



**Univerzitet u Nišu, Elektronski fakultet
Katedra za računarstvo**



Optimizacija upita u MySQL-u

Sistemi za upravljanje bazama podataka

Seminarski rad

Mentor:

Doc. dr Aleksandar Stanimirović

Student:

Vladana Stojiljković 1135

Niš, 2021.

Sadržaj

1. Uvod.....	1
2. MySQL optimizator upita	2
2.1. Primarne optimizacije	4
2.1.1. Optimizacija konstantnih relacija	4
2.1.2. Tipovi pristupa	7
2.1.3. ORDER BY klauzula.....	10
2.1.4. GROUP BY klauzula.....	11
2.2.Ostale optimizacije.....	13
2.2.1.Obrada NULL vrednosti	13
2.2.2.Particionisanje.....	14
3. Pisanje optimizovanih upita.....	15
3.1. Optimizacija SELECT naredbi	15
3.1.1. Optimizacija pretraživanja opsega	17
3.1.2. Spoljašnji spojevi	20
3.1.3. Optimizacija ORDER BY	21
3.1.4. Optimizacija GROUP BY klauzule	23
3.1.5. Optimizacija DISTINCT klauzule	25
3.2. Optimizacija INSERT, UPDATE i DELETE naredbi	26
4. Zaključak.....	27
5. Literatura.....	28

1. Uvod

Sistem za upravljanje bazama podataka (eng. *Database Management System*) ili DBMS je program ili skup programa kojim se vrši rukovođenje bazom podataka, velikom količinom podataka i izvršavanjem upita nad podacima koje zahtevaju krajnji korisnici. Upit (eng. *query*) je zahtev za informacijama iz baze podataka. Baze podataka imaju složenu strukturu, pa često i upiti mogu da budu kompleksni. Različiti upiti zahtevaju različita vremena izvršenja, u zavisnosti od načina pristupa bazi podataka. Vremena izvršenja upita utiču na efikasnost aplikacija, pa je u mnogim slučajevima potrebno izvršiti *optimizaciju upita*, tj. ubrzati rad DBMS-a. Svrha optimizacije upita je pronalaženje načina da se dati upit obradi u minimalnom vremenu [1]. Pronalaženje optimalnog načina izvršenja upita je dugotrajan proces jer prolazi kroz sve moguće načine izvršenja, pa iziskuje dosta vremena, a nekad je „najoptimalniji“ način nemoguće naći. Iz tih razloga se optimizacija upita vrši tako što se u obzir ne uzimaju sve varijante, već nekoliko varijanti koje se izvršavaju u razumnom vremenu, pa se umesto najoptimalnijeg načina traži dovoljno dobar način za izvršenje nekog upita.

MySQL je višenitni i višekorisnički relacioni DBMS. Radi kao server i obezbeđuje interfejs za interakciju sa bazom podataka. Poput drugih relacionih sistema i MySQL vrši automatsku optimizaciju upita, odnosno ima svoj optimizator (eng. *query optimizer*). Optimizator bira plan izvršenja za neki upit i detalji o optimizatora kod MySQL-a dati su u poglavlju 2. Pored automatske optimizacije, na efikasnost upita uticaj ima i sam korisnik, tj. način na koji zadaje upite. Zbog toga je prilikom pisanja upita, ukoliko je to moguće, potrebno definisati upit tako da se dobije dobar plan izvršenja. Preporuke za pisanje optimalnih upita uz odgovarajuće primere i komparaciju opisane su u poglavlju 3. Dok drugo poglavlje opisuje interne transformacije koje optimizator vrši da bi se poboljšale performanse izvršenja upita, treće poglavlje tiče se konkretnih slučajeva i saveta za postizanje veće efikasnosti. Ova dva poglavlja se prepliću jer se drugo bavi osnovnim transformacijama koje optimizator obavlja, a koje su sastavni deo dalje optimizacije upita, što je opisano u trećem poglavlju. Zaključak je dat u poglavlju 4, a korišćena literatura u poglavlju 5.

2. MySQL optimizator upita

Optimizator je skup rutina koje određuju koji plan izvršenja treba da koristi DMBS za upite [2]. Pored toga, optimizator može da izvrši transformaciju upita tako da transformisani upit daje iste rezultate kao originalni, ali se brže izvršava. Na slici 1 prikazan je kod koji izvršava MySQL optimizator.

```
handle_select()
mysql_select()
JOIN::prepare()
    setup_fields()
JOIN::optimize()          /* optimizer is from here ... */
    optimize_cond()
    opt_sum_query()
    make_join_statistics()
    get_quick_record_count()
    choose_plan()
        /* Find the best way to access tables */
        /* as specified by the user.          */
    optimize_straight_join()
        best_access_path()
    /* Find a (sub-)optimal plan among all or subset */
    /* of all possible query plans where the user   */
    /* controls the exhaustiveness of the search.   */
    greedy_search()
        best_extension_by_limited_search()
        best_access_path()
    /* Perform an exhaustive search for an optimal plan */
    find_best()
    make_join_select()      /* ... to here */
JOIN::exec()
```

Slika 1: Kod MySQL optimizatora [2]

Posle pripreme upita, sledi njegova optimizacija (deo koda od *JOIN::optimize*) . U okviru funkcija *optimize_cond()* i *opt_sum_query()* vrše se potrebne transformacije upita, nakon čega se prikupljaju informacije o indeksima koji mogu da budu korisni prilikom pristupa tabelama iz upita (*make_join_statistics*). Bira se odgovarajući plan izvršenja upita, a onda se upit izvršava. Svakom planu (ili delu plana) dodeljuje se cena. Cena plana predstavlja približno neophodne resurse za izvršenje upita prema planu, pri čemu je glavni faktor broj redova kojima će se pristupiti tokom izvršenja upita. Cene planova izvršenja su osnova za njihovu komparaciju. Zbog toga je cilj optimizatora pronalaženje plana izvršenja sa najnižom cenom. Kod spoja više tabela potrebno je spajati tabele redosledom koji daje optimalan pristup podacima. Optimizator vrši pretraživanje optimalnog plana odozdo nagore, tj. prvo uzima u obzir sve planove za jednu tabelu, onda za dve itd. dok ne dođe do optimalnog plana. Planovi izvršenja koji se sastoje od samo nekih tabela i predikata iz upita zovu se delimični planovi (eng. *partial plan*). Dodavanjem novih tabela u delimični plan, raste njegova cena. Zbog toga optimizator tabelama proširuje samo delimične planove čija je cena manja od trenutne najbolje.

Rad optimizatora može da se prati na dva načina: korišćenjem *EXPLAIN* naredbe ili pomoću tabele *OPTIMIZER_TRACE*. *EXPLAIN* naredba daje informacije o tome kako je MySQL izvršio upit i može da se koristi uz *SELECT*, *INSERT*, *UPDATE*, *DELETE* i *REPLACE* upite. Za svaku tabelu iz upita *EXPLAIN* vraća po jedan red informacija i to redosledom kojim MySQL čita tabele prilikom obrade upita. Format rezultata ove naredbe prikazan je na slici 2:

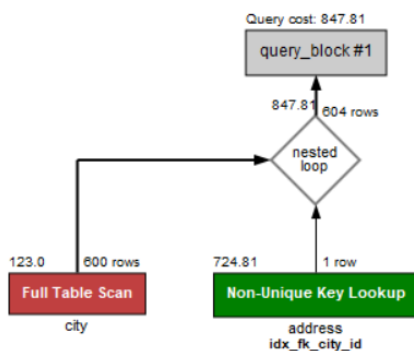
Column	JSON Name	Meaning
<u>id</u>	select_id	The SELECT identifier
<u>select_type</u>	None	The SELECT type
<u>table</u>	table_name	The table for the output row
<u>partitions</u>	partitions	The matching partitions
<u>type</u>	access_type	The join type
<u>possible_keys</u>	possible_keys	The possible indexes to choose
<u>key</u>	key	The index actually chosen
<u>key_len</u>	key_length	The length of the chosen key
<u>ref</u>	ref	The columns compared to the index
<u>rows</u>	rows	Estimate of rows to be examined
<u>filtered</u>	filtered	Percentage of rows filtered by table condition
<u>Extra</u>	None	Additional information

Slika 2: *EXPLAIN* naredba [2]

Tabela *OPTIMIZER_TRACE* sadrži informacije o upitu čije se izvršenje prati i ima 4 kolone:

- *query* – zadati upit
- *trace* – informacije od optimizatora u JSON formatu
- *missing_bytes_beyond_max_mem_size* – broj bajtova *trace*-a preko maksimalne, zadate veličine
- *insufficient_privileges* – označava da li korisnik može da vidi *trace* optimizatora.

Može da se koristi uz iste naredbe kao i *EXPLAIN* i kaže se da su informacije iz ove tabele komplementarne izlazu *EXPLAIN* naredbe. Drugim rečima, *EXPLAIN* prikazuje odabrani plan izvršenja, a *OPTIMIZER_TRACE* zašto je taj plan izabran. Pored toga, MySQL Workbench (integrirano okruženje za razvoj MySQL DBMS-a) može i vizuelno da prikaže izabrani plan izvršenja za postavljeni upit. Na slici 3 prikazan je primer ovakve vizuelizacije:



Slika 3: Primer plana izvršenja

Optimizacije i transformacije koje obavlja MySQL mogu da se podele na primarne i ostale [2]. Primarne optimizacije obuhvataju optimizaciju konstantnih relacija, tipova spoja, ORDER BY i GROUP BY klauzule, a ostale optimizacije obuhvataju specijalizovane slučajeve.

2.1. Primarne optimizacije

Pod primarnim optimizacijama podrazumevaju se najbitnije optimizacije koje obavlja MySQL server, što uključuje konstantne relacije, spojeve, ORDER BY i GROUP BY klauzule.

2.1.1. Optimizacija konstantnih relacija

Optimizacije konstantnih relacija podrazumevaju transformacije upita kod konstantne propagacije, eliminacije „mrtvog“ koda, preklapanja konstanti i konstantnih tabela.

Konstantna propagacija bazira se na zakonu tranzitivnosti (ako je $a = b$ i $b = c$, onda je i $a = c$). Ukoliko je dat upit

...WHERE column1 <op> column2 and column2 <op> 'x',

gde je <op> neki operator, transformiše se u sledeći oblik:

...WHERE column1 <op> 'x' and column2 <op> 'x'.

Transformacija se odvija ukoliko je operator neki iz skupa $\{=, <, >, >=, = <, < >, <= >, LIKE\}$, tj. ne važi samo za operator *BETWEEN* [2]. Slika 4 sadrži upit i deo trase optimizatora koji demonstriraju ovakvu transformaciju upita:

```
1 • use sakila;
2 • set optimizer_trace="enabled=on";
3 • select * from payment where customer_id=staff_id and staff_id = 1;
4 • select * from information_schema.optimizer_trace;

"steps": [
  {
    "condition_processing": {
      "condition": "WHERE",
      "original_condition": "((`payment`.`customer_id` = `payment`.`staff_id`) and (`payment`.`staff_id` = 1))",
```

Slika 4: Transformacija tranzivnost u WHERE klauzuli

Eliminacija mrtvog koda odnosi se na upite koji sadrže uslove koji su uvek tačni, netačni ili nemogući. Prilikom transformacije, takvi uslovi se otklanjaju. Neka je dat upit koji sadrži uslov koji je uvek ispunjen:

...*WHERE* 0 = 0 and column1 = 'x'.

Transformisani upit izgleda ovako:

...*WHERE* column1 = 'x'.

Ovakav slučaj demonstriran je na slici 5. U trasi optimizatora se vidi da se uslov 0=0 uopšte ne uzima u obzir, tj. da ga optimizator automatski uklanja.

```
1 • use sakila;
2 • set optimizer_trace="enabled=on";
3 • select * from payment where 0=0 and customer_id=1;
4 • select * from information_schema.optimizer_trace;

"condition_processing": {
  "condition": "WHERE",
  "original_condition": "((`payment`.`customer_id` = 1))",
```

Slika 5: Uklanjanje mrtvog koda

Ako neka kolona zbog ograničenja ne može da ima vrednost *NULL*, optimizator će da ukloni sve nevažne *IS NULL* uslove. Izuzetak su spoljašnji spojevi (eng. *outer join*) jer kod njih kolona koja ne može da bude *NULL*, ipak može da ima *NULL* vrednost. Optimizator ne može da uoči sve *nemoguće* uslove jer ima mnogo mogućnosti. To ilustruje sledeći primer (slika 6):

```
1 • use sakila;
2 • create table test_null_cond (label char(1));
   ...
3 • set optimizer_trace="enabled=on";
4 • select * from test_null_cond where label="test";
5 • select * from information_schema.optimizer_trace;

"condition_processing": {
  "condition": "WHERE",
  "original_condition": "(`test_null_cond`.`label` = 'test')",
```

Slika 6: Izuzetak kod uklanjanja NULL uslova

Optimizator neće da ukloni ovaj uslov, iako je on nemoguć zbog načina na koji je definisana tabela naredbom *CREATE TABLE*.

Kod preklapanja konstanti, izrazi koji sadrže više konstanti svode se na jednu, tj. uprošćavaju se. Ako je dat upit koji u *WHERE* klauzuli sadrži aritmetičku operaciju nad konstantama,

poput 1+2, transformacijom se dobija *WHERE kolona=3*. Ovakvi slučajevi su retki iz ugla korisnika koji zadaje upite, međutim mogu da se jave usled konstantne propagacije [2].

Konstantna tabela u MySQL-u je tabela koja ima jedan ili nijedan red, ili tabela zadata izrazom koji ima *WHERE* klauzulu tipa *column = const* oblika, za sve kolone primarnog ključa tabele ili za sve kolone bilo kog jedinstvenog ključa tabele (ako su definisani kao *NOT NULL*) [2]. Ako definicija neke tabele sadrži kolonu koja ne može da ima *NULL* vrednost, a pritom su vrednosti jedinstvene (*unique*), onda upit na osnovu te kolone vraća konstantnu tabelu. To znači da konstantne tabele imaju najviše jednu vrstu sa jedinstvenom vrednošću. Optimizator evaluira konstantnu tabelu unapred, da bi našao jedinstvenu vrednost, a onda tu vrednost ugrađuje u upit. Na slici 7 je prikazan primer ovakvog upita, kao i deo iz trase optimizatora koji sadrži transformisani uslov i koji pokazuje da se koristi konstantna tabela.

```
1 • use sakila;
2 • set optimizer_trace="enabled=on";
3 • explain select language.language_id, film.language_id from language, film
4   where language.language_id=film.language_id and language.language_id=1;
5 • select * from information_schema.optimizer_trace;
```

```
    "attaching_conditions_to_tables": {
      "original_condition": "(`film`.`language_id` = 1)",
      "attached_conditions_computation": [],
      "attached_conditions_summary": [
        {
          "table": "`film`",
          "attached": "(`film`.`language_id` = 1)"
        }
        ...
      ],
      "rows_estimation": [
        {
          "table": "`language`",
          "rows": 1,
          "cost": 1,
          "table_type": "const",
          "empty": false
        },
        ...
      ]
    }
```

Slika 7: Konstantne tabele

Najpre se uslov proverava, tj. proverava se da li postoji konstantna tabela i ako postoji, uslov se transformiše u *film.language_id=1 and 1=1* zbog konstantne propagacije, a onda se uslov *1=1* izbacuje jer je trivijalan. Na prethodnoj slici vidi se konačan uslov, kao i to da se za pristup koristi konstantna tabela koja ima jednu vrstu. Ako ne postoji vrsta iz prve tabele s vrednošću 5, optimizator evaluira *WHERE* uslov kao nemoguć, što se vidi iz poziva *EXPLAIN* naredbe u polju *Extra* (slika 8):


```

2 • explain select language.language_id, film.language_id from language, film
3   where language.language_id=film.language_id and language.language_id=100;
4
5
6

```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	no matching row in const table

Slika 8: Nemoguć WHERE kod konstantne tabele

2.1.2. Tipovi pristupa

Prilikom evaluacije izraza uslova, MySQL proverava koji tip spoja je u izrazu, pri čemu se tip spoja (eng. *join type*) odnosi na sve uslovne izraze, ne samo na spojeve tabela. Tipovi spojeva određuju na koji način se pristupa podacima, tj. tip pristupa i mogu da se poređaju po brzini od najboljeg do najgoreg na sledeći način: [2]

- *system* – sistemska, konstantna tabela
- *const* – konstantna tabela
- *eq_ref* – jedinstveni ili primarni indeks sa relacijom jednakosti
- *ref* – indeks s relacijom jednakosti koji ne može da bude *NULL*
- *ref_or_null* - indeks s relacijom jednakosti koji može da bude *NULL*
- *range* – indeks s relacijama poput *BETWEEN*, *LIKE*, *IN*, *>=* itd.
- *index* – sekvencijalno pretraživanje indeksa
- *ALL* – sekvencijalno pretraživanje tabele.



Na osnovu tipova spojeva optimizator bira drajver izraz kao najbolji spoj (eng. *driver expression*). Drajver je važan za odabir plana izvršenja. Na primer, loš plan izvršenja uključuje čitanje svake vrste iz tabele, a dobar plan uključuje pretragu pomoću indeksa. Zbog toga je bolje da drajver bude indeks, jer će pretraga biti kraća. Kod pretraživanja tabele po indeksiranoj koloni, optimizator može da vrati vrednosti iz samog indeksa, a ne iz tabele. Indeks koji se koristi na ovaj način zove se indeks pokrivanja (eng. *covering index*). Da bi neka kolona ili skup kolona mogao da se koristi kao indeks pokrivanja, potrebno je da postoji indeks nad tabelom i da on sadrži sve kolone iz *SELECT*-a.

Uslovi koji rade s indeksima nad opsegom ključeva zovu se uslovi opsega (eng. *range conditions*) i uključuju operatore *>*, *>=*, *<*, *<=*, *IN*, *LIKE*, *BETWEEN*. Za ovakve slučajeve optimizator koristi indekse, pri čemu vrši *range scan*. Na slikama 9 i 10 prikazan je izvršeni upit, *EXPLAIN* za dati upit gde se u polju *type* vidi da je tip pristupa *range*, kao i deo trase optimizatora koji ukazuje na to da se vrši *range scan*.

```

1 • use sakila;
2 • set optimizer_trace="enabled=on";
3 • explain select title from film where title like "a%";
4 • select * from information_schema.optimizer_trace;

```

Result Grid												
Filter Rows: <input type="text"/> Export:  Wrap Cell Content: 												
	id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
▶	1	SIMPLE	film	NULL	range	idx_title	idx_title	514	NULL	46	100.00	Using where; Using index

Slika 9: Upit sa LIKE uslovom opsega

```

"best_access_path": {
  "considered_access_paths": [
    {
      "rows_to_scan": 46,
      "filtering_effect": [],
      "final_filtering_effect": 1,
      "access_type": "range",
      "range_details": {
        "used_index": "idx_title"
      },
      "resulting_rows": 46,
      "cost": 10.163,
      "chosen": true
    }
  ]
}

```

Slika 10: Range scan pristup

U slučaju da je uslov takav da uprkos korišćenju indeksa zahteva iscrpno pretraživanje, optimizator menja tip spoja iz *range* u *all* (obično za uslove $>$ i $<$), ali nad indeksom. To znači da se i dalje pretražuje indeks, ali umesto pretraživanja opsega, prolazi se kroz ceo indeks. Na slici 11 prikazana su 2 upita koji koriste uslov $>$ i odgovarajuće trase optimizatora. Pošto kod drugog upita pretraživanje zahteva prolazak kroz ceo indeks, ne koristi se *range scan*.

```

1 • use sakila;
2 • set optimizer_trace="enabled=on";
3 • select rental_duration from film where rental_duration>3;

```

```

1 • use sakila;
2 • set optimizer_trace="enabled=on";
3 • select rental_duration from film where rental_duration>0;

```

```

"considered_access_paths": [
  {
    "rows_to_scan": 797,
    "access_type": "range",
    "range_details": {
      "used_index": "dur_index"
    },
    "resulting_rows": 797,
    "cost": 159.73,
    "chosen": true
  }
]

```

```

"considered_access_paths": [
  {
    "rows_to_scan": 1000,
    "access_type": "scan",
    "resulting_rows": 1000,
    "cost": 103,
    "chosen": true
  }
]

```

Slika 11: Razlika u pretraživanju opsega

Spajanje indeksa (eng. *index merge*) se koristi kad složeni uslov može da se konvertuje u oblik *uslov1 OR uslov2 OR ... uslovN*. Optimizator interno vrši ovu transformaciju ako se za svaki od uslova koristi *range scan* i ako nijedan par uslova ne koristi isti indeks. U slučaju da dva uslova koriste isti indeks, mogu da se kombinuju u jedan *range scan*, pa nema potrebe za spajanjem indeksa. Kod spajanja indeksa, MySQL povlači vrste za svaki indeks, a onda se uklanjaju duplikati. Na slici 12 prikazan je primer upita koji koristi spajanje indeksa, što se vidi iz trase optimizatora. Pošto za oba uslova iz upita postoje indeksi, koriste se i jedan i drugi i radi se njihovo spajanje.

```
1 • use sakila;
2 • set optimizer_trace="enabled=on";
3 • explain select * from film where language_id>1 or rental_duration>14;
4 • select * from information_schema.optimizer_trace optimizer_trace;
```

```

    "chosen_range_access_summary": {
      "range_access_plan": {
        "type": "index_merge",
        "index_merge_of": [
          {
            "type": "range_scan",
            "index": "idx_fk_language_id",
            "rows": 1,
            "ranges": [
              "1 < language_id"
            ]
          },
          {
            "type": "range_scan",
            "index": "dur_index",
            "rows": 1,
            "ranges": [
              "14 < rental_duration"
            ]
          }
        ]
      }
    }
  }
}

```

Slika 12: Spajanje indeksa

U slučaju da je složeni upit u formi *uslov1 AND uslov2 AND ... uslovN*, pri čemu su u pitanju uslovi opsega, ne vrši se spajanje indeksa. Proveravaju se mogući indeksi za date uslove i formiraju se mogući planovi izvršenja i njihove cene. Bira se indeks s najmanjom cenom i za njega se vrši *range scan*, što ilustruje primer na slici 13. Za oba uslova postoje indeksi, međutim cena pretraživanja prvog indeksa je mnogo veća nego za drugi, pa se bira indeks nad kolonom iz drugog uslova i vrši se *range scan*.

```

1 • use sakila;
2 • set optimizer_trace="enabled=on";
3 • explain select * from film where language_id>0 and rental_duration>14;
4 • select * from information_schema.optimizer_trace optimizer_trace;

```

```

"range_scan_alternatives": [
  {
    "index": "idx_fk_language_id",
    "ranges": [
      "0 < language_id"
    ],
    "index_dives_for_eq_ranges": true,
    "rowid_ordered": false,
    "using_mrr": false,
    "index_only": false,
    "rows": 1000,
    "cost": 350.26,
    "chosen": false,
    "cause": "cost"
  },
  {
    "index": "dur_index",
    "ranges": [
      "14 < rental_duration"
    ],
    "index_dives_for_eq_ranges": true,
    "rowid_ordered": false,
    "using_mrr": false,
    "index_only": false,
    "rows": 1,
    "cost": 0.61,
    "chosen": true
  }
]

```

Slika 13: Range scan

2.1.3. ORDER BY klauzula

Kad su u pitanju *ORDER BY* klauzule u upitu, optimizator preskače sortiranje rezultata ako uoči da su vrste svakako uređene. U slučaju da se u okviru klauzule nalazi indeksirana kolona, onda se koristi indeks. Ako indeks ne postoji, vrši se sortiranje tabele (*filesort*). Sledeći primer ilustruje upit s indeksiranom kolonom (slika 14). Pošto indeks postoji, ne vrši se sortiranje, pa polje *Extra* iz *EXPLAIN* tabele označava samo da se koristi indeks.

```

1 • use sakila;
2 • set optimizer_trace="enabled=on";
3 • explain select rental_duration from film order by rental_duration;

```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	film	<small>NULL</small>	index	<small>NULL</small>	dur_index	1	<small>NULL</small>	1000	100.00	Using index

Slika 14: ORDER BY nad indeksiranom kolonom

Ako je u pitanju mrtav kod, *ORDER BY* klauzula se uklanja. Na slici 15 prikazana su dva upita koji sadrže *ORDER BY*, pri čemu u prvom slučaju klauzula nema smisla. Kolona nije indeksirana, pa u *Extra* polju iz *EXPLAIN* tabele u drugom slučaju stoji *filesort* što znači da je klauzula izvršena, a u prvom slučaju je polje prazno, jer je klauzula uklonjena.

```

1 • use sakila;
2 • explain select rating from film order by "x";

```

	id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
▶	1	SIMPLE	film	<small>NULL</small>	ALL	<small>NULL</small>	<small>NULL</small>	<small>NULL</small>	<small>NULL</small>	1000	100.00	<small>NULL</small>


```

1 • use sakila;
2 • explain select rating from film order by rating;

```

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

Slika 15: Uklanjanje ORDER BY klauzule

U nekim slučajevima optimizator vrši sortiranje iako postoji indeks, što je ilustrovano na sledećem primeru (slika 16):

```

1 • use sakila;
2 • explain select rental_duration from film order by rental_duration+1;

```

	id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
▶	1	SIMPLE	film	<small>NULL</small>	index	<small>NULL</small>	dur_index	1	<small>NULL</small>	1000	100.00	Using index; Using filesort

Slika 16: Izuzetak kod indeksa za ORDER BY

Polje *Extra* označava da se koriste i indeks i sortiranje rezultata. Razlog je to što se indeks koristi za pribavljanje vrednosti (jeftinije je pretražiti indeks nego tabelu), a onda se na osnovu vrednosti dobijenim sabiranjem sa jedinicom vrši sortiranje.

2.1.4. GROUP BY klauzula

Optimizator poboljšava performanse izvršenja upita koji sadrži ovu klauzulu slično kao i za *ORDER BY*. U opštem slučaju se kod korišćenja ove klauzule skenira tabela i formira nova, privremena (eng. *temporary table*) u kojoj su sve vrste iz svake grupe uzastopne. Privremena tabela se onda koristi za određivanje grupa i eventualnu primenu funkcija agregacije. U slučaju da je kolona koja je u okviru klauzule indeksirana, prilikom pretraživanja se koristi indeks. Korišćenjem indeksa, izbegava se kreiranje privremene tabele, a samim tim se i upit izvršava brže. Preduslov za to je da su sve kolone iz *GROUP BY* deo jednog indeksa i da su ključevi u indeksu sortirani (B-stablo). Na slici 17 prikazani su odgovarajući upiti koji vrše grupisanje po indeksiranoj i po neindeksiranoj koloni i njihov *EXPLAIN*.

```

1 • use sakila;
2 • explain select rental_duration from film group by rental_duration;

```

	id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
▶	1	SIMPLE	film	<small>NULL</small>	range	dur_index	dur_index	1	<small>NULL</small>	6	100.00	Using index for group-by


```

1 • use sakila;
2 • explain select rating from film group by rating;

```

	id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
▶	1	SIMPLE	film	<small>NULL</small>	ALL	<small>NULL</small>	<small>NULL</small>	<small>NULL</small>	<small>NULL</small>	1000	100.00	Using temporary

Slika 17: Korišćenje indeksa i privremenih tabela kod GROUP BY klauzule

U poglavlju 3, koje se tiče preporuka za pisanje optimalnih upita, biće objašnjeni tipovi indeksa koji se koriste kod upita koji sadrže *GROUP BY* klauzulu, njihova efikasnost, kao i slučajevi korišćenja.

U određenim slučajevima optimizator može da transformiše upit tako da on sadrži *GROUP BY* klauzulu i to upite koji sadrže *DISTINCT*. Upit koji je u sledećem obliku:

SELECT DISTINCT kolona FROM tabela;

postaje:

SELECT kolona FROM tabela GROUP BY kolona.

Da bi ova transformacija mogla da se izvrši neophodna su dva uslova:

- postoji indeks nad kolonom uz *DISTINCT*, uz *FROM* stoji samo jedna tabela i nema *WHERE* klauzule
- nema *LIMIT* klauzule.

Na slici 18 prikazano je izvršeneje upita koji sadrži *DISTINCT* i deo trase optimizatora koji pokazuje da je izvršena transformacija u *GROUP BY*.

```
1 • use sakila;
2 • set optimizer_trace="enabled=on";
3 • select distinct title from film;
4 • select * from information_schema.optimizer_trace;

{
  "optimizing_distinct_group_by_order_by": {
    "changed_distinct_to_group_by": true,
    "simplifying_group_by": {
      "original_clause": "`film`, `title`",
      "items": [
        {
          "item": "`film`, `title`"
        }
      ]
    },
    "resulting_clause_is_simple": true,
    "resulting_clause": "`film`, `title`"
  }
},
{
```

Slika 18: Transformacija *DISTINCT* u *GROUP BY*

2.2.Ostale optimizacije

2.2.1. Obrada NULL vrednosti

Obrada NULL vrednosti se koristi kod *ref* i *eq_ref* tipova spoja (poglavlje 2.1.2). Svodi se na dodavanje *IS NOT NULL* uslova kod spojeva tabela na osnovu jednakosti primarnog ključa (ili njihovog dela ako su kompozitni) iz jedne tabele i neke kolone iz druge tabele. Kod spojeva dve ili više tabela po nekom uslovu *tabela1.ključ=tabela2.kolona*, čitaju se vrste iz obe tabele, a onda se proverava jednakost po zadatim kolonama. Primarni ključ sam po sebi ima ograničenje *NOT NULL*, međutim ukoliko je s druge strane jednakosti kolona koja nema to ograničenje, njihovo pribavljanje iz tabela može biti beskorisno. Zbog toga optimizator umeće dodatan uslov *IS NOT NULL* kod kolone u uslovu koja nije primarni ključ, čime se postiže da se prvo vrsta s tom kolonom čita iz tabele i proverava se da li je *NULL* i ako jeste, vrsta iz druge tabele se uopšte ne čita, već se ide na sledeću. Ovaj način obrade zove se rana obrada NULL vrednosti. Optimizator potvrđuje da je uslov dodat tako što u delu trase koji se odnosi na uslov postavlja *null_rejecting* na *true*, što ilustruje sledeći primer na slici 19:

```
1 • use sakila;
2 • set optimizer_trace="enabled=on";
3 • explain select staff_id, store.address_id from staff, store where staff_id=manager_staff_id;
4 • select * from information_schema.optimizer_trace;
```



```
  "ref_optimizer_key_uses": [
    {
      "table": "`staff`",
      "field": "staff_id",
      "equals": "`store`.`manager_staff_id`",
      "null_rejecting": true
    },
    {
      "table": "`store`",
      "field": "manager_staff_id",
      "equals": "`staff`.`staff_id`",
      "null_rejecting": true
    }
  ]
```

Slika 19: Dodavanje *IS NOT NULL* uslova

Prethodni primer odnosio se na *eq_ref* tip spoja. Optimizator može da obradi NULL vrednosti i kod *ref* tipa i ta obrada zove se kasna obrada NULL vrednosti. To je slučaj u kom se u *WHERE* klauzuli primarni ključ ili njegovi delovi upoređuju s nekim izrazima. Tada optimizator, pre pretraživanja indeksa, proverava da li je izraz s druge strane jednakosti NULL. U slučaju da jeste, indeks se ne pretražuje, već se vraća prazna tabela. Da je došlo do ovakve provere, optimizator signlizira isto kao i kod rane obrade NULL vrednosti postavljanjem atributa *null_rejecting* na *true* (slika 20):


```

1 • use sakila;
2 • set optimizer_trace="enabled=on";
3 • select * from film where film_id=5/0;
4 • select * from information_schema.optimizer_trace;

{
  "ref_optimizer_key_uses": [
    {
      "table": "film",
      "field": "film_id",
      "equals": "(5 / 0)",
      "null_rejecting": true
    }
  ]
}

```

Slika 20: Kasna obrada NULL vrednosti

2.2.2.Particionisanje

Particionisanje u MySQL-u omogućava podelu tabela na manje delove koji se distribuiraju u fajl sistemu na osnovu pravila koje zadaje korisnik [3]. Ta pravila predstavljaju funkciju particionisanja. Svaki deo neke tabele čuva se kao posebna tabela na različitoj lokaciji. MySQL optimizator može da izvrši odsecanje particija (eng. *partition pruning*), tj. da odredi minimalni skup particija koje treba pretražiti da bi se ispravno izvršio upit. Kardinalnost tog skupa može da bude manja od ukupnog broja particija neke tabele. Particijama van tog skupa se ne pristupa uopšte, čime se ubrzava izvršenje upita [2]. Odsecanje particija se vrši analiziranjem *WHERE* klauzule i konstruisanjem grafa intervala koji opisuje rezultat analize. Nakon toga se kretanjem kroz graf traže particije koje će da se koriste u svakom intervalu grafa, a na kraju se formira skup particija za ceo upit. Najprostiji slučaj za odsecanje je kod upita koji u uslovima u *WHERE* klauzuli imaju sve kolone na osnovu kojih je izvršeno particionisanje, pri čemu su uslovi povezani *AND* operatorom. Tada se računa funkcija particionisanja i na osnovu nje se određuje kojoj se particiji pristupa. Ukoliko je particionisanje izvršeno na osnovu neke kolone sa celobrojnim vrednostima i ta kolona se javi u *WHERE* klauzuli u formi *...WHERE const1 <= kolona <= const2*, optimizator uslov pretvara u više uslova povezanih *OR* operatorom (*kolona=const1+1 OR kolona=const1+2 OR ... kolona=const2*). Ova konverzija zove se obilazak intervala (eng. *interval walking*) [2]. Obilazak kratkih intervala nije skupa operacija jer se smanjuje broj particija, ali veći intervali mogu da uspore izvršenje. To može da se ubrza postavljanjem koraka (*#define MAX_RANGE_TO_WALK=const*) [2]. U slučaju particionisanja na osnovu opsega (*range*) neke kolone, svaka particija sadrži interval vrednosti te kolone i intervali su disjunkt. Kod pretraživanja po nekom opsegu, particiji se pristupa samo ako njen interval i interval pretraživanja imaju neprazan presek. Particionisanje može da se vrši i na osnovu skupa vrednosti. U tim slučajevima se particiji pristupa ako bar jedna tačka iz particije pripada intervalu po kom se vrši pretraživanje. Ako se particionisanoj tabeli pristupa pomoću indeksa (*eq_ref*, *ref*, *ref_or_null*), MySQL proverava da li ima potrebe da se pristupa indeksima za svaku particiju tabele ili broj particija može da se ograniči. Ova provera se vrši za svako pretraživanje indeksa.

3. Pisanje optimizovanih upita

3.1. Optimizacija SELECT naredbi

SELECT naredbe su osnova za pretraživanje podataka iz baze. Optimizacija ovih naredbi ima visok prioritet, bilo da su u pitanju kratki upiti na veb sajtovima koji treba da se izvršavaju što kraće ili upiti koji generišu izveštaje i izvršavaju se jako dugo [4]. Osnova za ubrzanje *SELECT* upita koji sadrži *WHERE* naredbu je korišćenje indeksa, odnosno indeksiranje kolona koje se javljaju u uslovima u *WHERE* klauzuli. Indeksi su značajni pogotovo za upite koji pribavljaju informacije iz više tabela. Dobra praksa je kreiranje malog skupa indeksa za srodne upite koji se često javljaju u aplikaciji [4]. Ukoliko optimizator ne uzima u obzir kreirane indekse, potrebno je ažurirati statistiku optimizatora s vremena na vreme pozivom naredbe *ANALYZE TABLES*, tako da optimizator uvek ima sveže informacije za formiranje optimalnog plana izvršenja upita. Često na loše performance izvršenja upita utiče samo jedan njegov deo, poput poziva neke funkcije. Te delove treba izolovati i optimizovati nezavisno od ostatka upita. Takođe, na loše performance utiče i pretraživanje velikog broja vrsti u velikim tabelama, pa upit treba napisati tako da se broj pribavljenih vrsti smanji. O izvršenju upita može da se sazna dosta pozivom *EXPLAIN* naredbe, koja je već pomenuta i korišćena u prethodnom poglavlju. Na osnovu polja koje vraća ova naredba, može se zaključiti gde treba kreirati indekse, kako transformisati uslov i sl. Pritom, treba imati u vidu da je nepotrebno vršiti transformacije koje ugrađeni optimizator (poglavlje 2) vrši sam jer upit može da postane manje razumljiv.

U nekim slučajevima, MySQL i pored postojanja indeksa može da se odluči za skeniranje cele tabele. To se obično dešava kada je tabela toliko mala da je brže izvršiti skeniranje tabele, nego da se pretražuje indeks. To je uobičajeno za tabele sa manje od 10 vrsti i malim brojem kolona. Ako je upoređivanjem indeksirane kolone sa konstantnim vrednostima MySQL izračunao (na osnovu stabla indeksa) da konstante pokrivaju preveliki deo tabele, optimizator može da odluči da bi skeniranje tabele bilo brže. Za male tabele često je pogodno izvršiti skeniranje tabele tako da uticaj na performanse bude zanemarljiv. Kad su u pitanju velike tabele, korišćenje indeksa je obično bolja opcija. U tu svrhu treba koristiti već pomenutu naredbu *ANALYZE TABLE tabela* za ažuriranje distribucije ključeva za skeniranu tabelu i naredbu *FORCE INDEX* za skeniranu tabelu kako bi se MySQL-u naglasilo da je bolje koristiti indeks nego skenirati tabelu [4]. Pored ove komande, moguće je “zabraniti” optimizatoru da koristi neki indeks pozivom *IGNORE INDEX*. Postoji i varijanta između ove dve, *USE INDEX*, koja samo sugeriše optimizatoru da iskoristi neki indeks, ali ga ne primorava da ga iskoristi. Sledeći primer ilustruje isti upit uz korišćenje ovih funkcija. U *EXPLAIN*-u se u polju mogućih indeksa javljaju *idx_address_id* i *idx_unique_manager*, pa optimizator u prvom slučaju samostalno, na osnovu cene, određuje koji indeks da iskoristi. Pošto je cena za *idx_address_id* manja, u drugom slučaju optimizatoru je sugerisano da koristi indeks *idx_unique_manager* iako je “skuplji” što je optimizator i prihvatio jer razlika u ceni nije bila velika. U trećem je nametnut indeks *idx_unique_manager*, a u četvrtom se optimizatoru ne dozvoljava korišćenje

ovog indeksa. Kao rezultat ovih nagoveštaja optimizatoru, iz polja *possible_keys* (*EXPLAIN*) se brišu odgovarajući indeksi. Pomenuti upiti prikazani su na slici 21:

```
1 • use sakila;
2 • explain select address_id from store where manager_staff_id<4 and address_id=4;
```

	id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
►	1	SIMPLE	store	NULL	ref	idx_unique_manager,idx_fk_address_id	idx_fk_address_id	2	const	1	100.00	Using where

```
1 • use sakila;
2 • explain select address_id from store use index (idx_unique_manager) where manager_staff_id<4 and address_id=4;
```

	id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
►	1	SIMPLE	store	NULL	range	idx_unique_manager	idx_unique_manager	1	NULL	2	50.00	Using index condition; Using where

```
• use sakila;
• explain select address_id from store force index (idx_unique_manager) where manager_staff_id<4 and address_id=4;
```

	id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
►	1	SIMPLE	store	NULL	range	idx_unique_manager	idx_unique_manager	1	NULL	2	50.00	Using index condition; Using where

```
1 • use sakila;
2 • explain select address_id from store ignore index (idx_unique_manager) where manager_staff_id<4 and address_id=4;
```

	id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
►	1	SIMPLE	store	NULL	ref	idx_fk_address_id	idx_fk_address_id	2	const	1	50.00	Using where

Slika 21: Force, Ignore i Use index funkcije

Kod korišćenja indeksa za pribavljanje podataka iz tabele, za optimizaciju se koristi potiskivanje uslova indeksa (eng. *Index Condition Pushdown*, *ICP*). Bez korišćenja ICP-a, mehanizam za skladištenje¹ (eng. *storage engine*) obilazi indeks za nalaženje vrsti iz tabele, a onda se vrši provera *WHERE* uslova. To znači da se pribavljanje vrsti obavlja tako što se pročita ključ iz indeksa, a onda se na osnovu njega pribavlja vrsta iz tabele, nakon čega se proverava uslov i odbacuju vrste koje ga ne zadovoljavaju. Kad se ICP omogući, pod uslovom da delovi *WHERE* uslova mogu da se provere samo na osnovu indeksiranih kolona, MySQL potiskuje ove delove do mehanizma za skladištenje. On onda može da evaluira uslov na osnovu indeksa i da pribavi vrste iz tabele samo kada je uslov ispunjen, čime se smanjuje broj pristupa tabeli. Ukoliko postoje *WHERE* uslovi koji nisu potisnuti, na kraju se proveravaju i oni. Omogućavanje i onemogućavanje ICP-a vrši se pomoću sledećih naredbi [4]:

```
SET optimizer_switch = 'index_condition_pushdown=off';
```

```
SET optimizer_switch = 'index_condition_pushdown=on';
```

¹ Mehanizam za skladištenje je MySQL komponenta koja obrađuje SQL operacije za različite tipove tabela

3.1.1. Optimizacija pretraživanja opsega

Range tip spoja (pristupa) podrazumeva korišćenje indeksa za pribavljanje podskupa vrsti tabele kod kojih je (indeksirana) kolona u okviru intervala indeksa koji se pretražuje. Može da se koristi i prost i složen (kompozitni) indeks. *Range scan* se koristi kod upita u sledećim slučajevima [4]:

- *range* pristup za proste indekse
- *range* pristup za složene indekse
- optimizacije opsega jednakosti kod viševrednosnih komparacija
- metoda preskakanja *range* pristupa (eng. *skip scan range access method*)
- optimizacija opsega za izraze za formiranje vrsti
- ograničavanje korišćenja memorije za optimizaciju opsega.

Kod prostih indeksa, koji se sastoje od jedne kolone, intervali vrednosti indeksa mogu da se predstave odgovarajućim uslovima u klauzuli *WHERE* koji se posmatraju kao uslovi opsega.

Pretraživanje opsega se vrši u upitima kod kojih se u *WHERE* uslovu javljaju operatori *>*, *<*, *=*, *<>*, *!=*, *LIKE* [4]. *LIKE* operator koristi pretraživanje opsega samo ako string po kom se vrši pretraživanje počinje konstantnim karakterom. Zbog toga treba izbegavati upite koji u *WHERE* klauzuli sadrže *LIKE* čiji je argument string koji počinje džoker znakom (eng. *wildcard*) [5]. Sledeća slika ilustruje razliku između upita koji kao argument *LIKE* operatora imaju string koji počinje konstantnim karakterom i koji počinje džoker znakom.

```
1 • use sakila;
2 • explain select title from film where title like "c%";
```

	id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
▶	1	SIMPLE	film	NULL	range	idx_title	idx_title	514	NULL	92	100.00	Using where; Using index


```
1 • use sakila;
2 • explain select title from film where title like "%c";
```

	id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
▶	1	SIMPLE	film	NULL	index	idx_title	idx_title	514	NULL	1000	11.11	Using where; Using index

Slika 22: Optimizacija *LIKE* operatora

MySQL pokušava da ekstrahuje uslove opsega iz klauzule *WHERE* za svaki od mogućih indeksa. Zbog toga upite koji treba da pretražuju neki opseg vrednosti treba pisati tako da kolone iz *WHERE* klauzule budu indeksirane. Tokom procesa ekstrakcije uklanjaju se uslovi koji ne mogu da se koriste za konstruisanje uslova opsega, kombinuju se uslovi koji proizvode preklapajuće opsege i uklanjaju se uslovi koji proizvode prazne opsege. Ekstrakcija se vrši tako što se u prvom koraku uslovi koji nisu uslovi opsega zamenjuju vrednošću *TRUE*, zatim se uslovi koji su uvek tačni ili netačni (npr. *TRUE OR uslov* je uvek ispunjeno, *FALSE AND uslov* je uvek netačno) evaluiraju na *TRUE* ili *FALSE* i na kraju se te konstante uklanjaju. Ovaj vid optimizacije može da izvrši i korisnik direktno prilikom pisanja upita, čime se smanjuje “posao” koji vrši optimizator.

Kad su u pitanju kompozitni indeksi, koji se sastoje od više kolona, uslov opsega postavlja restrikciju da vrste u indeksu moraju da budu deo jedne ili nekoliko torki (eng. *tuple*). Na primer, ako postoji indeks nad kolonama (*kolona1*, *kolona2*, *kolona3*), uslov *kolona1=5* ispunjavaju sve torke indeksa kod kojih je uslov ispunjen, tj. u obliku su (*5*, *kolona2*, *kolona3*). Da bi *range* pristup mogao da se koristi, uslovi moraju da se navode za kolone po redosledu iz indeksa. Drugim rečima, da je dat uslov *kolona3="nešto"*, nema opsega, pa ni korišćenja *range* pristupa.

Ako se nad delovima indeksa postavljeni uslovi jednakosti povezani *OR* operatorom, u pitanju su jednakosti opsega (iako je opseg jedna vrednost) i celokupan uslov je ispunjen ako je bar jedan od uslova tačan. Za ovakve upite optimizator procenjuje broj vrsti kojima se pristupa na osnovu tipa indeksa (za *unique* indeks najviše jedna vrsta može da ima traženu vrednost, a za *non-unique* indekse više njih). Procena broja vrsti vrši se pomoću statistike indeksa ili uranjanja u indeks (eng. *index dive*) [4]. Za ažuriranje statistike indeksa koristi se naredba *ANALYZE TABLE*. Kod uranjanja u indeks, optimizator pristupa krajevima zadatog opsega u indeksu i uzima broj vrsti u opsegu kao procenu. Ovakva procena je tačna, ali s porastom broja jednakosti opsega, proces postaje spor. Iz tog razloga, korisnik može da konfiguriše vrednost koja označava granicu za promenu načina procene, što se vrši postavljanjem sistemske promenljive *eq_range_index_dive_limit* [4]. Ako se vrednost postavi na *n*, to znači da će se uranjanje u indeks koristiti ako ima najviše *n-1* opsega jednakosti, inače se koristi statistika indeksa. Takođe, ukoliko korisnik želi da koristi isključivo statistiku indeksa za procenu broja vrsti, to može da postigne pisanjem upita koji imaju neku od sledećih karakteristika:

- upit se odnosi na jednu tabelu,
- postoji *FORCE INDEX* u upitu, što znači da nema potrebe za uranjanjem u indeks,
- nema podupita,
- indeks je *non-unique*,
- nema *GROUP BY*, *ORDER BY* ili *DISTINCT* klauzule u upitu.

Na slici 23 je dat sledeći upit i *EXPLAIN* za taj upit :

```

1 • use sakila;
2 • CREATE TABLE test_skip_scan (col1 INT NOT NULL, col2 INT NOT NULL, PRIMARY KEY(col1, col2));
3 • INSERT INTO test_skip_scan VALUES
4   (1,1), (1,2), (1,3), (1,4), (1,5),
5   (2,1), (2,2), (2,3), (2,4), (2,5);
6 • INSERT INTO test_skip_scan SELECT col1, col2 + 5 FROM test_skip_scan;
7 • INSERT INTO test_skip_scan SELECT col1, col2 + 10 FROM test_skip_scan;
8 • INSERT INTO test_skip_scan SELECT col1, col2 + 20 FROM test_skip_scan;
9 • INSERT INTO test_skip_scan SELECT col1, col2 + 40 FROM test_skip_scan;
10 • ANALYZE TABLE test_skip_scan;
11 • EXPLAIN SELECT col1, col2 FROM test_skip_scan WHERE col2 > 40;

```

	id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
►	1	SIMPLE	test_skip_scan	NULL	range	PRIMARY	PRIMARY	8	NULL	53	100.00	Using where; Using index for skip scan

Slika 23: Skip scan

U ovakvim slučajevima je pretraživanje opsega indeksa efektivnije od obilaska celog indeksa, ali je nemoguće jer je indeks kreiran nad *col1* i *col2*, a uslov se odnosi samo na *col2*. Kao što se vidi iz polja *Extra*, u izvršenju je iskorišćen indeks za preskakanje opsega (eng. *skip scan*). Preskakanje opsega optimizator izvršava kao veliki broj pretraživanja opsega, za svaku vrednost *col1*. Obavlja se tako što se preskaču iste vrednosti prvog dela indeksa *col1* (prefiksa indeksa). Za svaku (različitu vrednost) *col1* se proverava uslov koji se odnosi na *col2* (*col1*>40). Da bi optimizator mogao da izvrši preskakanje opsega, upit treba da bude napisan na sledeći način [4]:

4. tabela *T* ima najmanje jedan kompozitni indeks formata ([*A*₁, ..., *A*_k] *B*₁, ..., *B*_m, *C* [, *D*₁, ..., *D*_n]), gde *A* i *D* mogu biti prazni, ali *B* i *C* ne smeju biti prazni.
5. upit se odnosi samo na jednu tabelu;
6. upit ne sadrži *GROUP BY* ili *DISTINCT*;
7. upit se odnosi samo na indeksirane kolone;
8. predikati na *A*₁, ..., *A*_k moraju biti predikati jednakosti i moraju biti konstante, što uključuje i operator *IN*;
9. upit mora biti konjunktivni upit: (uslov1 (deo_ključa1) OR uslov2 (deo_ključa2)) AND (uslov1 (deo_ključa2) OR ...) AND ...;
10. postoji uslov opsega nad *C*;
11. ukoliko postoji uslovi nad *D*, moraju da budu u konjunktiji sa uslovima nad *C*.

Još jedna primena optimizacije pomoću pretraživanja opsega jeste kod upita koji sadrže izraze konstruktora vrsti (eng. *row constructor expression*) u obliku ...*WHERE* (*col1*, *col2*) *in* ((*"a"*, *"b"*), (*"c"*, *"d"*)). Da bi optimizator mogao da ubrza izvršenje ovih upita, upite treba pisati tako da sadrže *IN* operator (ne i *NOT IN*), da s leve strane operatora *IN* budu samo kolone iz tabele, da s desne strane operatora budu *runtime* konstante i više konstruktora. Da bi se kod ovakvih upita koristili indeksi, konstruktor mora da sadrži prefiks indeksa. To je ilustrovano na primerima na slici 24:

```

1 • use sakila;
2 • alter table customer add index row_const_ind (customer_id, store_id, address_id);
3 • analyze table customer;
4 • explain select customer_id, store_id from customer where (customer_id, store_id, address_id) in ((1, 1, 1), (2, 2, 1));

```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	customer	NULL	range	PRIMARY,idx_fk_store_id,idx_fk_address_id,ro...	row_const_ind	5	NULL	2	100.00	Using where; Using index

```

1 • use sakila;
2 • explain select customer_id, store_id from customer where (address_id, store_id) in ((1, 1), (2, 2));

```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	customer	NULL	range	idx_fk_store_id,idx_fk_address_id	idx_fk_address_id	2	NULL	2	100.00	Using where

Slika 24: Range scan kod konstruktora vrsti

U oba slučaja se koristi *range* tip pristupa, ali za različite indekse. Kod prvog upita se koristi kreirani indeks za pribavljanje vrsti, dok se u drugom upitu koristi već postojeći indeks *idx_fk_address_id* jer konstruktor ne sadrži prefiks indeksa (*customer_id*).

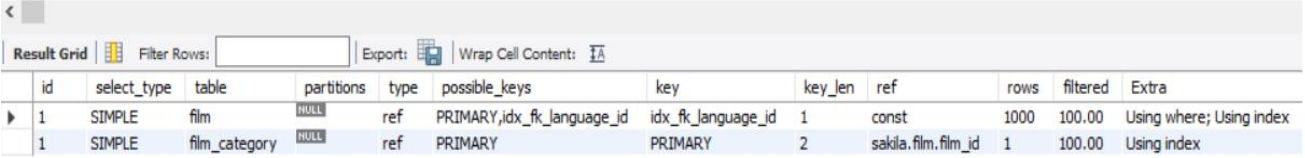
3.1.2. Spoljašnji spojevi

Spoljašnji spojevi podrazumevaju *LEFT JOIN* i *RIGHT JOIN*. MySQL implementira *A LEFT JOIN B* spajanje tabela na sledeći način [4]:

- Tabela B se postavlja da zavisi od tabele A i svih tabela od kojih zavisi A.
- Tabela A se postavlja tako da zavisi od svih tabela (osim B) koje se koriste u *LEFT JOIN* spoju.
- Na osnovu uslova uz *LEFT JOIN* utvrđuje se način pristupa redovima iz tabele.
- Izvode se sve standardne optimizacije spojeva, s tim što se tabela uvek čita nakon svih tabela od kojih zavisi. Ako postoji kružna zavisnost, dolazi do greške.
- Izvode se sve standardne *WHERE* optimizacije.
- Ako u A postoji red koji se podudara sa klauzulom *WHERE*, ali u B ne postoji red koji se podudara sa uslovom *ON*, generiše se dodatni B red sa svim kolonama postavljenim na NULL.

Implementacija *RIGHT JOIN* je analogna realizaciji *LEFT JOIN* sa obrnutim ulogama tabele. Desni spojevi se pretvaraju u ekvivalentna leva spajanja. Za *LEFT JOIN*, ako je *WHERE* uslov uvek netačan (eng. *null rejected*) za generisani NULL red, *LEFT JOIN* se menja u unutrašnji spoj. Na slici 25 dat je primer upita u kom se vrši ovakva transformacija. Upit sadrži i trivijalan uslov $0=1$ koji je uvek tačan, pa se on izbacuje, a onda se spoljašnji spoj tabela konvertuje u unutrašnji. Sad optimizator može da promeni redosled spajanja tabela tako da se dobije bolji plan izvršenja, a korisnik može optimizatoru i da zada redosled kojim treba da spoji tabele, pomoću klauzule *JOIN_ORDER* (*tabela1, tabela2...*) [4].

```
1 • use sakila;
2 • set optimizer_trace="enabled=on";
3 • explain select film.film_id, category_id from film_category left join film on film_category.film_id=film.film_id
4   where language_id=1 or 0=1;
5 • select * from information_schema.optimizer_trace;
```



```
{
  "transformations_to_nested_joins": {
    "transformations": [
      "outer_join_to_inner_join",
      "JOIN_condition_to_WHERE",
      "parenthesis_removal"
    ],
  },
}
```

Slika 25: Promena spoljašnjeg u unutrašnji spoj

Kod ugnježenih spojeva, uslov može da se evaluira kao nemoguć za jedan spoljašnji spoj, a moguć za drugi. Spoj za koji je uslov *null rejected* se menja u unutrašnji. Primer toga je upit na slici 26:

```

1 • use sakila;
2 • set optimizer_trace="enabled=on";
3 • select * from customer left join store on customer.store_id=store.store_id
4 • left join staff on staff.staff_id=store.manager_staff_id
5 • where staff.staff_id>0;

{
  "transformations_to_nested_joins": {
    "transformations": [
      "outer_join_to_inner_join",
      "JOIN_condition_to_WHERE",
      "parenthesis_removal"
    ],
    "expanded_query": "/* select#1 */ select `customer`.`customer_id` AS `customer_id`,`customer`.`store_id`
AS `store_id`,`customer`.`first_name` AS `first_name`,`customer`.`last_name` AS `last_name`,`customer`.`email`
AS `email`,`customer`.`address_id` AS `address_id`,`customer`.`active`AS `active`,`customer`.`create_date`
AS `create_date`,`customer`.`last_update` AS `last_update`,`store`.`store_id` AS `store_id`,`store`.`manager_staff_id`
AS `manager_staff_id`,`store`.`address_id` AS `address_id`,`store`.`last_update` AS `last_update`,`staff`.`staff_id`
AS `staff_id`,`staff`.`first_name` AS `first_name`,`staff`.`last_name` AS `last_name`,`staff`.`address_id`
AS `address_id`,`staff`.`picture` AS `picture`,`staff`.`email` AS `email`,`staff`.`store_id` AS `store_id`,`staff`.`active`
AS `active`,`staff`.`username` AS `username`,`staff`.`password` AS `password`,`staff`.`last_update` AS `last_update`
from `customer` join `store` join `staff` where ((`staff`.`staff_id` > 0) and
(`staff`.`staff_id` = `store`.`manager_staff_id`) and (`customer`.`store_id` = `store`.`store_id`))"
}

```

Slika 26: Ugnježdeni spoljašnji spojevi

Uslov u *WHERE* klauzuli je *null rejected* za drugi spoljašnji spoj, pa ga optimizator menja u unutrašnji. Nakon izvršenja upita dobijaju se isti rezultati kao da je odmah napisan upit:

```

SELECT * FROM customer LEFT JOIN store ON customer.store_id=store.store_id
INNER JOIN staff ON staff.staff_id=store.manager_staff_id
WHERE staff.staff_id>0.

```

Da bi optimizator konvertovao spoljašnji spoj u unutrašnji, on mora da ispita i evaluira uslove iz *WHERE* klauzule. Pored toga, ako je u pitanju desni spoj, prvo mora da ga konvertuje u levi. Zbog toga je generalno dobro izbegavati spoljašnje spojeve kad god je to moguće i koristiti unutrašnje umesto njih.

3.1.3. Optimizacija ORDER BY

Kao što je već u rečeno u drugom poglavlju, optimizator može da iskoristi indeks da bi izbegao *filesort* operaciju za *ORDER BY*. Indeks može da se koristi čak i ako se *ORDER BY* ne podudara tačno sa indeksom, sve dok su svi neiskorišćeni delovi indeksa i svi dodatne kolone *ORDER BY* konstante u klauzuli *WHERE* [4]. Ako indeks ne sadrži sve kolone kojima se pristupa u upitu, indeks se koristi samo ako je pristup indeksu jeftiniji od ostalih metoda pristupa.

Ukoliko postoji indeks *indeks(kolona1, kolona2)* i u upitu se javlja *ORDER BY kolona1, kolona2*, optimizator ima mogućnost da iskoristi indeks. Problem mogu da predstavljaju upiti

koji vrše pribavljanje velikog broja kolona kojih nisu u *ORDER BY* (npr. *SELECT **). U tim slučajevima, optimizator može da odluči da je pretraživanje tabele i sortiranje rezultata jeftinije od korišćenja indeksa. Kod InnoDB tabela, primarni ključ je implicitno deo indeksa, tako da i pribavljanje većeg broja kolona od broja indeksiranih koristi indeks čime se izbegava sortiranje. U upitu sa slike 27 *inventory_id* je primarni ključ, a kreiran je indeks nad (*film_id*, *store_id*). Iako se pribavlja i primarni ključ, nema sortiranja rezultata jer je on deo indeksa.

```

1 • use sakila;
2 • set optimizer_trace="enabled=on";
3 • alter table inventory add index ind_film_store (film_id, store_id);
4 • explain select inventory_id, film_id, store_id from inventory order by film_id, store_id

```

	id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
►	1	SIMPLE	inventory	<small>NULL</small>	index	<small>NULL</small>	ind_film_store	3	<small>NULL</small>	4581	100.00	Using index

Slika 27: Primarni ključ kod *ORDER BY* optimizacije

Ako sortiranje treba izvršiti po dve kolone, to je moguće uraditi u istom ili suprotnom smeru (*ASC* i *DESC*). Potreban uslov da bi se koristio indeks je da ima istu homogenost, dok pravci sortiranja mogu da budu različiti [4]. To znači da ako u upitu postoji sortiranje po suprotnom smeru, indeks se koristi ako se kolone indeksa poklapaju po smeru sa odgovarajućim kolonama po kojima se vrši sortiranje. Na slici 28 prikazan je *EXPLAIN* za tri upita koja se razlikuju po smeru sortiranja. Najpre je kreiran indeks koji sadrži rastuću i opadajuću kolonu, a onda je izvršen *ORDER BY* po kolonama iz indeksa tako da se smerovi sortiranja poklapaju sa smerovima u kreiranom indeksu. U polju *Extra* nema *filesort*, što znači da sortiranje nije izvršeno. Drugi slučaj koristi *ORDER BY*, ali po suprotnim smerovima u odnosu na one iz indeksa. Međutim, i u tom slučaju se ne vrši *filesort*, već se indeks pretražuje unazad. Treći slučaj pokriva ostale slučajeve i polje *Extra* označava da je došlo do sortiranja, što znači da upit nije optimizovan na pravi način. Zbog toga je, kod sortiranja po različitim smerovima, dobro kreirati indekse kojim će se izbeći sortiranje rezultata.

```

1 • use sakila;
2 • set optimizer_trace="enabled=on";
3 • CREATE TABLE test_order_dir (
4     col1 INT, col2 INT,
5     INDEX idx1 (col1 ASC, col2 DESC)
6 );

```

```

set optimizer_trace="enabled=on";
explain select * from test_order_dir order by col1 asc, col2 desc;

```

	id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
►	1	SIMPLE	test_order_dir	<small>NULL</small>	index	<small>NULL</small>	idx1	10	<small>NULL</small>	5	100.00	Using index

```
explain select * from test_order_dir order by col1 desc, col2 asc;
```

	id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
►	1	SIMPLE	test_order_dir	<small>NULL</small>	index	<small>NULL</small>	idx1	10	<small>NULL</small>	5	100.00	Backward index scan; Using index

```
explain select * from test_order_dir order by col1 desc, col2 desc;
```

	id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
►	1	SIMPLE	test_order_dir	<small>NULL</small>	index	<small>NULL</small>	idx1	10	<small>NULL</small>	5	100.00	Using index; Using filesort

Slika 28: *ORDER BY* po različitim smerovima

U nekim slučajevima optimizator ne koristi indeks za *ORDER BY*, ali koristi indeks za pribavljanje podataka iz tabele. To se dešava kad sortiranje treba izvršiti po različitim indeksima, kad u *ORDER BY* nisu uzastopni delovi indeksa, kad se koriste različiti indeksi za pribavljanje i *ORDER BY* ili kad se u klauzuli nađe funkcija nad indeksiranom kolonom. Takođe, uzrok mogu biti i različiti indeksi u *GROUP BY* i *ORDER BY*, indeksiranje prefiksa kolone (npr. prvih 5 bajtova tipa CHAR(20)) ili *HASH* indeksi koji ne koriste sortiranje [4].

Da bi se izbegla *filesort* operacija i prilikom postojanja uslova za korišćenje indeksa, ne treba koristiti alijase za kolone (slika 29):

```
1 • use sakila;
2 • explain select country_id from country order by country_id;
```

	id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
▶	1	SIMPLE	country	<small>NULL</small>	index	<small>NULL</small>	PRIMARY	2	<small>NULL</small>	109	100.00	Using index

```
1 • use sakila;
2 • explain select country_id+1 as country_id from country order by country_id;
```

	id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
▶	1	SIMPLE	country	<small>NULL</small>	index	<small>NULL</small>	PRIMARY	2	<small>NULL</small>	109	100.00	Using index; Using filesort

Slika 29: Alijas kod *ORDER BY* optimizacije

Kolona *country_id* je primarni ključ, pa je samim tim i indeksirana i u prvom slučaju se ne vrši sortiranje rezultata. Kod drugog upita se koristi alijask s istim nazivom *country_id*, ali se sad odnosi *country_id+1*, pa se vrši sortiranje rezultata.

Za neke upite pogodnije je da se izvršava *filesort*. Ukoliko nema dovoljno memorije za rezultate, koriste se privremeni fajlovi na disku. Optimizator može na efikasan način da izvrši *filesort* bez privremenih disk fajlova za upite koji u *ORDER BY* imaju neindeksiranu kolonu i čiji je broj rezultata ograničen *LIMIT* klauzulom, što je karakteristično za veb aplikacije [4].

Da bi se ubrzalo izvršenje *ORDER BY* potrebno je koristiti indekse kad god je to moguće. U slučaju da nije, jedan od načina je povećanje vrednosti promenljive *sort_buffer_size*. Idealna vrednosti te promenljive bi bila ona koja ne zahteva upis na disk i spajanje rezultata, već je ceo rezultat moguće smestiti u bafer. Drugi način jeste povećanje broj vrsti koje se pribavljaju istovremenom postavljanjem vrednosti promenljive *read_rnd_buffer_size*.

3.1.4. Optimizacija *GROUP BY* klauzule

Najvažniji preduslovi za upotrebu indeksa za *GROUP BY* su da sve kolone iz *GROUP BY* referenciraju attribute iz istog indeksa i da indeks čuva svoje ključeve u sortiranom redosledu [4]. Da li se upotreba privremenih tabela može zameniti pristupom indeksa takođe zavisi od toga koji se delovi indeksa koriste u upitu, koji su uslovi navedeni za te delove, kao i od odabrane funkcije agregacije. Postoje dva načina za izvršavanje upita *GROUP BY* korišćenjem indeksa. Prvi način vrši grupisanje sa svim predikatima, ako ih ima, i to je

slobodan indeks (eng. *loose index*), a drugi način vrši pretragu opsega ili celog indeksa, a onda grupiše rezultate i to je čvrst indeks (eng. *tight index*).

Najefikasniji način obrade *GROUP BY* je kada se slobodan indeks koristi za direktno pribavljanje kolona za grupisanje. Koristi se svojstvo indeksa da čuva ključeve u sortiranom redosledu, što omogućava pretraživanje grupa u indeksu umesto da se uzimaju u obzir svi ključevi u indeksu koji zadovoljavaju *WHERE* uslov. Ako ne postoji klauzula *WHERE*, slobodni indeks čita onoliko ključeva koliki je broj grupa, što može biti mnogo manji broj od broja svih ključeva. Kada klauzula *WHERE* sadrži predikate opsega, slobodan indeks traži prvi ključ svake grupe koji zadovoljava uslove opsega i ponovo čita najmanji mogući broj ključeva. Da bi to bilo moguće, potrebno je da budu ispunjeni sledeći uslovi [4]:

- upit se odnosi na jednu tabelu;
- u *GROUP BY* klauzuli se nalaze samo kolone koje čine krajnji levi prefiks indeksa i nijedna druga kolona. Na primer, ako tabela *tabela1* ima indeks na (*kol1*, *kol2*, *kol3*), labavo skeniranje indeksa je primenljivo ako upit ima *GROUP BY kol1*, *kol2*. Nije primenljivo ako upit ima *GROUP BY kol2*, *kol3* (kolone nisu krajnji levi prefiks) ili *GROUP BY kol1*, *kol2*, *kol4* (*kol4* nije u indeksu);
- od agregacionih funkcija u upitu mogu samo da se jave *MIN()* i *MAX()* i odnose se na istu kolonu koja mora da bude indeksirana i da prati kolone u *GROUP BY*;
- svi delovi indeksa sem onih referenciranih u *GROUP BY* klauzuli moraju da budu deo uslova jednakosti, osim kod *MIN()* i *MAX()*;
- cele kolone iz indeksa treba da budu indeksirane, ne njihovi prefiksi.

Ako su uslovi ispunjeni, *EXPLAIN* će u u *Extra* polju da prikaže “*use index for group by*”. Na sledećem primeru prikaza je razlika između upita od kojih prvi vrši grupisanje po krajnjem levom prefiksu, a drugi po drugoj koloni iz indeksa. Shodno tome, prvi upit s izvršava optimizovano. Primeri su prikazani na slici 30.

```

1 • use sakila;
2 • set optimizer_trace="enabled=on";
3 • alter table test_loose add index col1(col1, col2);
4 • explain select col1, max(col2) from test_loose group by col1;
5 • select * from information_schema.optimizer_trace;

```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	test_loose	NULL	range	col1	col1	5	NULL	2	100.00	Using index for group-by

```

1 • use sakila;
2 • set optimizer_trace="enabled=on";
3 • explain select min(col1), col2 from test_loose group by col2;
4 • select * from information_schema.optimizer_trace;
5

```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	test_loose	NULL	index	col1	col1	10	NULL	5	100.00	Using index; Using temporary

Slika 30: Slobodan indeks kod *GROUP BY* optimizacije

Čvrsto pretraživanje indeksa podrazumeva prolazak kroz ceo indeks ili neki opseg, u zavisnosti od uslova upita. Kada uslovi za skeniranje slobodnog indeksa nisu ispunjeni, još uvek je moguće izbeći stvaranje privremenih tabela za upite *GROUP BY*. Ako u klauzuli *WHERE* postoje uslovi opsega, ovaj metod čita samo ključeve koji zadovoljavaju ove uslove. U suprotnom, vrši skeniranje indeksa. Pomoću skeniranja čvrstog indeksa, operacija grupisanja se izvodi tek nakon pronalaska svih ključeva koji zadovoljavaju uslove opsega. Da bi ovaj metod funkcionisao, dovoljno je da u upitu postoje uslovi konstantne jednakosti koji se odnose na delove ključa koji dolaze pre ili između delova ključa *GROUP BY* [4]. Konstante iz uslova jednakosti popunjavaju sve „praznine“ u ključevima za pretragu tako da je moguće formirati kompletne prefikse indeksa. Ovi indeksni prefiksi se tada mogu koristiti za pretraživanje indeksa. Ako rezultat *GROUP BY* zahteva sortiranje i moguće je formirati ključeve za pretragu koji su prefiksi indeksa, MySQL takođe izbegava dodatne operacije sortiranja, jer pretraga s prefiksima u uređenom indeksu već preuzima sve ključeve po redu. U primeru sa slike 31 kreiran je indeks nad sve tri kolone iz tabele. Nema uslova za slobodan indeks, jer se grupisanje vrši po prvoj i trećoj koloni iz indeksa, ali se u ovom slučaju koristi čvrsti indeks, jer uslov u *WHERE* klauzuli „dopunjuje“ prazninu, pa se koristi indeks *all_ind*, a ne privremena tabela.

```

1 • use sakila;
2 • set optimizer_trace="enabled=on";
3 • alter table test_loose add index all_ind(col1, col2, col3);
4 • explain select * from test_loose where col2=1 group by col1, col3;
5

```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	test_loose	NULL	index	all_ind	all_ind	15	NULL	5	20.00	Using where; Using index

Slika 31: Čvrst indeks kod *GROUP BY* optimizacije

3.1.5. Optimizacija *DISTINCT* klauzule

DISTINCT u kombinaciji sa *ORDER BY* često zahteva kreiranje privremene tabele za sortiranje vrednosti. U mnogim slučajevima, *DISTINCT* može da se posmatra kao jedan vid *GROUP BY* klauzule. Zbog toga se sve optimizacije koje se odnose na *GROUP BY* mogu primeniti i ovde.

Ukoliko se u upitu javlja više tabela, ali se ne pribavljaju kolone iz svih tabela, MySQL prestaje sa skeniranjem nekorišćenih tabela čim nađe prvo poklapanje. U sledećem slučaju, čitanje se vrši prvo iz tabele *film*, onda iz tabele *film_category*, ali se sa čitanjem druge tabele staje odmah nakon pribavljanja prve vrste. Redosled čitanja tabela vidi se pozivom *EXPLAIN* naredbe. Odgovarajući primer prikazan je na slici 32.

```

1 • use sakila;
2 • set optimizer_trace="enabled=on";
3 • explain select distinct title from film, film_category where film.film_id=film_category.film_id;

```

	id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
▶	1	SIMPLE	film_category	NULL	index	PRIMARY	fk_film_category_category	1	NULL	1000	100.00	Using index; Using tempo
	1	SIMPLE	film	NULL	eq_ref	PRIMARY,idx_title	PRIMARY	2	sakila.film_category.film_id	1	100.00	NULL

Slika 32:Distinct optimizacija

3.2. Optimizacija INSERT, UPDATE i DELETE naredbi

Tradicionalne aplikacije za obradu transakcija (eng. *Online Transactional Processing, OLTP*) i moderne veb aplikacije obično vrše brojne male operacije promene podataka, gde je paralelna obrada od velikog značaja [4]. Aplikacije za analizu podataka i izveštavanje obično izvode operacije promene podataka koje utiču na više redova odjednom, pri čemu se u raymatranje uzima korišćenje U/I za upis velike količine podataka i redovno ažuriranje indeksa. Za umetanje i ažuriranje velike količine podataka se ponekad koriste druge SQL naredbe ili spoljne naredbe koje oponašaju efekte naredbi INSERT, UPDATE i DELETE.

Kad su u pitanju *INSERT* naredbe, da bi se povećala brzina izvršenja pogodno je kombinovati veći broj manjih operacija u jednu. Vreme potrebno da se doda jedna vrsta u tabelu zavisi od vremena za konektovanje, za slanje upita na server, parsovanje upita, dodavanje vrste, dodavanje indeksa i zatvaranje konekcije. U idealnom slučaju, treba kreirati konekciju, dodati podatke za veći broj vrsta u tabelu, a odložiti dodavanje indeksa i proveru ograničenja do samog kraja. Na sporije dodavanje indeksa utiče i veličina tabela i to s faktorom $\log N$. Ako treba izvršiti dodavanje velikog broja vrsti istovremeno, bolje je izvršiti *INSERT* sa većim brojem *values* listi jer je to brže od izvršenja pojedinačnih *INSERT* naredbi. U slučaju da se vrste dodaju u tabelu koja nije prazna, za brže izvršenje moguće je konfigurisati promenljivu *bulk_insert_buffer_size*. Ova promenljiva predstavlja veličinu keša koji se koristi za *bulk insert* [4]. Ako se podaci u tabelu unose iz fajla, uvek je bolja opcija koristiti *LOAD DATA* jer je približno 20 puta brže od višestrukih *INSERT* naredbi. Takođe, prilikom dodavanja podataka u tabelu u obzir treba uzeti podrazumevane vrednosti kolona. To znači da vrednosti treba dodavati eksplicitno samo kad je vrednost koja se dodaje različita od podrazumevane vrednosti.

UPDATE upiti optimiziju se isto kao *SELECT* upiti sa *WHERE* klauzulama uz dodatni trošak (eng. *overhead*) za upis podataka. Brzina upisa zavisi od količine podataka koji se ažuriraju i broja indeksa koji se ažuriraju. Drugi način za brzo ažuriranje je odlaganje ažuriranja i kasnije izvršavanje mnogih ažuriranja u nizu. Više zajedničkih ažuriranja je mnogo brže od pojedinačnih ako se tabela zaključa [4].

Vreme potrebno za brisanje pojedinačnih redova iz *MyISAM* tabele je proporcionalno broju indeksa. Za brže brisanje, moguće je povećati veličinu keš memorije ključeva povećavanjem sistemske promenljive *key_buffer_size*. Ako se brišu svi redovi iz tabele, bolje je koristiti *TRUNCATE* od *DELETE FROM*.

4. Zaključak

Performanse baze podataka zavise od nekoliko faktora na nivou baze podataka, kao što su tabele, upiti i postavke konfiguracije. Upiti koji se izvršavaju dugo imaju negativan uticaj na performanse, pa je to vreme potrebno redukovati, tj. izvršiti optimizaciju upita. Optimizacija upita podrazumeva pronalaženje optimalnog plana za izvršenje upita. Planovi izvršenja se evaluiraju i upoređuju na osnovu cene, tj. potrebnih resursa za njihovo izvršenje. Plan sa najmanjom cenom predstavlja optimalni plan izvršenja.

MySQL je relacioni sistem za upravljanje bazama podataka. Poput drugih relacionih DBMS-ova, on ima interni, ugrađeni optimizator upita koji vrši automatsku optimizaciju. Optimizator vrši neophodne transformacije kako bi ubrzao izvršenje upita i izbegao nepotrebna čitanje celih tabela, sortiranja, kreiranja privremenih tabela i sl. Osim automatske optimizacije, na optimalno izvršenje upita može da utiče i korisnik pisanjem upita koji imaju dobre planove izvršenja. Na osnovu trasa optimizatora i razmatranih planova za izvršenje nekog upita, korisnik može da zaključi gde je potrebno dodati indekse, promeniti uslove ili neke druge transformacije da bi izvršenje upita bilo efikasnije.

5. Literatura

- [1] *Query optimization*, https://en.wikipedia.org/wiki/Query_optimization, (april 2021)
- [2] Oracle, (2021), *MySQL Optimizer*, <https://dev.mysql.com/doc/internals/en/>, (april 2021)
- [3] Francisco Claria, (2017), *Everything you need you need to know about MySQL partitions*, <https://www.vertabelo.com/blog/everything-you-need-to-know-about-mysql-partitions/>, (april 2021)
- [4] Oracle, (2021), *MySQL Reference Manual*, <https://dev.mysql.com/doc/refman/8.0/en/>, (april 2021)
- [5] Kenny Gryp, (2012), *MySQL Query Optimization*, Washington DC