Elektronski fakultet, Univerzitet u Nišu

Sistemi za upravljanje bazama podataka

Seminarski rad

Optimizacija upita u MySQL sistemu za upravljanje bazama podataka

Mentor:

Aleksandar Stanimirović

Student:

Mila Mirović 1525

Niš, april 2023.

Sadržaj

Uvod

MySQL je besplatni relacioni sistem za upravljanje bazama podataka koji omogućava organizovanje, skladištenje i upravljanje velikim količinama podataka. ***DBMS*** *(Data Base Management Systems)* ili ***sistem za upravljanje bazama podataka*** je skup programa koji se koriste za upravljanje velikim količinama podataka, koji se sastoje od baza podataka, tabela, redova i kolona, a omogućavaju korisnicima manipulaciju i kreiranje upita nad podacima. Zbog svoje jednostavnosti korišćenja, skalabilnosti, pouzdanosti i efikasnosti u obradi velikih količina podataka, MySQL je jedan od najpopularnijih sistema za upravljanje bazama podataka koji se koristi širom sveta. Baze podataka mogu imati prilično kompleksnu strukturu, pa samim tim i upiti nad njima mogu biti složeni i mogu uticati na efikasnost i brzinu rada DBMS-a. Optimizacijom upita moguće je pronaći najefikasniji način obrade upita u minimalnom vremenskom periodu. Da se taj proces ne bi sveo na prolazak kroz sve moguće načine izvršenja, fokus optimizacije je na pronalasku dovoljno dobrog rešenja u razumnom vremenu. MySQL koristi ***SQL*** (*Structured Query Languag*e), standardni jezik za manipulaciju nad podacima i upravljanje bazom podataka. Performanse baze podataka zavise od nekoliko faktora na nivou baze podataka, kao što su tabele, upiti i podešavanja konfiguracije.

MySQL vrši automatsku optimizaciju upita i za to koristi ***MySQL optimizator upita*** koji analizira uput i generiše optimalan plan izvršenja upita. Skup operacija koje optimizator bira za izvođenje najefikasnijeg upita naziva se ***plan izvršenja upita*** ili EXPLAIN plan. Optimizator koristi informacije o strukturi tabela, indeksima i statističkim podacima kako bi izabrao najefikasniji način izvršavanja upita. MySQL automatski kešira plan izvršenja upita da bi izbegao nepotrebno ponavljanje optimizacije za česte upite. Naravno, u nekim situacijama automatska optimizacija neće biti dovoljna, pa je potrebno manuelno pisanje otpimizovanog upita od strane korisnika. Postoji više tehnika koje mogu biti primenjene za optimizaciju upita u MySQL-u, a neke od njih su indeksiranje i podupiti.

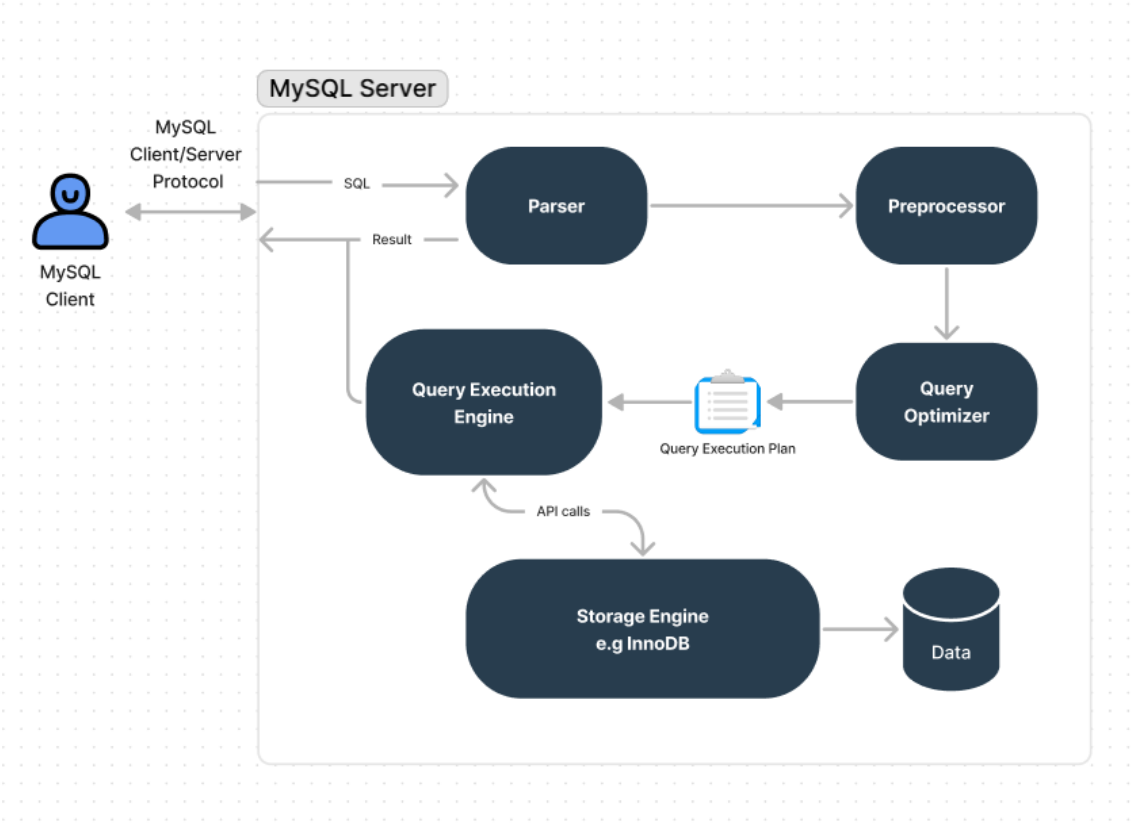
1. Optimizacija SQL upita u MySQL-u

MySQL optimizator je softverski mehanizam odgovoran za analizu SQL upita i izbor najefikasnijeg načina za izvršavanje upita, koji koirsti informacije o samoj strukturi baze podataka i rasporedu podataka kako bi pronašao optimalan način za obradu upita. Kada se neki upit pošalje MySQL serveru, optimizator se automatski pokreće i pokušava da nađe najbolji način za izvršavanje upita, transformiše originalni upit tako da daje isti rezultat kao i originalni ali brže. Dakle, cilj optimizacije upita je minimizirati količinu resursa i vremena potrebnog za izvršavanje upita, a maksimizirati brzinu izvršenja. U nastavku je dat kratak opis procesa izvršenja upita, kao i posao koji MySQL optimizator obavlja:

1. MySQL klijent šalje upit do MySQL servera koristeći MySQL Client/Server protokol.
2. Parsiranje i preprocesiranje: Optimizator prvo parsira upit kako bi odredio njegovu sintaksu i semantiku. Takođe proverava upit za eventualne greške u sintaksi i generiše internu reprezentaciju upita koju može koristiti za dalju obradu.
3. Analiza: optimizator vrši analizu strukture SQL upita tako što proverava da li je upit sintaksno i sematički korektan, kao i da li postoji odgovarajuća tabela ili indeks za svaki deo upita. Određuje koje tabele i kolone su uključene u upit, koji su dostupni indeksi i koje vrste su potrebne.
4. Transformacija: Na osnovu analize, optimizator transformiše upit u efikasniji oblik koji se može brže izvršiti.
5. Procena troškova: Optimizator procenjuje troškove izvršavanja upita pomoću različitih planova izvršavanja. Troškovi uključuju faktore kao što su I/O operacije na disku, korišćenje procesora i korišćenje memorije. Optimizator može da potraži statistike o tabelama iz upita pre samog izvršenja pomoću Storage engine-a.
6. Generisanje planova: Optimizator generiše skup mogućih planova izvršavanja, svaki sa drugačijom procenom troškova.
7. Selekcija plana: Optimizator bira plan izvršavanja sa najnižom procenom troškova i vraća ga MySQL serveru za izvršavanje.
8. ***Query Execution Engine*** [1]izvršava odabrani plan kreiranjem poziva ka ***Storage Engine***[2]-u preko specijalnih interfejsa.
9. MySQL server šalje rezultate ka MySQL klijentu.

[1] ***Query Execution Engine*** je softverska komponenta MySQL-a koja je zadužena za obradu SQL upita koje korisnik šalje bazi podataka. Kada korisnik pošalje upit, ona prvo analizira sintaksu upita kako bi utvrdio da li je ispravno napisan. Ako je upit ispravan, izvršni motor upita zatim analizira upit i generiše plan izvršavanja, kao što je opisano.

[2] ***Storage Engine*** u MySQL-u je softverska komponenta koja je odgovorna za upravljanje fizičkim skladištenjem podataka u bazi podataka. Zadužena je za organizovanje i manipulisanje podacima na disku, što omogućava efikasno čitanje i pisanje podataka u bazu podataka.



Slika 1. Proces izvršavanja upita u MySQL-u

Indeksi u MySQL DBMS-u su strukture koje se koriste za ubrzavanje pretrage podataka u bazi podataka. U osnovi, indeksi su slični indeksu u knjizi. Umesto pretraživanja svake stranice knjige u potrazi za određenim poglavljem, dovoljno je pogledati indeks koji pokazuje na kojoj stranici se nalazi to poglavlje.

Indeksi se sastoje od jedne ili više kolona u tabeli baze podataka, a svaka kolona je sortirana po vrednosti. Kada se pretražuje baza podataka, DBMS koristi indeks da bi brzo pronašao podatke koji odgovaraju zadatim kriterijumima pretrage. Bez indeksa, DBMS bi morao pregledati sve zapise u tabeli da bi pronašao odgovarajuće podatke, što bi bilo vrlo sporo za velike količine podataka. Postoje različite vrste indeksa u MySQL DBMS-u, uključujući primarni ključ (primary key), jedinstveni indeks (unique index), indeks sa više kolona (multi-column index), i puni tekstualni indeks (full-text index). Svaki od ovih indeksa se koristi za različite vrste pretraga i optimizaciju upita.

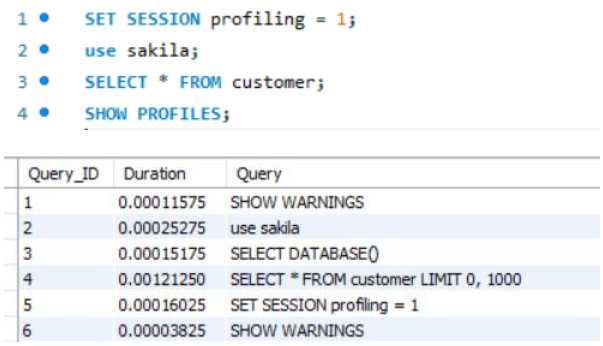
MySQL koristi indekse za sledeće operacije:

- Brzo pronalaženje redova koji odgovaraju WHERE klauzuli.
- Za eliminisanje vrsta iz razmatranja. Ako postoji više izbora za indekse, MySQL obično koristi indeks koji pronalazi najmanji broj vrsta (najselektivniji indeks).

- Ako tabela ima indeks sa više kolona, bilo koji levi prefiks indeksa može se koristiti od strane optimizatora da bi pronašao redove. Na primer, ako imate indeks sa tri kolone na (col1, col2, col3), možete pretraživati po (col1), (col1, col2) i (col1, col2, col3).
- Za dobijanje redova iz drugih tabela pri izvođenju spojeva. MySQL može efikasnije koristiti indekse na kolonama ako su oni deklarisani istim tipom i veličinom. U ovom kontekstu, VARCHAR i CHAR se smatraju istim ako su deklarisani istom veličinom. Na primer, VARCHAR (10) i CHAR (10) su iste veličine, ali VARCHAR (10) i CHAR (15) nisu.
- Za pronalaženje MIN () ili MAX () vrednosti za određenu indeksiranu kolonu key_col.
- Za sortiranje ili grupisanje tabele ako se sortiranje ili grupisanje vrši na levom prefiksu upotrebljivog indeksa (ORDER BY key_part1, key_part2).

Da bi se video proces izvršenja upita, može se koristiti `INFORMATION_SCHEMA.PROFILING` tabela koja sadrži informacije o upitima koji se izvršavaju u trenutnoj sesiji. Po default-u je onemogućeno njeno korišćenje, ali se lako omogućava komandom:

```
SET SESSION profiling = 1;
```



```
1 • SET SESSION profiling = 1;
2 • use sakila;
3 • SELECT * FROM customer;
4 • SHOW PROFILES;
```

Query_ID	Duration	Query
1	0.00011575	SHOW WARNINGS
2	0.00025275	use sakila
3	0.00015175	SELECT DATABASE()
4	0.00121250	SELECT * FROM customer LIMIT 0, 1000
5	0.00016025	SET SESSION profiling = 1
6	0.00003825	SHOW WARNINGS

Slika 2. Prikaz INFROMATION_SCHEMA.PROFILING tabele

SHOW PROFILES upit vraća listu upita iz trenutne sesije i ima 3 kolone:

- **query_id** – jedinstven identifikator upita
- **duration** – vreme trajanja izvršenja upita
- **query** – kolona prikazuje sadržaj upita koji se izvršio

Više informacija o konkretnom upitu možemo dobiti sledećom komandom:

```
SELECT * FROM INFORMATION_SCHEMA.PROFILING WHERE QUERY_ID=4;
```

```
1 •  SET SESSION profiling = 1;
2 •  use sakila;
3 •  SELECT * FROM customer;
4 •  SHOW PROFILES;
5 •  SELECT * FROM INFORMATION_SCHEMA.PROFILING WHERE QUERY_ID=4;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

QUERY_ID	SEQ	STATE	DURATION	CPU_USER	CPU_SYSTEM
4	2	starting	0.000055	0.000000	0.000000
4	3	Executing hook on transaction	0.000003	0.000000	0.000000
4	4	starting	0.000004	0.000000	0.000000
4	5	checking permissions	0.000006	0.000000	0.000000
4	6	checking permissions	0.000002	0.000000	0.000000
4	7	Opening tables	0.000058	0.000000	0.000000
4	8	init	0.000004	0.000000	0.000000
4	9	System lock	0.000008	0.000000	0.000000
4	10	optimizing	0.000003	0.000000	0.000000
4	11	statistics	0.000036	0.000000	0.000000
4	12	preparing	0.000021	0.000000	0.000000
4	13	executing	0.000953	0.000000	0.000000
4	14	end	0.000006	0.000000	0.000000
4	15	query end	0.000005	0.00 0.000000	000000
4	16	waiting for handler commit	0.000010	0.000000	0.000000
4	17	closing tables	0.000010	0.000000	0.000000
4	18	freeing items	0.000013	0.000000	0.000000
4	19	cleaning up	0.000019	0.000000	0.000000

Slika 3. Prikaz detalja o samom upitu

Optimizator koristi informacije o indeksima, statistici i rasporedu podataka da bi se procenilo koliko vremena će biti potrebno da se obradi upit na različite načine. Nakon što optimizator napravi plan za izvršavanje upita, server izvršava upit na najefikasniji način koji je predložio optimizator. Koriste se različite tehnike i algoritmi za generisanje najefikasnijeg plana izvršavanja upita. To uključuje heuristike, optimizaciju zasnovanu na pravilima, optimizaciju zasnovanu na troškovima i optimizaciju zasnovanu na statistici. Heuristike su jednostavna, opšta pravila koja optimizator koristi kako bi brzo generisao razumnu strategiju izvršavanja. Optimizacija zasnovana na pravilima predstavlja skup unapred definisanih pravila koje optimizator koristi za generisanje plana izvršavanja na osnovu strukture upita. Optimizacija zasnovana na troškovima koristi procene troškova kako bi se odredio najefikasniji plan izvršavanja. Optimizacija zasnovana na statistici koristi statističke podatke o tabelama i indeksima kako bi procenila troškove različitih planova izvršavanja.

U MySQL-u postoji nekoliko strategija koje se koriste za pronalaženje optimalnog plana za izvršavanje upita:

- **Potpuno pretraživanje tabele (full scan)** - ovaj plan uključuje pretraživanje svih redova u tabeli. Ova strategija je najsporija i koristi se samo kada tabela nema indekse u sebi ili kada se pretražuje mali broj redova.

- **Indeksiranje** – optimizator koristi indekse da bi pronašao potrebne redove u tabeli. Ovo je najčešća strategija i obično je najbrža. Indeksi su strukture podataka koje omogućavaju brži pristup podacima u bazi tako što se stvara lista vrednosti i adresa u tabeli. Oni pomažu MySQL serveru da brže izvrše upite jer smanjuju količinu podataka koju treba pregledati. Međutim, indeksiranje može imati i negativan uticaj na performanse ako je narušena struktura podataka ili ako je napravljen neadekvatan indeks. Važno je odabrati odgovarajuće kolone za indeksiranje i izbegavati indeksiranje velikih tekstualnih polja ili polja koja se retko koriste.
- **Korišćenje EXPLAIN naredbe** – EXPLAIN naredba omogućava korisnicima da prate kako MySQL planira da izvrši upit i da identifikuju potencijalne probleme sa upitom. Ovo može pomoći u identifikovanju nedostataka u dizajnu tabela ili upita koji mogu usporiti izvršavanje upita. Za detaljniji pregled rada MySQL optimizatora može se dodatno koristiti i OPTIMIZER_TRACE tabela. Pored obične EXPLAIN naredbe postoji i EXPLAIN ANALYZE naredba koja je vrlo slična običnoj, pri čemu je ključna razlika je u tome što je EXPLAIN ANALYZE detaljnija opcija jer pruža više statistike o izvršenju upita. Druga razlika je što EXPLAIN ne izvršava upite, dok EXPLAIN ANALYZE izvšava da bi dobio statistiku.
- **Normalizacija tabela** – podrazumeva organizaciju podataka u više tabela tako da se podaci ne ponavljaju. Ovo može pomoći u smanjenju veličine tabela i povećanju efikasnosti upita.
- **Korišćenje keširanja** – keširanje može smanjiti vreme potrebno za izvršavanje upita tako što se podaci skladište u keš memoriji umesto da se stalno čitaju iz baze podataka. MySQL ima nekoliko keša, uključujući keš za upita, keš za pohranu ključeva i keš za skladištenje metapodataka.
- **Optimizacija JOIN operacije** – kada se koristi JOIN operacija, optimizator mora da odluči kako da spoji dve ili više tabela. Optimizator može koristiti nekoliko strategija za JOIN operacije, uključujući spajanje pomoću indeksa, spajanje preko privremenih tabela ili spajanje preko spoljnih upita.
- **Optimizacija podupita** – kada se koristi podupit, optimizator mora da odluči kada da izvrši podupit i kako da to uradi na najbolji način.

1.1. Optimizacija upita EXPLAIN naredbom

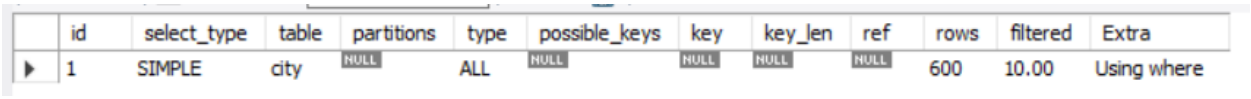
EXPLAIN naredba u MySQL-u omogućava nam da vidimo kako će MySQL optimizator izvršiti određenu naredbu, što može biti veoma korisno u identifikovanju neefikasnosti u upitu, kao što su sporo vreme izvršavanja, visoka upotreba CPU-a ili prekomerno korišćenje I/O operacija. Pomoću EXPLAIN naredbe, može se videti plan izvršavanja za upit, koji prikazuje kako će baza podataka pristupiti neophodnim podacima da bi se vratili rezultati upita. Ova naredba radi sa *SELECT, DELETE, INSERT, REPLACE* i *UPDATE* naredbama. EXPLAIN naredba prikazuje informacije o tome kako će MySQL izvršiti upit, uključujući koji će indeksi biti korišćeni, kako će se tabele spajati i kako će se podaci prikupljati. Plan izvršenja dobijen EXPLAIN naredbom daje ***mogući plan izvršenja*** za upit koji je naveden, a ne nužno tačan plan izvršenja. Ovaj plan se zasniva na statističkim podacima o tablicama i indeksima koje koristi upit, ali i drugim faktorima koji mogu uticati na način izvršenja upita, kao što su dostupna memorija, konfiguracija servera itd. Stoga, plan izvršenja može varirati u različitim situacijama, a samim tim i uticati na performanse upita. Potrebno je uzeti u obzir da je plan izvršenja samo alat koji pomaže u optimizaciji performansi upita, ali je potrebno testirati različite planove izvršenja kako bi se pronašao najbolji.

Rezultat EXPLAIN naredbe je tabela sa nekoliko kolona koje pružaju informacije o svakom koraku plana izvršavanja upita. Evo primera jednostavne SELECT naredbe:

```
SELECT * FROM city WHERE city = 'Nis';
```

Da bi se video plan izvršenja ovog upita, može se koristiti EXPLAIN naredba na sledeći način:

```
EXPLAIN SELECT * FROM city WHERE city = 'Nis';
```



	id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
▶	1	SIMPLE	city	NULL	ALL	NULL	NULL	NULL	NULL	600	10.00	Using where

Slika 4. Prikaz rezultata izvršenja EXPLAIN naredbe

Ovaj upit će vratiti informacije o tome kako MySQL obrađuje ovaj SELECT upit. Te informacije mogu biti korisne za optimizaciju upita, jer nam omogućava da vidimo kako MySQL obrađuje upit i šta može da se poboljša. EXPLAIN vraća red informacija za svaku tabelu koju koristi u SELECT naredbi i ređa tabele u rezultatu u redosledu u kome bi MySQL čitao dok obrađuje naredbu. Dakle, MySQL čita red iz prve tabele, zatim pronalazi odgovarajući red iz druge tabele, pa iz treće i tako dalje. Kada su sve tabele obrađene, MySQL izbacuje odabrane kolone i vraća se nazad kroz listu tabela dok ne pronađe tabelu za koju ima više odgovarajućih redova. Naredni red se čita iz ove tabele i proces se nastavlja sa sledećom tabelom. Rezultat ovog

upita će biti tabela koja sadrži informacije o tabelama i indeksima koji se koriste, kao i broju redova koji će biti skenirani, kolone su sledeće:

- **id** – redni broj koji označava redosled u kome će MySQL izvršiti operacije u upitu
- **select_type** – označava vrstu SELECT naredbe koja se izvršava, može biti:
    - **SIMPLE** – obična SELECT naredba bez unija i podupita
    - **PRIMARY** – spoljni upit
    - **UNION** – druga ili kasnija SELECT naredba u uniji
    - **SUBQUERY** – prva SELECT naredba u podupitu
- **table** – označava ime tabele na koju se odnosi jedna vrsta u izlazu, a može imati i sledeće vrednosti:
    - <union M, N> - vrsta se odnosi na uniju vrsta sa vrednostima id-eva M i N
    - <derived N> - vrsta se odnosi na izvedenu tabelu za red čiji je id jednak N. Izvedena tabela može biti rezultat podupita u FROM klauzuli.
    - <subquery N> - vrsta se odnosi na rezultat podupita za vrstu sa id-em N
- **partitions** - pokazuje broj particija (delova) tabele koje se koriste prilikom izvršavanja određenog upita. Particije su delovi tabele koje se mogu koristiti za organizaciju podataka po nekom kriterijumu, na primer po vremenu, geografskoj lokaciji ili vrednosti ključa. Kada se koriste particije, MySQL može brže pretraživati velike tabele, jer umesto pretraživanja cele tabele, pretražuje samo određeni deo. To može dovesti do značajnog poboljšanja performansi u određenim slučajevima. ko se prikazuje "NULL", to znači da se particije ne koriste za ovaj upit. Ukoliko se prikazuje broj veći od 1, to znači da se koristi više particija.
- **type** – označava tip spoja u izrazu (join type) i određuje na koji način se pristupa podacima u bazi. Odnosi se na sve uslovne izraze, ne samo na spojeve između tabela koje će MySQL koristiti za pribavljanje podataka. Moguće vrednosti su:
    - **system** – tabela ima samo jednu vrstu i to je specijalan slučaj const tipa spoja. Ovaj tip pristupa se koristi za interni MySQL sistemski upit koji često nije vidljiv korisnicima.
    - **const** – ovaj tip pristupa se koristi kada imamo konstatni izraz u WHERE klauzuli ili kada se koristi primarni ključ - kada se upoređuju svi delovi PRIMARY KEY ili UNIQUE indeksa sa konstantnim vrednostima. Ovo je najbrži tip pristupa tabelama jer se konstantne tabele čitaju samo jednom. Konstantna tabela ima najviše jedan red koji se poklapa ili nijedan, a čita se na početku upita. Pošto postoji samo jedan red, vrednosti iz ove kolone mogu biti tretirane kao konstante od strane ostatka optimizatora. Ova tabela se kreira kada se izvrši upit koji uključuje samo jednu tabelu i koristi se samo primarni ključ (primary key) ili jedinstveni indeks (unique index) za filtriranje podataka. U sledećem upitu, *table_name* može biti korišćena kao const tabela:

```
SELECT * FROM table_name WHERE primary_key=1;
```

Ove tabele su prviremene tabele koje se optimizuju se za brzu pretragu i pristup podacima. Kada se izvrši upit koji uključuje samo jednu tabelu i koristi se samo primarni ključ ili jedinstveni indeks, MySQL može koristiti const tabelu za brzo pristupanje podacima, što može biti izuzetno korisno za optimizaciju performansi upita, jer se podaci čitaju iz memorije, umesto da se čitaju sa diska. Ovo može značajno smanjiti vreme izvršavanja upita, posebno za tabele sa velikim brojem podataka.

- **eq_ref** – equal to refernece – Ovaj tip pristupa se koristi kada se spajaju dve tabele pomoću ***jedinstvenog ili primarnog ključa***. Ovo je efikasan tip pristupa, posebno za male tabele, a često je efikasniji nego korišćenje drugih tipova pretrage, kao što su "ref" ili "range", jer se pretraga fokusira samo na tačan odgovor koji se podudara sa jedinstvenim primarnim ključem ili jedinstvenim indeksom u drugoj tabeli. Eq_ref može koristiti samo ako postoji jedinstveni primarni ključ ili jedinstveni indeks u drugoj tabeli koja se povezuje sa osnovnom tabelom. Takođe je važno da se tačno podudara sa vrednošću u stranom ključu u osnovnoj tabeli. U sledećem primeru, MySQL može koristiti eq_ref spajanje za obradu ref_table:

```
SELECT * FROM ref_table, other_table WHERE ref_table.key_column =
other_table.column;
```

- **ref -** Ovaj tip pristupa se koristi kada se spajaju tabele na osnovu ***nejedinstvenog indeksa***. Ovo je efikasan tip pristupa, ali može biti manje efikasan od eq_ref za velike tabele. Ref pretraga se obično koristi kada se upit odnosi na jedan ili više indeksa, ali ne i na primarni ključ, a upit sadrži neke uslove koji ograničavaju broj redova koji se traže. Kada se izvrši upit koji koristi ref pretragu, MySQL koristi indeks da bi pronašao redove koji zadovoljavaju zadate uslove. Međutim, za razliku od eq_ref pretrage, ref pretraga može pronaći više redova u osnovnoj tabeli koji se podudaraju sa uslovima pretrage. Kada MySQL koristi ref pretragu, obično koristi ***B-tree indeks***[3], što omogućava brz pristup podacima. Međutim, ako se upit odnosi na veliki broj redova u tabeli, pretraga može biti spora i može dovesti do povećanja vremena izvršavanja upita. Ref pretraga može biti efikasna samo ako je indeks dobro dizajniran i ako se koristi na pravi način. Ako se indeks

[3] B-tree (Balanced Tree) je vrsta stabla koje se koristi za implementaciju indeksa u MySQL-u. B-tree indeks se koristi za brzo pronalaženje podataka u tabeli i efikasnije izvršavanje upita. Organizovan je kao stablo u kojem svaki čvor (osim listova) ima više od jednog deteta. Svaki čvor ima fiksnu veličinu i sadrži niz ključeva sortiranih u rastućem poretku. Ova organizacija omogućava brzu pretragu po indeksiranim ključevima, tako što se izbegava pretragu cele tabele. B-tree indeks se može kreirati nad jednom ili više kolona i obično se koristi za pretragu tačnog odgovora ili pretragu po određenom rasponu vrednosti. Kada se pretražuju velike količine podataka, B-tree indeks je efikasniji od linearnog pretraživanja. Ključne prednosti B-tree indeksa su brzina pretrage i minimalno zauzeće memorije.

ne koristi pravilno, pretraga može biti spora i može dovesti do povećanja vremena izvršavanja upita. Ref se koristi ako spajanje koristi samo skraćenu levu stranu indeksa ili ako indeks nije PRIMARY KEY ili UNIQUE indeks (drugim rečima, ako spajanje ne može da selektuje jedan red na osnovu vrednosti ključa). Ako indeks koji se koristi odgovara za samo nekoliko redova, ovo je dobar tip spajanja. Ref se može koristiti za indeksirane kolone koje se porede koristeći = ili <=> operator. U sledećim primerima, MySQL može koristiti ref spajanje za obradu ref_table:

```
SELECT * FROM ref_table WHERE key_column=expr;

SELECT * FROM ref_table, other_table WHERE ref_table.key_column =
other_table.column;
```

- **fulltext** – spajanje se vrši korišćenjem FULLTEXT indeksa[4].
- **ref_or_null** - ovaj tip spoja je sličan ref tipu, ali sa dodatkom da MySQL vrši dodatnu pretragu redova koji sadrže NULL vrednosti. Ova optimizacija tipa spajanja se najčešće koristi pri rešavanju podupita. U sledećim primerima, MySQL može koristiti ref_or_null tip spajanja za obradu ref_table:

```
SELECT * FROM ref_table WHERE key_column=expr OR key_column IS NULL;
```

- **index_merge** - Ovo znači da upit koristi više indeksa za pretragu, a zatim spaja rezultate. U ovom slučaju, kolona ključa u izlaznom redu sadrži listu korišćenih indeksa, a key_len sadrži listu najdužih delova ključa za korišćene indekse.
- **unique_subquery** – Ovo znači da upit koristi podupit za pretraživanje jednog reda, a podupit koristi jedinstveni indeks. Funkcija pretrage indeksa koja zamenjuje potpuno podupit, radi bolje efikasnosti, menja eq_ref za neke IN podupite sledećeg tipa:

```
value IN (SELECT primary_key FROM single_table WHERE some_expr)
```

[4] FULLTEXT je tip indeksa u MySQL-u koji se koristi za pretraživanje punog teksta (npr. reči, fraze, slogovi) u dugim tekstualnim poljima. Ovaj tip indeksa je koristan kada želimo da pronađemo sličnosti ili relevantne rezultate u tekstualnim sadržajima, kao što su članci, blogovi, knjige, itd. Fulltext indeksira ceo sadržaj polja, izdvojene reči se sortiraju i sprema se obrnuti indeks (inverse index) koji omogućava brzu pretragu teksta. Ovaj indeks se može kreirati samo na jednoj koloni i samo za MyISAM i InnoDB tabele. Kada se koristi FULLTEXT indeks, možemo koristiti operator MATCH AGAINST u WHERE klauzuli da bismo pronašli podudaranja sa pretraženim izrazima i on može biti korišćen za pretragu za jednu reč ili frazu (koristeći navodnike). FULLTEXT indeksiranje ne podržava stop reči (npr. "the", "and", "a"), tako da se ove reči obično neće koristiti prilikom pretraživanja. FULLTEXT indeksiranje je manje efikasno od B-tree indeksiranja i ne preporučuje se za tabele sa velikim brojem podataka, a i osetljivo je na dijakritičke znakove i može dovesti do netačnih rezultata u nekim slučajevima.

- **index_subquery** – Ovo znači da upit koristi podupit za pretraživanje više redova, a podupit koristi indeks. Slično kao unique_subquery menja IN podupite, ali za nejedinstvene indekse u podupitima sledećeg tipa:

```
value IN (SELECT key_column FROM single_table WHERE some_expr)
```

- **range** - upit koristi indeks za pretraživanje redova koji se podudaraju sa opsegom vrednosti. Samo one vrste koje se nalaze u određenom rasponu se pribavljaju, koristeći indeks za izbor vrsta. Kolona ključa u izlaznoj vrsti pokazuje koji se indeks koristi. key_len sadrži najduži ključni deo koji je korišćen. ref kolona je NULL za ovaj tip. range se može koristiti kada se ključna kolona uporedi sa konstantom koristeći bilo koji od operatora =, <>, >, >=, <, <=, IS NULL, <=>, BETWEEN, LIKE ili IN():

```
SELECT * FROM table_name WHERE key_column = 100;

SELECT * FROM table_name WHERE key_column BETWEEN 100 and 120;

SELECT * FROM table_name WHERE key_column IN (10,20,30);

SELECT * FROM table_name WHERE key_column1 = 10 AND key_column2
IN (10,20,30);
```

```
1 • use sakila;
2 • SET optimizer_trace='enabled=on';
3 • explain select last_name from customer where last_name LIKE "b%";
```

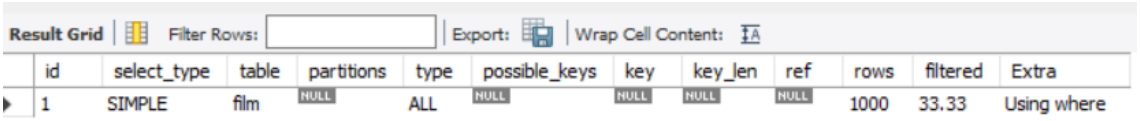
Result Grid | Filter Rows: | Export: | Wrap Cell Content:

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	customer	NULL	range	idx_last_name	idx_last_name	182	NULL	55	100.00	Using where; Using index

Slika 5. Prikaz rezultata primera

Postoje slučajevi kada je korišćenje indeksa ipak pogubnije od full scan-a, međutim i tada se vrši pretraživanje indeksa ali umesto da se posmatra opseg gleda se ceo indeks. Najčešće se ovo dešava pri korišćenju operatora > ili <.

```
1 • use sakila;
2 • SET optimizer_trace='enabled=on';
3 • explain select length from film where length > 100;
```



Slika 6. Prikaz rezultata primera

- **index** – upit koristi indeks za pretraživanje redova, ali ne koristi se jedinstveni ili opsežni indeks. Isto kao i ALL, samo što se prolazi kroz sablo indeksa. MySQL koristi ovaj tip spoja kada upit koristi samo kolone koje su deo jednog indeksa. Prolaženje kroz stablo indeksa se događa na 2 moguća načina:
  - Ako je indeks ***pokrivajući indeks***[5]za upite i može se koristiti za zadovoljavanje svih podataka potrebnih iz tabele, skenira se samo stablo indeksa. U ovom slučaju, kolona Extra kaže "Using index". Skeniranje samo indeksa obično je brže od ALL jer je veličina indeksa obično manja od podataka u tabeli.
  - Izvodi se potpuno skeniranje tabele (full scan) čitajući iz indeksa radi pretraživanja redova podataka u indeksiranom redu. Upotreba indeksa ne pojavljuje se u koloni Extra.

- **ALL** - za svaku kombinaciju redova iz prethodnih tabela vrši se potpuno pretraživanje tabele. Ovo obično nije dobro ako je tabela prva tabela koja nije označena kao "const", i obično je veoma loše u svim ostalim slučajevima. Obično se može izbeći korišćenje ALL tako što se dodaju indeksi koji omogućavaju povlačenje redova iz tabele na osnovu konstantnih vrednosti ili vrednosti kolona iz ranijih tabela.

- **possible_keys** – koji su indeksi mogući za pribavljanje podataka iz tabele. Ova kolona je potpuno nezavisna od redosleda tabela koji se prikazuju u izlazu iz EXPLAIN-a. Neki ključenvi u possible_keys mogu da budu neupotrebljivi u praksi. Ako je ova kolona jednaka NULL, onda ne postoje relevantni indeksi. Da bi se videli svi indeksi neke tabele, koristi se naredba:

```
SHOW INDEX FROM table_name;
```

[5] Pokrivajući indeks je indeks koji uključuje sve kolone potrebne za određeni upit. Kada se izvrši upit, umesto pristupa tabeli, MySQL može koristiti pokrivajući indeks kako bi u potpunosti zadovoljio upit, što može rezultirati bržim vremenom izvršenja upita. Na primer, pretpostavimo da imate tabelu sa kolonama A, B i C i često izvršavate upit koji bira samo kolone A i B. Ako kreirate indeks na kolonama A i B (u tom redosledu), ovaj indeks će biti pokrivajući indeks za upit jer uključuje sve potrebne kolone. U ovom slučaju, MySQL može koristiti pokrivajući indeks kako bi zadovoljio upit, izbegavajući potrebu za pristupom tabeli, što može rezultirati bržim izvršavanjem upita.

```
3 •  SHOW INDEX FROM filmj;
4
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

Table	Non_unique	Key_name	Seq_in_index	Column_name	Collation	Cardinality	Sub_part	Packed	Null	Index_type	Comment	Index_comment	Visibl
film	0	PRIMARY	1	film_id	A	1000	NULL	NULL		BTREE			YES
film	1	idx_title	1	title	A	1000	NULL	NULL		BTREE			YES
film	1	idx_fk_language_id	1	language_id	A	1	NULL	NULL		BTREE			YES
film	1	idx_fk_original_language_id	1	original_language_id	A	1	NULL	NULL	YES	BTREE			YES

Slika 7. Prikaz rezultata izvršenja SHOW INDEX naredbe

- **key** – označava koji je indeks zapravo MySQL odlučio da koristi za pribavljanje podataka iz tabele. Ako MySQL odluči da koristi jedan od indeksa iz possible_keys za pretraživanje redova, taj će indeks biti naveden kao vrednost ključa ili key-a. Može se desiti da key bude indeks koji nije prisutan u possible_keys, ako nijedan od mogućih ključeva nije prikladan za pretraživanje redova, ali su sve kolone izabrane upitom kolone nekog drugog indeksa. U svakom slučaju, pretraživanje indeksa je efikasnije od pretraživanja redova podataka.
- **key_len** – dužina indeksa tj. ključa koji će se koristiti za pribavljanje podataka iz tabele. Ako je kolona key jednaka NULL, onda je i key_len NULL.
- **ref** – označava kolonu ili kontstantu koja će se koristiti sa key indeksom da bi se pribavili podaci iz tabele. Ako je vrednost func, vrednost koja se koristi je rezultat neke funkcije ili operatora, a da bi se ustanovilo o kojoj funkciji je reč, koristi se SHOW WARNINGS nakon EXPLAIN naredbe da bi se dobio prošireni EXPLAIN prikaz rezultata.
- **rows** – procenjeni broj redova koje će MySQL morati procesuirati kako bi dobio konačan rezultat upita
- **filtered** – prikazuje procenat redova tabele koji su filtrirani uslovom tabele. Maksimalna vrednost je 100, što znači da nije bilo filtriranja redova. Vrednosti koje se smanjuju od 100 ukazuju na povećanje količine filtriranja. Kolona rows prikazuje procenjeni broj redova koji su pregledani, a rows x filtered prikazuje broj redova koji su povezani sa sledećom tabelom. Primer: ako je rows = 1000, filtered = 50%, broj redova koji će biti povezani sa sledećom tabelom je 1000 x 50% = 500 redova.
- **Extra** – dodatne informacije o tome kako će MySQL izvršiti upit, kao što je na primer sortiranje

MySQL procenjuje performanse upita kako bi pronašao najbolji način za izvršavanje upita i vratio rezultate što je brže moguće. Procena performansi upita vrši se pomoću EXPLAIN naredbe, međutim procene performansi upita nisu uvek tačne i mogu se razlikovati od stvarnog vremena izvršenja upita, pa je važno testirati performanse upita u stvarnom okruženju. U većini slučajeva, performanse upita u MySQL-u procenjuju se brojanjem traženja na disku. Za male tabele, obično se može pronaći vrsta u jednom traženju jer je indeks najverovatnije keširan, ali za veće tabele se koristi B stablo indeksa i potrebno je ovoliko traženja da bi se pronašla vrsta:

$$\frac{\log(broj\ vrsta)}{\log(\frac{dužina\ bloka\ indeksa}{3} * \frac{2}{dužina\ indeksa + dužina\ pokazivača\ podataka})} + 1$$

U MySQL-u blok indeksa obično iznosi 1024 bajta, a pokazivač podataka 4 bajta, dok za tabelu od 500.000 vrsta sa dužinom vrednosti ključa od 3 bajta formula pokazuje da je potrebno:

$$\frac{\log(500.000)}{\log(\frac{1024}{3} * \frac{2}{3+4})} + 1 = 4\ traženja$$

Ovo ne znači da se performanse izvršavanja polako pogoršavaju po log N, jer sve dok je sve keširano u MySQL serveru stvari postaju samo marginalno sporije sa porastom veličine tabele. Tek nakon što podaci podastanu preveliki za keširanje, stvari počinju mnogo sporije da se izvršavaju, pa da bi se to izbeglo potrebno je povećavati veličinu keša sa porastom podataka.

1.2. OPTIMIZER_TRACE tabela

Tabela **OPTIMIZER_TRACE** u MySQL-u se koristi za prikazivanje detalja o tome kako MySQL query optimizer izvršava upit. Dok EXPLAIN naredba daje prikaz odabranog plana izvršenja, OPTIMIZER_TRACE daje objašnjenje zašto je taj plan izabran.

Da bi se uključilo praćenje rada optimizatora koristi se sledeća komanda:

```
SET optimizer_trace='enabled=on';
```

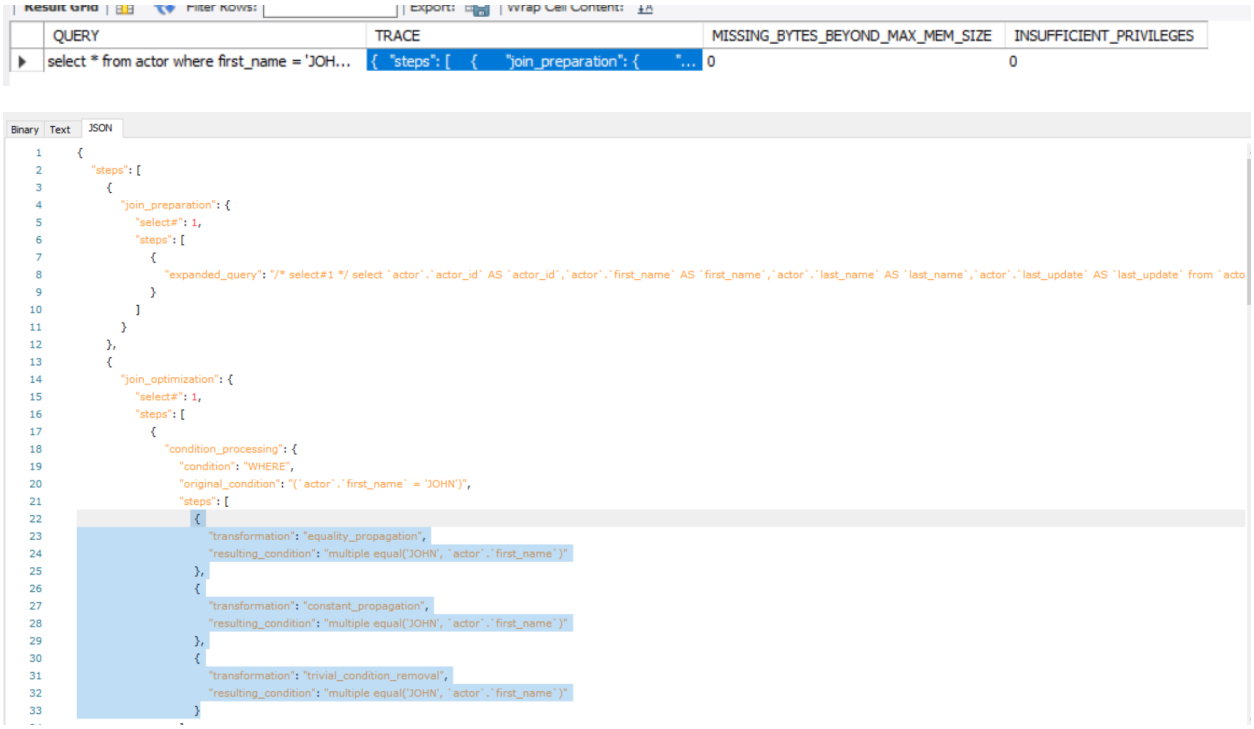
Tabela OPTIMIZER_TRACE ima 4 kolone:

- **query** – tekst SQL upita koji se posmatra
- **trace** – pribavljene informacije u JSON formatu
- **missing_bytes_beyond_max_mem_size** – broj bajtova trace-a preko maksimalne zadate veličine memorije. Svaki zapamćeni trace je string koji se proširuje kako optimizacija napreduje i dodaju se podaci u njega. Promenljiva optimizer_trace_max_mem_size postavlja ograničenje na ukupnu količinu memorije koju koriste svi trenutno zapamćeni trace-evi. Ako se dostigne ovo ograničenje, trenutni trace se ne proširuje i ostaje nepotpun, a ova kolona prikazuje broj bajtova koji nedostaju u trace-evima.
- **insufficient_privileges** – govori da li MySQL korisnik može da vidi trace optimizatora ili ne, ako ne može vrednost je 1, inače je 0.

Primer:

```
1 • SET optimizer_trace='enabled=on';
2 • select * from actor where first_name = 'JOHN';
3 • select * from information_schema.optimizer_trace;
```

Rezultat:

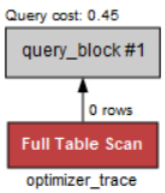


QUERY	TRACE	MISSING_BYTES_BEYOND_MAX_MEM_SIZE	INSUFFICIENT_PRIVILEGES
select * from actor where first_name = 'JOH...	{ "steps": [{ "join_preparation": { "...	0	0


```
{
  "steps": [
    {
      "join_preparation": {
        "select#": 1,
        "steps": [
          {
            "expanded_query": "/* select#1 */ select `actor`.`actor_id` AS `actor_id`,`actor`.`first_name` AS `first_name`,`actor`.`last_name` AS `last_name`,`actor`.`last_update` AS `last_update` from `acto
          }
        ]
      }
    },
    {
      "join_optimization": {
        "select#": 1,
        "steps": [
          {
            "condition_processing": {
              "condition": "WHERE",
              "original_condition": "(`actor`.`first_name` = 'JOHN')",
              "steps": [
                {
                  "transformation": "equality_propagation",
                  "resulting_condition": "multiple equal('JOHN', `actor`.`first_name`)"
                },
                {
                  "transformation": "constant_propagation",
                  "resulting_condition": "multiple equal('JOHN', `actor`.`first_name`)"
                },
                {
                  "transformation": "trivial_condition_removal",
                  "resulting_condition": "multiple equal('JOHN', `actor`.`first_name`)"
                }
```


Slika 8. Prikaz rezultata OPTIMIZER_TRACE tabele

MySQL Workbench pruža sve formate za izvršavannje upita, pa tako i vizuelizaciju. Da bi se video vizuelni plan izvršenja, potreno je odabrati Execution Plan tab u polju sa rezultatima upita i dobija se sledeći prikaz:



Slika 9. Prikaz vizuelizacije plana izvršenja upita

1.3. EXPLAIN ANALYZE naredba

Kao što je već rečeno, ova naredba je detaljnija i daje statistike koje ne daje obična EXPLAIN naredba, a za te statistike je neophodno da se zapravo izvrši upit. Primer poziva komande i rezultat koji se dobija je:

```
use sakila;
EXPLAIN ANALYZE SELECT first_name, last_name, city, country
FROM customer
INNER JOIN address USING(address_id)
INNER JOIN city USING(city_id) INNER JOIN country USING(country_id);
```

```
-> Nested loop inner join  (cost=643 rows=604) (actual time=0.469..6.06 rows=599 loops=1)
   -> Nested loop inner join  (cost=432 rows=604) (actual time=0.271..3.22 rows=603 loops=1)
      -> Nested loop inner join  (cost=221 rows=600) (actual time=0.185..1.79 rows=600 loops=1)
         -> Table scan on country  (cost=11.2 rows=109) (actual time=0.0741..0.127 rows=109 loops=1)
         -> Index lookup on city using idx_fk_country_id (country_id=country.country_id)  (cost=1.38 rows=5.5) (actual time=0.00702..0.0149 rows=5.5 loops=109)
      -> Covering index lookup on address using idx_fk_city_id (city_id=city.city_id)  (cost=0.25 rows=1.01) (actual time=0.00187..0.00229 rows=1 loops=600)
   -> Index lookup on customer using idx_fk_address_id (address_id=address.address_id)  (cost=0.25 rows=1) (actual time=0.00415..0.00456 rows=0.993 loops=603)
```

Slika 10. Prikaz rezultata izvršenja EXPLAIN ANALYZE naredbe

Kao što se vidi, rezultat je opis toga kako je MySQL server izvršio plan ali uvodi i novine: pravo vreme za pribavljanje prvog reda tabele u milisekundama, pravo vreme za pribavljanje svih vrsta tabele u milisekundama, estimiran trošak za upit, pravi broj čitanja vrsta, pravi broj petlji koje su nastale. "ANALYZE" deo naredbe EXPLAIN ANALYZE se koristi za prikazivanje detaljnih informacija o tome koliko dugo MySQL izvršava svaki korak upita, kao i koliko redova svaki korak vraća. Ova informacija može biti korisna za identifikovanje sporih delova upita i pronalaženje načina za poboljšanje performansi.

1.4. Izbegavanje potpunog pretraživanja tabele (full scan)

Rezultat EXPLAIN naredbe u type koloni ima vrednost "ALL" kada MySQL koristi potpuno pretraživanje tabele (full scan) za rešavanje upita. Ovo se obično dešava u sledećim uslovima:

- Tabela je tako mala da je brže izvršiti potpuno pretraživanje tabele nego tražiti ključ. To je uobičajeno za tabele sa manje od 10 redova i kratkom dužinom reda.
- Nema upotrebljivih ograničenja u ON ili WHERE klauzuli za indeksirane kolone.
- Upoređuju se indeksirane kolone sa konstantnim vrednostima i MySQL je utvrdio da konstante pokrivaju prevelik deo tabele, pa je full scan brži.
- Kada se koristi ključ sa niskom kardinalnošću (mnogo redova se podudara sa vrednošću ključa) preko neke druge kolone, MySQL pretpostavlja da korišćenje ključa verovatno zahteva mnogo pretraga ključa i da će pretraga pune tabele biti brža.

Za velike tabele treba pokušati sprečiti optimizator da odabere full scan metodu za pretraživanje tabele, a neke od opcija su:

- Korišćenje **ANALYZE TABLE table_name** radi ažuriranja raspodele ključeva za pretraženu tabelu.
- Pokretanje MySQL servera sa opcijom **--max-seeks-for-key=1000** ili **SET max_seeks_for_key=1000** da bi optimizator pretpostavio da nijedna pretraga ključa ne izaziva više od 1000 pretraga ključa.

2. Optimizacija SELECT naredbe

Ugrađeni optimizator MySQL upita izvrsno obavlja posao optimizacije izvršenja upita. Međutim, loše napisani upiti mogu sprečiti optimizator da dobro obavi posao. Čak i ako se primenjuju druge tehnike optimizacije poput dobrog dizajna šeme ili indeksiranja, ako su napisani upiti pogrešni, oni će i dalje uticati na performanse baze podataka.

Sve operacije pretrage u okviru baze podataka se obavljaju pomoću SELECT naredbe, stoga je optimizacija te naredbe od velikog značaja.

Prvo što se predlaže kao strategija za ubrzanje spore SELECT … WHERE naredbe jeste ***indeksiranje***. Kako bi se ubrzalo izvršavanje, filtriranje i pribavljanje rezultata potrebno je postaviti indekse na kolone koje se koriste u WHERE klauzuli. Dobra praksa je smanjiti broj prolazaka kroz celu tabelu u upitima, posebno za velike tabele. Ako se pak radi o malim tabelama sa manje od 10 kolona, MySQL optimizator se može odlučiti za skeniranje cele tabele (full scan) jer se ono može brže izvršiti od pronalaska indeksa. Statistiku tabele možemo periodično održavati pomoću ANALYZE TABLE naredbe, tako da optimizator ima potrebne informacije za konstrukciju efikasnog plana izvršenja.

Pored SELECT naredbe, optimizacija se vrši i za naredbe koje kombinuju operacije upisa i čitanja, poput CREATE TABLE … AS SELECT, INSERT INTO … SELECT i WHERE klauzula u DELETE naredbama.

2.1. Optimizacija WHERE klauzule

WHERE klauzula u MySQL-u omogućava filtriranje redova u bazi podataka prema određenim kriterijumima u okviru SELECT, DELETE i UPDATE naredbi. MySQL mora da prođe kroz sve vrste tabele kako bi pronašao one koji odgovaraju zadatkom kriterijumu, što može biti vrlo sporo ako je tabela velika ili je kriterijum složen. Stoga je optimizacija WHERE klauzule u okviru SQL upita od ključnog značaja. MySQL vrši automatske optimizacije upita, kao što su:

- **Uklanjanje nepotrebnih zagrada ili suvišnog koda**

Originalni upit: ((a AND b) AND c OR (((a AND b) AND (c AND d))))

Tranformisan upit: (a AND b AND c) OR (a AND b AND c AND d)

Pri eliminaciji suvišnog koda treba imati na umu upite koji sadrže uslove koji imaju konstante vrednosti, a to su upiti oblika:

$$SELECT \dots WHERE\ 1 = 1\ AND\ col1 =' value' AND\ 'apple' =' apple'$$

Ovakav upit bi se transformisao u sledeći oblik:

$$SELECT \dots WHERE\ col1 = 'value'$$

```
1 •  SET optimizer_trace='enabled=on';
2 •  select * from actor where 1 = 1 AND first_name = 'JOHN';
3 •  select * from information_schema.optimizer_trace;
```

Pomoću trace-a možemo videti da je uslov 1 = 1 iz datog upita automatski uklonjen od strane optimizatora:

Slika 11. Prikaz trace-a pri uklanjanju suvišnog koda

MySQL vrši i automatsko uklanjanje suvišnih IS NULL uslova u upitima kada kolona od značaja ima ograničenje da ne može imati NULL vrednost. Kada se ovaj operator koristi u upitu, MySQL pretražuje bazu podataka kako bi pronašao sve redove koji imaju NULL vrednosti u određenom polju. Ovaj proces pretrage može biti vrlo spor, posebno ako je tabela velika ili ako postoji puno redova sa NULL vrednostima. Kako bi se ubrzao ovaj proces pretrage, MySQL koristi "IS NULL" optimizaciju koja radi na način da kreira tzv. "null bitmap" strukturu. Ova struktura čuva informaciju o tome koji redovi u tabeli sadrže NULL vrednosti, što omogućava MySQL-u da izbegne pretragu tih

redova kada se koristi "IS NULL" operator. Ovo može značajno ubrzati pretragu i smanjiti vreme izvršavanja upita.

Ako WHERE klauzula sadrži IS NULL uslov za kolonu koja je deklarisana kao NOT NULL, taj izraz se u svakom slučaju optimizuje, ali se optimizacija ne dešava u slučajevima kada je kolona produkt tabele koja je na desnoj strani LEFT JOIN-a. MySQL optimizira kombinaciju oblika: column_name = expression OR column_name IS NULL i u EXPLAIN rezultatu možemo videti ref_or_null oznaku kao tip.

Primeri upita:

```
SELECT * FROM table_name WHERE key_column IS NULL;

SELECT * FROM table_name WHERE key_column <=> NULL;

SELECT * FROM table_name WHERE key_column=const1 OR key_column=const2
OR key_column IS NULL;
```

- **Preklapanje konstanti**

```
Originalni upit:    (a<b AND b=c) AND a=5
Transformisan:      b>5 AND b=c AND a=5
Originalni upit:    SELECT ... WHERE col1 = 2 * 5
Transformisan:      SELECT ... WHERE col1 = 10
```

Ovi slučajevi se ređe javljaju, ali nastaju zbog ljudske prirode razmišljanja. Izrazi koji sadrže više konstanti uprošćavaju se i svode na jednu.

- **Uklanjanje konstantnih uslova**

```
Originalni upit:(b>=5 AND b=5) OR (b=6 AND 5=5) OR (b=7 AND 5=6)
Transformisani:     b=5 OR b=6
```

Pri eliminaciji konstantnih uslova, ono što treba imati na umu jeste ***zakon tranzitivnosti*** koji glasi:

Relacija ρ nad skupom A je tranzitivna ako za svako $x, y, z \in$ A važi $(x, y) \in \rho \wedge (y, z) \in \rho \Rightarrow (x, z) \in \rho$. Drugim rečima, ako je x = y i y = z, onda je i x = z.

Ukoliko je dat upit koji je oblika:

$$SELECT \ \ldots \ WHERE \ col1 \ < operator > \ col2 \ AND \ col2 \ < operator > \ 'value'$$

On se transformiše u upit sledećeg oblika:

$$SELECT\ \ldots\ WHERE\ col1\ <operator>\ 'value'\ AND\ col2\ <operator>\ 'value'$$

Pri čemu je $operator\ \epsilon\ \{=, >, <, \geq, \leq, <>, <=>, LIKE\}$.

```
1 •  SET optimizer_trace='enabled=on';
2 •  select * from city where country_id = city_id AND city_id = 1;
3 •  select * from information_schema.optimizer_trace;
```

```
"steps": [
  {
    "condition_processing": {
      "condition": "WHERE",
      "original_condition": "((`city`.`country_id` = `city`.`city_id`) and (`city`.`city_id` = 1))",
      "steps": [
        {
          "transformation": "equality_propagation",
          "resulting_condition": "(multiple equal(1, `city`.`country_id`, `city`.`city_id`))"
        },
        {
          "transformation": "constant_propagation",
          "resulting_condition": "(multiple equal(1, `city`.`country_id`, `city`.`city_id`))"
        },
        {
          "transformation": "trivial_condition_removal",
          "resulting_condition": "multiple equal(1, `city`.`country_id`, `city`.`city_id`)"
        }
      ]
    }
  },
```

Slika 11. Prikaz trace-a pri uklanjanju konstantnih uslova

Saveti za pisanje optimizovanih WHERE klauzula u MySQL-u:

- **Indeksiranje**

Indeksiranje je jedan od najvažnijih načina za optimizaciju WHERE klauzule, kao što je već napomenuto ranije. Indeksiranje dodaje strukturu podacima, što omogućava MySQL-u da brže pretražuje podatke. Kada MySQL pretražuje podatke bez indeksa, to se naziva punim pregledom tabele, što može biti vrlo sporo ako je tabela velika. Na primer, ako želimo pronaći sve redove u tabeli "customers" gde je "first_name" jednak "John", upit bi izgledao ovako:

```
SELECT * FROM customers WHERE first_name = 'John';
```

Ako tabela "customers" sadrži veliki broj redova, ovaj upit bi mogao biti spor. Međutim, ako indeksiramo kolonu "first_name", MySQL će moći brzo pronaći sve redove u tabeli sa vrednošću "John". Indeksiranje možemo uraditi na sledeći način:

```
ALTER TABLE customers ADD INDEX idx_first_name (first_name);
```

Sada će upit biti brži, jer MySQL koristi indeks "idx_first_name" za brzo pronalaženje redova sa vrednošću "John".

- **Korišćenje operatora umesto funkcija**

Korišćenje funkcija u WHERE klauzuli može takođe usporiti upit. Na primer, ako želimo pronaći sve redove u tabeli "orders" gde je "order_date" manji od jučerašnjeg datuma, upit bi izgledao ovako:

```
SELECT * FROM orders WHERE DATE(order_date) < DATE_SUB(NOW(), INTERVAL 1 DAY);
```

Funkcija DATE() u ovoj WHERE klauzuli izvlači samo datum iz kolone "order_date". Međutim, ova funkcija usporava upit, jer MySQL mora izvršiti ovu funkciju za svaki red u tabeli. Umesto toga, možemo napisati upit na sledeći način:

```
SELECT * FROM orders WHERE order_date < DATE_SUB(NOW(), INTERVAL 1 DAY);
```

Ovaj upit će biti brži, jer MySQL može direktno uporediti kolonu "order_date" sa datumom dobijenim funkcijom. Dakle, korišćenje operatora umesto funkcija z WHERE klauzuli može poboljšati performanse.

Ukoliko želimo izvršiti ovakav upit:

```
SELECT * FROM users WHERE UPPER(first_name) = 'JOHN';
```

Ovo opet može biti spor upit jer se koristi funkcija UPPER() kojom se konvertuje first_name u upper case, pa će MySQL obraditi svaki red u tabeli pre nego što primeni funkciju. Umesto toga se može koristi operator LIKE:

```
SELECT * FROM users WHERE first_name LIKE 'John';
```

Ovo će pronaći sve vrste u kojima je kolon first_name John bez potrebe za funkcijama.

- **Korišćenje LIMIT klauzule**

Korišćenje LIMIT klauzule u WHERE upitu može smanjiti broj vrsta koje MySQL mora da obradi, primer:

```
SELECT * FROM users WHERE age > 25 LIMIT 10;
```

Ovako će se pronaći tačno 10 vrsta u kojima je starost veća od 25, a bez LIMIT dela, broj vrsta bi mogao da bude znatno veći.

- **Korišćenje podupita**

Korišćenje podupita u WHERE klauzuli može poboljšati performanse i olakšati čitanje koda, primer:

```
SELECT * FROM users WHERE user_id IN (SELECT user_id FROM orders WHERE total_price > 100);
```

Ovo pronalazi sve korisnike čiji su ID-evi prisutni u podupitu koji vraća ID-eve korisnika koji su napravili narudžbinu sa ukupnom cenom većom od 100. podupit se izvršava samo jednom i vraća samo potrebne ID-eve, što pobljšava performanse.

- **Korišćenje EXPLAIN naredbe**

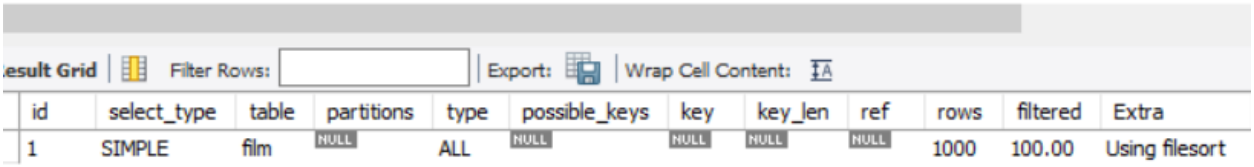
Naredba EXPLAIN može pomoći optimizaciji, kao što je ranije pomenuto, jer prikazuje kako MySQL izvršava upit i omogućava nam da vidimo koje indekse koristi, koliko vrsta obrađuje i koliko vremena je potrebno za izvršavanje upita. Primer:

```
EXPLAIN SELECT * FROM users WHERE username = 'jovan';
```

2.2. Optimizacija ORDER BY klauzule

Pre nego se dođe do same ORDER BY klauzule prvo se razrešavaju ranije navedeni problemi koji mogu da se jave u uslovu, kao što je uklanjanje suvišnog koda ili konstantnih uslova. Pa ukoliko se ustanovi da klauzula nema smisla u upitu, ona se uklanja, kao u sledećem primeru:

```
1 •  use sakila;
2 •  SET optimizer_trace='enabled=on';
3 •  EXPLAIN SELECT description FROM film ORDER BY 1;
```

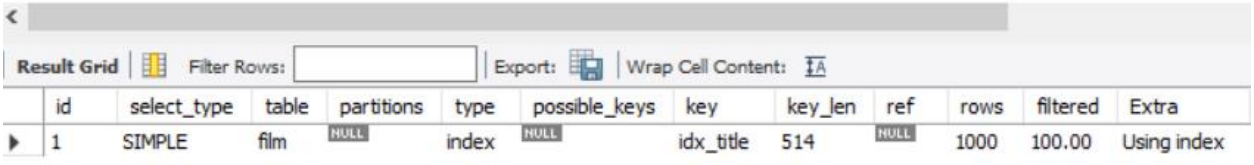


id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	film	NULL	ALL	NULL	NULL	NULL	NULL	1000	100.00	Using filesort

Slika 12. Prikaz rezultata optimizacije ORDER BY naredbe pomoću filesort-a

Ključno kod optimizacije ORDER BY klauzula je da optimizator ume da prepozna uređenost vrsta i ukoliko su već uređene da ne izvrši traženo sortiranje. MySQL može da koristi indekse za zadovoljavanje ORDER BY klauzule i tako izbegne dodatno sortiranje koje je potrebno za izvođenje operacije filesort, ali naravno samo ako je to isplativije odnosno efikasnije. Sortiranje tabele se izvršava ukoliko se u upitu ne nalazi indeksirana kolona, inače se koristi indeks. Informaciju o tome možemo videti u Extra koloni rezultata EXPLAIN naredbe:

```
1 •  use sakila;
2 •  SET optimizer_trace='enabled=on';
3 •  EXPLAIN SELECT title FROM film ORDER BY title;
```



id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	film	NULL	index	NULL	idx_title	514	NULL	1000	100.00	Using index

Slika 13. Prikaz rezultata optimizacije ORDER BY klauzule pomoću indeksiranja

U upitima sledećeg oblika:

$$SELECT * FROM\ table1\ ORDER\ BY\ key\ part1, key\ part2$$

Postojanje indeksa za key_part1 i key_part2 omogućava optimizatoru da izbegne sortiranje, ali pošto upit vrši selekciju svih kolona tabele table1, što nisu samo kolone key_part1 i key_part2, skeniranje celog indeksa i traženje vrsta tabele radi pronalaženja kolona koji nisu indeksirani može biti skuplje od skeniranja tabele i sortiranja rezultata. Ako je to slučaj, optimizator verovatno neće koristiti indeks. U slučaju da je SELECT naredba pokrivena indeksom, tj. postoji odogovarajući pokrivajući indeks za SELECT naredbu, onda se koristi indeks i sortiranje filesort-om se izbegava.

Dve kolone u ORDER BY klauzuli mogu sortirati u istom smeru (obe ASC ili obe DESC), a mogu i u suprotnim smerovima (jedna ASC, druga DESC ili obrnuto), pa je uslov za upotrebu indeksa tada da indeks mora imati istu homogenost, ali ne mora imati isti stvarni smer. Šta to znači? Ako upit kombinuje ASC i DESC, optimizator može koristiti indeks na kolonama ako indeks takođe koristi odgovarajuće mešovite rastuće i opadajuće kolone:

$$SELECT * FROM\ table1\ ORDER\ BY\ key\ part1\ DESC, key\ part2\ ASC$$

Optimizator može korisiti indeks nad key_part1 i key_part2 ako je key_part1 opadajući, a key_part2 rastući, ali može koristiti indeks na tim kolonama i ako je key_part1 rastući i key_part2 opadajući.

Slučajevi kada MySQL ne može da koristi indekse za razrešavanje ORDER BY klauzule:

- Kada upit koristi ORDER BY nad različitim indeksima:

```
SELECT * FROM t1 ORDER BY key1, key2;
```

- Kada upit koristi ORDER BY nad neuzastopnim delovima indeksa:

```
SELECT * FROM t1 WHERE key2=constant ORDER BY key1_part1, key1_part3;
```

- Kada se indeks za pribavljanje redova razlikuje od indeksa koji se koristi za ORDER BY:

```
SELECT * FROM t1 WHERE key2=constant ORDER BY key1;
```

- Kada upit koristi ORDER BY sa izrazom koji sadrži izraze koji nisu naziv indeksa kolone:

```
SELECT * FROM t1 ORDER BY ABS(key);
SELECT * FROM t1 ORDER BY -key;
```

- Kada upit ima različite ORDER BY i GROUP BY klauzule.
- Kada indeks ne pamti redove u nekom uređenom redosledu.

Ako se indeks ne može korisiti za ispunjavanje klauzule ORDER BY, MySQL izvodi operaciju filesort koja čita redove tabele i sortira ih. Da bi se obezbedila memorija za filesort operaciju, optimizator inkrementalno dodeljuje memorijske bafere po potrebi do veličine koji odredi ***sort_buffer_size*** bajtova. Korisnici tako mogu da postave *sort_buffer_size* na veće vrednosti kako bi ubrzali veća sortiranja, bez brige o prekomerenoj upotrebi memorije. Operacija filesort koristi privremene disk fajlove po potrebi ako je skup rezultata prevelik da bi stao u

memoriju. Neki tipovi upita su posebno pogodni za filesort operacije koje se potpuno izvršavaju u memoriji, na primer upiti sledećeg oblika:

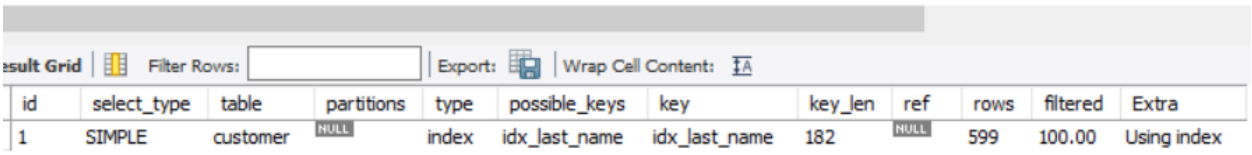
```
SELECT ... FROM single_table ... ORDER BY non_index_column [DESC] LIMIT [M,]N;
```

2.3. Optimizacija GROUP BY klauzule

Optimizacije GROUP BY klauzule je slična kao i za ORDER BY klauzulu. Grupisanje po jednoj ili više kolona se može koristiti u MySQL upitima kako bi se izvršilo sažimanje podataka i prikazalo više redova sa grupisanim podacima, umesto svakog pojedinačnog reda. Međutim, izvršavanje ovakvih upita može biti resursno zahtevno i dugotrajno, posebno kada je potrebno izvršiti grupisanje nad velikim skupovima podataka. MySQL optimizuje takve upite automatski kako bi smanjio njihov uticaj na performanse sistema. Optimizacija se izvršava tako što MySQL procenjuje upit i određuje da li je moguće koristiti indeksiranje kako bi se ubrzao postupak grupisanja. Ako je grupisanje moguće izvršiti korišćenjem postojećih indeksa, MySQL će koristiti tu strategiju kako bi izbegao dodatne korake sortiranja. U slučaju da ne postoji odgovarajući indeks, MySQL će koristiti ***privremene tabele***. Privremena tabela je tabela koja se privremeno kreira tokom izvršavanja upita radi efikasnijeg grupisanja podataka. Kada se koristi klauzula GROUP BY, MySQL prvo sortira podatke po grupisanju, a zatim vrši grupisanje. Ovo može biti veoma sporo ako se radi sa velikim količinama podataka. Korišćenjem privremene tabele, baza podataka može sortirati podatke jednom i zatim grupisati sortirane podatke. Ovo može biti mnogo efikasnije od sortiranja i grupisanja podataka u jednom koraku. Privremena tabela se obično briše nakon izvršavanja upita. U MySQL verziji 8.0 i novijim, optimizacija upita sa GROUP BY klauzulom je dodatno unapređena korišćenjem novog optimizatora upita (MySQL Cost Model). Ovaj optimizator koristi statističke informacije o podacima kako bi procenio i odabrao najbolju strategiju izvršavanja upita. Ova promena omogućava MySQL-u da bolje iskoristi indekse i druge tehnike optimizacije za grupisanje podataka.

Ako je izvršeno indeksiranje kolona koje se koriste u GROUP BY klauzuli, onda se koristi taj indeks. Ako se koristi indeks onda nema potrebe da se kreira privremena tabela pa je i ceo upit brži, naravno uz ograničenje da su sve kolone potrebne za GROUP BY deo jednog indeksa i da su ključevi u indeksu sortirani. U koloni Extra može se videti koji način je odabran, da li indeksiranje ili privremena tabela:

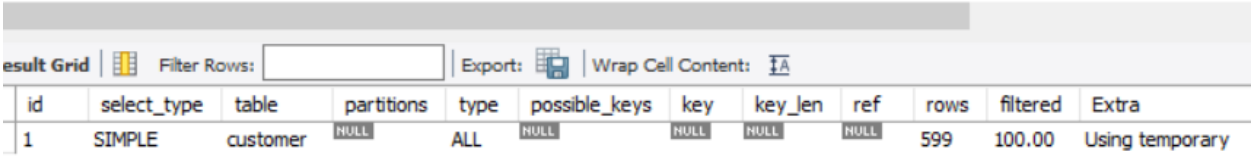
```
use sakila;
EXPLAIN SELECT last_name FROM customer GROUP BY last_name;
```



id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	customer	NULL	index	idx_last_name	idx_last_name	182	NULL	599	100.00	Using index

Slika 14. Prikaz rezultata EXPLAIN naredbe nad GROUP BY klauzulom kada se koristi indeksiranje

```
use sakila;
EXPLAIN SELECT email FROM customer GROUP BY email;
```



id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	customer	NULL	ALL	NULL	NULL	NULL	NULL	599	100.00	Using temporary

Slika 15. Prikaz rezultata EXLAIN naredbe pri korišćenju privremene tabele

2.4. Optimizacija DISTINCT klauzule

Za DISTINCT u kombinaciji sa ORDER BY vrlo često je potrebno korisiti privremenu tabelu radi optmizacije upita. U većini slučajeva DISTINCT se može posmatrati kao poseban slučaj GROUP BY klauzule po sledećem pravilu:

Ukoliko nema WHERE ni LIMIT klauzule, postoji indeks nad kolonom koja je potrebna DISTINCT klauzuli i koriste se podaci iz samo jedne tabele, upit oblika:

```
SELECT DISTINCT column FROM table
```

Može se prevesti u upit oblika:

```
SELECT column FROM table GROUP BY column
```

Tako da su sledeća dva upita ekvivalentna:

```
SELECT DISTINCT col1, col2, col3 FROM table1 WHERE col1 > const;
```

```
SELECT col1, col2, col3 FROM table1 WHERE col1 > const GROUP BY col1, col2, col3;
```

Zbog ovoga se optimizacije koje koristimo za GROUP BY mogu primeniti i na DISTINCT klauzulu, što je prilično korisna stvar.

```
1 •  use sakila;
2 •  SET optimizer_trace="enabled=on";
3 •  EXPLAIN SELECT DISTINCT last_name FROM customer;
4 •  SELECT * FROM information_schema.optimizer_trace;
```

```
{
  "optimizing_distinct_group_by_order_by": {
    "changed_distinct_to_group_by": true,
    "simplifying_group_by": {
      "original_clause": "`customer`.`last_name`",
      "items": [
        {
          "item": "`customer`.`last_name`"
        }
      ],
      "resulting_clause_is_simple": true,
      "resulting_clause": "`customer`.`last_name`"
    }
  }
},
```

Slika 16. Prikaz trace-a za DISTINCT klauzulu

2.5. Optimizacija LIMIT klauzule

LIMIT klauzula se koristi kada u rezultatu nisu potrebne sve moguće vrste, već samo određeni broj vrsta. Kada se aktivira automatska optimizacija LIMIT upita, MySQL će izvršiti sledeće radnje:

1. Izvršiće se upit i naći se redovi koji odgovaraju uslovima upita.
2. Nakon toga, MySQL će koristiti tzv. "***quick select***" algoritam za sortiranje redova. Quick select je efikasan algoritam koji omogućava sortiranje samo prvih nekoliko redova, što je u skladu sa zahtevima klauzule LIMIT.
3. Nakon sortiranja redova, MySQL će vratiti samo prvih nekoliko redova koji su potrebni u skladu sa LIMIT klauzulom. Ostatak redova se neće vratiti, što čini izvršavanje upita bržim i efikasnijim.

Vredno je napomenuti da se automatska optimizacija vrši samo kada se koristi LIMIT klauzula sa određenim brojem redova koji se vraćaju. Ako se koristi LIMIT klauzula bez broja, automatska optimizacija se neće primeniti. Takođe, ako postoji kompleksan upit sa JOIN-ovima, podupitima i složenim uslovima, automatska optimizacija LIMIT upita možda neće biti moguća. U ovom slučaju, potrebno je ručno optimizovati upit kako bi se postigle bolje performanse.

Ako se kombinuje **LIMIT row_count** sa ORDER BY klauzulom, MySQL zaustavlja sortiranje čim pronađe prvih row_count redova sortiranog rezultata, umesto da sortira ceo skup rezultata. Ako se sortiranje vrši pomoću indeksa, ovo je vrlo brzo. Ako je potrebno uraditi filesort, svi redovi koji se podudaraju sa upitom bez LIMIT klauzule su izabrani, i većina ili svi oni su sortirani pre nego što se pronađe prvih row_count redova. Nakon što se pronađu početni redovi, MySQL ne sortira preostali skup rezultata.

Ako se kombinuje LIMIT row_count sa DISTINCT, MySQL zaustavlja pretragu čim pronađe row_count jedinstvenih redova.

4. Optimizacija INSERT, DELETE i UPDATE naredbi

Kada se koristi INSERT naredba u MySQL-u, baza podataka će automatski optimizovati upit na najefikasniji način kako bi se ubrzao proces ubacivanja podataka u tabelu. MySQL će pokušati da ubrza proces ubacivanja podataka tako što će koristiti optimizacije poput grupnog ubacivanja (batching), što znači da će umesto jednog po jednog unosa, grupisati više njih i ubaciti ih odjednom, čime će se smanjiti broj operacija upisivanja na disk. Pogodno je kombinovati veći broj manjih operaciju u jednu. Ako se vrste dodaju u tabelu koja nije prazna može se koristiti varijabla bulk_insert_buffer_size koja predstavlja veličinu keša koji se koristi za bulk ubacivanje podataka. Takođe, ako se koristi AUTO_INCREMENT kolona u tabeli, MySQL će automatski generisati sledeći broj za ovu kolonu kada se izvrši INSERT naredba, što omogućava automatsko generisanje jedinstvenih identifikatora za svaki novi unos. MySQL takođe koristi tehnike keširanja memorije kako bi smanjio broj upita ka disku, tako da će često pristupani podaci biti keširani u memoriji kako bi se ubrzao proces ubacivanja u tabelu. Međutim, iako MySQL automatski optimizuje INSERT upite, važno je i dalje koristiti najbolje prakse za efikasno upisivanje podataka, kao što su korišćenje transakcija, odabir odgovarajućeg tipa podataka i normalizacija baze podataka.

Na brzinu upisivanja podataka u bazu podataka utče više faktora, a to su vreme konektovanja, vreme potrebno za slanje upita ka serveru, dodavanje nove vrste, kreiranje indeksa, zatvaranje konekcije, veličina tabela itd.

MySQL automatski optimizuje UPDATE naredbu kako bi poboljšao njeno izvršavanje. U suštini, optimizacija UPDATE naredbi se vrši slično kao i optimizacija SELECT naredbi sa

WHERE klauzulama, samo što postoji dodatni overhead zbog upisa podataka. Logično, brzina upisa zavisi od količine podataka koji se ažuriraju, tako da je više ažuriranja odjednom mnogo brže od pojedinanih. Neki primeri optimizacija koje MySQL može primeniti na UPDATE naredbu su:

- Index Merge Optimization - MySQL može koristiti više indeksa kako bi pronašao redove koji će biti ažurirani. Ako nema pojedinačnog indeksa koji će pronaći sve relevantne redove, MySQL može kombinovati više indeksa da bi došao do potrebnih redova.
- Index Condition Pushdown Optimization - Ova optimizacija omogućava MySQL-u da primeni WHERE uslov pretraživanjem indeksa. Umesto da prođe kroz sve redove u tabeli, MySQL koristi indeks da bi pronašao samo redove koji ispunjavaju uslov i zatim ih ažurira.
- Multi-Range Read Optimization - Ova optimizacija je slična Index Merge Optimization-u, ali umesto kombinovanja indeksa, MySQL može koristiti više pojedinačnih indeksa i izvršiti više opsežnih čitanja kako bi pronašao i ažurirao odgovarajuće redove.
- Batched Key Access Optimization - Ova optimizacija se koristi kada se ažurira više redova u istoj tabeli. MySQL može skupiti sve ključeve koji se ažuriraju i dohvatiti sve redove odjednom, smanjujući ukupan broj operacija koje je potrebno izvršiti.

Korišćenje indeksa i dobro napisani WHERE uslov mogu poboljšati performanse UPDATE naredbe u MySQL-u. Takođe, preporučuje se korišćenje transakcija za više UPDATE operacija kako bi se izbegli problemi sa konzistentnošću podataka.

MySQL vrši i automatsko optimizovanje brisanja (DELETE) podataka iz baze podataka. Neki od načina na koje MySQL optimizuje brisanje podataka:

- Optimizacija indeksa - Ako tabela ima indeks na koloni koja se koristi u WHERE klauzuli za DELETE upit, MySQL će koristiti indeks da pronađe redove koji treba da budu obrisani umesto da pretražuje celu tabelu. To smanjuje vreme potrebno za brisanje podataka.
- Pisanje podataka u binarne dnevnike - MySQL piše podatke koji se brišu u binarne dnevnike (binlog) umesto da ih odmah briše. Ovo omogućava drugim procesima da ih čitaju i izvrše neke druge radnje.
- Optimizacija memorije - MySQL koristi mehanizam keširanja memorije (cache) za čuvanje nedavno korišćenih podataka u memoriji. Ako se isti upit za brisanje ponavlja više puta, MySQL će koristiti keširanu memoriju umesto da ponovo čita podatke iz baze.
- Brisanje u delovima - Ako se velika količina podataka briše iz tabele, MySQL će obično podeliti DELETE upit u manje delove kako bi smanjio opterećenje baze. Na ovaj način se smanjuje uticaj brisanja na ostale procese koji se izvršavaju u bazi podataka.
- Održavanje integriteta podataka - MySQL obezbeđuje da se podaci brišu na način koji održava integritet podataka u bazi. Na primer, ako postoji strani ključ između dve tabele, MySQL će obrisati redove iz povezane tabele pre nego što obriše redove iz glavne tabele kako bi se izbeglo kršenje integriteta.

Sve ove tehnike automatske optimizacije DELETE upita u MySQL-u pomažu u poboljšanju performansi baze podataka i smanjuju vreme potrebno za brisanje podataka.

Zaključak

Optimizacija upita je važna tema u svetu baza podataka, posebno kada se radi sa velikim skupovima podataka. MySQL, kao popularni sistem za upravljanje bazama podataka, ima brojne opcije za optimizaciju upita koje su detaljno opisane u njihovoj dokumentaciji. U ovom radu su istražene različite tehnike optimizacije upita u MySQL-u, uključujući indeksiranje, upotrebu EXPLAIN i EXPLAIN ANALYZE naredbi, OPTIMIZER_TRACE i PROFILING tabela, itd.

Kao najvažniji alat za optimizaciju upita u MySQL-u izdvojila bih EXPLAIN naredbu koja pruža informacije o tome kako MySQL izvršava upit. Koristeći ovaj alat, mogu se videti indeksi koje koristi MySQL i kako izvršava pridruživanje i sortiranje. Takođe, osvrnuli smo se i na ostale tehnike optimizacije upita, poput indeksiranja, korišćenja LIMIT i GROUP BY klauzula, korišćenja podupita, itd. Utvrđeno je da je dobra praksa indeksiranja polja koja se često koriste u WHERE klauzulama. LIMIT klauzula se može koristiti za vraćanje samo nekoliko redova, što može biti korisno kada se radi s velikim skupovima podataka.

Međutim, važno je napomenuti da čak i kada koristimo sve ove navedene tehnike, loše napisani upiti mogu sprečiti optimizator da radi dobro. Stoga je ključno da se fokusiramo na pisanje efikasnih upita koji koriste optimalne tehnike. Potrebno je provoditi testiranje performansi kako bismo bili sigurni da se upiti izvršavaju brzo i efikasno. Naravno, ne treba se fokusirati na pisanje upita tako da se optimizacija ugleda na ono što MySQL optimizator već automatski radi.

Razumevanje optimizacije upita je od suštinskog značaja za razvoj softverskih sistema. Poznavanje MySQL-a i njegovih funkcionalnosti omogućava stvaranje brze i efikasne baze podataka, što je od velikog značaja u modernom softverskom inženjeringu. Stoga bi programeri definitivno trebalo da se upoznaju sa svim tehnikama optimizacije upita u MySQL-u kako bi izgradili performantne i skalabilne aplikacije.

Literatura

1. How to optimize query performance in MySQL databases, Elvis Duru https://coderpad.io/blog/development/optimize-query-performance-mysql/ (pristup 10.4.2023.)
2. MySQL Documentation – Chapter 8 Optimization, https://dev.mysql.com/doc/refman/8.0/en/optimization.html (pristup 10.4.2023.)
3. Optimizing Queries in MySQL: Optimizing Reads, https://www.red-gate.com/simple-talk/databases/mysql/optimizing-queries-in-mysql-optimizing-reads/ (pristup 10.4.2023.)
4. Understanding MySQL Queries with Explain, Sanja Bonic, https://www.exoscale.com/syslog/explaining-mysql-queries/ (pristup 10.4.2023.)
5. How to get optimizer trace for a query, http://oysteing.blogspot.com/2016/01/how-to-get-optimizer-trace-for-query.html (pristup 10.4.2023.)
6. MySQL Query performance Optimization Tips, Benson Karikuki, https://www.section.io/engineering-education/mysql-query-performance-optimization-tips/#:~:text=Optimizing%20Queries%20with%20EXPLAIN&text=According%20to%20the%20MySQL%20documentation,rows%20scanned%20in%20each%20table. (pristup 10.4.2023.)