nested_loop=on* ili *block_nested_loop=off* kao deo podešavanja za sistemsku promenljivu servera *optimizer_switch*.

Hash join se može koristiti kada postoji jedan ili više indeksa koji se mogu koristiti za predikate jedne tabele.

Od *MySQL* 8.0.19 verzije, *hash_join* flag u *optimizer_switch*-u, kao ni *HASH_JOIN* i *NO_HASH_JOIN hint*-ovi više nemaju uticaja. *Hash join* je obično brži od algoritma blok ugnježdene petlje (*block nested loop*) korišćenog u prethodnim verzijama *MySQL*-a i zato je i namenjen da se koristi u takvim slučajevima. Počevši od *MySQL* 8.0.20, podrška za blok ugnježdenu petlju je uklonjena, a server koristi *hash join* gde god bi se ona ranije koristila.

Na slici ispod prikazano je korišćenje *hash join*-a prilikom spajanja 2 tabele.

1 • EXPLAIN SELECT * FROM courses JOIN courselines ON courses.courseDescription = courselines.textDescription;

	id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
▶	1	SIMPLE	courselines	NULL	ALL	NULL	NULL	NULL	NULL	7	100.00	NULL
	1	SIMPLE	courses	NULL	ALL	NULL	NULL	NULL	NULL	110	10.00	Using where; Using join buffer (hash join)

Slika 3.8. Korišćenje *hash join* optimizacije

3.1.12. Engine Condition Pushdown optimizacija

Ova optimizacija poboljšava efikasnost direktnih poređenja između neindeksirane kolone i konstante. U takvim slučajevima, uslov se „gura dole“ u mehanizam za skladištenje radi evaluacije. Ovu optimizaciju može da koristi samo *NDB* mehanizam za skladištenje. *NDB* (*Network DataBase*) je visoko dostupna i visoko redundantna verzija *MySQL*-a.

Za *NDB* klaster, ova optimizacija može eliminisati potrebu za slanjem nepodudarnih redova preko mreže između čvorova podataka klastera i *MySQL* servera koji je izdao upit i može ubrzati upite tamo gde se koristi za 5 do 10 puta u odnosu na slučajeve gde bi uslov mogao biti, ali se ne koristi.

Ukoliko se u *WHERE* klauzuli nalazi indeksirana kolona onda se koristi indeks, a ne *engine condition pushdown*, dok ukoliko je kolona neindeksirana, onda će se najverovatnije koristiti.

3.1.13. Index Condition Pushdown optimizacija

ICP je optimizacija za slučaj kada *MySQL* preuzima redove iz tabele koristeći indeks. Bez *ICP*-a, mehanizam za skladištenje prelazi indeks da bi locirao redove u osnovnoj tabeli i vraća ih *MySQL* serveru koji procenjuje *WHERE* uslov za redove. Kada je *ICP* omogućen, i ako se delovi uslova *WHERE* mogu proceniti korišćenjem samo kolona iz indeksa, *MySQL* server gura ovaj deo *WHERE* uslova dole u mašinu za skladištenje. Mehanizam za skladištenje tada procenjuje uslov gurnutog indeksa koristeći unos indeksa i samo ako je to zadovoljeno, red se čita iz tabele. *ICP* može smanjiti broj pristupa osnovnoj tabeli od strane mašine za skladištenje, kao i broj pristupa *MySQL* servera mašini za skladištenje.

Podrazumevano je omogućen i kod *InnoDB*-a i kod *MyISAM* tabela, a može se uključivati i isključivati po potrebi korišćenjem *index_condition_pushdown* *flag*-a kod sistemske promenljive *optimizer_switch*.

3.1.14. *Multi-Range Read* optimizacija

Čitanje redova korišćenjem skeniranja opsega na sekundarnom indeksu može rezultirati mnogim nasumičnim pristupima disku osnovnoj tabeli kada je tabela velika i nije uskladištena u kešu mehanizma za skladištenje. Sa optimizacijom za čitanje više opsega (*MRR*), *MySQL* pokušava da smanji broj nasumičnih pristupa disku za skeniranje opsega skenirajući samo indeks i prikupljajući ključeve za relevantne redove. Zatim se ključevi sortiraju i konačno se redovi preuzimaju iz osnovne tabele koristeći redosled primarnog ključa. Motivacija za *MRR* je smanjenje broja nasumičnih pristupa disku i postizanje sekvencijalnog skeniranja podataka osnovne tabele.

MRR omogućava pristup redovima podataka sekvencijalno, a ne nasumičnim redosledom. Server dobija skup indeksnih torki koji zadovoljavaju uslove upita, sortira ih prema redosledu *ID*-a redova podataka i koristi sortirane torke za preuzimanje redova podataka. Ovo čini pristup podacima efikasnijim i jeftinijim. Nije neophodno nabaviti sve indeksne torke pre početka čitanja redova podataka.

Sistemska promenljiva *optimizer_switch* ima *mrr* *flag* za optimizaciju čitanja u više opsega. Podrazumevano ovaj *flag* je postavljen i optimizator pravi izbor na osnovu troškova da li će biti korišćen ili ne.

3.1.15. Filtriranje uslova

U obradi spajanja, prefiksni redovi su oni redovi koji se prosleđuju iz jedne tabele u spoju na sledeću. Optimizator pokušava da stavi tabele sa malim brojem prefiksa rano u redosled spajanja kako bi sprečio brzo povećanje broja kombinacija redova. Optimizator bira najbolji plan izvršenja – plan izvršenja sa najmanjim troškovima.

Bez filtriranja uslova, broj redova prefiksa za tabelu je zasnovan na procenjenom broju redova izabranih klauzulom *WHERE* u skladu sa metodom pristupa koju optimizator odabere. Filtriranje uslova omogućava optimizatoru da koristi druge relevantne uslove u klauzuli *WHERE* koji nisu uzeti u obzir metodom pristupa i na taj način poboljša svoje procene broja redova prefiksa. Iako mogu postojati metod pristupa zasnovan na indeksu koji se može koristiti za odabir redova iz trenutne tabele u spoju, mogu postojati i dodatni uslovi za tabelu u klauzuli *WHERE* koji mogu filtrirati procenu za kvalifikacione redove prenete na sledeću tabelu.

Uslov doprinosi proceni filtriranja samo ako se odnosi na trenutnu tabelu, ako zavisi od konstante ili pak vrednosti iz ranijih spojeva ili ako nije uzet metodom pristupa.

3.1.16. *Constant-Folding* optimizacija

Poređenja između konstanti i vrednosti kolone u kojima je vrednost konstante van opsega ili pogrešnog tipa u odnosu na tip kolone sada se obrađuju jednom tokom optimizacije upita, a ne red po red tokom izvršavanja. Poređenja koja se mogu tretirati na ovaj način su $>$, $>=$, $<$, $<=$, $<>$, $!=$, $=$ i $<=>$.

Preklapanje se vrši za konstante u poređenju sa podržanim tipovima *MySQL* kolona na sledeći način – primera radi za tip kolone *integer* – ukoliko je konstanta van opsega tipa kolone, poređenje se preklapa samo sa 1 ili pak *IS NOT NULL*.

Ovde postoje određena ograničenja – ova optimizacija se ne može izvršiti ukoliko u poređenju imamo *BETWEEN* ili *IN*, ukoliko se treba izvršiti nad kolonom tipa datum (*date*, *time*), kao ni tokom faze pripreme jer u tom trenutku vrednost konstante nepoznata.

3.1.17. *Function Call* optimizacija

MySQL funkcije mogu biti kako determinističke, tako i nedeterminističke. Za razliku od determinističkih koje uvek za iste parametre vraćaju istu vrednost, nedeterminističke za iste parametre mogu vraćati različite vrednosti (na primer *RAND()*, *UUID()*). Deterministička funkcija koja kao parametar ima kolonu tabele, mora se ponovo izvršiti nakon promene vrednosti te kolone.

Nedeterminističke funkcije se računaju za svaki red u tabeli, te tako se ne koriste indeksi već potpuno skeniranje tabele. Ukoliko se u *WHERE* klauzuli nalazi neka ovakva funkcija, *SELECT* može vratiti 0, 1 ili čak više redova.

Ažuriranja koja se ne izvode deterministički nisu bezbedna za replikaciju.

Poteškoće proizilaze iz činjenice da se funkcija evaluira za svaki red pojedinačno. U cilju sprečavanja prethodno navedenog, predlaže se korišćenje sledećih tehnika:

- Premestiti izraz koji sadrži nedeterminističku funkciju u zasebnu naredbu, čuvajući vrednost u promenljivoj. U originalnoj naredbi zameniti izraz referencom na promenljivu, koju optimizator može da tretira kao konstantnu vrednost.
*SET @keyval = FLOOR(1 + RAND() * 49);*
UPDATE t SET col_a = some_expr WHERE id = @keyval;
- U *WHERE* delu imati deo ključa na osnovu koga će se koristiti indeks i evaluacija nedeterminističke funkcije će biti izvršene samo nekoliko puta.

3.1.18. Izbegavanje potpunog skeniranja tabela

Prilikom izvršenja *full table scan* operacije u koloni *type* u izlazu *EXPLAIN* naredbe se nalazi *ALL*. Obično se dešava pod sledećim uslovima:

- Za male tabele (manje od 10 redova koji imaju malu dužinu) jer je dosta brže u poređenju sa pretraživanjem indeksa;
- Nema upotrebljivih ograničenja u *ON* ili *WHERE* klauzulama za indeksirane kolone;
- Prilikom upoređivanja indeksirane kolone sa konstantom i *MySQL* je ustanovio da je pokriven preveliki deo tabele, te da mu se više isplati potpuno skeniranje;
- Korišćenje ključa sa niskom kardinalnošću (mnogi redovi odgovaraju vrednosti ključa).

Za male tabele, skeniranje tabele je često prikladno i uticaj na performanse je zanemarljiv. Za velike tabele treba isprobati sledeće tehnike u cilju navođenja optimizatora da ne izabere potpuno skeniranje tabele:

- Izvršavati naredbu *ANALYZE TABLE ime_tabele* radi ažuriranja distribucije ključeva;
- Koristiti *FORCE INDEX* kako bi se *MySQL* dao nagoveštaj da je skeniranje tabele dosta skupo i da koristi navedeni indeks;
- Pokrenuti *mysqld* sa parametrom *--max-seeks-for-key=1000* ili pak izvršiti *SET max_seeks_for_key=1000* kako bi optimizator zaključio da nijedno skeniranje ključa ne uzrokuje više od 1.000 traženja ključa.

3.1.19. Optimizacija ugnježđenih upita

S obzirom na činjenicu da je razvoj u toku, gotovo nijedan savet za optimizaciju nije stopostotno pouzdan na duže staze. U nastavku su dati neki od saveta koji se preporučuju u trenutnoj verziji *MySQL*-a.

Zameniti *join* sa podupitom:

```
SELECT DISTINCT column1 FROM t1 WHERE t1.column1 IN (SELECT column1 FROM t2);
```

umesto:

```
SELECT DISTINCT t1.column1 FROM t1, t2 WHERE t1.column1 = t2.column1;
```

Za podupite koji nisu u korelaciji i koji uvek vraćaju samo jedan red, bolje je koristiti = umesto *IN*. *MySQL* izvršava upite koji nisu u korelaciji samo jednom.

Prebaciti klauzule u ugnježđeni deo.

```
SELECT * FROM t1 WHERE s1 IN (SELECT s1 FROM t1 UNION ALL SELECT s1 FROM t2);
```

a ne:

```
SELECT * FROM t1 WHERE s1 IN (SELECT s1 FROM t1) OR s1 IN (SELECT s1 FROM t2);
```

```
SELECT (SELECT column1 + 5 FROM t1) FROM t2;
```

a ne:

```
SELECT (SELECT column1 FROM t1) + 5 FROM t2;
```

3.2. *INSERT*, *UPDATE* i *DELETE* optimizacija

Kombinovanjem mnogo malih operacija u jednu veliku vrši se optimizacija *INSERT* naredbe. U idealnom slučaju, napraviti jednu konekciju, poslati podatke za mnogo novih redova odjednom i odložiti sva ažuriranja indeksa i proveru doslednosti do samog kraja. Prilikom učitavanja tabele iz tekstualne datoteke koristiti *LOAD DATA* koja je i do 20 puta brža od *INSERT* naredbe. Iskoristiti činjenicu da kolone imaju podrazumevane vrednosti. Eksplicitno umetati vrednosti koje se razlikuju od podrazumevanih. Ovo smanjuje raščlanjivanje koje *MySQL* mora uraditi i poboljšava brzinu umetanja.

Brzina izvršenja *UPDATE* naredbe zavisi od količine podataka koje ažuriramo i da li su i indeksi pogođeni istim. Treba odložiti ažuriranja, a zatim uraditi mnogo ažuriranja zaredom kasnije. Izvođenje više ažuriranja zajedno je mnogo brže nego jedno po jedno ukoliko se zaključava tabela.

U cilju bržeg brisanja redova savetuje se povećanje vrednosti sistemske promenljive *key_buffer_size*. Ukoliko se vrši brisanje svih redova kod *MyISAM* tabele, savetuje se korišćenje *TRUNCATE TABLE tbl_name* jer je brže od *DELETE FROM tbl_name*.

3.3. Ostali optimizacioni saveti

U nastavku su dati još neki optimizacioni saveti:

- Ukoliko postoje podaci koji nisu u skladu sa strukturom tabele, mogu se skladištiti u *BLOB* koloni. U tom slučaju je neophodno obezbediti u aplikaciji pakovanje i raspakivanje podataka. Broj *I/O* operacija čitanja i pisanja je smanjen.
- U bazi podataka čuvati putanju do datoteke, a ne samu datoteku (primera radi neku sliku ili drugi binarni file).
- Ukoliko želimo ogromnu brzinu, možemo pristupati *storage engine*-u direktno umesto *SQL interface*-u.

3.4. *InnoDB* – optimizacija upita

Za razliku od *MyISAM storage engine*-a koji je veoma brz prilikom čitanja i jako dobar kod skeniranja potpunog teksta, ali koji prilikom ažuriranja zaključava celu tabelu, *InnoDB* podržava transakcije i premašuje po performansama pisanja prethodno navedeni

storage engine, pogotovo kod opterećenih baza podataka. Podrazumevani *storage engine* u verziji 8.0 je *InnoDB* (do verzije 5.5 bio je to *MyISAM*).

Kako bi optimalno bili podešeni upiti kod *InnoDB* tabela potrebno je kreirati odgovarajući skup indeksa za svaku tabelu. Smernice koje pomažu pri tome su:

- S obzirom da svaka *InnoDB* tabela ima primarni ključ (bez obzira da li se traži ili ne), navesti skup kolona primarnog ključa za svaku tabelu (kolone koje se koriste u najvažnijim i vremenski kritičnim upitima).
- Ne navoditi previše kolona u primarnom ključu ili pak da budu predugačke jer se vrednosti ovih kolona dupliraju u svakom sekundarnom indeksu. Kada indeks sadrži nepotrebne podatke, *I/O* za čitanje ovih podataka i memorija za njihovo keširanje smanjuju performanse i skalabilnost servera.
- Ne kreirati poseban sekundarni indeks za svaku kolonu jer svaki upit može da koristi samo jedan indeks. Indeksi na retko testiranim kolonama ili kolonama sa samo nekoliko različitih vrednosti možda neće biti od pomoći ni za jedan upit. Ako postoji mnogo upita za istu tabelu, testirajući različite kombinacije kolona, pokušati sa kreiranjem malih broja spojenih indeksa umesto velikog broja indeksa u jednoj koloni. Ako indeks sadrži sve kolone potrebne za skup rezultata (indeks pokrivanja), upit bi možda mogao u potpunosti da izbegne čitanje podataka tabele.
- Ukoliko indeksirana kolona ne može sadržati nijednu *NULL* vrednost, deklarirati je kao *NOT NULL* pri kreiranju tabele. Optimizator može bolje da odredi koji je indeks najefikasniji za korišćenje prilikom upita.

4. OPTIMIZATOR I PLAN IZVRŠENJA UPITA

Na osnovu detalja o tabelama, kolonama, indeksima, uslovima u *WHERE* klauzuli, *MySQL* optimizator razmatra različite tehnike radi što efikasnijeg pretraživanja upita (primera radi pri spajanju velikog broja tabela ne mora se vršiti poređenja kombinacija svih mogućih redova, ili pak ne moraju se čitati svi redovi kod ogromnih tabela). Skup operacija za koje se optimizator odlučuje se naziva plan izvršenja upita (*Query Execution Plan*), odnosno *EXPLAIN* plan. Cilj je razumeti plan izvršenja kako bi se uvidelo na neke neefikasne operacije i poboljšao isti.

4.1. *EXPLAIN* naredba

EXPLAIN naredba pruža informacije o tome kako *MySQL* izvršava naredbe:

- Radi sa *SELECT*, *DELETE*, *INSERT*, *REPLACE* i *UPDATE* naredbama;
- Kada naredbi prethodi *EXPLAIN*, *MySQL* prikazuje informacije iz optimizatora o planu izvršenja naredbe;
- Podrazumevano, plan se prikazuje u tabeli, ali to se može promeniti postavljanjem opcije *FORMAT* na, primera radi, *JSON* i na taj način se dobijaju informacije u *JSON* formatu.

Pomoću *EXPLAIN*-a možemo videti gde bismo mogli dodati indeks radi bržeg pronalaska redova i samim tim bržeg izvršavanja upita. Takođe, može se koristiti i za proveru optimalnog redosleda spajanja tabela.

Ukoliko se smatra da se pojedini indeksi ne koriste kada bi trebalo, pokrenuti *ANALYZE TABLE* naredbu radi ažuriranja statistike tabele. Ovo može uticati na buduće izbore koje optimizator donosi.

Objasnićemo na primeru. Nakon izvršenja naredbe sa slike ispod, dobili smo rezultat u vidu tabele. Kolona *select_type* ima vrednost *SIMPLE* što znači da je reč o jednostavnom upitu koji nema spojeva ili podupita. Vrednost kolone *type* je *ALL* što znači da je vršeno potpuno skeniranje tabele. U našem primeru je reč o veoma maloj tabeli (sa samo 16 redova) i u tom slučaju je to bolje u poređenju sa korišćenjem indeksa (koji prethodno mora biti kreiran i uskladišten). Za velike tabele sa ogromnim brojem torki, tehnika potpunog skeniranja je gotovo neprihvatljiva i preporučuje se kreiranje indeksa nad kolonom/-ama koja/-e se često pretražuje/-u (još bolje bi bilo ako nema/-ju mali skup vrednosti). Ukoliko bi se koristio indeks, vrednost kolone *type* bi bila *ref*. Vrednost kolone *filtered* je 10,00 i predstavlja procenu optimizatora u vidu procenta broja redova koji bi mogli biti vraćeni kao rezultat. U našem primeru očekuje između 1 i 2 reda (10% od 16 redova = 1,6), ali ipak smo kao krajnji rezultat dobili 3 reda što se da videti na slici ispod.

1 • `EXPLAIN SELECT studentNumber, LastName, FirstName FROM students WHERE country='France';`

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	students	NULL	ALL	NULL	NULL	NULL	NULL	16	10.00	Using where

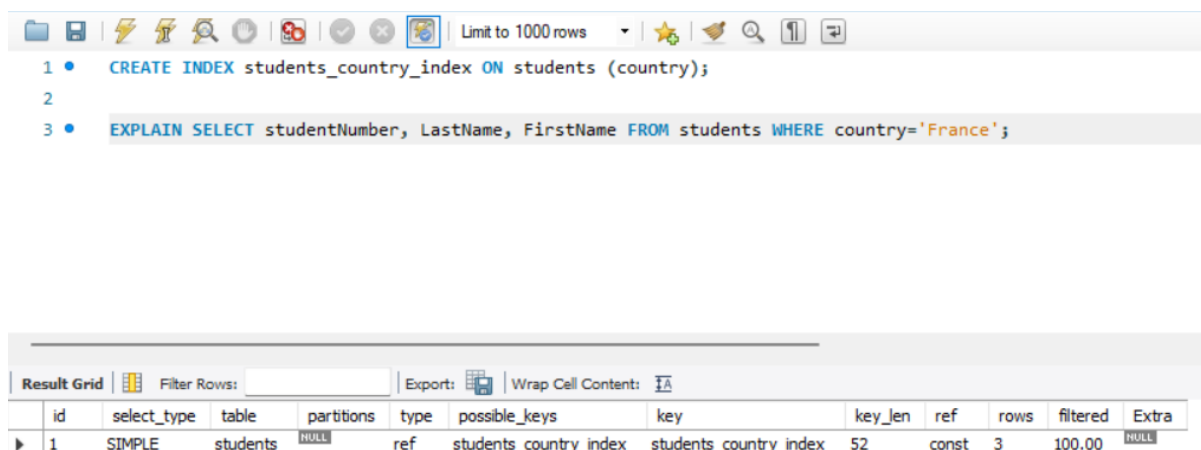
Slika 4.1. Rezultat izvršenja naredbe *EXPLAIN*

1 • `SELECT studentNumber, LastName, FirstName FROM students WHERE country='France';`

	studentNumber	LastName	FirstName
▶	103	Schmitt	Carine
	119	Labrune	Janine
	146	Saveley	Mary
*	NULL	NULL	NULL

Slika 4.2. Rezultat izvršenja naredbe pribavljanja studenata iz Francuske

Nakon kreiranja indeksa nad kolonom *country* i izvršenjem naredbe *EXPLAIN*, dobili smo rezultat u vidu tabele koju vidimo na slici ispod. S obzirom da je vrednost kolone *type* *ref*, reč je o korišćenju indeksa i to *students_country_index* (vrednost kolone *key*).



Slika 4.3. Rezultat izvršenja naredbe *EXPLAIN* nakon kreiranja indeksa

Ukoliko želimo prikaz u *JSON* formatu, izvršiti sledeću naredbu:

```
EXPLAIN FORMAT=JSON SELECT studentNumber, LastName, FirstName FROM
students WHERE country='France';
```

4.2. Kontrolisanje optimizatora

MySQL omogućava kontrolu optimizatora kroz sistemske promenljive (utiču na kreiranje plana izvršenja), davanje nekih *hint*-ova optimizatoru. Server održava statistiku histograma o vrednostima kolona.

Zadatak optimizatora je pronalaženje optimalnog plana za izvršenje upita. S obzirom na činjenicu da razlika u dobrom i lošem planu može mnogo varirati, *MySQL* vrši iscrpnu pretragu najboljeg plana proučavajući sve moguće. Kod spojeva, sa porastom broja tabela broj planova izvršenja eksponencijalno raste. Kod složenijih upita, vreme provedeno u optimizaciji upita može postati glavno usko grlo u performansama servera.

Fleksibilniji metod za optimizaciju upita omogućava korisniku da kontroliše opsežnost optimizatora u svojoj potrazi za najboljim planom. Ideja je da što manje planova optimizator istraži, kako bi manje vremena trošio na sastavljanje upita. S druge strane, s obzirom da optimizator preskače neke planove, može propustiti najbolji.

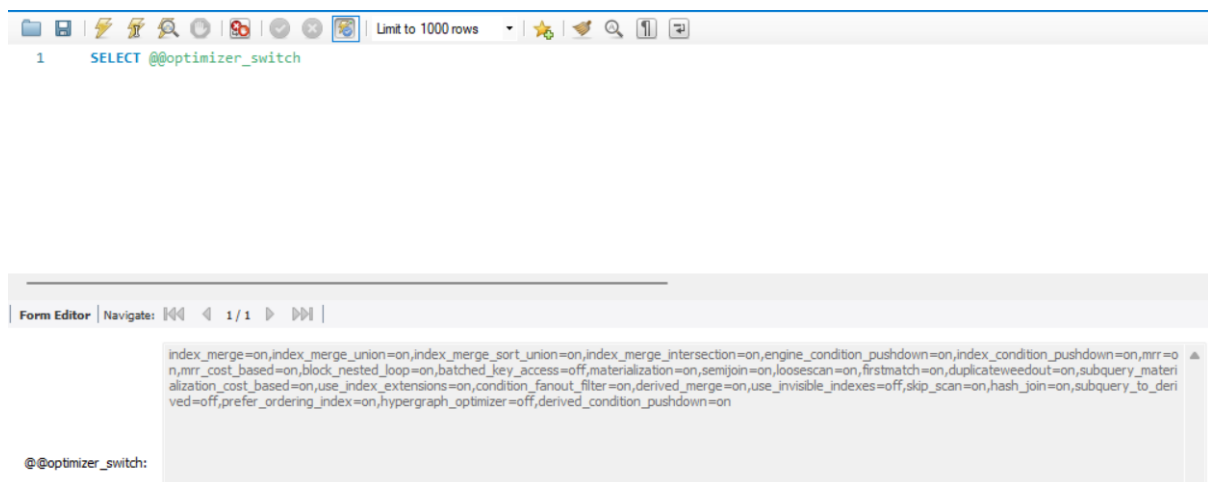
Prethodno opisano se podešava korišćenjem sledećih sistemskih promenljivih:

- *optimizer_prune_level* – promenljiva govori optimizatoru da preskoči određene planove na osnovu procene broja redova kojima se pristupa za svaku tabelu. Iskustvo pokazuje da retko promašuje optimalne planove i može dramatično smanjiti vreme

kompilacije upita. Zbog toga je ova opcija podrazumevano uključena. Naravno, ova opcija se može isključiti uz rizik da kompilacija upita može potrajati mnogo duže.

- *optimizer_search_depth* – podrazumevano je vrednost 0 što znači da optimizator sam procenjuje vrednost. Naravno *developer* može postaviti željeni broj dubine. Što je ovaj broj manji to kompilacija upita traje kraće, ali se najverovatnije neće dobiti najbolji plan izvršenja. Sa druge strane, ukoliko imamo na primer 15-ak tabela uključenih u upitu i postavimo dubinu na 15 (heuristički mnogo), kompilacija ovakvog upita može trajati i do nekoliko sati, čak i dana.

Sistemska promenljiva *optimizer_switch* omogućava kontrolu ponašanja optimizatora. Vrednost ove promenljive je skup *flag*-ova čiji redosled nije bitan. Vrednost *flag*-a može biti podrazumevano ili *on* ili *off*. Može se promeniti tokom izvršenja. Naredbom `SELECT @@optimizer_switch` možemo videti trenutni skup *flag*-ova. Rezultat izvršenja sa trenutnom vrednošću *flag*-ova se vidi na slici ispod.



Slika 4.4. Vrednosti *flag*-ova sistemske promenljive *optimizer_switch*

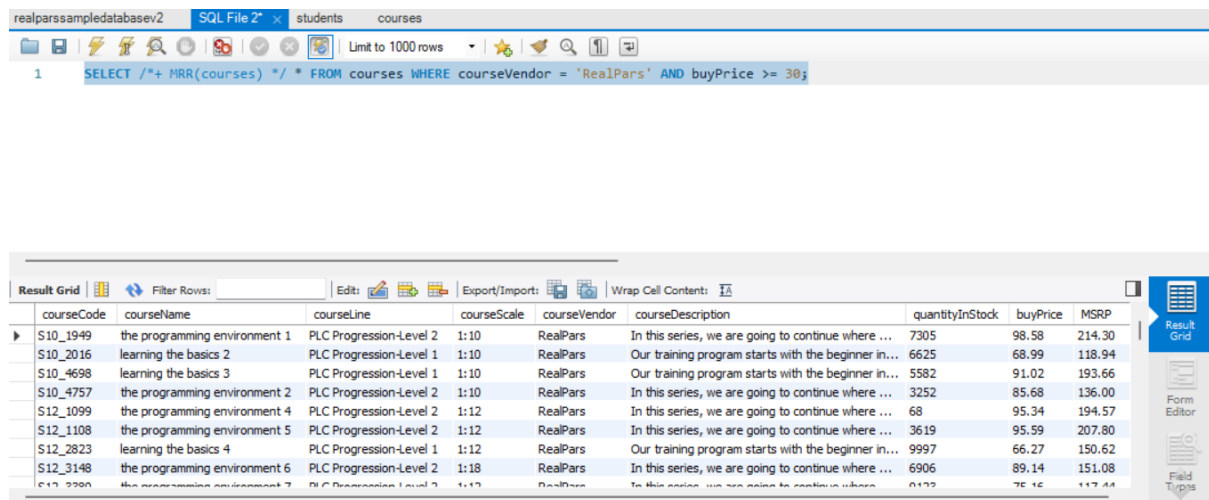
Primer promene vrednosti nekog *flag*-a:

```
SET optimizer_switch='index_merge_union=off,index_merge_sort_union=off';
```

Bitno je napomenuti da vrednost *flag*-ova koji u gore navedenoj naredbi nisu pomenuta, ostaju nepromenjena.

Na ovaj način utičemo na izvršenje svih narednih upita. Ukoliko želimo da utičemo samo na pojedinačni upit ili pak neki njegov deo, savetuje se korišćenje *hint*-ova optimizatora. Pružaju finiju kontrolu optimizatora i imaju prvenstvo nad *flag*-om sistemske promenljive *optimizer_switch*. Specificiraju se unutar `/*+ ... */` u upitu. Moramo biti oprezni jer se mi na ovaj način praktično mešamo u rad optimizatora i možemo pogoršati izvršenje istog.

Primer korišćenja *hint*-a je dat na slici ispod. Zahtevamo da optimizator koristi *Multi-Range Read* optimizaciju nad tabelom *courses* u cilju smanjenja broja nasumičnih pristupa disku.

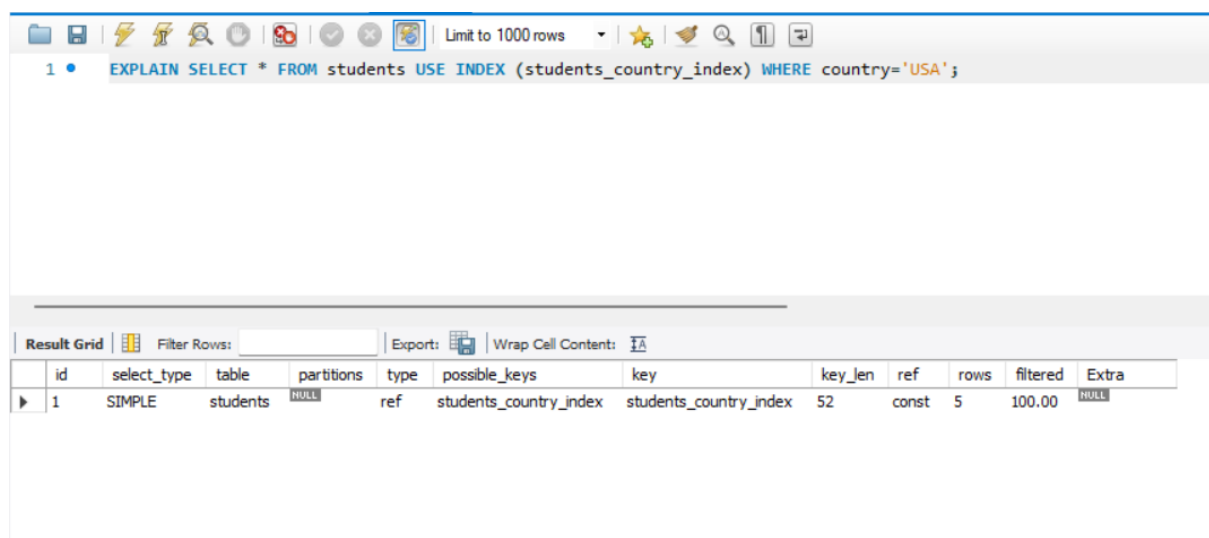


Slika 4.5. *Hint* optimizatora - *Multi-Range Read* optimizacija

Hint-ovi indeksa pružaju optimizatoru informacije o načinu izbora indeksa prilikom obrade upita. *Hint*-ovi optimizatora i indeksa se razlikuju i mogu se koristiti zajedno ili odvojeno. Za razliku od *hint*-ova optimizatora koji se svuda mogu koristiti, *hint*-ovi indeksa se mogu koristiti samo sa *SELECT* i *UPDATE* naredbama (i *DELETE* koji radi sa više tabela).

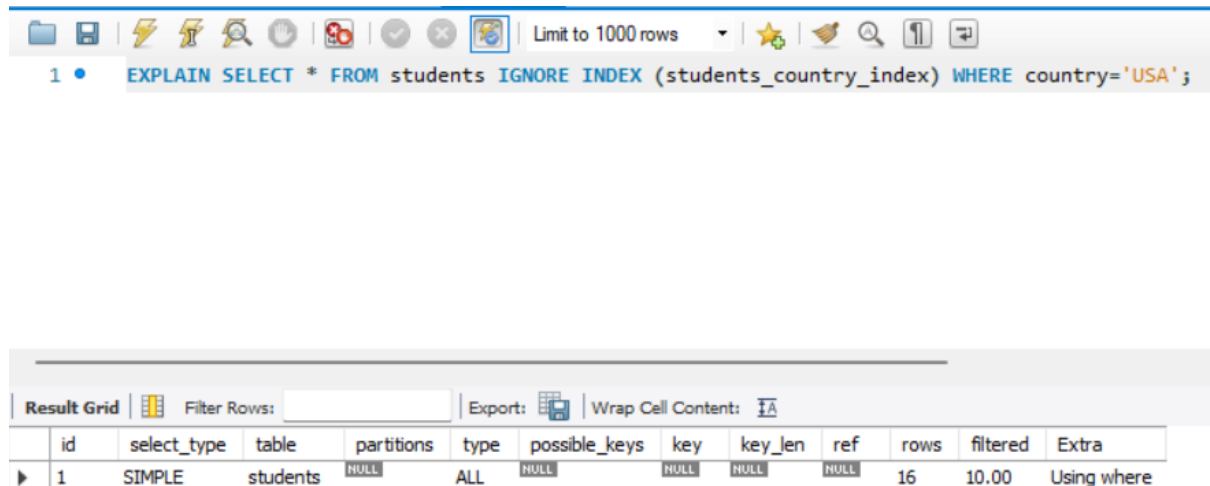
Primeri:

```
SELECT * FROM students USE INDEX (students_country_index) WHERE country='USA';
```



Slika x.6. Zahtevanje korišćenja *students_country_index* indeksa prilikom upita

Na slici iznad, mi govorimo optimizatoru da koristi postojeći indeks za pretragu. Za razliku od ovog primera, daćemo jedan primer u nastavku u kome kazujemo da prilikom upita ne koristi indeks (što se da videti u tabeli ispod – kolona *type* ima vrednost *ALL* – vrši se potpuno skeniranje tabele).



The screenshot shows a SQL IDE interface. The top toolbar includes icons for file operations, execution, and a 'Limit to 1000 rows' dropdown. The SQL editor contains the following query:

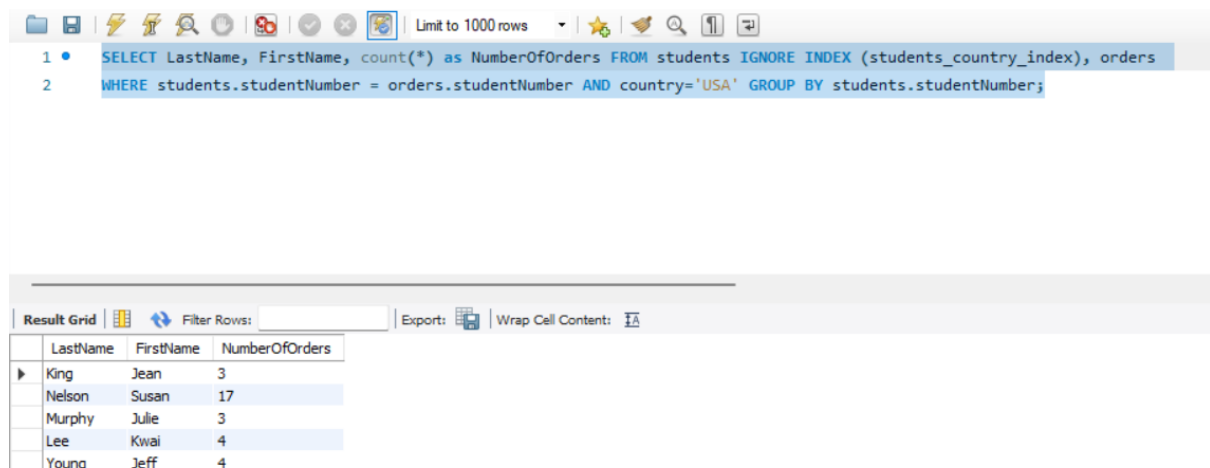
```
1 • EXPLAIN SELECT * FROM students IGNORE INDEX (students_country_index) WHERE country='USA';
```

Below the editor is the 'Result Grid' tab. It displays the execution plan for the query. The columns are: id, select_type, table, partitions, type, possible_keys, key, key_len, ref, rows, filtered, and Extra.

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	students	NULL	ALL	NULL	NULL	NULL	NULL	16	10.00	Using where

Slika 4.7. Zahtevanje ignorisanja *students_country_index* indeksa prilikom upita

Na slici ispod je prikazan jedan složeniji upit (želimo da vidimo ukupan broj narudžbina studenata iz Amerike). S obzirom da znamo da je tabela *students* mala želimo da se ignoriše kreirani indeks nad kolonom *country* jer verujemo da će potpuno skeniranje tabele odraditi brže posao.



The screenshot shows a SQL IDE interface. The top toolbar is the same as in the previous image. The SQL editor contains the following query:

```
1 • SELECT LastName, FirstName, count(*) as NumberOfOrders FROM students IGNORE INDEX (students_country_index), orders
2 WHERE students.studentNumber = orders.studentNumber AND country='USA' GROUP BY students.studentNumber;
```

Below the editor is the 'Result Grid' tab. It displays the results of the query. The columns are: LastName, FirstName, and NumberOfOrders.

LastName	FirstName	NumberOfOrders
King	Jean	3
Nelson	Susan	17
Murphy	Julie	3
Lee	Kwai	4
Young	Jeff	4

Slika 4.8. Zahtevanje ignorisanja *students_country_index* indeksa prilikom složenijeg upita

Za generisanje plana izvršenja, optimizator koristi modela troškova (*the optimizer cost model*) koji se zasniva na proceni troškova različitih operacija koje se javljaju u upitu. Optimizator ima skup podrazumevanih konstanti troškova dostupnih pri donošenju odluke. Optimizator takođe ima bazu podataka sa procenama troškova koju koristi tokom izrade plana izvršenja. Ove procene se čuvaju u tabelama *server_cost* i *engine_cost* u sistemskoj bazi podataka i mogu se konfigurisati u bilo kom trenutku. Vrednosti koje se nalaze u ovim tabelama se ažuriraju tokom vremena. Cilj je prilagođavanje procena troškova koje optimizator koristi kada pokušava da dođe do planova izvršenja upita.

4.3. Statistika optimizatora

Tabela rečnika podataka *column_statistics* čuva statistiku histograma o vrednostima kolona i optimizator je koristi za konstruisanje planova izvršenja upita. Naredbom *ANALIZE TABLE* vršimo upravljanje histogramom.

Tabela *column_statistics* ima sledeće karakteristike:

- Tabela sadrži statistiku za kolone svih tipova podataka osim za tipova geometrije (prostorni podaci) i *JSON*;
- Tabela je postojana tako da se statistika kolona ne mora kreirati svaki put nakon pokretanja servera;
- Server vrši ažuriranja tabele, a ne korisnici.

Optimizator koristi statistiku histograma (ukoliko je primenljiva) za kolone bilo kog tipa podataka za koje se prikuplja ista. Optimizator primenjuje statistiku histograma radi određivanja procene redova na osnovu selektivnosti (efekata filtriranja) poređenja vrednosti kolone sa konstantnim vrednostima.

Statistika histograma je korisna prvenstveno za neindeksirane kolone. Dodavanje indeksa koloni za koju je primenljiva statistika histograma takođe može pomoći optimizatoru da napravi procene redova. Kompromisi su:

- Indeks se mora ažurirati nakon modifikovanja podataka;
- Histogram se kreira ili ažurira samo na zahtev, tako da nema dodatnih troškova nakon modifikovanja podataka (može se desiti da imamo zastarelu statistiku).

Optimizator preferira procene redova optimizatora opsega od onih dobijenih iz statistike histograma. Ako optimizator utvrdi da se optimizator opsega primenjuje, on ne koristi statistiku histograma.

Za indeksirane kolone, indeksi daju bolje procene od histograma. Ukoliko imamo vrednost u koloni *filtered* prilikom izvršenja *EXPLAIN*, to znači da je korišćena statistika. Ako smatramo da imamo zastareli histogram, treba izvršiti naredbu *ANALIZE TABLE*.

Za isključivanje statistike izvršiti:

```
SET optimizer_switch='condition_fanout_filter=off';
```

5. ZAKLJUČAK

Na kraju rada, ostalo je izvesti zaključke iz celokupnog istraživanja date teme. Optimizacija upita ima ogromnu primenu u produkciji, a *MySQL* je jedna od najpopularnijih baza podataka. Treba biti veoma oprezan pri davanju *hint*-ova optimizatoru kako ne bismo drastično pogoršali performanse. Istražujući ovu temu, stekao sam veliko praktično znanje koje će mi bez sumnje značiti u budućem radu.

LITERATURA

[1] *MySQL*, <https://dev.mysql.com/>, april 2023.