UNIVERZITET U NIŠU ELEKTRONSKI FAKULTET

Milan Stanković

**SEMINARSKI RAD**

**Obrada i optimizacija upita kod PostgreSQL baze podataka**

Sistemi za upravljanje bazama podataka

|  |  |
| --- | --- |
| Student: | Mentor: |
| Milan Stanković, br. ind. 1276 | Prof. dr Aleksandar Stanimirović |

Niš, 2022. god.

Sadržaj

Contents

[1. Uvod 3](#_Toc101985162)

[2. Obrada upita kod PostgreSQL 4](#_Toc101985163)

[2.1. Parser 5](#_Toc101985166)

[2.2. Analizator (Semantički analizator/Analzyer) 6](#_Toc101985167)

[2.3. Prepisivanje ( Transformacije / Rewritting) 6](#_Toc101985172)

[2.4. Planiranje (Planning) 7](#_Toc101985178)

[2.5. Izvršenje upita (Execution) 11](#_Toc101985185)

[3. Optimizacija upita 14](#_Toc101985186)

[3.1. Metode preuzimanja podataka 14](#_Toc101985190)

[Algoritam potpuno skeniranje (Full Scan) 15](#_Toc101985191)

[Algoritam pristupa podacima zasnovan na indeksu (Index based table access) 15](#_Toc101985192)

[Skeniranja indeksa (Index-Only scan) 16](#_Toc101985193)

[3.2. Algoritmi spajanja 18](#_Toc101985194)

[Strategija pridruživanja ugnežđene petlje (Nested Loops JOIN) 18](#_Toc101985195)

[Strategija heš pridruživanja (Hash Join) 19](#_Toc101985196)

[Strategija Marge pridruživanje (Marge Join) 20](#_Toc101985197)

[Primer izbora strategije za spajanje relacija 21](#_Toc101985198)

[4. Zaključak 23](#_Toc101985199)

[5. Literatura 24](#_Toc101985200)

# Uvod

Glavni cilj stvaranja baze podata je skladištenje, pristup i obrada povezanih podataka na jednom mestu. Pristup podacima od strane korisnika, njihova obrada i manipulacija nad tim podacima trebalo bi da se obavlja što efikasnije, odnosno potrebno je obezbedeti koriscnima što jednostavniji i brži pristup željenim podacima.

Sistem za upravljanje bazama podataka (eng. Database Management System) ili DBMS  
je program ili skup programa kojim se vrši rukovođenje bazom podataka, čuvanje velike količine podataka i izvršavanjem upita nad podacima koje zahtevaju krajnji korisnici. Upit (eng.query) je zahtev za informacijama iz baze podataka i za pisanje različitih upita koristi se SQL jezik. SQL je jezik visokog nivoa stvoren za izgradnju mosta između korisnika i DBMS-a odnosno omogućava njihovu komunikaciju.

Obrada upita i optimizacija su osnovni, ako ne i kritični delovi bilo kog DBMS-a. Osnovni sistemi u DMBS-u ne razumeju SQL jezik, koji korisnik koristi za pisanje upita, već kao posledica toga mora postojati jezik na niskom nivou koji ovi sistemi mogu da razumeju kako bi izvršili željene akcije. Upit napisan korišćenjem SQL-a se pretvara u jezik niskog nivoa koji sistetem može razumeti. Korisnik piše upite koristeći SQL jeziku, zatim se proverava kod koji je napsao koristik od strane samog sistema baze, a onda se vrši pretvaranje jezika visokog nivoa u jezik niskog nivoa koji DMBS razume. DMBS bira najbolju putanju izvršenja, izvršava upit i dostavlja željene podatke korisniku. Ovaj proces koje sprovodi baza podataka naziva se obrada upita.

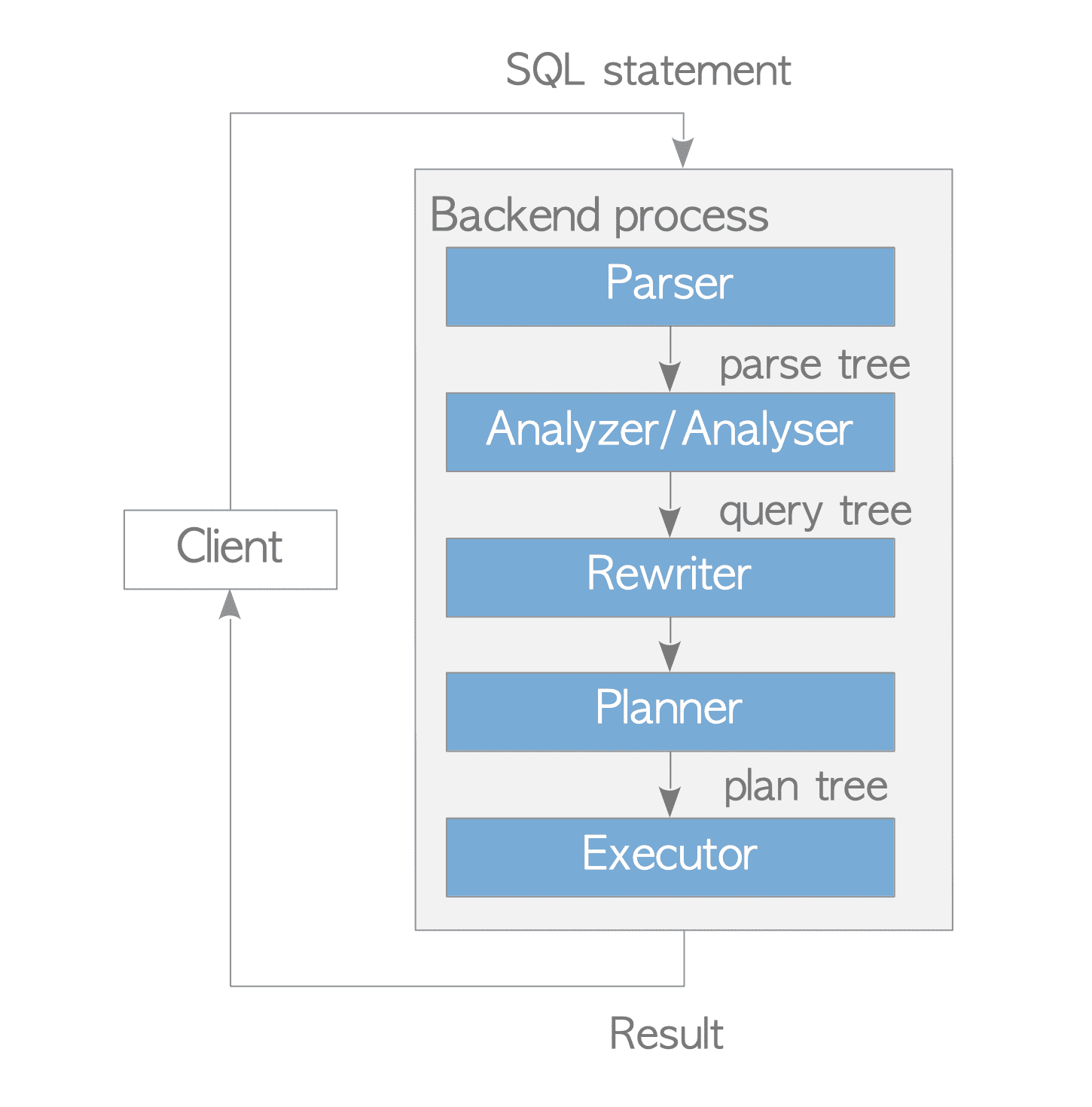
U okviru ovog rada biće opisan koncept obrade upita, faze kroz koje prolazi upit prilikom obrade, kao i načini optimizacije tih upita kod sistema za upravljanje relacionim bazama podataka - PostgreSQL .

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. Odlikuje se dobrim karakteristikama performansi i pouzdanošću koju pruža svojim korisnicima. PostgreSQL pruža jedan od najboljih mehanizama za optimizaciju i obradu upita u odnosu na druge sisteme otvorenog koda i komercijalne sisteme sličnih namena, zbog čega je zapravo i izabran kao osnova izvornog koda za razvoj pojedinih delova nekih komercijalnih sistema baza.

# Obrada upita kod PostgreSQL

Sama obrada upita može se predstaviti kao proces gde se korak po korak vriši razbijanje jezika visokog nivoa u jezik niskog nivoa u kojem mašina može razumeti i izvršiti traženu radnju za korisnika. Kada se upit pošalje bazi podataka, prima ga kompajler upita. Zatim skenira upit I deli ga na pojedinačne tokene. Jednom kada su tokeni generisani, analizira ih i veritifikuje njihovu ispravnost. Tada su tokenizirani upiti transofmisani u razliite moguće relacijske izraze, relacijska stabla i relacijski grafovi (planovi upita). Optimizator upita ih zatim bira kako bi identifikovao najbolji plan upita za obradu. Proverava se sistemski katalog za ograničenja i indekse, zatim se odlučuje o najboljem planu upita. Generiše različite planove izvršenja za plan upita. Plan izvršenja upita tada odlučuje o najboljem i optimizovanom planu izvršenja koji se treba izvršiti. Procesor naredbi tada koristi ovaj plan izvršenja za prikupljanje podataka iz baze podataka i vraća rezultat korisniku. Ovo je način na koji tipičnna obrada upita kod DBMS-a funkcioniše. Dakle u opštem slučaju obrade upita u DMBS-u postoje četri faze prilikom same obrade upita, a to su: Raščlanjivanje i prevod, optimizacija upita, procena ili generisanje koda upita i zvršenje u DB runtime procesoru.

Osnovna svrha PostgreSQL klijent-server protokola je dvostruka: on šalje SQL upite serveru i kao odgovor prima ceo rezultat izvršenja. Upit koji server primi za izvršenje prolazi kroz nekoliko faza. Celokupan put od slanja upita do vraćanja rezultata kroz faze prikazan je na sledećoj slici.



Slika 1 – faze obrade upita kod PostgreSQL

Kao što je prikazano na slici 1. vidimo da se sam pozadinski proces obrade upita može podeliti u pet podsistema: parser, analizator, prepisivač (transformacije), planer i izvršilac.



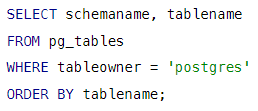
## Parser

Nakon što se prosledi upit od strane klijenta, prvo se izvršava parsiranje tog upita, tako da server razume šta tačno treba da odradi. Komponente koje se koriste u ovoj fazi su:

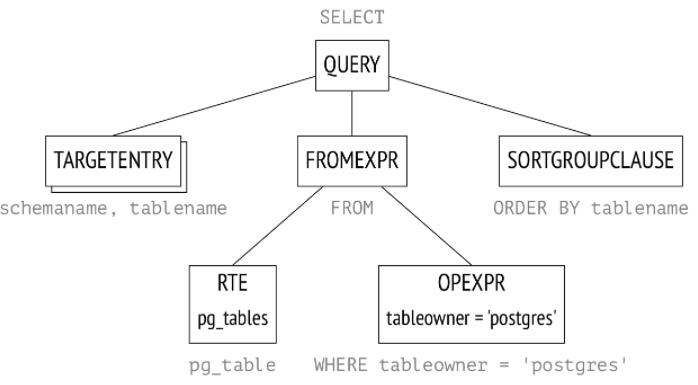
* + Lexer - vrši leksičku analizu, odnosno odgovoran je za prepoznavanje leksema u nizu upita (kao što su SQL ključne reči, string, numerički literali)
  + Parser - stara se da je rezultujući skup leksema gramatički validan

Parser generiše stablo raščlanjivanja koje mogu da čitaju naredni podsistemi iz SQL naredbe u običnom tekstu.

Primer upita:



Na osnovu ovog upita biće prikazano kreirano sintaksno stablo generisano od strane parsera na slici ispod. Čvorovi upita označeni su odgovarajućim delovima upita.



Slika 2 - Primer sintaksnog stabla

Kod stabla generisanog pri parsiranju, koren čvor je SelectStm struktura definisana u parsenodes.h. RTE je skraćenica koja označava „Unos tabele opsega“. Naziv „tabela opsega“ u PostgreSQL izvornom kodu odnosi se na tabele, podupite, rezultate spajanja.

Zbog činjenice da parser samo proverava sintaksu ulaza prilikom generisanja stabla parsiranja, ona vraća samo grešku ako postoji greška u sintaksi u upitu.  
Parser ne proverava semantiku ulaznog upita. Na primer, čak i ako upit sadrži ime tabele koje ne postoji, parser ne vraća grešku. Semantičke provere vrši se analizatorom / analizatorom.

## Analizator (Semantički analizator/Analzyer)

Semantički analizator utvrđuje da li u bazi podataka postoje tabele i drugi objekti na koje se upit odnosi po imenu i da li korisnik ima pravo pristupa ovim objektima. Sve informacije potrebne za semantičku analizu se čuvaju u katalogu sistema.

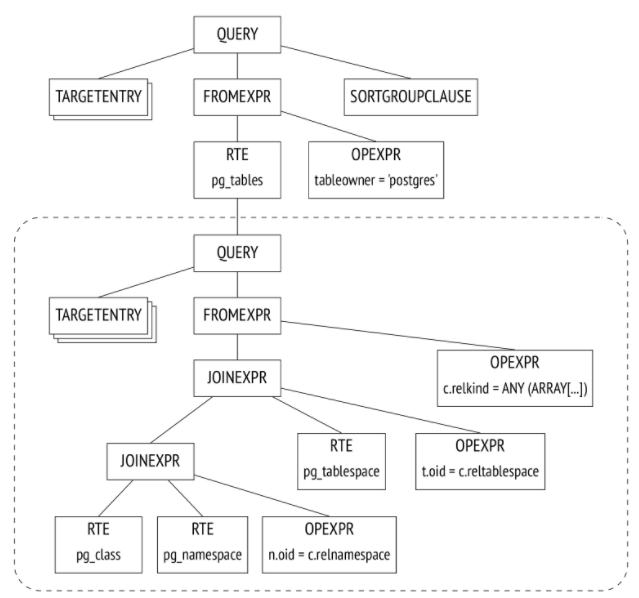
Semantički analizator prima stablo raščlanjivanja od parsera i ponovo ga gradi, dopunjujući ga referencama na specifične objekte baze podataka, informacijama o tipu podataka itd.



## Prepisivanje ( Transformacije / Rewritting)

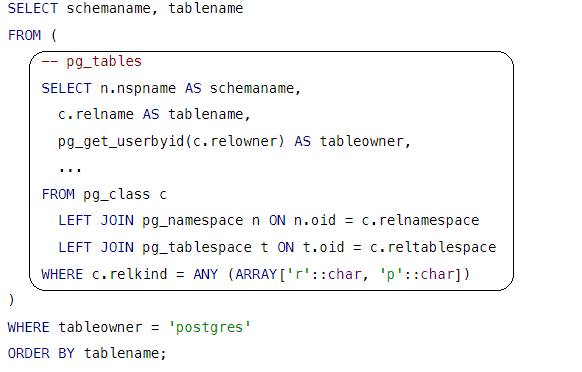
Transformacije koriste jezgro sistema u nekoliko namena. Jedna od njih je zamena imena pogleda (View) iz stabla rasčlanjivanja sa podstablom koje odgovara upitu ovog pogleda.

Za primer koji je naveden iznad pogled je *pg\_tables,* nakon transformacije stablo rasčlanjivanja izgledaće kao na slici ispod:



Slika 3 – stablo rasčlanjivanja nakon transofrmacije

Ovo stablo rasčlanjivanja odgovara sledećem upitu (iako se sva manipulacija vrši na samom stablu rasčlanjivanja, ne i na tekst upita):

  
Kada se izda upit koji sadrži pogled, parser kreira stablo raščlanjivanja na osnovu pravila za taj pogled koje je bilo generisano i skladišteno u katalog prilikom kreiranja tog pogleda naredbom Create View. Stablo rasčlanjivanja održava sintaksnu strukturu upita, ali ne i redosled kojim će se operacije izvršiti .

Dugi primer upotrebe transformacija od strane jezgra sistema je implementiranje klauzula Search i Cycle za rekurzivne upite. PostgreSQL takođe podržava prilagođene transformacije koje korisnik može sam napisati korišćenjem *rewrite rule system*.



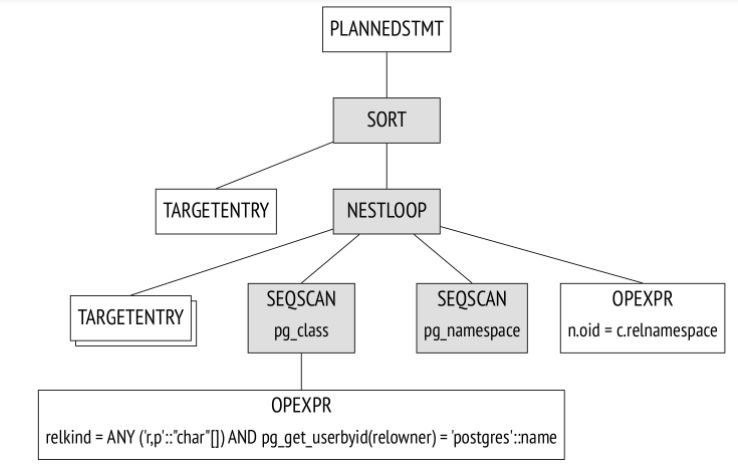
## Planiranje (Planning)

Planer prima stablo kreirano u okviru prethodnog podsistema (Rewriter) i generiše stablo plana (upita) koje se najefikasnije može obraditi od strane izvršioca (Executor-a).

Bilo koji upit se može izvršiti na više načina. Svaka operacija u stablu raščlanjivanja ima više opcija za izvršenje. Na primer, možete da preuzmete određene zapise iz tabele tako što ćete pročitati celu tabelu i odbaciti redove koji vam nisu potrebni, ili možete koristiti indekse da pronađete redove koji odgovaraju vašem upitu. Skupovi podataka se uvek spajaju u parove. Varijacije u redosledu spajanja rezultiraju mnoštvom opcija izvršenja. Zatim postoje različiti načini da spojite dva seta redova zajedno. Na primer, možete da prođete kroz redove u prvom skupu jedan po jedan i potražite odgovarajuće redove u drugom skupu, ili možete prvo da sortirate oba skupa, a zatim ih spojite. Različiti pristupi rade bolje u nekim slučajevima i lošije u drugim.

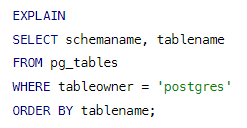
Optimalni plan može da se izvrši brže od neoptimalnog za nekoliko redova veličine. Zbog toga je planer, koji optimizuje raščlanjeni upit, jedan od najsloženijih elemenata sistema u RDMBS-u.

Plan izvršenja se takođe može predstaviti kao stablo, ali sa svojim čvorovima kao fizičkim, a ne kao logičkim operacijama nad podacima.. Primer stabla planiranja biće prikazan na sledećoj slici:

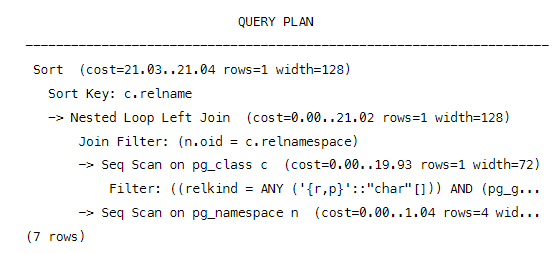


Slika 4 – primer stabla planiranja

Analiza planiranja može se pored stabla planiranja obaviti i korišćenjem komande EXPLAIN, na sledeći način:



Pri čemu će se kao rezultat upita dobiti plan izvršenja upita:



Prikazana slika stabla planiranja (slika 4) sadrži glavne čvorove stabla, isti ti čvorovi su označeni strelicama u izlanom rezultatu koji je dobijen izvršenjem naredbe EXPLAIN.

Čvor Seq Scan predstavlja operaciju čitanja tabele, dok Nested Loop predstavlja operaciju spajanja. Ovde treba obratiti pažnju na dve stvari:

* Jedna od početnih tabela je nestala iz stabla plana jer je planer shvation da nije potrebno obraditi upit i uklonio ga.
* Postoji procenjeni broj redova za obradu i cena obrade pored svakog čvora.

Planer upita je ono što određuje najbolji način za procenu izvršenja upita, i takođe je poznat kao Optimizator upita (Query Optimizer).

Da bi pronašao optimalni plan, PostgreSQL koristi optimizator upita zasnovan na troškovima (*cost-based optimization*). Optimizator prolazi kroz različite dostupne planove izvršenja i procenjuje potrebne količine resursa, kao što su I/O operacije i CPU ciklusi. Ova izračunata procena, pretvorena u proizvoljne jedinice, poznata je kao trošak plana. Za izvršenje se bira plan sa najnižim rezultujućim troškovima.

Optimizator vrši dve vrste transformacija: zamenjuje logičke operacije sa njihovim algoritmima izvršenja i eventualno menja strukturu logičkog izraza promenom redosleda u kome će biti logičke operacije pogubljen.

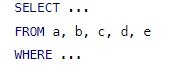
Nijedna od ovih transformacija nije jednostavna. Logička operacija može biti izračunata korišćenjem različitih algoritama, a optimizator pokušava da izabere najbolji. Isti upit može biti predstavljen sa nekoliko ekvivalentnih izraza koji proizvode isti rezultat ali zahteva znatno drugačiju količinu računarskih resursa za izvršenje. Optimizator pokušava da pronađe logički plan i fizičke operacije koji minimiziraju potrebne resurse, uključujući vreme izvršenja. Izlaz optimizatora je izraz koji sadrži fizičke operacije. Ovaj izraz se naziva (fizički) plan izvršenja. Iz tog razloga, PostgreSQL optimizator se naziva još i planer upita. Konačno, plan izvršenja upita tumači mehanizam za izvršenje upita, koji se često pominje kao izvršilac u PostgreSQL zajednici, a izlaz je vraćen u klijentsku aplikaciju.

Problem je u tome što broj mogućih planova koji se mogu izvršiti raste eksponencijalno kako se broj operacija spajanja povećava (*JOIN* operacija), a pregledavanje svih planova jedan po jedan je nemoguće čak i za relativno jednostavne upite jer bi oduzelo previše vremena i memorije. Zbog toga se dinamičko programiranje i heuristika koriste za ograničavanje obima pretrage. Ovo omogućava precizno rešavanje problema za veći broj tabela u upitu u razumnom roku, ali nije garantovano da će izabrani plan biti zaista optimalan jer planer koristi pojednostavljene matematičke modele i može koristiti neprecizne početne podatke.

Upit se može strukturirati na specifične načine kako bi se značajno smanjio obim pretrage (uz rizik da propustite priliku da pronađete optimalni plan). Uobičajeni tabelarni izrazi se obično optimizuju odvojeno od glavnog upita. Od verzije 12, ovo se može forsirati klauzulom MATERIALIZE. Upiti iz ne-SQL funkcija optimizuju odvojeno od glavnog upita. (SQL funkcije se u nekim slučajevima mogu umetnuti u glavni upit.) .

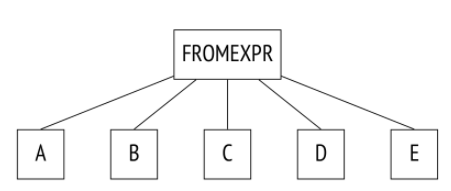
Parametar join\_collapse\_limit zajedno sa eksplicitnim JOIN klauzulama, kao i parametar from\_collapse\_limit zajedno sa podupitima mogu definisati redosled nekih spajanja, u zavisnosti od sintakse upita.

Na sledećem primeru možemo videti pojašnjenje za slučaj naveden u tekstu iznad.



Navedeni upit prikuplja podatke iz više različitih tabela koristeći klauzolu FROM, bez korišćenja eksplicitnih JOIN klauzola.

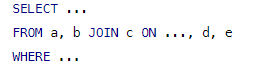
Stablo koje se dobija nakon parsiranja za ovaj upit prikazan je na sledećoj slici:



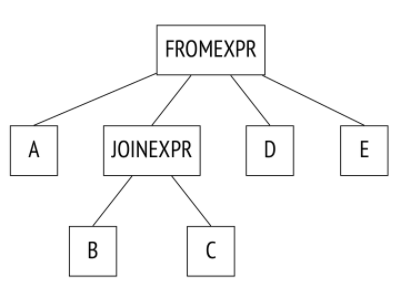
Slika 5. Stablo dobijeno kao rezultat podsistema za parsiranje

Za obaj upit planer će razmotriti sve moguće redoslede izvršenja operacije spajanja (JOIN), što će znatno uticati na rad.

Delimična optimizacija ovakvog upita može se izvršiti izmenom upita tako što će se eksplictino iskoristiti klauzola JOIN za spajanje nekih tabela, na način prikazan ispod:



Stablo parsiranja za ovaj upit bi izgledalo kao na slici ispod:



Planer skuplja stablo spajanja, efektivno ga transformišući u stablo iz prethodnog primera. Algoritam rekurzivno prelazi kroz stablo i zamenjuje svaki JOINEXPR čvor ravnom listom njegovih komponenti. Tabela B mora biti spojena sa tabelom C (ili obrnuto, redosled spajanja u paru nije ograničen). Tabele A, D, E i spoj B sa C mogu se spojiti bilo kojim redosledom, čime se umanjuje broj mogućih kombinacija redosleda izvršenja spajanja odnosno JOIN operacije.

Definicija najboljeg plana varira u zavisnosti od nameravane upotrebe. Kada je potreban kompletan izlaz, plan mora da optimizuje preuzimanje svih redova koji odgovaraju upitu. Sa druge strane, ako želite da izvučete iz baze samo prvih nekoliko podudarnih redova optimalni plan bi mogao biti potpuno drugačiji.

Dodatni načini optimizacije upita kod PostgreSQL-a biće navedeni u zasebnoj oblasti u okviru ovog rada.

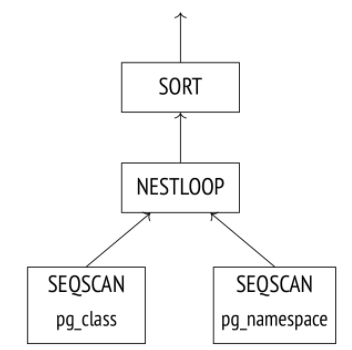


## Izvršenje upita (Execution)

Optimizovani upit kreiran od strane planera / optimizatora se prosleđuje izvršiocu (executoru) i izvršava se u skladu sa kreiranim planom.

Objekat koji se zove portal kreira se u pozadinskoj memoriji. Portal čuva stanje upita dok se izvršava. Ovo stanje je predstavljeno kao drvo, po strukturi identično stablu plana.

Čvorovi stabla deluju kao montažna linija, tražeći i isporučujući redove jedni drugima.



Slika 6 – primer plan stabla koje se treba izvršiti

Izvršenje počinje od osnovnog (root) čvora koji se rekurzivno obrađuje sve dok se ne obrade njegovi podplanovi tj. podčvorovi i isporuče se odgovarajući redovi. Svaki podčvor ukoliko sadrži sopstvene podčvorove postaje za njih osnovni čvor.

Za stablo prikazano iznad osnovni čvor (u ovom primeru to je čvor za sortiranje SORT ) zahteva podatke od podređenog čvora. Kada primi sve tražene podatke, vrši operaciju sortiranja, a zatim isporučuje podatke naviše, do klijenta.

Neki čvorovi (kao što je čvor NESTLOOP u ovom primeru) pridružuju podatke iz različitih izvora. Ovaj čvor zahteva podatke od dva podređena čvora. Nakon što primi dva reda koja odgovaraju uslovu spajanja, čvor odmah prosleđuje rezultujući red roditeljskom čvoru (za razliku od sortiranja, koje mora da primi sve redove pre nego što ih obradi). Čvor se tada zaustavlja sve dok njegov roditeljski čvor ne zatraži drugi red. Zbog toga, ako je potreban samo delimičan rezultat (kao što je podešeno LIMIT-om, na primer), operacija se neće izvršiti u potpunosti.

Dva SEQSCAN lista su skeniranja tabele. Na zahtev roditeljskog čvora, lisni čvor (SEQSCAN) čita sledeći red iz tabele i vraća ga.

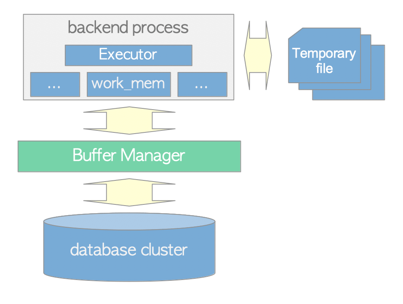
Složeni upiti mogu uključiti više nivoa čvorova, ali opšti pristup je isti: svaki čvor vraća svoj sledeći red svaki put kada je pozvan. Čvorovi su takođe odgovorni za primenu bilo kog izraza za selekciju ili projekciju koji mu je dodelio planer

Ovaj čvor i kao i neki drugi uopšte čvorovi ne čuvaju redove, već ih samo isporučuju i odmah zaboravljaju. Drugi čvorovi, kao što je sortiranje, mogu potencijalno morati da skladište ogromne količine podataka u isto vreme. Da bi se to realizovalo, work\_mem tj. radna memorija se dodeljuje pozadinskoj memoriji. Njegova podrazumevana veličina 4MB. Kada ponestane memorije, višak podataka se šalje u privremenu datoteku na disku.

Plan može da sadrži više čvorova sa zahtevima za skladištenje, tako da može imati nekoliko dodeljenih delova memorije, svaki veličine work\_mem. Ne postoji ograničenje ukupne veličine memorije koju proces upita može zauzeti.

Izvršitelj čita i upisuje tabele i indekse u klaster baze podataka preko menadžera bafera. Prilikom obrade upita, izvršilac koristi neke memorijske oblasti, kao što su temp\_buffers i work\_mem, unapred dodeljene i kreira privremene datoteke ako je potrebno.

Pored toga, kada pristupa torkama (redovima), PostgreSQL koristi mehanizam kontrole istovremenosti da bi održao konzistentnost i izolaciju tekućih transakcija.



Slika 7 - Odnos između izvršioca, menadžera bafera i privremenih datoteka.

# Optimizacija upita

Već nekoliko puta smo pomenuli da jedna SQL izjava (upit) može biti izvršen na mnogo načina, koristeći različite planove izvršenja. U stvari, moglo bi biti stotine, hiljade ili čak milioni mogucćih načina da se izvrši jedna izjava! Ovo poglavlje prikazuje da na planove planiranja i optimizacija mogu najviše uticati:

• Redosled operacija

• Metode pristupa i preuzimanja podataka (npr. korišćenje indeksa, punog skeniranja)

• Algoritmi koji se koriste za spajanja i druge operacije (npr. ugneždene petlje, hash join)



## Metode preuzimanja podataka

Kada pređemo sa logičkog na fizički nivo, matematički odnosi su transformisan u tabele koje se čuvaju u baze podataka, a mi moramo da identifikujemo načine za preuzimanje podataka iz tabela.

Da bi započeo izvršavanje upita, mehanizam baze podataka mora da izdvoji sačuvane podatke korišćenjem jednog od algoritama za pristup podacima iz baze. Ovaj odeljak odnosi se na algoritme koji se koriste za čitanje podataka iz objekata baze podataka. Korišćenjem ovih algoritama moguće je uštedeti vreme izvršenja izbegavajući čitanje koje će naknadno biti filtrirano, čime se postiže optimizacija. Efikasnost ovih operacija zavisi od odnosa redova koji se preuzimaju iz baze u odnosu na ukupan broj redova u skladištenoj tabeli. Ovaj odnos se naziva selektivnost. Izbor algoritma za datu operaciju čitanja zavisi od selektivnosti filtera

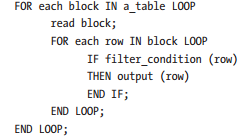
Metode za preuzimanje podataka iz baze koje se koriste kod PostgreSQL-a su:

* Full Scan
* Index Based table acess
* Index-Only scan

### Algoritam potpuno skeniranje (Full Scan)

Prilikom potpunog skeniranja (full scan), mehanizam baze podataka uzastopno čita sve redove u tabeli i proverava da li je uslov filtriranja ispunjen za svaki red preuzet iz baze.

Pseudo kod full scan algoritma prikazan je na slici ispod:



Ukupna cena korišćenja ovog algoritma za skeniranje podataka može se predstaviti preko forumule: *c1 \* BR + c2 \* TR + c3 \* S\* TR*, gde BR - označava broj I/O pristupa bazi, TR - broj unutrašnjih iteracija petlje, S – selektivnost, c1,c2,c3 konstante koje predstavljaju karakteristike korišćenog hardvera.

Algoritam potpunog skeniranja može se koristiti kod bilo koje tabele i nije potrebna nikakva dodatna struktura podataka, dok ostali algoritmi za skeniranje podataka zavise od postojanja indeksa u tabeli.

### Algoritam pristupa podacima zasnovan na indeksu (Index based table access)

Sve relacione baze podataka, uključujući PostgreSQL, dozvoljavaju izgradnju dodatnih, redundantnih struktura podataka, čineći pristup podacima znatno bržim nego jednostavno sekvencijalno čitanje. Ove dodatne strukture se nazivaju indeksi. Indeksi su „redudantni“ objekti baze podataka što znači da oni ne čuvajte nikakve dodatne informacije koji bi se skladištili u samoj tabeli. Oni obezbeđuju dodatne putanje za pristup podacima, dozvoljavaju nam da odredimo koje vrednosti se čuvaju u redovima tabele bez stvarnog čitanja tabele - ovo je način kako funkcioniše pristup zasnovan na indeksu, i izvršavanje je u potpunosti nevidljivo za aplikaciju.

Ako je uslov (ili uslovi) filtriranja enkapsuliran indeksom na tabeli, indeks se može koristiti za pristup podacima iz te tabele. Algoritam izvlači listu pokazivača na blokove koji sadrže redove sa vrednostima koje zadovoljavaju uslov filtriranja, i samo ovi blokovi se čitaju iz tabele.

Da biste dobili red tabele iz pokazivača, blok koji sadrži ovaj red mora biti pročitan. Osnovna struktura podataka tabele je gomila, to jest, redovi se čuvaju neuređeni. Postoje dve odvojene fizičke operacije koje PostgreSQL koristi za preuzimanje redova preko indeksa: skeniranje indeksa i bitmap skeniranje gomile.

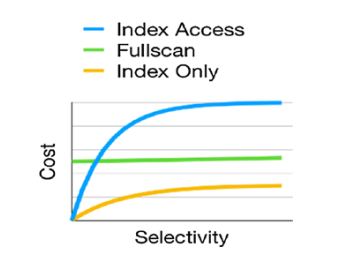
U skeniranju indeksa, mašina baze podataka čita svaki unos indeksa koji zadovoljava uslov filtera i preuzima blokove po redosledu indeksa. Pošto je osnovna tabela gomila, više unosa indeksa može ukazivati na isto blokirati. Da bi se izbeglo višestruko čitanje istog bloka, implementirano je bitmap skeniranje gomile u PostgreSQL-u koje pravi bitmapu koja označava blokove koji sadrže potrebne redove.

Model troškova ovog algoritma je mnogo složeniji i može biti opisano na ovaj način: za male vrednosti selektivnosti, svi redovi koji zadovoljavaju uslov filtriranja će se nalaziti u različitim blokovima, u tom slučaju trošak je proporcionalan broju redova rezultata. Za veće vrednosti selektivnosti, broj obrađenih blokova približava se ukupnom broju blokova. U poslednjem slučaju, trošak postaje veći od cena korišćenja potpunog skeniranja zbog potrebnih dodatni resursa za pristup preko indeksa.

### Skeniranja indeksa (Index-Only scan)

Operacije pristupa podacima ne moraju da vraćaju obavezno cele redove. Ako neke kolone nisu potrebne za ispunjavanje upita, mogu se preskočiti čim redovi prođu uslove filtriranja (ukoliko postoje uslovi za te redove). Ovo je posebno korisno ako su indeksi korišćeni za uslove filtriranja svih kolona koje su potrebne u okviru upita. Ovaj algoritam čita podatke iz indeksa i primenjuje preostale uslove filtriranja ako je potrebno. Obično nije potreban dodatni pristup podacima iz tabele, ali u nekim slučajevima je potrebna dodatna provera. Cena korišćenja algoritma zasnovanog na skeniranju samo indeksa je slična skeniranju tabela korišćenjem indeksa, osim što u ovom slučaju nije potrebnan stvarni pristup tabeli. Za male vrednosi selektivnosti cena je proporcionalna broju vraćenih redova. Obično je cena skeniranja indeksa manja od cene potupnog skeniranja tabele jer sadrži manje podataka.

Izbor najbolje algoritma za pristup podacima zavisi najčešće od selektivnosti upita. Veza između cene i selektivnosti za nabrojane algoritme pristupa podacima je prikazana na slici ispod:



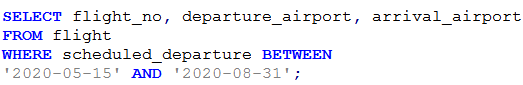
Slika 8. veza između cene i selektivnosti upita za algoritame pristupa podacima

Linija za potpuno skeniranje (full scan) je linearna i skoro je horizontalna jer je rast posledica generisanje izlaza. Trošak generisanja izlaza je zanemarljiv u poređenju sa ostalim troškovima za ovaj algoritam, tako da cena najmanje zavisi od selektivnosti.

Linija koja predstavlja cenu pristupa tabeli zasnovanog na indeksu počinje od (skoro) 0 i brzo raste sa porastom selektivnosti.

Linija koja predstavlja skeniranje samo indeksa je najniža, što znači da je ovaj algoritam najpoželjniji ako se može primeniti (tj. sve potrebne kolone su u indeksu).

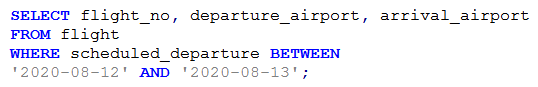
Neki slučajevi korišćenja navedenih algoritama za pristup podacima iz tabele biće prikazani u delu koji sledi:

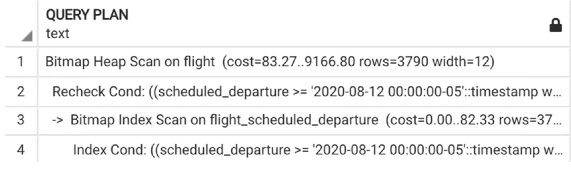




U ovom slučaju optimizator je izabrao algoritam za potpuno skeniranje (full scan) tabele.

Za isti upit, samo sa kraćim opsegom za filtiraranje optimizator je izabrao drugi algoritam za pristup podacima (koristio je pristup podacima zasnovan na indeksima).





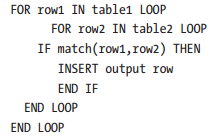
Posao optimizatora upita je mnogo složeniji: uslov filtriranja može biti podržan sa više indeksa sa različitim vrednostima selektivnosti. Višestruki indeksi se mogu kombinovati da bi se proizveo bitmap blok sa manjim brojem blokova koje je potrebno skenitrati . Kao rezultat toga, broj opcija dostupnih optimizatoru je značajan veći od tri algoritma. Dakle, među algoritmima za pristup podacima nema konkretnog izbora najboljeg algoritma, svaki algoritam pod određenim uslovima može biti izabran kao najbolji.

## Algoritmi spajanja

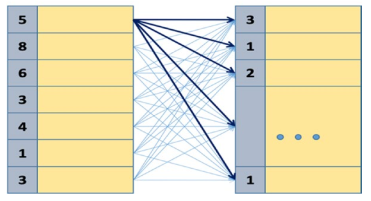
Prava moć teorije relacija i SQL baza podataka oslanja se na kombinovanje (pridruživanje) podataka iz različitih tabela. Postoje tri strategije pridruživanja u PostgreSQL-u koje funkcionišu sasvim drugačije. Ako PostgreSQL izabere pogrešnu strategiju za pridruživanje, može se izgubiti na performansama prilikom izvršenja upita. Ovaj deo objašnjava strategije pridruživanja, kako ih možete podržati sa indeksima, šta može da pođe po zlu sa njima i kako možete da podesite svoje spojeve radi boljih performansi i optimizacije upita.

### Strategija pridruživanja ugnežđene petlje (Nested Loops JOIN)

Ovo je najjednostavnija i najopštija strategija pridruživanja od svih. PostgreSQL skenira spoljnu relaciju uzastopno, i za svaki red rezultata skenira kompletnu unutrašnju relaciju. Za svaki red iz tabele kojoj se pridružuju podaci, skenira kompletnu tabelu koja se pridružuje. Pseudo kod Nested strategije za pridruživanje dve tabele prikazan je ispod:



Vreme potrebno za izvršenje ovog algoritma proporcionalno je proizvodu veličina ulaznij tabela.



Slika 9. Spajanje dve tabela korišćenjem Nested loop strategije

* Indeksi koji mogu pomoći pri spajanju ugnežđenih petlji

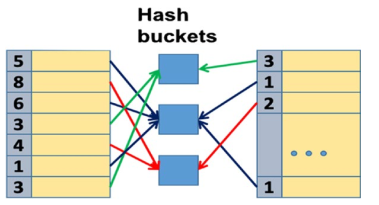
Pošto spoljnu relaciju skeniramo sekvencijalno, nijedan indeks na spoljnoj relaciji neće pomoći. Ali indeks na ključ spajanja unutrašnje relacije može značajno ubrzati spajanje ugnežđene petlje.

* Slučajevi korišćenja strategije pridruživanja ugnežđene petlje

Spajanja ugnežđene petlje je posebno efikasna ako je spoljna relacija mala, jer tada unutrašnja petlja neće biti izvršavana prečesto. To je tipična strategija pridruživanja koja se koristi u OLTP radnim opterećenjima sa normalizovanim modelom podataka, gde je veoma efikasan. Ako je spoljna relacija velika, spojevi ugnežđene petlje su obično veoma neefikasni, čak i ako oni su podržani indeksom o unutrašnjem odnosu. Osim toga, to je jedina strategija pridruživanja koja se može koristiti ako nijedan uslov pridruživanja ne koristi operator =. Tako da takođe služi kao rezervna strategija ako se ne može koristiti nijedna druga strategija.

### Strategija heš pridruživanja (Hash Join)

Ova strategija izvršava se kroz dve faze. U okviru prve faze koja se naziva building phase PostgreSQL skenira unutrašnju relaciju sekvencijalno i pravi heš tabelu prema vrednostima heš funkcije za sve redove te tabele. Zatim u drugoj fazi (probe phase) skenira spoljnu relaciju uzastopno šalje redove te tabele u heš tabelu i ispituje podudaranje na osnovu kriterijuma. Ove dve faze prikazane su kao odvojene fizičke operacije u planu izvršenja.



Slika 10. Spajanje dve tabele korišćenjem Hash strategije

Cena korišćenja ovog algoritma može se predstaviti preko forumule:

cost(hash,inner,outer)=size(inner)+size(outer)+size(inner)\*size(outer)/size(JoinAttribute)

Ovo je donekle slično spojajanju korišćenjem ugnežđene petlje. Izgradnja heš tabele je dodatni početni napor, ali ispitivanje heša je mnogo brže od skeniranja cele unutrašnje relacije.

* Indeksi koji mogu pomoći u heš spojevima

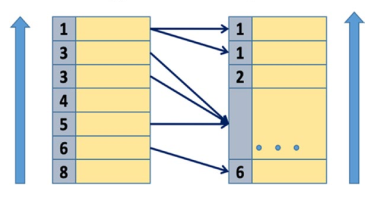
Pošto skeniramo obe relacije uzastopno, indeks na uslovu spajanja neće pomoći pri heš pridruživanju.

* Slučajevi korišćenja strategije heš pridruživanja

Heš spajanja su najbolja ako nijedna od uključenih relacija nije mala, ali heš tabela zahteva resurse za manju tabelu u work\_mem. To je zato što bi inače PostgreSQL napravio heš u nekoliko serija i uskladištio ih u privremenim disk, što šteti performansama. U takvim slučajevima optimizator obično bira drugačiju strategiju pridruživanja kao što je marge spajanje. Traženje vrednosti u heš tabeli funkcioniše samo ako je operator u uslovu pridruživanja =, tako da vam treba najmanje jedan uslov spajanja sa tim operatorom.

### Strategija Marge pridruživanje (Marge Join)

U Marge join strategiji, PostgreSQL bira sve uslove spajanja pomoću = operatora. Zatim sortira obe tabele na osnovu spoljnih ključeva (što znači da tipovi podataka moraju biti sortirani). Zatim se kreće kroz obe sortirane liste i upoređuje ključeve.



Slika 11. Spajanje dve tabele korišćenjem Marge strategije

Cena korišćenja ovo algoritma je predstavljena preko formule:

size(outertable)\*log(size(outertable)) + size(innertable)\*log(size(innertable))

* Indeksi koji pomažu pri spajanju spajanja

Indeks na ključevima za sortiranje mogu ubrzati sortiranje, tako da indeks na spojnim ključevima na obe relacije može ubrzati Marge spajanja. Međutim, eksplicitno sortiranje je često jeftinije osim ako se ne može koristiti samo skeniranje indeksa.

* Slučajevi korišćenja strategije spajanja

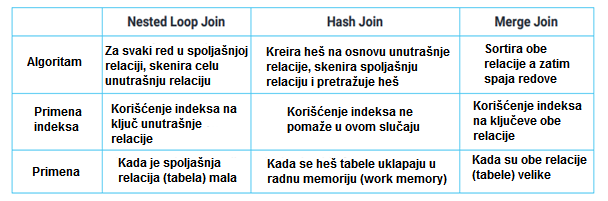
Optimizator obično bira Marge spajanja ako su uključene relacije prevelike za heš koji zahteva prostora u work\_mem. Dakle, ovo je najbolja strategija za spajanje zaista velikih tabela. Kao i heš pridruživanje, Marge pridruživanje je izvodljivo samo ako postoji bar jedan uslov spajanja sa = operatorom.

Odabir pogrešne strategije pridruživanja dovodi do lošeg učinka:

* Ako optimizator potceni broj redova, može greškom izabrati spajanje ugnežđene petlje. Zatim skenira unutrašnje relacije češće nego što je procenjeno, što dovodi do lošeg uticaja na performanse.
* Ako optimizator preceni broj redova, može greškom izabrati heš ili Marge spajanja. Onda je potrebno da se u potpunosti skeniraju obe relacije, što može da radi mnogo lošije od spajanja ugnežđene petlje sa indeksom na unutrašnjem relacijama.

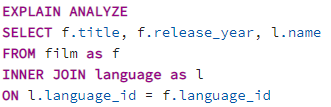
U oba slučaja, uzrok problema je loša procena broja redova.

U tabeli ispod vidimo poređenje između ovih tri metoda spajanja:



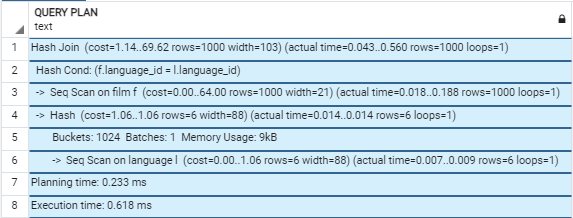
### Primer izbora strategije za spajanje relacija

U ovom delu biće naveden konkretan primer spajanja relacija, analiza i izbor strategije za spajanje kao i poređenje vremena izvršenja različitih strategija za isti upit.

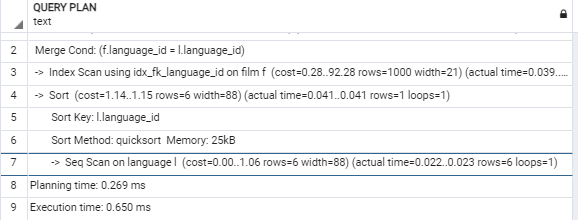


Za navedeni upit uporedićemo vreme potrebno za izvršavanje korišćenjem različitih strategija spajanja.

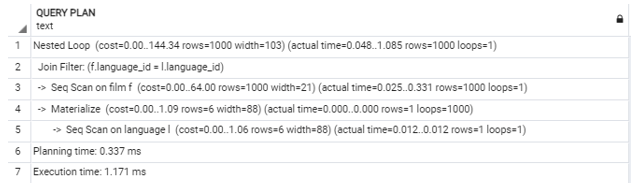
Prilikom izvršavanja ovog upita sam optimizator je izabrao Hash strategiju kao najbolju za ovaj slučaj. Razlog je što spoljašnja tabela nije mala, takođe se ne koriste indeksi (čime bi se favorizovala Nasted loop strategija), a u odnosu na Marge join podaci nisu sortirani. Na slici ispod prikazano je potrebno vreme za izvršenje navedenog upita korišćenjem Hash strategije.



Za isti upit korišćenjem Marge join strategije vreme planiranja i izvršavanja upita bi bilo:



Korišćenjem Nested Loop strategije za ovaj upit potrebno vreme planiranja i izvršenja bi bilo:



Male izmene samog upita (poput dodavanja indeksa) mogle bi dovesti do potupno drugačijeg izbora strategije za pridruživanje, čime se uticalo na performanse izvršenja tog upita.

# Zaključak

Obrada i optimizacija upita predstavlja jednu od najvažnijih, ako ne i najvažniji koncept rada sa bazom podataka, jer danas važnu ulogu u informacionim tehnologijama igra brzina obrade podataka. Sam proces obrade je vrlo komplikovan i sastoji se iz više faza. Vreme obrade može varirati od jednog delića sekunde do više sati, zavisno od izabranog načina pristupa bazi podataka i načinu izvršenja upita. Svrha optimizacije upita, koji je najteži deo celokupnog sistema za obradu upita, jeste da se pronađe način da se dati upit obradi u minimalnom vremenu. Pronalaženje tačnog optimalnog puta za izvršavanje upita, među svim mogućim, je veoma složen, dugotrajan proces, koji može puno da košta po ceni vremena a često i ne može da se pronađe.

Poznavanje načina na koji se vrši obrada i optimizacija upita i 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.

PostgreSQL pruža veoma bogate mogućnosti za optimizaciju upita. Zahvaljujući naredbi EXPLAIN koji je najbolji pristup osnovnim podacima i zahvaljujući razumevanju kako PostgreSQL radi, administrator može odlučiti da li je potrebno izmeniti neki upit, koristiti neke druge strategije za pridruživanje ili je možda potrebno korisitit indekse. PostgreSQL može da generiše plan izvršenja i pruža mogućnost nadgledanja i optimizacije performansi prilikom izvršenja upita.

# Literatura

* Understanding How PostgreSQL Executes a Query

<http://etutorials.org/SQL/Postgresql/Part+I+General+PostgreSQL+Use/Chapter+4.+Performance/Understanding+How+PostgreSQL+Executes+a+Query/>

* Query Processing (<https://www.interdb.jp/pg/pgsql03.html>)
* Queries in PostgreSQL – executio stages ( <https://postgrespro.com/blog/pgsql/5969262> )
* PostgreSQL Query Optimization – Building Efficient Queries
* JOIN STRATEGIES AND PERFORMANCE IN POSTGRESQL (<https://www.cybertec-postgresql.com/en/join-strategies-and-performance-in-postgresql/> )