**Univerzitet u Nišu, Elektronski fakultet**

**Katedra za računarstvo**

logo_1960_4

**Obrada upita kod PostgreSQL baze podataka**

Stefan Živić 698

**Predmet: Sistemi za upravljanje bazama podataka**

# Sadržaj

[Sadržaj 2](#_Toc69065017)

[Uvod 3](#_Toc69065018)

[Faze u obradi upita kod PostgreSQL-a 4](#_Toc69065019)

[Uspostavljanje konekcije 5](#_Toc69065020)

[Parsiranje 6](#_Toc69065021)

[Faza transformacije 6](#_Toc69065022)

[Faza prerađivanja upita 6](#_Toc69065023)

[Faza planiranja 7](#_Toc69065024)

[Generisanje mogućih planova 7](#_Toc69065025)

[Faza izvršavanja upita 8](#_Toc69065026)

[Primer obrade upita kroz faze 10](#_Toc69065027)

[Keširanja podataka pri izvršavanju upita 17](#_Toc69065028)

[Primeri korišćenja keš memorije na konkretnim upitima 19](#_Toc69065029)

[Primer sa sekvencijalnim skeniranjem 21](#_Toc69065030)

[Realizacija parametrizovanih upita i pripremljenih naredbi 23](#_Toc69065031)

[Primeri obrade složenijih upita 25](#_Toc69065032)

[Zaključak 25](#_Toc69065033)

[Literatura 25](#_Toc69065034)

# Uvod

Svi sistemi baza podataka moraju biti u stanju da odgovore na zahtev za informacijama od korisnika - tj. upit. Upit predstavlja način zadavanja komandi sistemu za upravljanje bazom podataka. Ove komande mogu biti čitanje podataka iz memorije, upis novih podataka u memoriju, prepisivanje postojećih ili brisanje. Dobijanje željenih informacija iz sistem baza podataka na predvidljiv i pouzdan način predstavlja obradu upita. U ovom radu će biti ovrađeni koncepti obrade upita kod sistema za upravljanje relacionim bazama podataka - PostgreSQL. Biće prikazano kako server obrađuje upit kroz faze, koji se sve algoritmi i pravila koriste, uz demonstraciju na primerima.

PostgreSQL( sreće se i pod imenom Postgres) je, kao što je to rečeno, sistem za upravljanje objektno relacionim bazama podataka. Poznat je po svojoj pouzdanosti, robusnosti, performansama. Kao i kod ostalih relacionih baza, podaci se čuvaju u redovima, a redovi se organizuju u tabele. Ima multiplatformsku podršku, može raditi na svim poznatijim sistemima kao što su Linux, Widnows i ostali. Njegov izvorni kod je dostupan svakom ko želi da ima uvid u to kako je određeni deo sistema implementiran, ili možda nadogradi i prepravi neki deo. Proširljiv je tako da korisnici mogu dodati svoje tipove podataka, funkcije, operatore itd. Pored toga, postoji mnoštvo alata i programa koji olakšavaju rad sa PostgreSQL-om. Za rad sa bazom koja je korišćena za praktični deo ovog rada korišćen je *pgAdmin* i njegova opcija analize i grafičke analize upita.

Rad je organizovan tako da je u prvom delu dato teorijski objašnjenje obrade upita, a u drugom, praktičnom delu, kako to izgleda na primerima. U prvom poglavlju objašnjena obrada upita u generalnom smislu, a potom, u narednom poglavlju, kako ona konkretno izgleda kod PostgreSQL-a. U tom poglavlju se nalaze pobrojane sve faze u obradi, šta se dešava u svakoj od njih, šta je specifično za svaku itd. Nakon toga, u narednda dva poglavlja, obrađene su dve teme koje su jako bitne kada se govori o obradi upita i načinu na koji će upit biti izvršen. To su *keširanje podataka* i *parametrizovnai upiti.* Prva je bitna jer se različiti upiti različito i ponašaju po pitanju koriščćenja keš memorije i da bismo shvatili kako se upit izvršava i mogli da utičemo na njegove performanse, bitno je ukapirati tu spregu. Sa druge strane, parametrizovani upiti su značajni jer se njihova obrada može u velikoj meri razlikovati od obrade “običnih”, neparametrizovanih upita. Ovim poglavljem se završava “teorijski” deo rada.

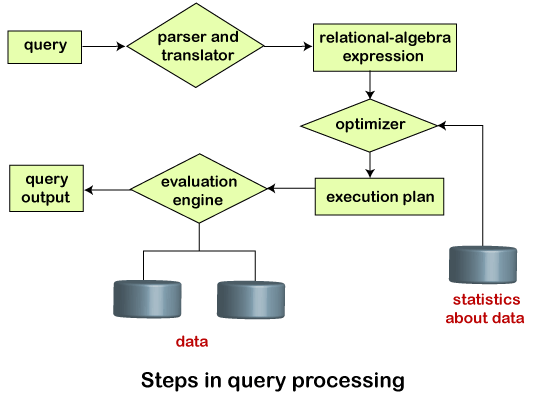
Na kraju, tu je poglavlje sa primerima. Ovo poglavlje predstavlja praktični deo rada jer prikazuje konkretne upite koji se izvršavaju nad realnom bazom i podacima. Organizovano je tako da prati tok teorijskog dela, pa je prvo dat primer obrade upita kroz faze. Nakon toga sledi analiza složenijih upita i primeri upita sa analizom korišćenja keš memorije. Za rad sa bazom korišćen je *pgAdmin,* i njegov alat sa analizu i grafičku analizu obrade upita. Za svaki od primera data je slika koja je na neki način rezultat izvršenja ( najčešće prikazan plan ) i dodatno objašnjenje.

# Obrada upita

Obrada upita označava kompajliranje i izvršavanje specifikacije upita koja se obično izražava u deklarativnom jeziku upita baze podataka, kao što je SQL. Ona podrazumeva prevode upita na visokom nivou u izraze koji se mogu razumeti na fizičkom nivou, optimizaciju upita i izvršenje upita da bi se dobio stvarni rezultat.Na najvišem nivou obrada upita sastoji se od faze kompajliranja, ili prevođenja, kako se često naziva, i faze izvršavanja. U fazi kompajliranja, prevodilac upita prevodi specifikaciju upita u izvršni program. Ovaj proces prevođenja sastoji se od leksičke, sintaksičke i semantičke analize specifikacije upita, kao i faze optimizacije upita i generisanja koda. Generirani kod se obično sastoji od fizičkih operatora koje sistem za upravljanje bazom “zna” kako da izvrši. Ovi operatori implementiraju pristup podacima, spajanja, selekciju, projekcije, grupisanje i agregaciju. Tokom izvođenja, mehanizam baze podataka tumači i izvršava program koji implementira specifikaciju upita kako bi proizveo rezultat upita.

Najčeće se faza kompajliranja upita deli u fazu parsiranja i fazu optimizacije, tako da se obrada upita po koracima prikazuje ovako :

1. Parsiranje i prevođenje
2. Optimizacija
3. Izvršenje



Slika - Obrada upta

Dakle tokom prve faze vrši se provera sintakse, kao i semantička provera. Korisnički upit se prvo prevodi u jezik visokog nivoa baze podataka, npr SQL. Ako je sintaksno ispravan, nakon toga se prevodi u jezik niskog nivoa koji se može razumeti na fizičkom nivou, za šta se koriste izrazi relacione algebre.

U sledećem koraku se bira najefikasniji plan izvršenja za upit. Tokom analiziranja optimalnosti gleda se koliko je “skupa” svaka od operacija koja će se izvršiti da bi se dobili rezultati i na osnovu toga se formira put koji je najbolji po pitanju brzine, zauzeća memorije, iskorišćenja CPU-a itd. U ovaj fazi se pristupa statistici, odnosno katalozima sistema, kako bi planer imao sve potrebne informacije pre “donošenja” odluke koji je put najbolji. Obzirom da će planer ispitati više mogućih planova (nekad i sve), oni koji su obrađeni se čuvaju u katalozima da bi se ne kraju izabrao jedan. Plan se dalje šalje izvršiocu, koji ga obrađuje, izvršava i vraća rezultat.

# Obrada upita kod PostgreSQL-a

Podsistem za obradu upita je najsloženiji deo PostrgreSQL-a. Da bi jedan upit bio uspešno izvršen on mora da prođe kroz nekoliko faza. Pre svega, mora se uspostaviti veza između aplikacije, odnosno programa i PostgreSQL servera, da se pošalje upit, obradi, parsuje, izvuku informacije, pribave rezultati koji se vraćaju nazad. Dakle, nakon uspostavljanja veze i slanja teksta upita serveru, slede faze vezane za samu obradu upita, tako da celokupan put od slanja upita do vraćanja rezultata kroz faze izgledao ovako :

1. Uspostavlja se veza između aplikativnog programa i PostgreSQL servera. Aplikativni program prenosi upit na server i čeka da dobije rezultate koje je server poslao nazad.
2. Faza parsiranja proverava ispravnost sintakse upita koji prenosi aplikativni program i kreira stablo upita.
3. Faza analize upita u kojoj se proverava semantička ispravnost upita i izvlače informacije o tome kojoj tabeli i kolonam se pristupa
4. Sistem za prerađivanje( prepisivanje) uzima stablo upita kreirano u fazi parsiranja i traži bilo koja pravila (uskladištena u sistemskim katalozima) koja se primenjuju na stablo upita. Obavlja transformacije date u pravilima.

Jedna primena sistema za prepisivanje je u kreiranju pogleda( view ). Kad god se napravi upit pogledu, sistem za prerađivanje preradi korisnikov upit u upit koji umesto toga pristupa osnovnim tabelama datim u definiciji pogleda

1. Planer / optimizator uzima prerađeno stablo upita i kreira plan upita koji će biti ulaz izvršiocu.

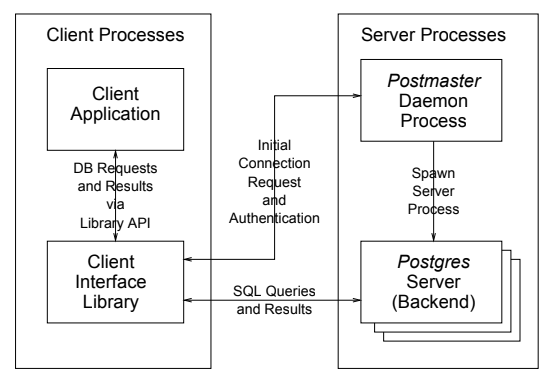
To čini tako što prvo kreira sve moguće puteve koji vode do rezultata. Na primer, ako postoji indeks na tabeli koja se pretražuje, postoje dve putanje za izvršenje. Jedna mogućnost je jednostavno sekvencijalno petraživanje, a druga mogućnost je korišćenje indeksa. Zatim se procenjuje optimalnost izvršenja svake putanje i bira se najefikasnija putanja.

1. Izvršitelj upita se kreće rekurzivno kroz stablo plana i preuzima redove na način zadat planom. Izvršitelj koristi sistem za skladištenje dok petražuje tabele, vrši sortiranje i spajanje i na kraju vraća redove.

## Uspostavljanje konekcije

Prilikom izvršavanja upita program, odnosno korisnička aplikacija u bilo kom obliku, prenosi upit na server i čeka da dobije rezultate koje je server poslao nazad. PostgreSQL je implementiran tako da koristi jednostavan model *klijent-server* arhitekture -  *„proces po korisniku“*. U ovom modelu postoji jedan klijentski proces povezan sa tačno jednim serverskim procesom. Kako ne znamo unapred koliko će veza biti uspostavljeno, moramo koristiti glavni proces koji svaki put kada se zahteva veza kreira[[1]](#footnote-1) novi proces servera. Ovaj glavni proces se naziva *postgres* i on osluškuje određeni TCP/IP port za nadolazeće konekcije. Kad god se otkrije novi zahtev za vezu, postgres proces generiše novi proces servera. Procesi servera međusobno komuniciraju koristeći semafore i zajedničku memoriju kako bi se osigurao integritet podataka tokom istovremenog pristupa podacima.

Klijentski proces može biti bilo koji program koji implementira PostrgreSQL protokol. Jednom kada se uspostavi veza, klijentski proces može poslati upit serveru. Upit se prenosi pomoću običnog teksta, tj. na klijentskoj strani se ne vrši nikakva konverzija, niti parsiranje. Server analizira upit, kreira plan izvršenja, izvršava plan i vraća preuzete redove klijentu prenoseći ih preko uspostavljene veze.



Slika - Klijent-Server arhitektura kod PostgreSQL-a

## Parsiranje

Parser proverava sintaksu upita koji je stigao kao tekst. Ako je sintaksa ispravna, odnosno nije pronađena greška, kreira se ***stablo parsiranja - parse tree***, a u suprotnom se vraća greška koja je pronađena. U ovoj fazi učestvuju dve komponente, *parser* i *lexer*. Oni su definisani pomoću Unix-ovih alata *bison* i *flex.*

*Lexer*  je definisan u fajlu *scan.l* i odgovoran je za pronalaženje identifikatora, ključnih SQL reči itd. Za svaki pronađeni identifikator i za svaku ključnu reč generiše se token koji se prosleđuje parseru.

Parser je definisan u fajlu *gram.y* i sastoji se od seta gramatičkih pravila i akcija koje se izvršavaju svaki put kada se određeno pravilo ispuni, ili kako se često u literaturi kaže - okine. Kod za akcije, koji je ustvari C kod, se koristi za kreiranje *stabla parsiranja*.

Fajl scan.l se prevodi u C fajl *scan.c* pomoću alata flex, dok se gram.y prevodi u *gram.c* pomoću bison-a.

## Faza semantiče analize / faza prevođenja (transformacije) upita

Obzirom da se tokom faze parsiraranja kreira se stalo parsiranjapomoću fiksnih, unapred određenih, sintaksnih pravila, u toj fazi se ne pristupa katalozima sistema, pa se tokom nje ne može odrediti semantička ispravnost prosleđenih upita. Zato je tu faza semantičke analize( negde se može sresti pod imenom faza transformacije upita), u kojoj se prethodno kreirano stablo semantički obrađuje, odnosno intepretira. Tada se dobijaju informacije kojima se otkriva na koju tabelu, funkciju, operaciju se odnosi upit. Kao rezultat ove faze kreira se ***stablo upita - query tree****.*

Razlog za razdvajanje početnog sintaksnog parsiranja od semantičke analize je taj što se pretraživanja sistemskog kataloga može izvršiti samo u okviru transakcije,a nije dobro započeti transakciju odmah po prijemu niza upita. Tokom faze parsiranja mogu se identifikovati komande za kontrolu transakcija (BEGIN, ROLLBACK, itd.), i one se zatim mogu pravilno izvršiti bez ikakve dalje analize. Jednom kada znamo da imamo posla sa konkretnim upitom (kao što je SELECT ili UPDATE), u redu je započeti transakciju ako već nismo u njoj. Tek tada se može započeti proces transformacije.

Stablo upita kreirano procesom transformacije strukturno je slično stablu parsiranja, ali ima mnogo razlika u detaljima. Na primer, čvor FuncCall u stablu parsiranja predstavlja nešto što sintaksički liči na poziv funkcije. Ovo se može transformisati u čvor FuncExpr ili Aggref, u zavisnosti od toga da li je referencirano ime uobičajena funkcije ili agregatne funkcije. Takođe, u stablo upita dodaju se informacije o tipovima podataka kolona i rezultatima izraza.

## Faza prerađivanja upita

Tokom faze prerađivanje upit se prerađuje na osnovu definisanih pogleda. PostgreSQL podržava moćan *sistem pravila (rule system)* za specifikaciju pogleda(view) i  ažuriranja pogleda. Kroz istoriju postojale su dve implementacije sistema za pravila:

* Prvi je radio pomoću obrade na nivou reda i implementiran je u izvršiocu. Sistem pravila se pozivao kad god se pristupilo pojedinačnom redu. Ova primena je uklonjena 1995. godine kada je poslednje zvanično izdanje projekta Berkelei Postgres transformisano u Postgres95.
* Druga implementacija sistema pravila je tehnika koja se naziva *prerađivanje upita( query rewriting)*. Sistem za prerađivanje upita je modul koji postoji između faze parsiranja i faze planiranja i optimizovanja. Ova tehnika se i dalje primenjuje.

Ulaz i izlaz faze prerađivanja upita su ***stabla upita***, odnosno nema promena u strukturi ili nivou semantičkih detalja u stablima. Prerađivanje upita se može smatrati oblikom proširenja makro blokova

## Faza planiranja

Zadatak planera / optimizatora je da kreira optimalan plan izvršenja. Dati SQL upit, samim tim i stablo upita, može se zapravo izvršiti na mnogo različitih načina, od kojih će svaki dati isti skup rezultata. Ako je to izvodljivo, optimizator upita će ispitati svaki od mogućih planova izvršenja i na kraju izabrati onaj za koji se očekuje da će se najbrže i najoptimalnije izvesti. U nekim situacijama ispitivanje svakog mogućeg načina na koji se upit može izvršiti oduzelo bi previše vremena i memorije. To se naročito događa kada se izvršavaju upiti koji uključuju veliki broj operacija spajanja tj *JOIN* operacija. Da bi odredio dovoljno dobar plan upita u razumnom vremenskom roku, PostgreSQL koristi *Genetic Query Optimizer,* kada broj operacija spajanja premaši prag - *geqo\_threshold.*

Tokom postupka pretraživanja planer zapravo radi sa strukturama podataka koje se nazivaju *putanje - paths*, a koje su jednostavno skraćeni, odnosno uprošćeni,  prikazi planova koji sadrže samo onoliko informacija koliko je planeru potrebno za donošenje odluka. Nakon što se odredi najoptimalnija putanja, gradi se ***stablo plana - plan tree***,  koje će se proslediti izvršiocu. Stablo sadrži dovoljno informacija potrebnih izvršiocu.

### Generisanje mogućih planova

Planer / optimizator započinje generisanjem planova za skeniranje svake pojedinačne relacije (tabele) koja se koristi u upitu. Mogući planovi su određeni dostupnim indeksima za svaku tabelu. Obzirom da uvek postoji mogućnost izvođenja sekvencijalnog skeniranja nad tabelom, plan sekvencijalnog skeniranja se uvek kreira. Pretpostavimo da je indeks definisan nad tabelom (npr. indeks B-stabla), a upit sadrži uslov tipa :

“*tabela.atribut OPR konstanta”* .

Ako se tabela.atribut podudara sa ključem indeksa B-stabla i OPR je jedan od operatora navedenih u klasi operatora indeksa, kreira se drugi plan pomoću indeksa B-stabla za skeniranje tabele. Ako postoji još indeksa i ako se ograničenja u upitu podudaraju sa ključevima tih indeksa, kreiraće se dalji planovi. Planovi skeniranja pomoću indeksa takođe se generišu za indekse koji imaju redosled sortiranja koji se može podudarati sa klauzulom ORDER BY upita (ako postoji) ili redosled sortiranja koji bi mogao biti koristan za spajanje tabela.

Ako upit zahteva spajanje dve ili više tabela( join upit), planovi za spajanje tabela razmatraju se nakon što se pronađu svi izvodljivi planovi za skeniranje pojedinačnih tabela. Tri dostupne strategije spajanja su:

* **nested loop join** - desna tabela se skenira jednom za svaki red koji se nalazi u levoj. Ovu strategiju je lako implementirati, ali može biti neoptimalna u smislu vremena izvršenja, ako se ne koriste indeksi
* **merge join** - svaka tabela se sortira na atributima koji su prisutni u uslovu za spajanje tabela pre nego što spajanje započne. Zatim se paralelno skeniraju dve tabele, a redovi koji se podudaraju se kombinuju da bi se formirali zajednički. Kod ove vrste se svaka tabela skenira samo jednom.
* **hash join** - desna tabela se prvo skenira i učitava u hash tabelu, koristeći atribute iz uslova za spajanje kao hash ključeve. Sledeće se skenira leva tabela i odgovarajuće vrednosti svakog pronađenog reda koriste se kao hash ključevi za pronalaženje odgovarajućih redova u tabeli.

Gotovo *stablo plana* sastoji se od sekvencijalnih ili indeksnih skeniranja osnovnih tabela, plus čvorova za spajanje tabela( bilo koje od tri vrste)i, kao i svih potrebnih pomoćnih čvorova, kao što su čvorovi za sortiranje ili čvorovi za izračunavanje agregatne funkcije. Na najvišem novou čvorovi se dela na:

* čvorove spajanja
* čvorove pretraživanja/skeniranja
* čvorove materijalizacije ( agregacija, soritranje, grupisanje itd. )

## Faza izvršenja upita

Izvršilac (Executor) uzima plan koji je kreirao planer / optimizator i rekurzivno ga obrađuje kako bi izdvojio potreban skup redova. Ovo je u osnovi mehanizam koji se naziva *demand-pull pipeline mechanism*. Svaki put kada se pozove čvor iz stabla plana, on mora isporučiti još jedan red ili da prijavi da je završio s isporukom redova.

Da bismo bolje razumeli kako rekurzija u ovom slučaju radi, videćemo kratak primer. Pretpostavimo da je glavni čvor *MergeJoin* čvor. Pre nego što se izvrši bilo kakvo spajanje, potrebno je preuzeti dva reda (po jedan iz svakog potplana koji predstavljaju tabele koje se spajaju). Dakle, izvršilac se rekurzivno poziva da obrađuje potplanove (započinje potplanom koji je vezan za levo stablo). Novi glavni čvor (glavni čvor levog potplana) je, recimo, čvor *Sort* i ponovo je potrebna rekurzija da bi se dobio red. “Dete” čvora Sort može biti čvor *SeqScan*, koji zapravo predstavlja stvarno čitanje podataka iz tabele. Izvršenje ovog čvora dovodi do toga da izvršilac preuzme red iz tabele i vrati ga do pozivajućeg čvora, a to je Sort. Čvor Sort će više puta pozivati svoje dete čvor da dobije sve redove koje bi trebalo da sortira. Kada nema više redova ,odnosno kada podređeni čvor vrati NULL umesto reda, kod Sort čvora vrši sortiranje i konačno može da vrati svoj prvi izlazni red, što će biti prvi red u sortiranom redosledu. Preostale redove čuva uskladištenim kako bi ih mogao dostaviti u sortiranom redosledu kao odgovor na kasnije zahteve.

Čvor MergeJoin na sličan način zahteva prvi red iz svog desnog potplana. Zatim upoređuje dva reda da bi se videlo da li se mogu spojiti. Ako mogu, vraća spojeni red. Pri sledećem pozivu, ili odmah ako se ne može izvršiti spajanje redova koje je dobio, prelazi u sledeći red jedne ili druge tabele (u zavisnosti od načina na koji je poređenje izvedeno) i ponovo proverava da li se podudaraju. Na kraju, jedan ili drugi potplan nema više redova, a čvor MergeJoin vraća NULL da ukaže da se više ne mogu formirati spojeni redovi.

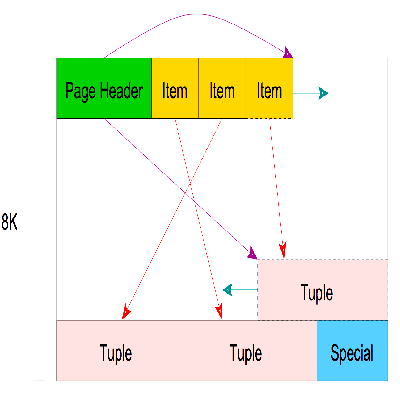
Složeni upiti mogu uključiti više nivoa čvorova, ali opšti pristup je isti: svaki čvor vraća svoj sledeći red svaki put kada je pozvan. Čvorovi su takođe odgovorni za primenu bilo kog izraza za selekciju ili projekciju koji mu je dodelio planer.

Mehanizam izvršioca koristi se za procenu sva četiri osnovna tipa SQL upita: SELECT, INSERT, UPDATE i DELETE. Za SELECT, izvršni kod najvišeg nivoa samo treba da pošalje klijentu svaki red koji je vratilo stablo plana upita. INSERT ... SELECT, UPDATE i DELETE su zapravo vrste SELECT-a sa posebnim glavnim čvorom stabla plana. Kod INSERT ... SELECT upita svi redovi za umetanje dolaze do ModifyTable. Za UPDATE, planer uređuje da svaki selektovani red uključi sve ažurirane vrednosti kolona, plus TID(tupleID), ili ID reda,  originalnog reda. Ovi podaci su vraćaju do čvora ModifyTable, koji koristi informacije za kreiranje novog ažuriranog reda i označavanje starog kao izbrisanog. Za DELETE, jedina kolona koju plan zapravo vraća je TID, a čvor ModifyTable jednostavno koristi TID da dobije svaki ciljni red i označi ga kao izbrisan. Sa druge strane, jednostavna naredba INSERT ... VALUES kreira trivijalno stablo plana koje se sastoji od jednog *Result* čvora, koji izračunava samo jedan red rezultata, i vraća ga do ModifyTable za umetanje.

# Keširanje podataka pri obradi upita

Iako Postgres ima nekoliko podešavanja koja direktno utiču na alokaciju memorije u svrhu keširanja, većim delom keš memorije koju Postgres koristi upravlja operativni sistem. Postgres, za razliku od većine drugih sistema baza podataka, u velikoj meri koristi keš memoriju operativnog sistema za veoma veliki broj operacija. Postoji nekoliko različitih načina na koje Postgres alocira keš memoriju. Ono što uvek važi je to da se veći deo memorije koji “dobije” instanca PostgreSQL servera dodeljuje operativnom sistemu na upravljanje( OS upravlja njome na isti način kao sa svojom keš memorijom). Delom memorije koji se naziva „Shared Buffer Cache“, upravlja PostrgreSQL, i tu memoriju koristi interno za čuvanje podataka i indeksa u memoriji. To je obično konfigurisano da iznosi oko 25% ukupne sistemske memorije dodeljenje Postgres instanci. Preostalu raspoloživu memoriju Postgres koristi u dve svrhe: za keširanje podataka i indeksa na disku putem keša operativnog sistema i za interne operacije ili strukture podataka. Najčešće Postgres instanci bude dodeljeno 500MB, što znači da Shared buffers bude veličine 128MB.

Nezavisno od veličine memorije, u PostgreSQL-u podaci su organizovani u stranicama. Te stranice su najčešće veličine 8 KB( ako nike eksplicitno rečeno drugačije), a svaka takva stranica može sadržati više redova (u zavisnosti od veličine reda). Pojednostavljeni prikaz mogao bi biti kao u nastavku:



Slika - Prikaz jedne stranice u memoriji

Postgres kešira sledeće podatke :

* Podatke iz tabele

Stvarni sadržaj tabela u vidu redova skladištenim u njoj

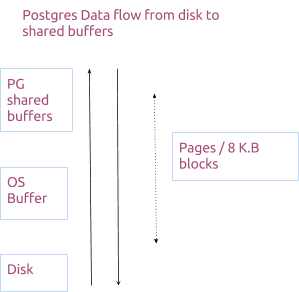
* Indekse

Indeksi se takođe čuvaju u 8K blokovima. Čuvaju se na istom mestu kao i podaci tabele.

* Planovi izvršenja upita

PostgreSQL u pojedinim situacijama može keširati planove izvršenja vezane za neku pripremljenu naredbu, odnosno upit. Ako se ne koristi keširani plan, svaki poziv takve naredbe bi imao za posledicu kreiranje novog plana izvršenja, kome će biti prosleđene vrednosti parametara kako bi mogla da se izvrši optimizacija i izabere “najjeftiniji” put izvršenja. Ako naredba nema parametre, ili se izvrši više puta, Postgres će pokušati da kreira generički plan, koji ne zavisi od konkretnih vrednosti parametara. Takav plan se čuva u keš memoriji i može se upotrebiti iznova i iznova, čime se štedi vreme. Detaljnije objašnjenje ovog proces biće dato u narednom poglavlju - Realizacija parametrizovanih upita i pripremljenih naredbi .

Što se organizacije memorije tiče Postgres se kao multiplatformska baza podataka  u velikoj meri se oslanja na operativni sistem za svoje keširanje. Shared\_buffers u okviru PostgreSQL-a zapravo na neki način duplira ono što OS radi. U nastavku je data tipična slika kako podaci prolaze kroz postgres:



Slika - Organizacija memorije

Ovo je u početku zbunjujuće, jer keširanjem upravljaju i OS i Postgres, ali za to postoje razlozi. Glavni razlog leži u tome što PostrgreSQL i OS koriste različite strategije za dodavanje i uklanjanje stranica iz keša. Shared buffers deo memorije, kojim upravlja Postgres, koristi ***Clock sweep***algoritam. Ovaj algoritam se zasniva na brojanju pristupa svakom bloku. Sa druge strane, kod OS-a se najčešće koristi *Last Recently Used* algoritam, koji se zasniva na informacijama o tome koji je podatak skorije korišćen. Već smo rekli da kod Postgres-a svaka konekcija ima svoj nativni OS proces koji je opslužuje. Kada se upit izvršava takav proces traži podatke iz LRU keša operativnog sistema. Tada se, pre svega, gleda da li taj podatak postoji u Shared buffers memoriji, i ako postoji, onda se povećava brojač pristupa za taj podatak i on se vraća. Na taj način će se u keš memoriji PostrgreSQL-a zadržati podaci koji se najčešće koriste.

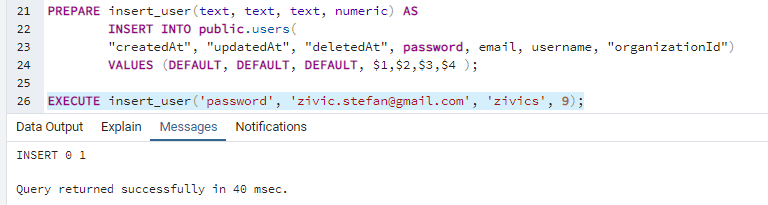
Kod čitanja podataka, ako se oni nalaze u Shared buffers i ne dolazi do keš memorije operativnog sistema. Kada se traži podatak koji nije prisutan ni u jednom od ova dva keša, onda se on pribavlja iz memorije, upisuje u OS keš, a zatim i u Postgres keš. Kod upisa, podaci se prvo upisuju u Shared buffers. Nakon toga se ta stranica označi da je promenjena( dirty bit) i upisuje se u keš operativnog sistema, odakle se kasnije upisuje i u memoriju. Na ovaj način je prepušteno operativnom sistemu da redosled I/O operacija organizuje na najbolji mogući način.

Organizovanje memorije na ovaj način, gde OS i Postgres koriste različite strategije keširanja podataka dovodi do toga da aplikacije imaju daleko bolji procenat iskorišćenosti keša, a samim tim i bolje performanse.

# Obrada parametrizovanih upita i pripremljenih naredbi

Često se dešava da se pri radu sa bazom tokom jedne sesije se izvrši veliki broj istih ili sličnih upita. Neretko su ovi upiti složeni, treba vremena da se parsiraju, analiziraju, prepišu i pogotovo da se kreira plan za njih. Srećom PostrgreSQL nudi rešenje za ovaj problem u vidu parametrizovanih upita i pripremljenih naredbi.

Pripremljene naredbe su upiti koji umesto konkretnih vrednosti parametara imaje tzv. “placeholdere”, odnosno simbole koji označavaju da će se na njihom mestu naći parametar sa konkretnom vrednošću. Kreiraju se komandom PREPARE i predstavljaju objekat na strani servera koji se koristi za optimizaciju performansi. Naredba pripremljena sa PREPARE se izvršava EXECUTE komandom. Kada se pozove prepare naredba se parsira, analizira i prepisuje. Kada se izda execute, pripremljena naredba se planira i izvršava. Ono što pripremljenim naredbama pruža mogućnost boljeg izvršenja po pitanju performansi je upravo to što se mogu parametrizovati. Tokom pripremanja umesto konkretnih parametra koriste se simboli “$n” koji se tokom faze planiranja i izvršenja menjaju konkretnim vrednostima parametara.



Slika - Primer pripremljene naredbe i njenog izvršenja

Ovakvom podelom posla u dve faze sa prepare i execute se dolazi do toga da se izbegava stalno izvršavanje prve 3 faze, pogotovo faze analize. Istovremeno se omogućava da se kreira plan izvršenja koji zavisi od specifičnih vrednosti parametara, ali čak i to da se i faza planiranja preskače, u zavisnosti od upita, performansi, podešavanja( slučaj koji nas zanima).

Kao što je rečeno, pripremljene naredbe potencijalno imaju najveću prednost u performansama kada se jedna sesija koristi za izvršavanje velikog broja sličnih upita. Razlika u performansama biće posebno značajna ako su upiti složeni za planiranje ili prepisivanje - npr. ako upit uključuje spajanje mnogih tabela ili zahteva primenu nekoliko pravila. Ako je naredba relativno jednostavna za planiranje i prepisivanje, ali relativno skupa za izvršavanje, prednost performansi pripremljenih izjava biće manje primetna.

PostgreSQL može izabrati dva načina za izvršavanje pripremljenog upita:

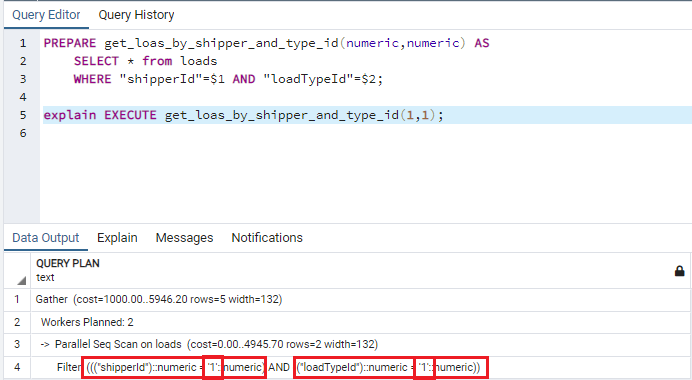
1. Mogao bi stvoriti novi, *prilagođeni plan* za svako izvršavanje koje uzima u obzir trenutne vrednosti parametra - *custom query*. To će dovesti do toga da se uvek dobija najoptimalniji mogući plan izvršenja, ali planiranje upita za svako izvršavanje može ukloniti osnovnu prednost korišćenja pripremljenih upita - izbegavanje potrebe za planiranjem istog upita iznova i iznova.
2. Mogao bi stvoriti *generički plan* koji ne uzima u obzir vrednosti parametara – *generic query*. Time ćemo izbeći ponovno planiranje upita svaki put, čime ćemo dobiti na performansama u velikoj meri. Generički plan se, kako je to objašnjeno i u prethodnom poglavlju) čuva u keš memoriji Postgresa. Problem sa ovim pristupom je taj što je u pojedinim slučajevima prilagođeni plan daleko efikasniji od generičkog.

Što se tiče strategije koju koristi PostgreSQL, može se reći da ona predstavlja kompromis ova dva pristupa. Naime, ako nije eksplicitno drugačije podešeno, PostgreSQL će generisati prilagođeni plan tokom *prvih pet* izvršavanja pripremljene naredbe koja uzima vrednosti parametara u obzir i izračunaće prosečnu “cenu” planova. Naredni, šesti, put se kreira generički plan i njegova procenjena cena se upoređuje sa izračunatom prosečnom cenom prilagođenog plana. Ako cena generičkog plana nije mnogo veća (toliko da se više isplati raditi re-planiranje upita svaki put) nastavlja se sa njegovim korišćenjem.

Strategija se može menjati, jer je od verzije 12 PostgreSQL-a uvedena promenljiva *plan\_cache\_mode* koja može imati vrednosti :

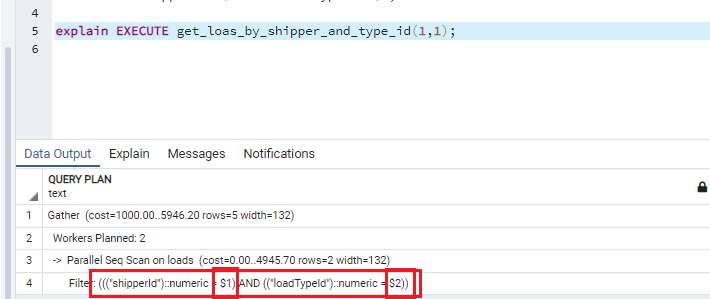
* *force\_generic\_plan*  za generički plan, ili
* *force\_custom\_plan* za prilagođeni.

Ako se koristi generički plan, on će sadržati simbole parametara označene sa “$n”, gde je n redni broj parametra, dok će prilagođeni plan imati zamenjene isporučene vrednosti parametara. Ovo možemo videti i na primeru :



Slika - Upit tokom prvih pet izvršenja

Ovde se vidi kako izgleda plan tokom prvih pet izvršenja, a evo kako izgleda od šestog:



Slika - Upit tokom šestog izvršenja

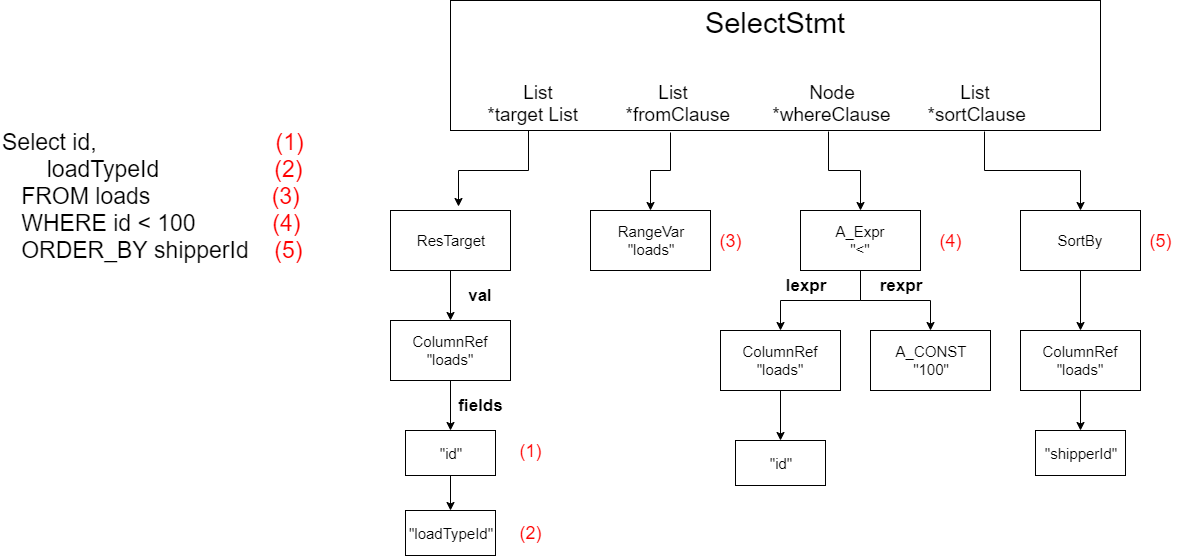
# Primeri obrade upita

U ovom delu videćemo kako obrada upita funkcioniše na konkretnim primerima, pomoću *explain* opcije kojoj se dodaju različiti parametri. Kao što je objašnjeno u uvodnom poglavlju. počećmo obradom upita kroz faze, obzirom da je to najvažniji deo u razumevanju kako sama obrada funckioniše. Videćemo kako izgleda izlaz u vidu stabla svake od faza, koje strukture i čvorovi učestvuju u generisanju tih stabla, kako se obrađuju itd. Nakon toga su primeri obrade složenijih upita za koje je prikazana grafička analiza pomoću pgAdmin programa. Ove je bitno videti skup različitih planova, koje sve vrste čvorova se nalaze u njima i zašto su baš oni prisutni u planu. Slede primeri vezani za poglavlje 3 Keširanje podataka pri obradi upita*,* kojima se ovo poglavlje zaključuje.

Za primer obrade upita kroz faze uzet je jednostavniji upit kako bi bilo lakše ispratiti, a i prikazati  stabla koja se kreiraju kao rezultati pojedinih faza. Upit koji ćemo uzeti za primer je :

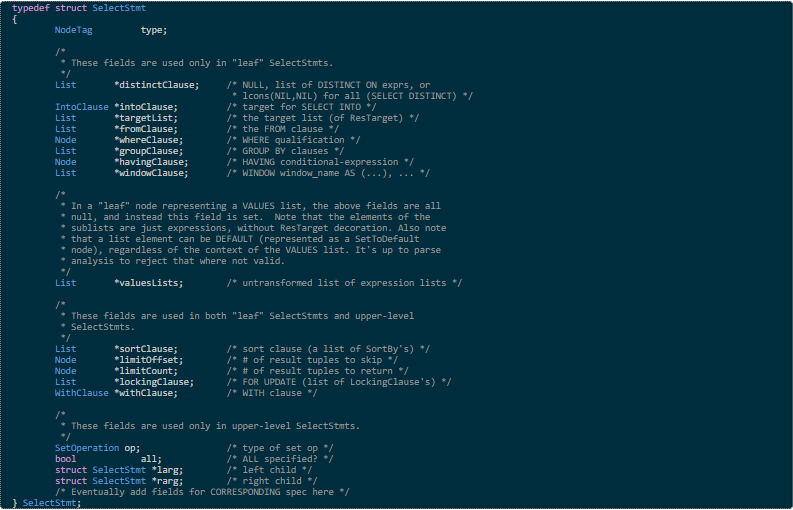
***SELECT id, loadTypeId FROM loads WHERE id < 100 ORDER\_BY shipperId;***

Prva faza je, kako je to već rečeno, parsiranje. Za navedini upit stablo parsiranja koje predstavlja izlaz ove faze izgleda ovako:



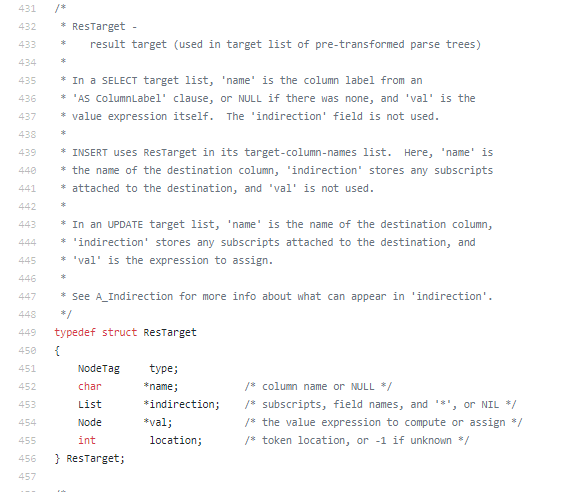
Slika 3 - Stablo parsiranja

Na levoj polovini slike mogu se videti označeni delovi upita i kako su oni predstavljeni u stablu. Koren ovog stabla je SelectStmt struktura koja izgleda ovako :



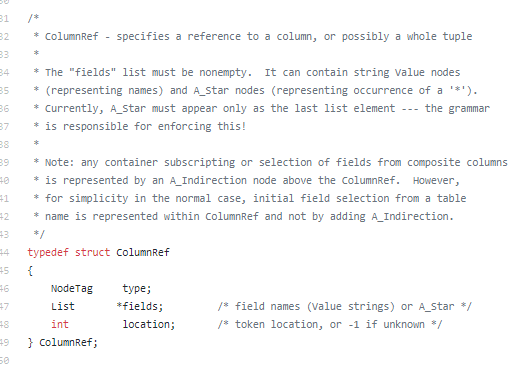
Slika 4 - SelectStmt struktura

Nama od značaja su polja targetList koje će biti pokazivač strukturu *ResTarget* čije polje *va*l predstavlja tabelu nad kojom se izvršava upit.



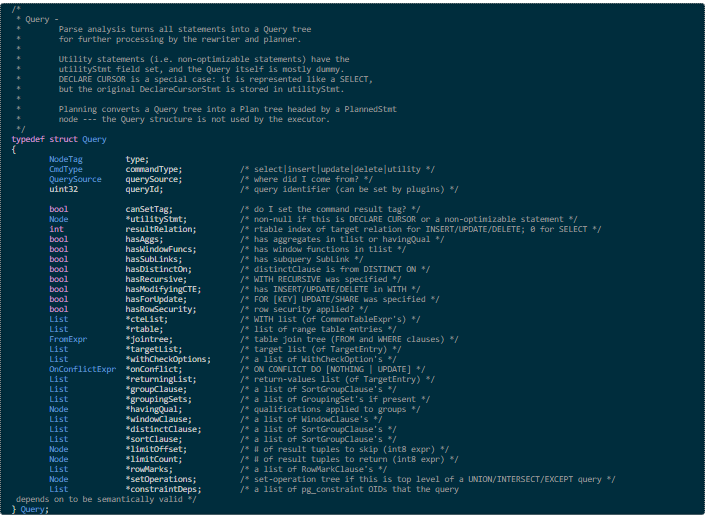
Slika 5 - ResTarget struktura

Polja u tabeli nad kojima se vrši upit su određena strutkurom *ColumnRef,* odnosno njenim atributom *fields,* kao što se vidi na primeru. Ako je ime tabele, ili neke od kolona pogrešno to se u ovoj fazi neće otkriti i stablo će svakako biti generisano na isti način.



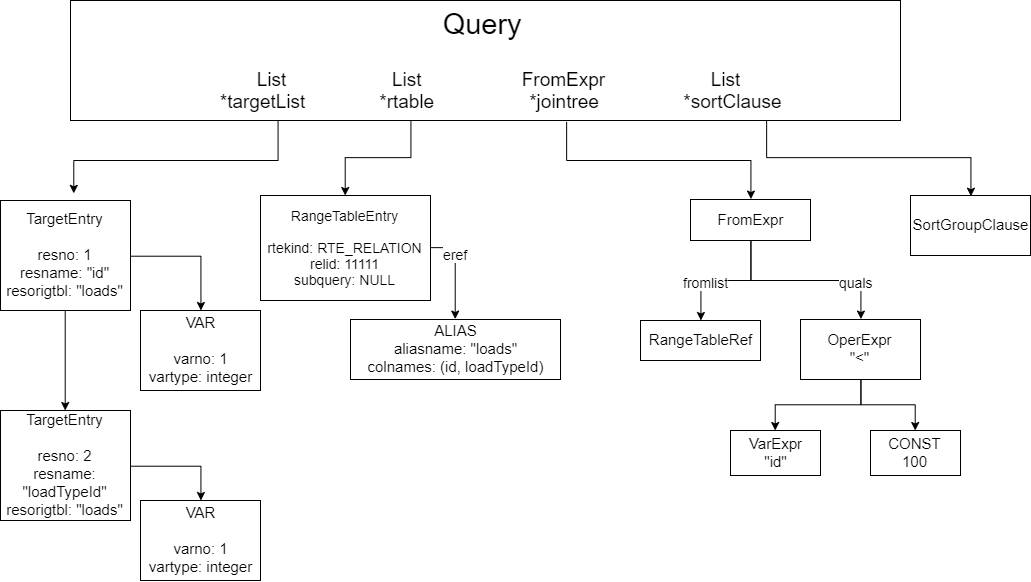
Slika 6 - ColumnRef struktura

Sledeća je faza analiziranja i stablo upita kao izlaz. Koren stabla upita je *Query* struktura koja izgleda ovako:



Slika 7 - Query struktura

Što se samog stabla tiče evo kako ono izgleda:



Slika 8 - Stablo upita

* *TargetList* predstavlja listu kolona koje su rezultati upita. Za upit koji sam uzeo kao primer to su kolone “id” i “loadTypeId”. Da je umesto eksplicitno navedenih kolona u upitu \*, ona bi bila zamenjena listom svih kolona te tabele.
* *rTable* koji je tipa *RangeTableEntry* predstavlja listu tabela koje se obrađuju. Tu su upisane infromacije kao što su jedinstveni id tabele i njeno ime
* *joinTree* sadrži informacije iz FROM i WHERE dela upita
* *sortClause* je pokazivač na listu *SortGroupClause* struktura kojom se predstavljaju ORDER BY, GROUP BY, PARTITION BY, DISTINCT, DISTINCT ON ITEMS.

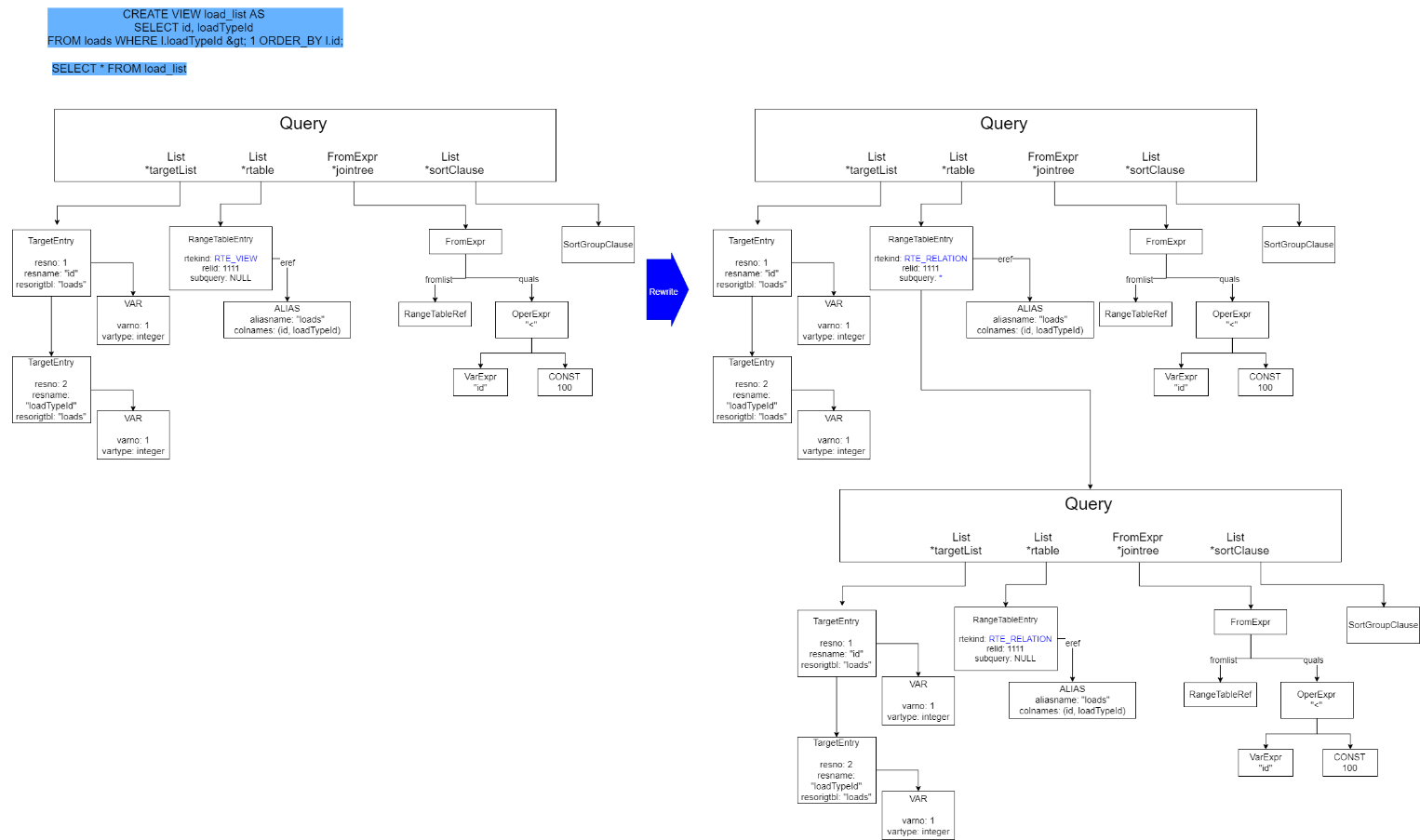
Sledeća je faza prepisivanja, odnosno prerađivanja upita, tokom koje se primenjuju definisana pravila i pogledi, i stablo upita se prerađuje. Obzirom da se upit koji sam dao kao primer vrši nad tabelom *loads,* biće kreiran i pogled nad poljima iz te tabele kako bi bio demonstriran rad prerađivača upita. Pogled je kreiran na sledeći način:

CREATE VIEW load\_list AS

SELECT id, loadTypeId

FROM loads WHERE l.loadTypeId > 1 ORDER\_BY id;

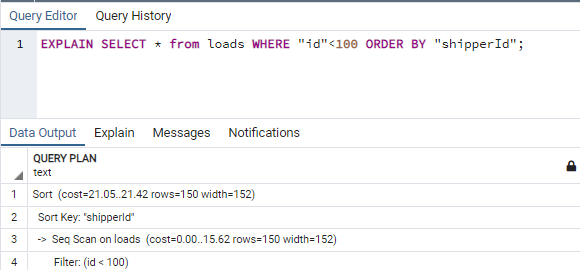
Nakon toga je izvršen upit :SELECT \* FROM load\_list;



Slika 9 - Stablo upita nakon prerađivanja

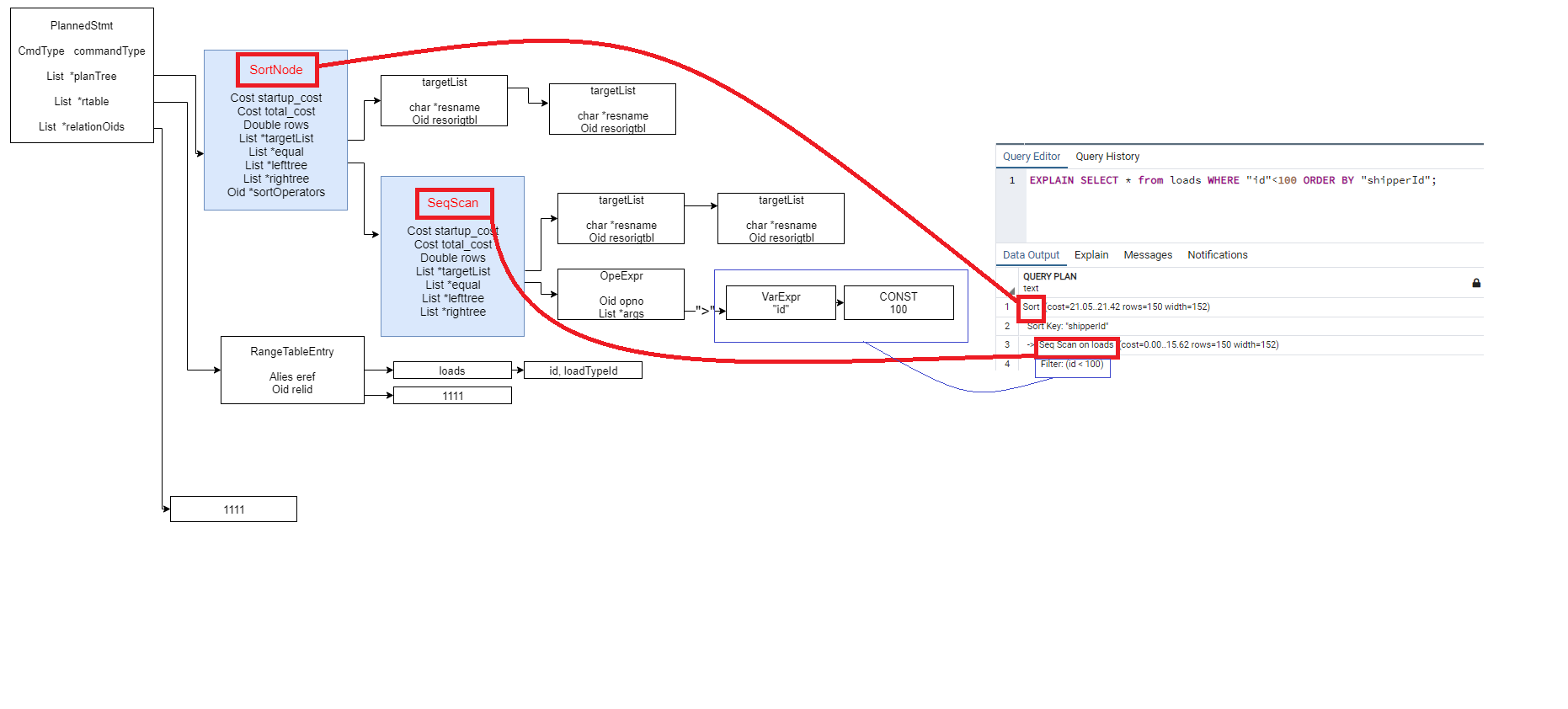
Ovde možemo videti kako se čvor *RangeTableEntry* obrađuje i od njega kreira dodatni pod-upit. Ono što ovde treba pomenuti je da je ostatak stabla isti, osim što se deo koji se odnosi na pogled „razvija“ na ceo novi upit.

Poslednje dve faze su planiranje i izršavanje upita. Planer prima stablo parsiranja koje mu je prosleđeno nakon faze prerađivnja upita, i generiše stablo planiranja koje izršilac može da obradi. Ako upit koji je dat kao primer bude analiziran sa EXPLAIN može se videti rezultat kao na sledećoj slici:



Slika 10 - Primer upita analiziranog sa EXPLAIN

Da bismo videli kako se plan realizuje tokom izvršenja, uporedićemo ovaj rezultat sa stablom plana.



Slika 11 - Stablo plana izvršavanja u poređenju sa analiziranim upitom

Koren ovog stabla je *PlannedStmt* struktura, dok se čvorovi staba, tzv. *Plan nodes*, čuvaju u *planTree* listi. Svaki od čvorova stabla sadrži sve informacije potrebne za izvršavanje upita. Izvršavanje se obavlja počeći od listova ka korenu stabla. U zavisnosti od samog upita stablo će biti popunjeno različitim vrstama čvorova. Za primer koji pratimo to su čvorovi *SortNode* i *SeqScan,* prvi zadužen za sortiranje, drugi za pretraživanje. U ovom slučaju prvo će se izvršiti sekvencijalno skeniranje tabele, a zatim i sortiranje.

Deo sa analaizom obrade složenijih upita počinjemo primerom :

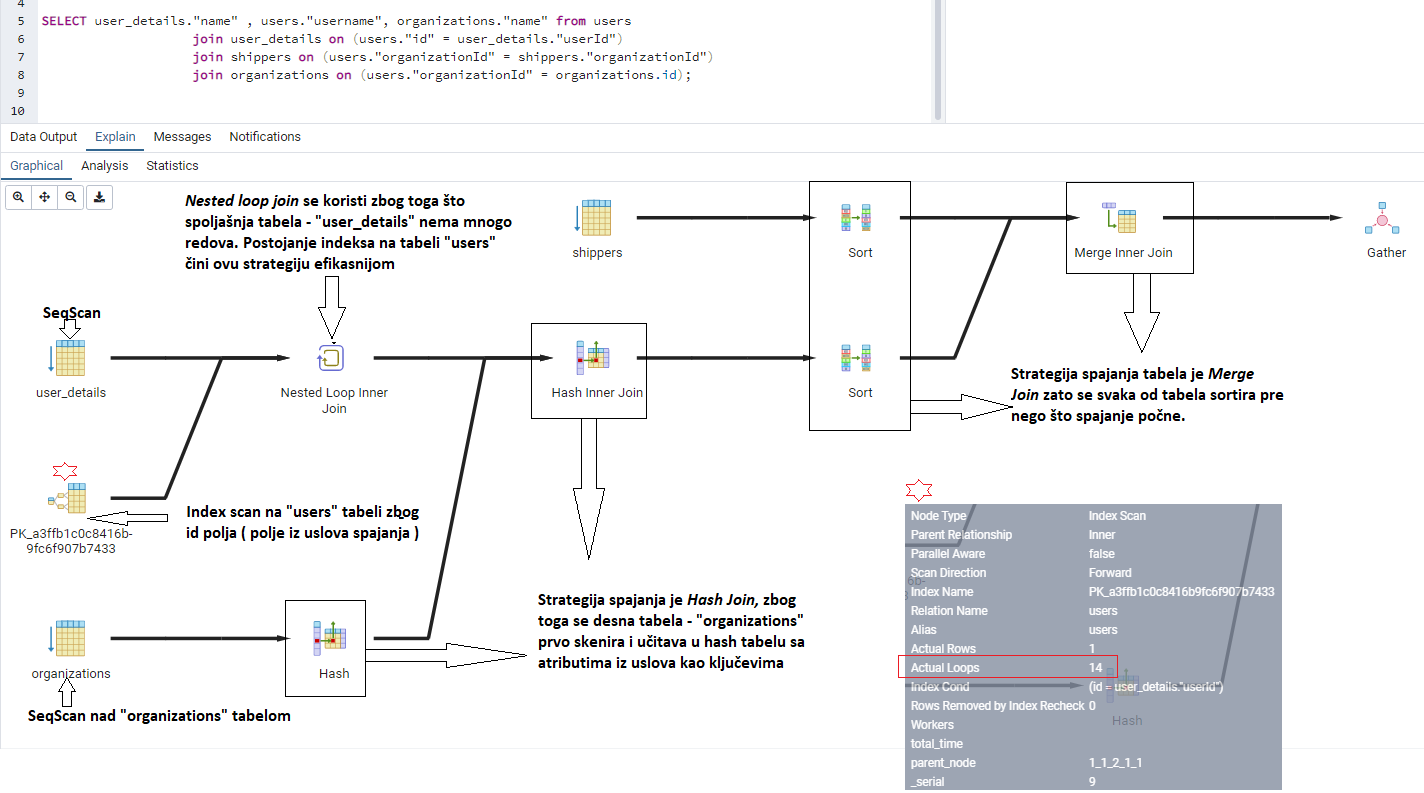
*SELECT user\_details.”name”, users.”username”, organization.”name”*

*from users*

*join user\_details on ( users.id = user\_details.”userId” )*

*join shippers on ( users.”organizationId” = shippers.”organizationId”*

*join organizations on ( users.”organizationsId” = organizations.id ) ;*

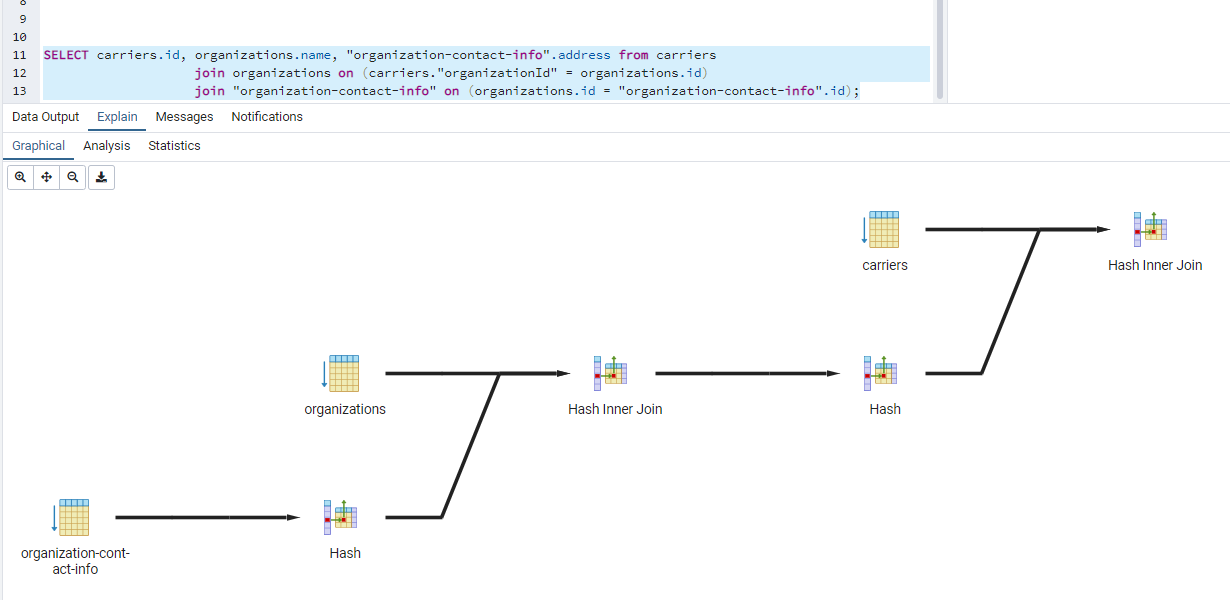


Slika - Grafička analiza upita

Na grafičkoj analizi upita koja se vidi na slici bitni je sledeće :

* tabele “organizations”, “user\_details” i “shippers” se pretražuju pomoću sekvencijalnog skeniranja
* tabela “users” se pretražuje indeksnim skeniranjem zbog podignutog indeksa nad poljem “id” koji je i uslov spajanja
* za spajanje tabela “users” i “user\_details” planer je izabrao strategiju *Nested loop join.* Ova strategija je izabrana zbog toga što spoljašnja tabela “user\_details” ima mali broj redova i zbog toga što je nad unutrašnjom tabelom podignut indeks što će ovo spajanje učiniti daleko efikasnijim. Sa slike se može videti da je broj prolaza kroz tabelu “users” zapravo *14,* iako ima preko 700.000 redova
* sa dve prethodno spojene tabele “organizations” se spaja pomoću *Hash Join* strategije, pa se zbog toga ona prvo skenira i učitava u *hash* tabelu, sa poljima iz uslova za spajanje (u ovom slučaju *id*) kao *hash ključem.*  Razlog za odabir Hash Join strategije je taj što se kreirana hash tabela može smestiti u ***work\_mem***, što je maksimalna količina memorije koju jedna operacije može da iskoristim pre nego podaci počnu da se upisuju na disk. Upis hash tabele na disk bi doveo do pogoršanja u velikoj meri
* na kraju, prethodno spojene tabele se sa “shippers” spajaju korišćenjem *Merge join*. Razlog je što “shippers” ima mnogo redova i hash tabela ne bi stala u work\_mem. Ono što je bitno je da se pre spajanja obe tabele sortiraju, kao što se to na slici vidi

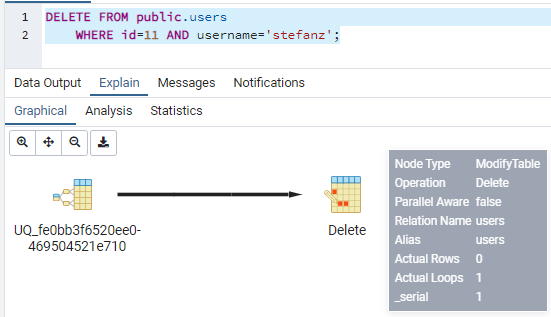
Razlog zbog koga je baš ovaj upit obrađen kao primer, pored njegove složenosti, je upravo taj što se koriste sve moguće strategije spajanja tabela. Ako za primer uzmemo upit u kome ni jedna od tabela koje se spajaju nema veliki broj redova, dobićemo ovakakv rezultat:



Slika - Primer upita sa tabelama sa malim brojem redova

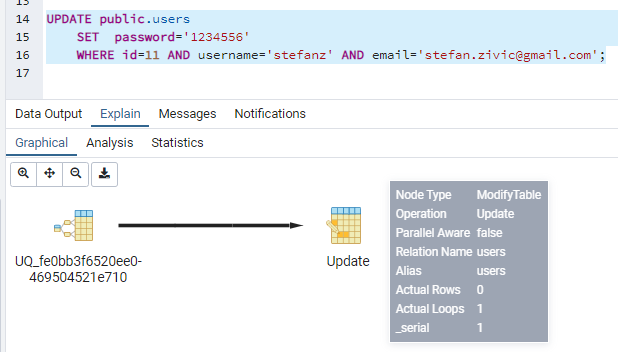
Ovde možemo da vidimo da se oba spajanja rade pomoću *Hash join* strategije, jer nema mnogo redova, hash tabele staju u work\_mem pa ova strategija daje najoptimalnije rezultate.

U oba prethodna primera smo zbog prirode upita videli samo čvorove za sortiranje i spajanje. Sa druge strane kod upita koji sadrže UPDATE ili DELETE sa tabelama radi čvor *ModifyTable* . To se može videti i na primeru *delete* upita :



Slika - Delete upit i modify table čvor

kao i na primeru *update* upita :



Slika - Update upit i modify table čvor

Da bismo na primeru videli i ostale vrste čvorova pri obadi upita, kao i to kako izgleda plan, za primer ćemo uzeti upit koji vraća 3 lokacije sa najvećom latitudom :

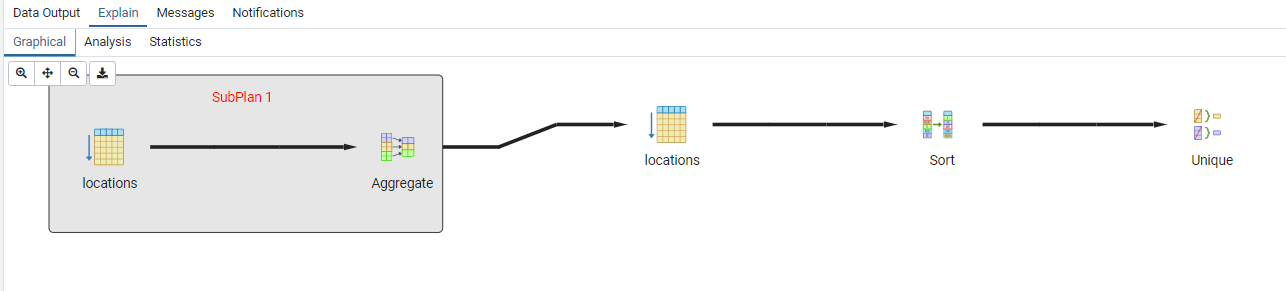
*select distinct latitude from locations a*

*where 3 >= (select count(distinct latitude) from locations b*

*where a.latitude <= b.latitude)*

*order by a.latitude desc;*

Kada se ovaj upit izvrši sa analyze i uključi grafički prikaz dobijamo:

**

Ono što treba istaći u vezi sa ovim primerom je to je kreiran “SubPlan” koji se odnosi na deo *(select count(distinct latitude) from locations b where a.latitude <= b.latitude)* jer će se on izvršiti više puta. Sort se radi metodom *quick sort,* a čvor koji se nalazi na kraju *Unique* je tu zbog *distinct* dela upita.

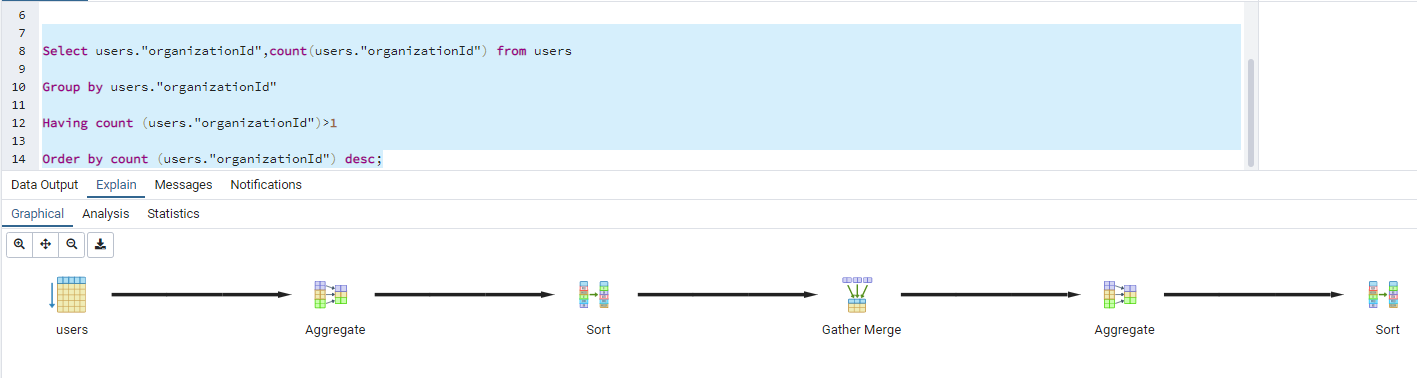
Naredni primer koji ćemo obraditi je upit koji koristi GroupBy i spada u grupu ***paralelnih*** upita:

*Select users.”organizationId”, count(user.”organizationsId”) from users*

*GROUP BY users.”organizationId”*

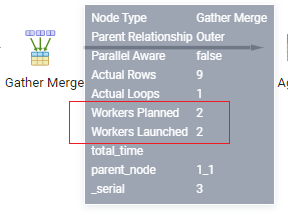
*HAVING count (users.”organizationId”) > 1*

*Order by count ( users.”organizationId” ) desc;*



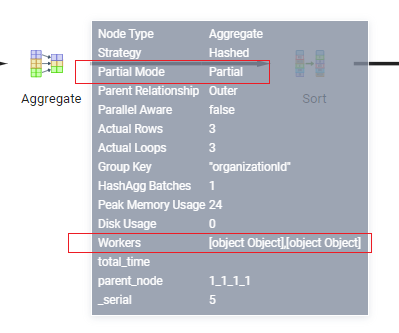
Slika - Analiza upita sa GROUB BY i HAVING klauzulama

Ovde vidimo da u izvršenju upita učestvuje čvor *Gather Merge.* Naime, kada optimizator odluči da je paralelni upit najbolja strategija izvršenja za određeni upit, kreiraće plan koji uključuje *Gather* ili *GatherMerge* čvor. Ova dva čvora uvek imaju samo jedan „dete plan (potplan)“, i to je deo plana koji će se paralelno izvršiti. Ako je Gather ili GatherMerge na vrhu plana, ceo upit će se paralelno izvršiti, a ako je negde unutar stabla, onda se samo deo „ispod“ njega paralelno izvršava. Kada prikažemo informacije o GatherMerge čvoru sa primera možemo videti, između ostalog, koliko radnih niti je planirano i koliko njih je iskorišćeno:



Slika - GatherMerge čvor

Na primeru možemo videti da se *Aggregate* čvor u stablu nalazi ispod GatherMerge čvora, tako da će se i on izvršiti paralelno. To je prikazano i u informacijama o ovom čvoru :



Slika - Aggregate čvor koji se paralelno izvršava

Vidimo da je *mode* izvršavanja *partial* i da ima dva *workers* objekta koji rade agregaciju.

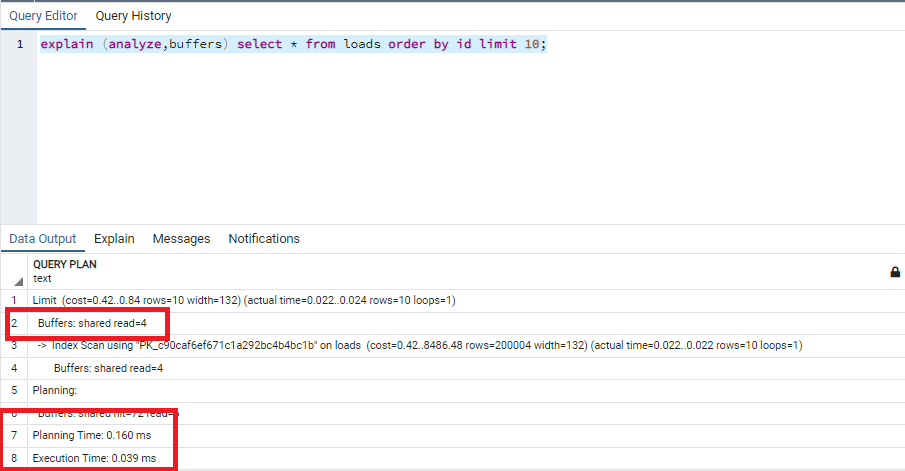
Na kraju, obradićemo primere u kojima se analizira keširanje podataka. Kada analiziramo izvršenje upita pomoću EXPLAIN naredbe, između ostalih podataka, možemo videti i koliko blokova podataka dobijamo sa diska, a koliko iz shared buffers keš memorije. Za primer ćemo uzeti upit koji pribavlja prvih 10 redova iz “loads” tabele :

*select \* from loads order by id limit 10;*

Ovaj upit će biti analiziran tako da će biti vidljiva osnovna statistika, vreme izvršenja, kao i informacije o korišćenju keš memorije u vidu broja blokova podataka koji dolaze iz keš memorije i  sa diska - kada je u pitanju čitanje, ili broj blokova upisanih u keš i na disk - kod upisa. Ovo se postiže dodavanjem parametara *ANALYZE* i *BUFFERS* u naredbu explain.

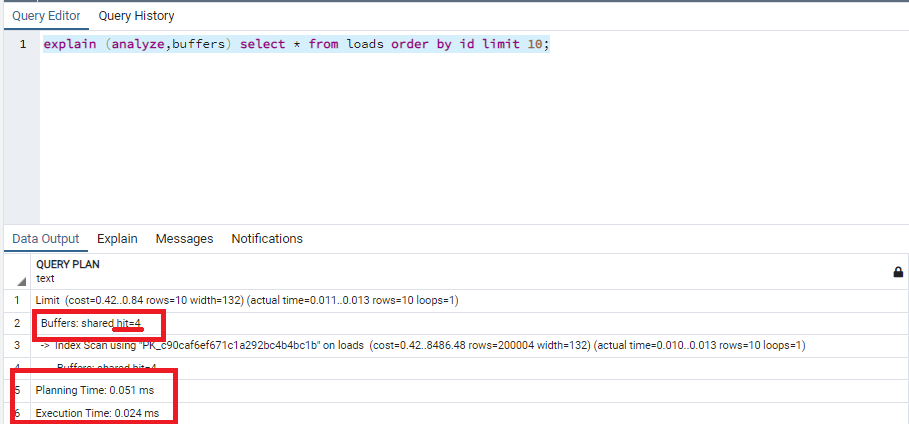
*explain (analyze,buffers) select \* from loads order by id limit 10;*

Kao rezultat dobijamo :



Slika 14 - Primer upita sa analiziranom upotrebom keš memorije

*Shared read* znači da je blok podataka dobijen sa diska, a ne iz keš memorije. Kao što se vidi na primeru, sve podatke smo dobili sa diska( 4 bloka) i vreme izvršenja upita je 0.039ms. Ako je keš konfigurisan kako treba( što bi trebalo da je uvek tačno) i isti upit pokrenemo ponovo redovi bi trebalo da se nalaze u keš memoriji i trebalo bi da umesto shared read dobijemo *shared hit*. To bi značilo da su podaci pročitani iz keš memorije. Kada se upit pokrene po drugi put dobija se sledeći rezultat:



Slika 15 - Primer upita sa analiziranom upotrebom keš memorije 2

Pre svega, možemo da potvrdimo da su podaci pročitani iz keš memorije jer sada imamo 4 bloka u shared HIT, a nijedan u shared read. Ono što se takođe može videti je da je vreme izvršavanja upita sada 0.024ms, što je skoro 40% manje. Na ovako malom uzorku od 10 redova to predstavlja jako veliki faktor ubrzanja i govori nam koliko je keš memorija bitna kada se radi o performansama.

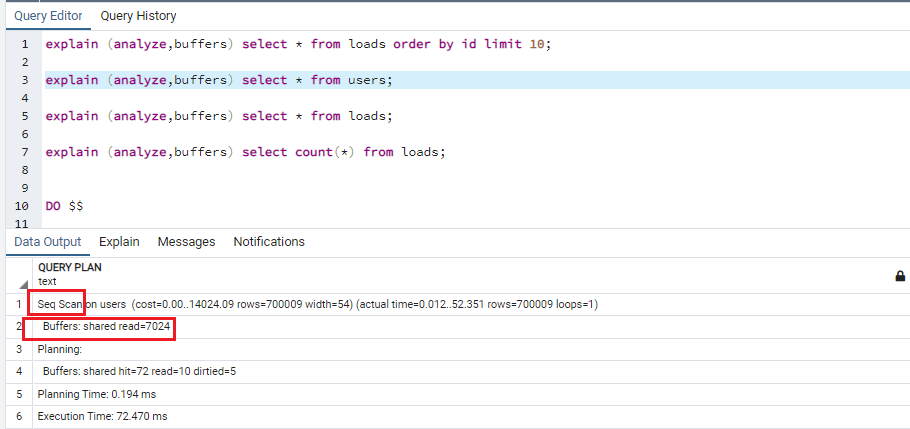
Sekvencijalno skeniranje( čitanje) podataka koje se dešava kada nema podignutog indeksa nad određenom tabelom i svi podaci se čitaju sa diska predstavlja ozbiljan problem ako je keš memorija organizovana na način da koristi algoritme koje smo videli do sada. Obzirom da jedno takvo skeniranje može da prebriše čitavu keš memoriju, ako se radi nad tabelom sa velikom brojem redova, takvi slučajevi se u Postgres-u tretiraju drugačije. Iz tog razloga ćemo primer takvog upita obraditi posebno.

Naime, kod sekvencijalnog skeniranja je veoma mala verovatnoća da će se stranici podataka koja se trenutno obrađuje opet pristupiti. Zbog toga se umesto uobičajenih *clock sweep* i *last recently used* algoritama koji bi prepisali celu keš memoriju, alocira serija bafera podataka, organizovanih tako da čine prsten – *ring buffer,* ukupne veličine *256KB* koji se koriste se čitanje podataka. Svaki od njih je dovoljno mali da može da stane u L2 nivo keš memorije što prebacivanje podataka iz keša operativnog sistema i shared buffers keš čini efikasnim.

Kako ovaj mehanizam radi demonstiraćemo na primeru pribavljanja svih redova iz tabele “users”. Ono što je bitno napomenuti je da ova tabela sadrži preko 500.000 redova, što je čini pogodnom za demonstraciju procesa o kome smo pričali. Pokretanjem upita:

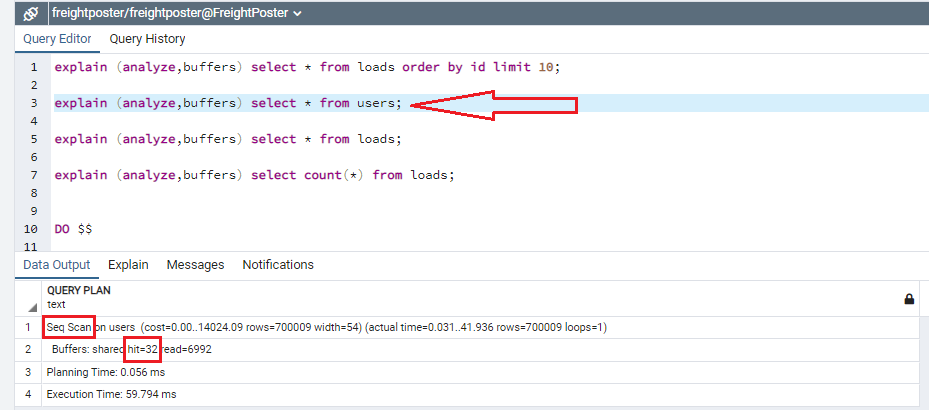
*explain (analyze,buffers) select \* from users;*

dobijamo:



Slika 16 - Primer upita koji koristi sekvencijalno skeniranje sa analiziranom upotrebom keš memorije

U rezultatu se može videti da je u pitanju sekvencijalno pretraživanje - *seq scan,* kao i to da su svi podaci pročitani sa diska. Kada se isti upit pokrene još jednom, dobija se sledeće:



Slika 17 - Primer upita koji koristi sekvencijalno skeniranje sa analiziranom upotrebom keš memorije 2

Sada vidimo da je shared hit iznosi 32. To je zbog toga što su stranice *8KB,* tako da je onda

*32 \* 8KB = 256KB,*

koliko je i veličina jednog bafera.

# Zaključak

Obrada upita predstavlja jedan od najvažnijih, ako ne i najvažniji koncept rada sa bazom podataka. Ovaj proces je komplikovan, sastoji se iz faza, zavisi od mnogo faktora i nije ga lako u potpunosti razumeti. Svaki od podsistema koji učestvuje u obradi je kompleksan i zahteva da se posebno prouči kako funkcioniše. Međutim, što bolje poznajemo kako obrada funckioniše to ćemo biti u boljoj poziciji za rad sa bazom. Znati šta se sve dešava u pozadini kada se neki upit izvršava može pomoći u lociranju greški i problema, optimizaciji, odnosno pisanju upita koji će imati bolje performanse, nalaženju boljih rešenja kod pisanja komplikovanih upita itd.

Što se PostgreSQL-a tiče, uz dostupnu dokumentaciju koja objašnjava teorijski deo obrade, programe za upravljanje bazom poput pgAdmin-a, dodatne alate za grafički prikaz, mogućnost pregleda interne statistike koju Postgres vodi, proces obrade upita postoje mnogo razumljiviji

# Literatura

[1] Ibrar Ahmed, Gregory Smith, Enrico Pirozzi , “PostgreSQL 10 High Performance Expert techniques for query optimization, high availability, and efficient database maintenance”, Packt Publishing, 2018

[2] Hans-Jurgen Schonig, “Mastering PostgreSQL 11 Expert techniques to build scalable, reliable, and fault-tolerant database applications (Secon edition) “, Packt publishing, October 2018

[https://www.postgresql.org/docs/current/overview.html](https://www.postgresql.org/docs/current/overview.html%20)

[3] An overview of Caching for PostgreSQL [https://severalnines.com/database-blog/overview-caching-postgresql](https://severalnines.com/database-blog/overview-caching-postgresql%20) [poslednji pristup: 11.04.2021.]

[4] Understanding caching in PostgreSQL [https://madusudanan.com/blog/understanding-postgres-caching-in-depth/#Contents](https://madusudanan.com/blog/understanding-postgres-caching-in-depth/%23Contents%20) [poslednji pristup: 11.04.2021.]

[5] How PostgreSQL 12 handled prepared plans [https://www.cybertec-postgresql.com/en/tech-preview-how-postgresql-12-handles-prepared-plans/](https://www.cybertec-postgresql.com/en/tech-preview-how-postgresql-12-handles-prepared-plans/%20) [poslednji pristup: 09.04.2021.]

[6] Join strategies and performance in PostgreSQL <https://www.cybertec-postgresql.com/en/join-strategies-and-performance-in-postgresql/> [poslednji pristup: 12.04.2021.]

[7] Query processing <http://www.interdb.jp/pg/pgsql03.html> [poslednji pristup: 08.04.2021.]

## [8] Overview of PostgreSQL Internals

[https://www.postgresql.org/docs/current/overview.html](https://www.postgresql.org/docs/current/overview.html%20) [poslednji pristup: 12.04.2021.]

[9] Salahaldin Juba, Andrey Volkov, “Learning PostgreSQL 11 ( Third edition) “, Packt publishing, January 2019

[10] Execution plan <http://postgresguide.com/performance/explain.html> [poslednji pristup: 11.04.2021.]

1. ne mora nužno da kreira ispočetka, može samo da “probudi” proces koji je već prethodno kreiran i koji je najverovatnije već opslužio klijenta i sada se može povezati na novi klijentski proces [↑](#footnote-ref-1)