UNIVERZITET U NIŠU ELEKTRONSKI FAKULTET

Mario Žalac

**SEMINARSKI RAD**

**Optimizacija upita kod PostgreSQL baze podataka**

Studijski program: Računarstvo i informatika

|  |  |
| --- | --- |
| Student: | Mentor: |
| Mario Žalac, br. ind. 1170 | Prof. dr Aleksandar Stanimirović |

Niš, april 2021. god.

Sadržaj

[1. Uvod 3](#_Toc69326925)

[2. PostgreSQL baza podataka 5](#_Toc69326926)

[3. Optimizacija performansi 7](#_Toc69326927)

[3.1. Faze izvršavanja SQL upita 7](#_Toc69326928)

[3.2. Optimizator 9](#_Toc69326929)

[3.3. Sekvencijalni čvorovi 9](#_Toc69326930)

[3.3.1 Sekvencijalno skeniranje 10](#_Toc69326931)

[3.3.2 Indeks čvorovi 10](#_Toc69326932)

[3.3.3 Join čvorovi 11](#_Toc69326933)

[3.4. Paralelno izvršenje upita 14](#_Toc69326934)

[3.4.1 Gather čvorovi 14](#_Toc69326935)

[3.4.2 Paralelno skeniranje 15](#_Toc69326936)

[3.4.3 Paralelni join 15](#_Toc69326937)

[3.4.4 Paralelna agregacija 15](#_Toc69326938)

[3.4.5 Primeri paralelnih upita 15](#_Toc69326939)

[3.5. EXPLAIN naredba 17](#_Toc69326940)

[3.5.1 EXPLAIN ANALYZE naredba 19](#_Toc69326941)

[3.5.2 Primeri korišćenja EXPLAIN naredbe 20](#_Toc69326942)

[3.6. Genetska optimizacija upita 22](#_Toc69326943)

[3.7. Generalni primeri optimizacije upita 23](#_Toc69326944)

[4. Zaključak 25](#_Toc69326945)

[5. Literatura 26](#_Toc69326946)

1. Uvod

Optimizacija upita (engl. *Query optimization*) je komponenta procesora upitnog jezika koja je neophodna kod sistema baza podataka, da bi sistem uopšte mogao da zadovolji traženu efikasnost.

Optimizator pokušava da generiše najoptimalniji plan izvršenja za SQL izraz. Optimizator bira plan sa najnižom cenom među svim razmatranim planovima. Optimizator koristi raspoložive statistike za izračunavanje troškova. Za određeni upit u datom okruženju, izračunavanje troškova uzima u obzir faktore izvršavanja upita kao što su I/O, CPU i komunikacija.

Na primer, upit može tražiti informacije o zaposlenima koji su menadžeri. Ako statistika optimizatora ukazuje da su 80% zaposlenih menadžeri, tada optimizator može odlučiti da je skeniranje kompletne tabele najefikasnije. Međutim, ako statistika ukazuje da je vrlo mali broj zaposlenih menadžera, čitanje indeksa praćenog pristupom tabeli0 od strane id-a reda može biti efikasnije od skeniranja pune tabele.

Budući da baza podataka ima na raspolaganju mnogo internih statistika i alata, optimizator je obično u boljoj poziciji od korisnika da odredi optimalan način izvršavanja izraza. Iz tog razloga, svi SQL izrazi koriste optimizator.

Optimizacija upita je ukupan proces izbora najefikasnijeg sredstva za izvršavanje SQL izraza. SQL je neproceduralan jezik, tako da optimizator može da spaja, reorganizuje i obradi upit u bilo kojem redosledu.

Baza podataka optimizuje svaki SQL izraz na osnovu statistike prikupljene o podacima kojima se pristupa. Optimizator određuje optimalan plan za SQL izraz ispitivanjem metoda višestrukog pristupa, kao što su skeniranje pune tabele ili skeniranje indeksa, različite metode spajanja kao što su ugneždene petlje i heš spajanja, različiti redosledi pridruživanja i moguće transformacije.

Za dati upit i okruženje, optimizator dodeljuje relativni numerički trošak svakom koraku mogućeg plana, a zatim te vrednosti zajedno faktoriše kako bi generisao ukupnu procenu troškova za plan. Nakon izračunavanja troškova alternativnih planova, optimizator bira plan sa najnižom procenom troškova. Iz tog razloga, optimizator se ponekad naziva i optimizator zasnovan na troškovima (engl. *cost-based optimizer*).

Moderni sistemi za upravljanje bazama podataka (engl. *DBMS*) vrše optimizaciju na fizičkom i logičkom nivou.

Obično sam korisnik ne može da pristupi direktno samom optimizatoru upita jer se on odvija automatski unutar sistema za upravljanje bazama podataka. Da bi se pisali efikasniji upiti potrebno je razumeti na koji način optimizator upita radi i koje tehnike koristi za izradu plana izvršenja upita.

U daljem radu biće ukratko sagledana PostgreSQL baza podataka i detaljno obrađena obrada i optimizacija upita, kao i tehnike za optimizovano izvršenje upita na konkretnom primeru PostgreSQL baze podataka.

1. PostgreSQL baza podataka

Online rečnik Merriam-Webster (*http://www.merriam-webster.com*) definiše bazu podataka kao veliku zbirku podataka posebno organizovanu za brzo pretraživanje i pronalaženje.

Sistem za upravljanje bazom podataka je skup biblioteka, aplikacija i uslužnih programa koji oslobađaju programera aplikacije tereta brige o detaljima skladištenja i upravljanja podacima. Takođe, sistem za upravljanje bazom podataka nudi mogućnosti pretraživanja i ažuriranja zapisa. Sistemi za upravljanje bazom podataka dolaze u velikom broju modela razvijenih tokom godina kako bi rešili određene vrste problema sa skladištenjem podataka.

PostgreSQL takođe poznat i kao Postgres je besplatan, otvorenog koda relacioni sistem za upravljanje bazama podataka (engl. RDBMS) koji koristi relacioni model za svoje baze podataka i koji podržava SQL standardni jezik upita. PostgreSQL je takođe veoma sposoban i pouzdan, i odlikuje se dobrim karakteristikama svojih performansi. Radi na bilo kojoj UNIX platformi, uključujući sisteme kao što su FreeBSD, Linux, Mac OS i Microsoft Windows.

PostgreSQL može se uporediti sa ostalim sistemima za upravljanje bazom podataka, pri čemu će biti u prednosti u odnosu na ostale sisteme. PostgreSQL sadrži gotovo sve funkcije koje se mogu naći u drugim komercijalnim ili otvorenim bazama podataka i nekoliko dodataka koji se možda neće naći nigde drugde.

PostgreSQL poseduje sledeće alate:

• Transakcije (engl. Transactions)

• Podselekti (engl. Subselects)

• Prikazi (engl. Views)

• Referentni integritet stranih ključeva (engl. Foreign key referential integrity)

• Sofisticirano zaključavanje (engl. Sophisticated locking)

• Korisnički definisani tipovi (engl. User-defined types)

• Nasleđivanje (engl. Inheritance)

• Pravila (engl. Rules)

• Kontrola paralelnosti više verzija (engl. Multiple-version concurrency control)

PostgreSQL se pokazao vrlo pouzdanim u upotrebi. Svako izdanje je veoma pažljivo kontrolisano, a beta izdanja su podvrgnuta testiranju najmanje jednom mesečno. Zahvaljujući velikoj zajednici korisnika i univerzalnom pristupu izvornom kodu, greške se mogu brzo popraviti.

Performanse PostgreSQL-a su se poboljšavale sa svakim izdanjem, a najnoviji indikatori ukazuju da se PostgreSQL, u određenim uslovima, dobro upoređuje sa komercijalnim proizvodima. Neki manje opremljeni sistemi za upravljanje bazom podataka će ga nadmašiti, čak i po cenu niže ukupne funkcionalnosti [2][3].

1. Optimizacija performansi

Podešavanje performansi jedan je od najsloženijih zadataka u svakodnevnom poslu administratora baze podataka. SQL je deklarativni jezik i zato ne definiše kako pristupiti osnovnim podacima, ta odgovornost je prepuštena mašini baze podataka. PostgreSQL, prema tome, mora za svaki iskaz odabrati najbolji dostupni pristup podacima. Određena komponenta, planer, odgovorna je za donošenje odluke o najboljoj od svih dostupnih putanja do osnovnih podataka, a druga komponenta, optimizator, je odgovorana za izvršavanje iskaza sa takvim određenim pristupnim planom.

SQL je deklarativni jezik. Dakle, prilikom zahteva od baze podataka da izvrši upit nad podacima koje sadrži ne određuje se kako bi baza podataka trebala da dovrši SQL izraz. Na primer, kada se zahteva da se vrate neki podaci, izvršiće se SELECT naredba. Njom se specificiraju samo klauzule koje određuju koji podskup podataka je potreban, a ne kako baza podataka treba da izvuče podatke iz svog trajnog skladišta. PostgreSQL ima sposoban da razume (i donekle protumači) SQL iskaze kako bi omogućio pristup podacima na najbrži način. Međutim, pronalaženje najbržeg put do podataka često zahteva ravnotežu između traženja apsolutno najbržeg puta i vremena provedenog u rasuđivanju o ovom putu; drugim rečima, PostgreSQL ponekad bira kompromis kako bi dostavio podatke na dovoljno brz način, čak i ako to nije apsolutno najbrži. Ponekad PostgreSQL ne može dobro da razume kako da pronađe najbrži put do podataka, i zato mu je potrebna pomoć administratora baze podataka. Obično se u sporom iskazu skriva pogrešno napisan upit, što znači iskaz napisan sa pogrešnim ili suprotnim klauzulama. U drugim slučajevima, spor upit se javlja zbog obrazloženja PostgreSQL-a o pogrešnoj veličini skupa podataka koji mora da obrađuje. U svim ovim slučajevima, administrator baze podataka mora da obezbedi neko podešavanje baze podataka ili izraza kako bi pomogao PostgreSQL-u da donese najbolje odluke [4][5].

* 1. Faze izvršavanja SQL upita

SQL izraz - skraćeni upit - obrađuje se u četiri glavne faze:

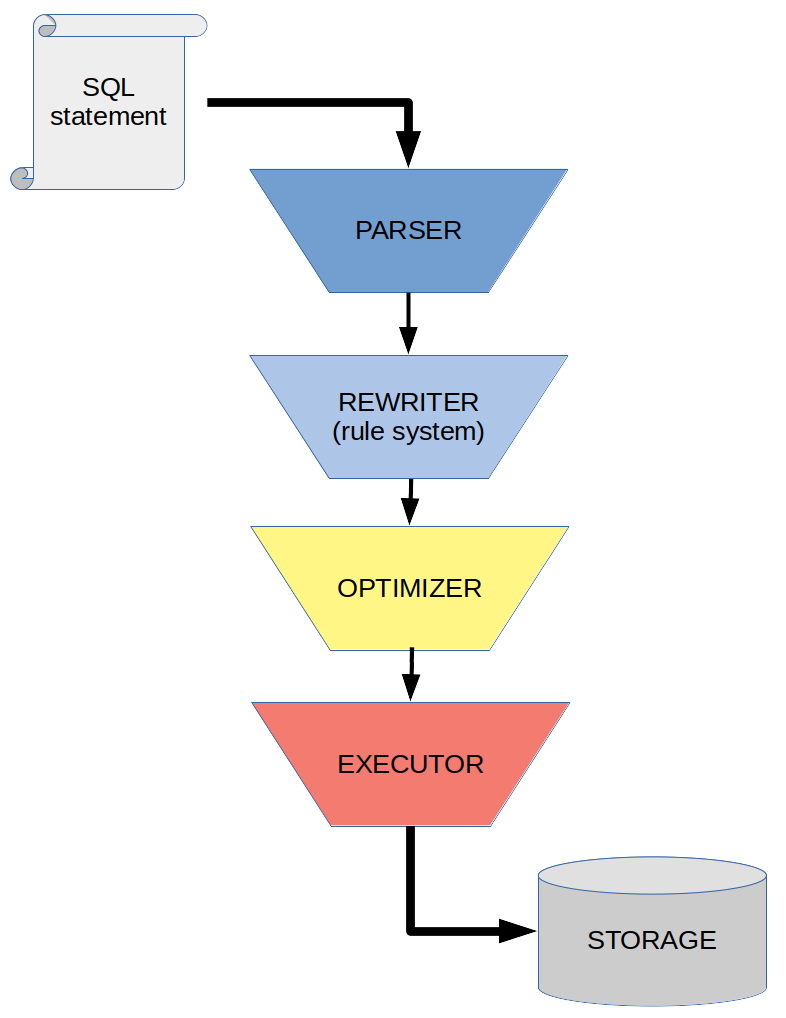
Prva faza je parsiranje; namenska komponenta, parser, obrađuje tekstualni oblik iskaza (SQL tekst) i proverava da li je tačan ili ne. Ako izraz sadrži bilo kakve sintaksne greške, izvršenje se zaustavlja u ovoj fazi. U suprotnom, parser rastavlja izjavu na njen glavni deo, na primer, spisak uključenih tabela i kolona, klauzule za filtriranje podataka, sortiranje itd.

Nakon što je parsiranje uspešno završeno, iskaz prelazi u drugu fazu: faza prepisivanja (eng. *rewriting*). Prepisivač je odgovoran za primenu svih sintaksičkih pravila za prepisivanje originalnog SQL izraza u ono što će se efikasno izvršiti. Prepisivač je posebno odgovoran za primenu pravila. Kada prepisivač završi svoj zadatak, dajući efektivni iskaz koji će baza podataka obrađivati, ovaj iskaz prelazi u fazu optimizacije.

U fazi optimizacije, upit obrađuje optimizator koji je odgovoran za pronalaženje samog najbržeg puta do podataka koji su potrebni iskazu. Pronalaženje najbržeg puta do podataka nije jednostavan zadatak: optimizator mora odlučiti kako će, između svih dostupnih metoda pristupa, poput indeksa, doći do podataka. Rasuđivanje i ponavljanje među svim dostupnim metodama pristupa troše vreme i resurse, pa zadatak optimizatora nije samo da pronađe najbrži način pristupa već i da ga pronađe u kratkom vremenu.

Napokon, kada je optimizer odlučio kako da pristupi podacima, upit ide u poslednju fazu: izvršenje. Fazom izvršenja upravlja komponenta izvršioca, koja je odgovorna za efikasan odlazak do skladišta i preuzimanje (ili umetanje) podataka pomoću metode pristupa koju je odredio izvršitelj (eng. *executor*).

Dakle, jedan SQL izraz prolazi kroz četiri faze, sve prikazane na sledećem dijagramu: faza parsiranja koja proverava sintaksu izraza, faza prepisivanja koja upit pretvara u nešto konkretnije, optimizaciona faza koja odlučuje kako pristupiti podacima koje zahteva upit, i na kraju, faza izvršenja, koja dobija fizički pristup podacima. To se može prikazati na sledećem dijagramu:



Slika 3.1 Dijagram sa fazama izvršenja SQL upita

Administrator baze podataka može se priključiti samo u fazi optimizacije, pokušavajući da pomogne PostgreSQL-u da bolje razume upit i pravilno je optimizuje kad god PostgreSQL ne radi optimalan posao [4].

* 1. Optimizator

Optimizator je komponenta odgovorna za odlučivanje šta će koristiti za pristup podacima što je brže moguće. Optimizator odlučuje da zavisi od koncepta troškova: svakom načinu pristupa podacima dodeljuje se trošak, a način koji ima najnižu cenu pobeđuje i bira se kao najbolji način pristupa. Iz tog razloga se PostgreSQL optimizator naziva optimizator zasnovan na troškovima.

PostgreSQL je konfigurisan da dodeljuje određeni trošak svakoj operaciji koju izvodi: traženje podataka iz skladišta, izvođenje neke operacije zasnovane na procesoru (na primer, sortiranje u memoriji) itd. Optimizator ponavlja sve moguće načine pristupanja podacima i rukovanja njima da bi korisniku vratio željeni rezultat, računajući ukupan iznos troškova za svaki način - to je zbir troškova svake operacije koju će PostgreSQL izvršiti. Nakon toga, plan sa najnižim troškovima se prenosi izvršitelju kao niz radnji koje treba izvršiti, a samim tim i podacima se upravlja.

Ipak, ovo je samo polovina priče. Postoje slučajevi kada je posao optimizatora zaista jednostavan: ako postoji samo način pristupa, trivijalno je odlučiti kako pristupiti podacima. Međutim, postoje iskazi koje uključuju toliko objekata i tabela da bi ponavljanje svih mogućnosti zahtevalo puno vremena, toliko vremena da će rezultat preteći vreme provedeno u računanju optimalnog načina pristupa podacima. Iz tog razloga, ako iskaz uključuje više od 12 spajanja tabela, optimizator ne ponavlja sve mogućnosti već izvršava genetski algoritam za pronalaženje kompromisnog načina za pristup podacima. Kompromis je između vremena provedenog u računanju putanje do podataka i pronalaženju ne previše lošeg pristupnog puta.

Od PostgreSQL 9.6, izvršilac takođe može da vrši pristup podacima koristeći paralelne poslove. To, na primer, znači da se preuzimanje vrlo velikog skupa podataka može izvršiti podelom količine rada između različitih paralelnih radnika (na primer, niti).

U svim slučajevima, optimizator deli skup radnji koje se prenose izvršiocu u čvorovima: čvor je akcija koju treba izvršiti kako bi se dobio konačni ili srednji rezultat. Na primer, recimo da izvršavate generički upit koji traži podatke u određenom redosledu, kao u sledećem primeru:

SELECT \* FROM categories ORDER BY description;

Optimizator će izvršitelju proslediti dve radnje, a samim tim i čvorove: jedan za preuzimanje svih podataka i jedan za sortiranje podataka [4][5].

* 1. Sekvencijalni čvorovi

Postoje različiti čvorovi za svaku operaciju koja se može izvršiti i za svaku različitu metodu pristupa koju prihvata PostgreSQL.

Važno je napomenuti da se čvorovi mogu složiti: izlaz čvora može se koristiti kao ulaz u drugi čvor. To omogućava izgradnju vrlo složenih planova izvršenja koje su napravili različiti čvorovi, a koji mogu proizvesti metod finog pristupa podacima.

Sekvencijalni čvorovi su oni čvorovi koji će se izvršavati sekvencijalno, jedan za drugim, kako bi se postigao konačni rezultat. Ovde su navedeni glavni čvorovi koji će biti objašnjeni u nastavku:

* Sekvencijalno skeniranje (engl. Sequential Scan)
* Indeks skeniranje (engl. Index Scan), skeniranje samo indeksom (engl. Index Only Scan) i skeniranje indeksa bitmape (engl. Bitmap Index Scan)
* Ugnježdena petlja (engl. Nested Loop), Hash pridruživanje (engl. Hash Join) i spajanje pridruživanja (engl. Merge Join)
* Paralelni čvorovi okupljanja i spajanja (engl. Gather and Merge parallel nodes)
  + 1. Sekvencijalno skeniranje

Sekvencijalno skeniranje je jedini čvor koji je uvek dostupan optimizatoru i izvršiocu, posebno kada ne postoji druga vredna alternativa. U sekvencijalnom skeniranju, izvršilac će ići na početak skupa podataka na disku - na primer, početak datoteke koja odgovara tabeli - i čitaće sve podatke jedan po jedan blok u sekvencijalnom redosledu.

Ovaj čvor se često koristi kada tražite sadržaj tabele bez ikakve posebne klauzule za filtriranje, kao u sledećem primeru:

SELECT \* FROM categories;

* + 1. Indeks čvorovi

Skeniranje indeksa ima pristup podacima koji uključuju indeks kako bi se brzo pronašao traženi skup podataka. U PostgreSQL-u su svi indeksi sekundarni, što znači da žive uz tabelu; stoga se u skladištu nalazi datoteka podataka za tabelu i po jedna za svaki indeks koji se gradi na tabeli. To znači da indeksno skeniranje uvek zahteva dva različita pristupa skladištu: jedan za čitanje diska i izdvajanje informacija o tome gde se u tabeli nalaze traženi tuples-i, i drugi za pristup disku radi traženja tuples-a na koje ukazuje indeks.

Iz ovoga bi trebalo da bude jasno da PostgreSQL izbegava upotrebu indeksa kada nisu korisni, a to je kada prethodno pomenuti pristup dvostrukom skladištu ima više nedostataka nego prednosti. Međutim, kada PostgreSQL veruje da bi pristup podacima putem indeksa mogao biti dragocen, proizvešće indeksni čvor koji se može specijalizovati za tri različita tipa.

Indeks skeniranje je, kao što i samo ime kaže, „klasični“ metod pristupa indeksu: PostgreSQL čita izabrani indeks i odatle kreće u potragu za tuples-ima, čitajući ponovo iz skladišta.

Skeniranje samo indeksom je posebna varijanta indeksnog skeniranja. Ako zahtevani podaci uključuju samo kolone koje pripadaju indeksu, PostgreSQL je dovoljno pametan da izbegne drugo putovanje u skladište jer može izvući sve potrebne informacije direktno iz indeksa.

Poslednji tip čvora zasnovan na indeksu je skeniranje indeksa bitmape: PostgreSQL gradi memorijsku bitmapu gde su tuples koji zadovoljavaju klauzule izraza, a zatim se ova bitmapa koristi za dolazak iz skladišta u tuples. Skeniranje indeksa bitmapa obično je povezano sa skeniranjem bitmap heap-a.

* + 1. Join čvorovi

Kada PostgreSQL izvrši spajanje dve (ili više) tabela, koristi jedan od tri moguća čvora. U nastavku sledi opis čvorova za spajanje, uzimajući u obzir spajanje dve tabele: spoljne tabele (levo od spajanja) i unutrašnje (tabele na desnoj strani spajanja).

Najjednostavniji čvor za razumevanje je ugnježđena petlja (engl. Nested Loop): obe tabele se skeniraju sekvencijalnom ili indeksiranom metodom i svaki tuples se proverava da li postoji podudaranje. U osnovi, algoritam se može opisati sledećim delom pseudo-Java koda:

for ( Tuple o : outerTable )

for ( Tuple i : innerTable )

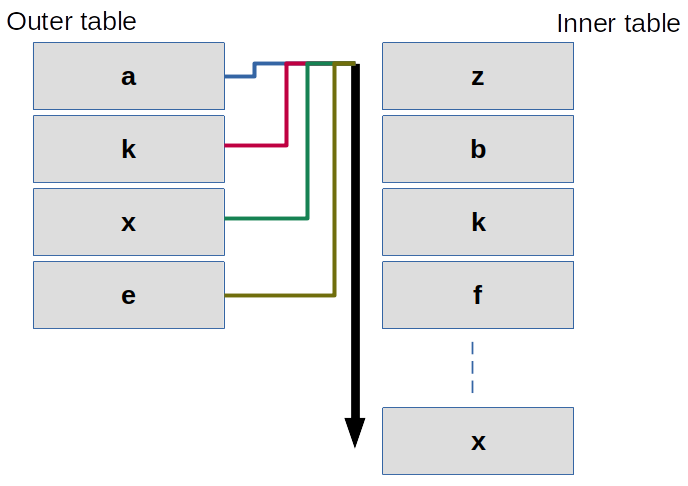
if ( o.matches( i ) )

appendTupleToResultDataSet( o, i );

Kao što vidite iz prethodnog pseudo-koda, ugnježđena petlja je imenovana po ugnežđavanju petlji koje izvodi kako bi se procenio svaki tuple između unutrašnje i spoljašnje tabele.

U svakom slučaju, ugnežđena petlja nije primorana da vrši sekvencijalno skeniranje obe tabele, i zapravo, u zavisnosti od konteksta, kroz svaku tabelu se može proći sekvencijalnim ili indeksiranim pristupnim metodom. Međutim, jezgro ugnežđene petlje se ne menja: uvek će biti ugnežđena dvostruka petlja za traženje podudaranja među tuples-ima.

Sledeći dijagram prikazuje ponašanje spoja ugnežđene petlje:



Slika 3.2 Dijagram Nested Loop join-a

PostgreSQL bira ugniježđenu petlju samo ako je unutrašnja tabela dovoljno mala, tako da petlja svaki put prema njoj ne uvodi nikakve posebne penale.

Drugi način izvođenja spajanja je upotreba čvora Hash Join: unutrašnja tabela je preslikana u heš, koji je skup bucket-a koje sadrže tuples-e tabele. Zatim se prolazi kroz spoljašnju tabelu i za svaki tuple izvučen iz spoljne tabele pretražuje se heš da bi se videlo da li postoji podudaranje. Sledeći deo pseudo-Java koda ilustruje mehaniku Hash Join-a:

Hash innerHash = buildHash( innerTable );

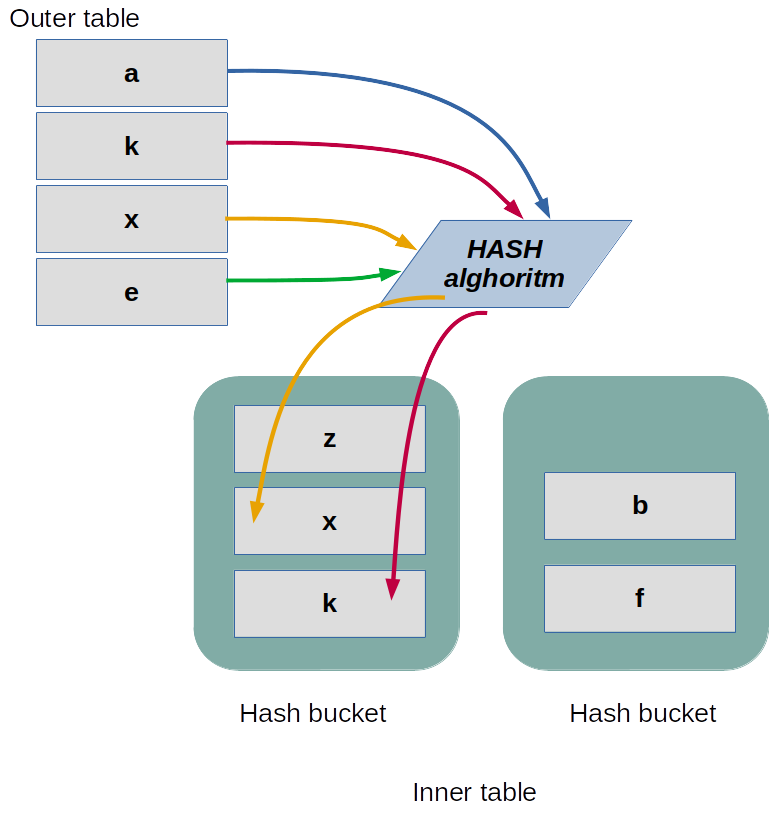
for ( Tuple o : outerTable )

if ( innerHash.containsKey( buildHash( o ) ) )

appendTupleToResultDataSet( o, i );

Kao što možete videti iz prethodnog primera, prvi korak uključuje posedovanje unutrašnje tabele, a zatim prolazak kroz spoljašnju tabelu kako bi se videlo da li se bilo koji od njenih tuples-a može podudarati sa vrednostima na heš mapi unutrašnje tabele.

Sledeći dijagram prikazuje algoritam Hash Join:



Slika 3.3 Dijagram Hash join-a

Poslednji tip spajanja koji postoji kod PostgreSQL-a je Merge Join. Kao što i samo ime govori, Merge Join uključuje korak sortiranja. Obe tabele se prvo sortiraju po join ključevima, a zatim se kroz njih prolazi sekvencijalno. Za svaku tuple spoljne tabele izdvajaju se svi tuples-ovi koji se podudaraju u unutrašnjoj tabeli. Budući da su obe tabele sortirane, nepodudarajući se tuple pokazuje da je vreme da pređete na sledeći join ključ.

Sledeći pseudo-Java kod ilustruje Merge Join:

outerTable = sort( outerTable );

innerTable = sort( innerTable );

int innerIdx = 0;

for ( Tuple o : outerTable )

for ( ; innerIdx < innerTable.length(); innerIdx++ ){

Tuple i = innerTable[ innerIdx ];

if ( o == i )

appendTupleToResultSet( o, i );

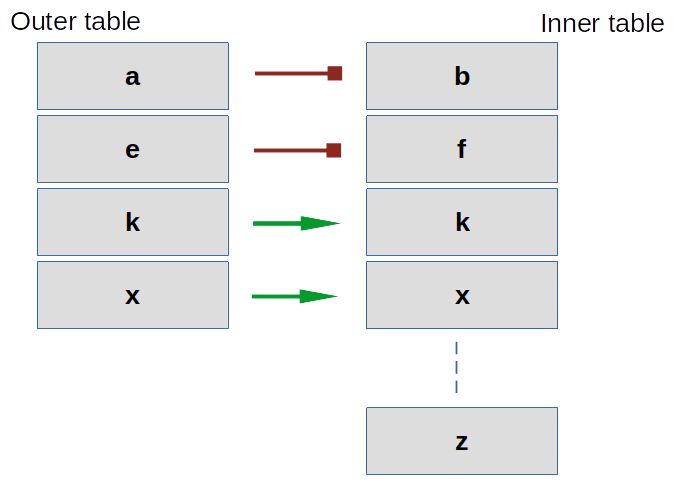
else

break;

}

Kada se tabele sortiraju, iz spoljne tabele se izvlači tuple koja se upoređuje sa svim tuples-ima unutar unutrašnje tabele. Čim se tuples-i više ne podudaraju, izvlači se još jedan tuple iz spoljne tabele i unutrašnja tabela ponovo pokreće svoju petlju sa prethodne pozicije. Drugim rečima, obe tabele se prelaze tačno jednom i samo jednom.

Sledeći dijagram prikazuje Merge Join, gde su obe tabele prikazane nakon koraka sortiranja:



Slika 3.4 Dijagram Merge Join-a

* 1. Paralelno izvršenje upita

Paralelni čvorovi su oni čvorovi koje PostgreSQL može izvršiti i koji distribuiraju količinu posla među paralelnim procesima, pa brže dolaze do konačnog rezultata. Važno je napomenuti da paralelno izvršavanje nije uvek pravi izbor: postoji određeno vreme za distribuciju posla među paralelnim procesima, kao i vreme i resursi potrebni za vraćanje rezultata svakog pojedinačnog procesa. Iz tog razloga, PostgreSQL omogućava paralelno izvršavanje određenih čvorova samo ako će procenjena paralelna verzija pružiti prednost nad sekvencijalnim izvršenjem.

Jednostavan primer je mala tabela koju čini samo nekoliko tuples-a, kao što su četiri. Ako vam je potreban sav sadržaj tabele, resursi i vreme provedeno u pokretanju i sinhronizaciji paralelnih procesa biće mnogo veći od odlaska direktno u tabelu i sekvencijalnog vraćanja skupa rezultata. Osnovno pravilo je: ako je traženi skup podataka dovoljno mali, PostgreSQL nikada neće odabrati paralelno izvršavanje.

Važno je shvatiti činjenicu da samo zato što planer izrađuje paralelni plan, koji je plan izvršenja napravljen od paralelnih čvorova, on to čini ne znači da će izvršilac slediti ovaj paralelizam. Mogli bi postojati uslovi, naročito tokom izvođenja, koji sprečavaju PostgreSQL u izvršavanju paralelnog plana, čak i ako bi to bio optimalan izbor (na primer, PostgreSQL nema dovoljno prostora da omogući potrebnu količinu paralelnih procesa).

U nastavku će biti ukratko predstavljeni najvažniji paralelni čvorovi [4][5].

* + 1. Gather čvorovi

Paralelni plan izvršenja uvek uključuje dve vrste Gather čvorova: običan Gather čvor i Gather Merge čvor.

Gather čvorovi su odgovorni za prikupljanje rezultata iz paralelnih izvršnih čvorova, njihovo okupljanje da bi se dobio konačni rezultat. Razlika je u tome što čvor Gather Merge zahteva paralelne procese da bi mu pružio sortirani izlaz, tako da se sastavljanje niza rezultata vrši po redosledu podataka.

Obični čvor Gather ne zahteva sortiranje rezultata serije, tako da on jednostavno sastavlja sve delove kako bi pružio konačni rezultat.

* + 1. Paralelno skeniranje

Svi glavni čvorovi koji se mogu naći u metodu sekvencijalnog pristupa mogu se napraviti paralelno.

U paralelnom skeniranju sekvence (engl. Parallel Seq), sve stranice podataka tabele podeljene su na sve dostupne paralelne procese, tako da svaki proces prolazi kroz manji deo tabele na sekvencijalni način. U skeniranju paralelnog indeksa (engl. Parallel Index scan) i paralelnom skeniranju samo indeksa (engl. Parallel Index Only scan) blokovi indeksa se dodeljuju svakom paralelnom procesu tako da svaki proces prolazi kroz podskup indeksa i, prema tome kroz tabelu.

* + 1. Paralelni join

Kada PostgreSQL odluči da koristi metodu paralelnog spajanja (engl. Parallel join), pokušava da zadrži pristup unutrašnjoj tabeli na neparalelni način (pod pretpostavkom da je takva tabela dovoljno mala) i vrši paralelni pristup spoljnoj tabeli koristeći jedan od čvorova.

Međutim, u slučaju Hash Join-a, unutrašnja tabela se izračunava kao heš pomoću svakog paralelnog procesa, što stoga zahteva da svaki paralelni proces koji radi na spoljnoj tabeli izračuna iste rezultate za unutrašnju tabelu. Iz tog razloga postoji i Parallel Hash Join, koji omogućava paralelno izračunavanje heš mape unutrašnje tabele u svakom procesu koji radi na spoljnoj tabeli.

* + 1. Paralelna agregacija

Kada se konačni rezultat dobije agregacijom različitih paralelnih podupita, mora postojati paralelna agregacija, što je agregacija svakog pojedinačnog paralelnog dela.

Ova agregacija se događa u različitim koracima: prvo, postoji Partial Aggregate čvor, izvršen svakim paralelnim procesom koji daje delimični skup rezultata. Nakon toga, Gather čvor (ili Gather Merge) prikuplja sve delimične rezultate i prosleđuje ceo skup čvoru Finalize Aggregate, koji sekvencijalno sastavlja konačni rezultat.

* + 1. Primeri paralelnih upita

Počevši od verzije 9.6, PostgreSQL podržava paralelne upite. Ova podrška za paralelizam se postepeno poboljšavala tokom vremena, a verzija 11 je dodala još više funkcionalnosti ovoj važnoj funkciji.

U nastavku ćemo pogledati neke primere paralelizma i kako on funkcioniše u praksi:

test=# explain SELECT \* FROM t\_parallel;

QUERY PLAN

----------------------------------------------------------------------

Seq Scan on t\_parallel (cost=0.00..360621.20 rows=25000120 width=4)

(1 row)

Ovde PostgreSQL ne koristi paralelni upit iako je on dostupan. Razlog je cena međuprocesne komunikacije. Ako PostgreSQL mora da šalje redove između procesa, upit zapravo može biti sporiji nego u režimu pojedinačnog procesa. Optimizator koristi parametre troškova za kažnjavanje međuprocesne komunikacije:

#parallel\_tuple\_cost = 0.1

Svaki put kada se tuple premešta između procesa, izračunavanju će se dodati 0,1 poena. Sledeći primer ilustruje ponašanje PostgreSQL-a u slučaju kada je prisiljen da izvršava paralelne upite:

test=# SET force\_parallel\_mode TO on;

SET

test=# explain SELECT \* FROM t\_parallel;

QUERY PLAN

----------------------------------------------------------------------------

Gather (cost=1000.00..2861633.20 rows=25000120 width=4)

Workers Planned: 1

Single Copy: true

-> Seq Scan on t\_parallel (cost=0.00..360621.20 rows=25000120 width=4)

(4 rows)

Kao što vidite, troškovi su veći nego u jednojezgrenom režimu. U praksi ovo je važno pitanje, jer se postavlja pitanje zašto PostgreSQL koristi jedno jezgro. U stvarnom primeru, takođe je važno videti da više jezgara ne dovodi automatski do veće brzine. Za pronalaženje savršenog broja jezgara potreban je delikatan postupak uravnoteženja.

PostgreSQL paralelni plan neće ni uzeti u odabir ako je očekivana veličina skupa rezultata premala. Konkretno, ako tabela za koju se traže podaci ima dimenziju nižu od parametra min\_parallel\_table\_scan\_size (podrazumevano je 8 MB) ili je indeks kroz koji treba proći manji od min\_parallel\_indek\_scan\_size (podrazumevano 512 kB), PostgreSQL neće uzeti u obzir paralelnu obradu.

Kao što smo videli u prethodnom primeru PostgreSQL je moguće prisiliti da izvrši paralelni plan, čak i ako prethodne vrednosti nisu zadovoljene, dodatnim parametrom konfiguracije - force\_parallel\_mode - koji je podrazumevano isključen.

U svakom slučaju, kada PostgreSQL razmatra paralelni plan kao opciju, ona nije podrazumeva. Radije pažljivo procenjuje troškove sekvencijalnog plana i troškove paralelnog plana kako bi se utvrdilo da li je i dalje vredno dodatnog napora oko postavljanja.

Međutim, postoje i druga ograničenja za primena paralelnog plana: PostgreSQL mora da obezbedi da ne stvara previše paralelnih procesa, pa ako na sistemu već radi previše paralelnih procesa, paralelno izvršavanje neće biti uzeto u razmatranje. Štaviše, bilo koji iskaz koji proizvodi upisivanje podataka - odnosno bilo šta različito od SELECT izjave - neće biti valjani kandidat za paralelni plan, kao ni bilo koji iskaz koji se može suspendovati i nastaviti, kao što je upotreba kursora.

Na kraju, bilo koji upit koji uključuje pozivanje funkcije označene kao PARALLEL UNSAFE neće biti kandidat za paralelni plan [4][5].

* 1. EXPLAIN naredba

EXPLAIN je naredba koja prikazuje kako će PostgreSQL izvršiti određeni upit. Prosleđivanjem upita EXPLAIN naredbi prikazaće se plan izvršenja.

Postoji nekoliko važnih stvari koje treba znati pre korišćenja EXPLAIN naredbe:

* Prikazaće se samo najbolji plan, onaj sa najnižom cenom među razmatranim planovima.
* Neće izvršiti upit za koji se traži plan, stoga je EXPLAIN izvršenje brzo i prilično konstantno svaki put.
* Prikazaće se svi čvorovi izvršenja koje će izvršitelj koristiti da pruži skup podataka.

Sledi primer EXPLAIN naredbe na delu radi boljeg razumevanja. Zamislite da moramo da razumemo plan izvršenja izjave SELECT \* FROM categories. U ovom slučaju, naredbi treba da prethodi naredba EXPLAIN, kako sledi:

forumdb=> EXPLAIN SELECT \* FROM categories;

QUERY PLAN

-----------------------------------------------------------

Seq Scan on categories (cost=0.00..1.01 rows=1 width=68)

(1 row)

Kao što se vidi, izlaz EXPLAIN naredbe izveštava o planu upita. Postoji jedan izvršni čvor, tipa Seq Scan, praćen tabelom prema kojoj se čvor izvršava (na kategorijama).

Za svaki čvor EXPLAIN naredba će izvesti neke dodatne informacije između zagrada: trošak, broj redova i širina. Trošak predstavlja količinu napora potrebnog za izvršavanje čvora i uvek se izražava kao „početni trošak“ i „konačni trošak“. Trošak pokretanja je koliko posla PostgreSQL mora da uradi pre nego što počne da izvršava čvor; u prethodnom primeru trošak je 0, što znači da izvršenje čvora može početi odmah. Konačni trošak je koliki napor PostgreSQL mora da učini da obezbedi poslednji bit skupa podataka - odnosno da dovrši izvršenje čvora.

Polje redova označava koliko tuples-a čvor treba da pruži u konačnom skupu podataka i on je čista procena. Kao procena, vrednost nikada neće biti tačna i treba imati na umu da nikada neće biti nula: kada PostgreSQL proceni vrlo mali broj tuples-a, uvek daje 1 kao broj redova.

Na kraju, polje širine ukazuje koliko bitova će zauzeti svaki tuple, u proseku. U osnovi, ove informacije se koriste za procenu mrežnog saobraćaja koji će upit proizvesti: u prethodnom primeru je moguće proceniti 68 bajtova po tuple-u.

Sledi još jedan primer korišćenja EXPLAIN naredbe:

forumdb=> EXPLAIN

SELECT title

FROM categories ORDER BY description DESC;

QUERY PLAN

-----------------------------------------------------------------

Sort (cost=1.02..1.02 rows=1 width=64)

Sort Key: description DESC

-> Seq Scan on categories (cost=0.00..1.01 rows=1 width=64)

(3 rows)

Ovde imamo dva različita čvora: prvi na vrhu je čvor Sort (zbog klauzule ORDER BY), a drugi čvor je Seq Scan, kao u prethodnom primeru. Prvi red u izlistanom planu je uvek čvor, a ostali čvorovi reda uvučeni su udesno i imaju strelicu kao prefiks (->). Ostale linije u planu pružaju informacije o čvoru pod kojim se nalaze; stoga je u prethodnom primeru red Sort Key dodatna informacija čvoru Sort.

Drugi pristup razlikovanju redova čvorova od dodatnih informacija je uzet u obzir da svaka linija čvora ima u zagradama atribute cena, redovi i širina.

Seq Scan je prvi izvršen čvor. Ovaj čvor radi isto što je objašnjeno u prethodnom primeru: primorava izvršitelja da ode do tabele u fizičkom skladištu i preuzme, po sekvencijalnom redosledu, sav sadržaj tabele. Jedna stvar se, međutim, razlikuje u prethodnom primeru: prosečna širina se smanjila, a to je zbog činjenice da upit ne zahteva sve kolone svakog tuple-a, već samo naslovnu.

Kada se čvor sekvencijalnog skeniranja završi, njegov izlaz se koristi kao ulaz za čvor Sort, koji izvodi željenu operaciju ORDER BY. Kao što se vidi, sortiranje originalne izjave je prikazano kako bi imali uvid na šta će izvršilac sortirati podatke. Čvor Sort ima početni trošak koji je veći kao konačni trošak prethodnog čvora: sekvencijalno skeniranje ima konačni trošak 1,01, a sortiranje počinje s troškovima 1,02. Ovo ponovo naglašava kako se čvorovi izvršavaju u kontinuitetu, a takođe vam govori da sortiranje ne može započeti pre nego što se završi drugi čvor. Čvor sortiranja ima konačni trošak jednak početnom trošku, što znači da je ovaj čvor jednostavan za izvršavanje PostgreSQL-u [3][4].

* + 1. EXPLAIN ANALYZE naredba

Režim ANALYZE mod EXPLAIN naredbe poboljšava komandu efikasnim pokretanjem upita za objašnjenje. Prema tome, naredba radi dvostruki zadatak: ispisuje najbolji plan za izvršavanje upita i pokreće upit, takođe izveštavajući neke statističke podatke.

U sledećem kodu ćemo uporediti izlaz EKSPLAIN ANALIZE naredbe u poređenju sa izlazom rezultata obične EKSPLAIN naredbe:

forumdb=> EXPLAIN SELECT \* FROM categories;

QUERY PLAN

-----------------------------------------------------------

Seq Scan on categories (cost=0.00..1.01 rows=1 width=68)

(1 row)

forumdb=> EXPLAIN ANALYZE SELECT \* FROM categories;

QUERY PLAN

-----------------------------------------------------------------------------------------------------

Seq Scan on categories (cost=0.00..1.01 rows=1 width=68) (actual time=0.023..0.025 rows=1 loops=1)

Planning Time: 0.102 ms

Execution Time: 0.062 ms

(3 rows)

Izlaz naredbe EXPLAIN ANALYZE pojačan je „stvarnim“ delom svakog čvora: izvršitelj izveštava kako je tačno teklo izvršenje čvora. Prema tome, dok EXPLAIN može proceniti samo troškove čvora, EXPLAIN ANALYZE pruža povratne informacije o vremenu izvršavanja (izraženom u milisekundama), efektivnom broju redova i koliko je puta čvor izvršen (petlje).

Vreme čvora se izražava, slično trošku, u vremenu pokretanja i završnom vremenu, koje je potrebno vremenu da čvor dovrši svoje izvršavanje. Prema tome, u prethodnom primeru, PostgreSQL-u je trebalo 0,023 milisekunde da se „zagreje“ i dovršio je izvršavanje upita za 0,025 milisekundi, tako da je čvoru trebalo 0,002 milisekunde da dovrši svoj posao.

Na samom kraju izlaza komande, EXPLAIN ANALYZE pruža ukupno informacije o vremenu, koje uključuju vreme planiranja, odnosno vreme koje je optimizator proveo izrađujući najbolji plan pristupa kandidatu i vreme izvršavanja, što je ukupno vreme provedeno u izvršavanju upita (isključujući vreme raščlanjivanja i planiranja). prethodnom primeru trebalo je 0,062 milisekunde da „preuzme podatke“ i 0,102 milisekunde pri odlučivanju o načinu preuzimanja podataka, pa je tako trajalo 0,164 milisekunde.

* + 1. Primeri korišćenja EXPLAIN naredbe

U ovom delu biće izloženi neki osnovni koncepti svakodnevne upotrebe EXPLAIN naredbe kao moćnog alata za određivanje gde i kako da se PostgreSQL koristi u radu za brži pristup podacima. Naravno, podešavanje upita je vrlo složena tema i često zahteva stalnu optimizaciju zasnovanu na istraživanjima.

Ponekad podešavanje upita uključuje jednostavno prepisivanje na način koji je ugodniji - ili bolji, razumljiviji - u PostgreSQL-u, ali najčešće to znači korišćenje odgovarajućeg indeksa za ubrzavanje pristupa osnovnim podacima.

U narednom primeru želimo da izdvojimo sve postove poređane prema danu kreiranja, pa je upit sledeći:

SELECT \* FROM posts ORDER BY created\_on;

Propuštanjem kroz EXPLAIN naredbu dobijamo sledeći rezultat:

forumdb=> EXPLAIN SELECT \* FROM posts ORDER BY created\_on;

QUERY PLAN

-----------------------------------------------------------------------

Sort (cost=1009871.99..1022372.04 rows=5000000 width=55)

Sort Key: created\_on

-> Seq Scan on posts (cost=0.00..111729.20 rows=5000000 width=55)

(3 rows)

Kao što se može viditi, prvi čvor koji treba izvršiti je sekvencijalno skeniranje (početni trošak je 0), koji će proizvesti 5.000.000 tuples-a kao izlaz. Sekvencijalno skeniranje se koristi zato što ne postoji klauzula za filtriranje - želimo da preuzmemo sve podatke uskladištene u tabeli - i drugo, u tabeli ne postoji pristupni metod (nema indeksa). Budući da smo tražili da sortiramo izlaz, sledeće čvor za izvršavanje je čvor za sortiranje koji kao rezultat daje isti broj tuples-a.

Koliko tačno treba da bi se izvršio ovaj upit najbolje se može prikazati naredbom EXPLAIN ANALYZE:

forumdb=> EXPLAIN ANALYZE SELECT \* FROM posts ORDER BY created\_on;

QUERY PLAN

------------------------------------------------------------------------------------------------------------

Sort (cost=1009871.99..1022372.04 rows=5000020 width=55) (actual time=31321.217..32806.986 rows=5000000 loops=1)

Sort Key: created\_on

Sort Method: external merge Disk: 367024kB

-> Seq Scan on posts (cost=0.00..111729.20 rows=5000020 width=55) (actual time=0.078..2546.672 rows=5000000 loops=1)

Planning Time: 0.148 ms

Execution Time: 33138.966 ms

Vreme izvršenja je 33 sekunde. Moguće je redukovati ukupno vreme izvršenja dodavanjem indeksa created\_on polju:

forumdb=> CREATE INDEX idx\_posts\_date ON posts( created\_on );

CREATE INDEX

forumdb=> EXPLAIN ANALYZE SELECT \* FROM posts ORDER BY created\_on;

QUERY PLAN

------------------------------------------------------------------------------------------------------------

Index Scan using idx\_posts\_date on posts (cost=0.43..376844.33 rows=5000000 width=55) (actual time=0.768..8694.662 rows=5000000 loops=1)

Planning Time: 6.357 ms

Execution Time: 9128.668 ms

(3 rows)

Upit se sada izvodi u gotovo trećini vremena potrebnog bez indeksa, a zapravo se plan upita promenio iz sekvencijalnog u indeksno skeniranje sa novostvorenim novim indeksom.

Naravno, novostvoreni indeks ima negativnu stranu u pogledu prostora za skladištenje. Povećanje brzine dolazi sa dodatnim troškovima prostora koji se mogu proveriti na sledeći način:

forumdb=> SELECT pg\_size\_pretty( pg\_relation\_size( 'posts' ) ) AS table\_size,

pg\_size\_pretty( pg\_relation\_size( 'idx\_posts\_date' ) ) AS index\_size;

table\_size | index\_size

------------+------------

482 MB | 107 MB

(1 row)

Indeks zahteva oko 22% prostora na disku koji zauzimaju podaci cele tabele.

Da li je ovaj dodatni prostor na disku previše ili ne, zavisi od resursa i konačnog cilja: u prethodnom slučaju, pod pretpostavkom da se upit izvršava često, povećana brzina opravdana je dodatnim zauzimanjem prostora [3][5]..

* 1. Genetska optimizacija upita

Rezultat procesa planiranja je ključan za postizanje superiornih performansi. Planiranje nije daleko od jednostavnog i uključuje različite složene proračune. Što se više tabela dodirne upitom, planiranje će postati složenije. Što više tabela ima, to će veći izbor imati planer. Logično, vreme planiranja će se povećati. U jednom trenutku planiranje će potrajati toliko dugo da izvođenje klasične iscrpne pretrage više nije izvodljivo. Povrh svega, greške koje se javljaju tokom planiranja su toliko velike da pronalaženje teorijski najboljeg plana ne mora nužno dovesti do najboljeg plana u pogledu vremena izvođenja.

U takvim slučajevima u pomoć može doći genetska optimizacija upita (engl. Genetic Query Optimization - GEQO). Genetska optimizacija upita crpi ideju i inspiraciju iz prirode i podseća na prirodni proces evolucije.

Prvo, genetski optimizator će generisati nasumični skup planova. Ti planovi se zatim pregledaju. Loši se odbacuju i generišu novi na osnovu gena dobrih. Na ovaj način se generišu potencijalno još bolji planovi. Ovaj postupak se može ponavljati koliko god često želite. Na kraju dana, ostaje nam plan za koji se očekuje da će biti mnogo bolji od pukog korišćenja slučajnog plana. GEQO se može uključiti i isključiti podešavanjem geqo promenljive, kao što je prikazano u sledećim redovima koda:

test=# SHOW geqo;

geqo

------

on

(1 row)

test=# SET geqo TO off;

SET

Spisak pokazuje kako se GEQO može uključiti i isključiti. Podrazumevano, geqo promenljiva se uključuje ako izraz prelazi određeni nivo složenosti, koji kontroliše sledeća promenljiva:

test=# SHOW geqo\_threshold ;

geqo\_threshold

----------------

12

(1 row)

Ako su upiti toliko veliki da se dostiže ovaj prag, svakako ima smisla igrati se sa ovom postavkom da biste videli kako planer menja plan ako promenite te promenljive.

Međutim, nepisano pravilo je da se izbegne GEQO sve dok postoji mogućnost i pokuša prvo sa popravkom stvari u upitu [4].

* 1. Generalni primeri optimizacije upita

U ovom delu biće prezentovani neki generalni primeri optimizacije upita koji se mogu primeniti i na PostgreSQL bazu podataka. Oni predstavljaju neke primere dobre prakse koje administratori baze podataka mogu da primenjuju i tako izbegnu lose performanse.

Upotreba imena kolona umesto \* u SELECT iskazu. Ukoliko želimo da prikažemo samo određene kolone iz tabele poželjno je ne koristiti SELECT \* i ako je ovu komandu lakše napisati. Štedi se na obimu prikazanih podataka, smanjuje se mrežni saobraćaj i ukupna brzina pronalaženja podataka. Sledi primer ove vrste upita:

SELECT \* FROM SH.Sales;

Bolje rešenje u ovom slučaju bi bilo sledeće:

SELECT s.prod\_id FROM SH.sales s;

U konkretnom slučaju ušteda u vremenu se kreće i do 27 procenata.

Izbegavanje DISTINCT izraza posebno u situacijama kada je rezlutat isti sa i bez DISTINCT ključne reči a to se dešava kada u rezultatu imamo kolonu koja je primarni ključ. Primer ovakvog upita je sledi:

SELECT DISTINCT \* FROM SH.sales s

JOIN SH.customers c ON s.cust\_id= c.cust\_id

WHERE c.cust\_marital\_status = 'single';

Bolje rešenje u ovom slučaju bi bilo sledeće:

SELECT \* FROM SH.sales s

JOIN SH.customers c ON s.cust\_id = c.cust\_id

WHERE c.cust\_marital\_status='single';

U konkretnom slučaju ušteda u vremenu se kreće čak do 85 procenata u zavisnosti od broja redova.

Korišćenje JOIN iskaza umesto ugnježdenih upita je efikasnije u pogledu izvršenja:

SELECT \* FROM SH.products p

WHERE p.prod\_id = (SELECT s.prod\_id FROM SH.sales s WHERE s.cust\_id = 100996

AND s.quantity\_sold = 1 )

Bolje rešenje u konkretnom slučaju bi bilo sledeće:

SELECT p.\* FROM SH.products p, sales s

WHERE p.prod\_id = s.prod\_id AND s.cust\_id = 100996 AND s.quantity\_sold = 1;

U ovom slučaju efikasnost raste sa povećanjem broja redova i do 61 odsto.

Koristiti UNION ALL umesto UNION. UNION ALL se brže izvršava jer ne traži duplikate za razliku od UNION koji traži duplikate bez obzira da li postoje ili ne postoje:

SELECT cust\_id FROM SH.sales UNION SELECT cust\_id FROM customers;

Bolje rešenje u konkretnom slučaju bi bilo sledeće:

SELECT cust\_id FROM SH.sales UNION ALL SELECT cust\_id FROM customers;

U ovom slučaju efikasnost raste sa povećanjem broja redova i do 81 odsto.

Ovo su bili neki primeri optimizacije upita koji se generalno mogu primeniti, pa se tako I na PostgreSQL bazi podataka mogu implementirati[6].

1. Zaključak

Optimizacija upita se upotrebljava u većini baza podataka, jer danas važnu ulogu u informacionim tehnologijama igra brzina obrade podataka.

PostgreSQL pruža veoma bogate mogućnosti za optimizaciju upita. Zahvaljujući naredbi EXPLAIN, administrator baze podataka može pregledati spor upit i vidite kako je optimizator razmišljao o tome koji je najbolji pristup osnovnim podacima i zahvaljujući razumevanju kako PostgreSQL radi, administrator može odlučiti koje indekse će kreirati kako bi podesio performanse. PostgreSQL takođe pruža bogat skup statistika koja se koristi za uvid u kvalitet i količinu podataka u svakoj tabeli, što znači da može da generiše plan izvršenja i da nadgleda koji se indeksi koriste i kada.

Vreme obrade može varirati od jednog delića sekunde do više sati, zavisno od izabranog načina pristupa bazi podataka. Svrha optimizacije upita, koji je automatizovan proces, jeste da se pronađe način da se dati upit obradi u minimalnom vremenu. Važno je naglasiti da je podešavanje performansi jedno od najsloženijih zadatke u administraciji baze podataka, te da ne postoji jedno zlatno rešenje ili jednoznačno rešenje, pa su potrebni iskustvo i puno prakse.

1. Literatura
2. Database SQL Tuning Guide, <https://docs.oracle.com/database/121/TGSQL/tgsql_optcncpt.htm#TGSQL195> , (pristupljeno 11. aprila 2021.).
3. Beginning Databases with PostgreSQL: From Novice to Professional, Richard Stones, Neil Matthew, 2005.
4. PostgreSQL 13.0 Documentation, <https://www.postgresql.org/docs/13/> , (pristupljeno 11. aprila 2021.).
5. Learn PostgreSQL, Luca Ferrari, Enrico Pirozzi, Packt Publishing, 2020., <https://learning.oreilly.com/library/view/learn-postgresql/9781838985288/> , (pristupljeno 12. aprila 2021.).
6. Mastering PostgreSQL 13 - Fourth Edition, Hans-Jurgen Schonig, Packt Publishing, 2020., <https://learning.oreilly.com/library/view/mastering-postgresql-13/9781800567498/> , (pristupljeno 13. aprila 2021.).
7. Optimizacija upita, <https://vtsnis.edu.rs/wp-content/plugins/vts-predmeti/uploads/5.6_TehnikeZaOptimizacijuUpita.pdf?script=lat> , (pristupljeno 14. aprila 2021.).