Elektronski fakultet u Nišu

Seminarski rad

PostgreSQL - Obrada upita

Predmet: Sistemi za upravljanje bazama podataka

|  |  |
| --- | --- |
| Mentor: Aleksandar Stanimirović | Student: Marko Đokić 1022 |

*April, 2020.*

Sadržaj

[1.Pregled](#_742nebewpb76)

[1.1. Parser](#_5dwxiae30xow)

[1.2. Alalyzer](#_n09j3qylfq1v)

[1.3. Rewriter (Prepisivač)](#_aq07ssvhjzx2)

[1.4. Planner i executor (planer i izvršilac)](#_dd9r301whmrz)

[2. Cena troškova kod upita nad jednom tabelom](#_hda7jljew7zk)

[3. Izrada stabla plana kod upita nad jednom tabelom](#_i7o5ck9a2jkg)

[3.1. Preprocesiranje](#_1gdhaudii7hy)

[3.2. Dobijanje najjjeftinije pristupne putanje](#_xyl6uahd32jy)

[3.3. Kreiranje stabla plana](#_fxnksy1g3glu)

[4. Kako izvršilac (executor) radi](#_e90wyfos4eaw)

[5. Join (pridruživanje) operacije](#_5ptnf15zav40)

[5.1. Join sa ugnjezdenom petljom](#_h395kuwi17gc)

[5.1.1. Osnovni join sa ugnježdenom petljom](#_7avfz3l1c051)

[5.1.2. Materijalizovani join sa ugnježdenom petljom](#_hjb13ki0sqmc)

[5.1.3. Indeksirani join sa ugnježdenom petljom](#_kt2zqj1s854t)

[5.1.4. Ostale varijante](#_kuaha1mcyc8o)

[5.2. Megre Join](#_zcjg2spw7fez)

[5.2.1. Osnovni merge join](#_crurhskpvn87)

[5.2.2. Materijalizovani merge join](#_wo6p0gxo4lgs)

[5.2.3. Ostale varijacije](#_60c8frpqxs4u)

[5.3. Hash Join](#_j70l09p8695v)

[5.3.1. In memory hash join](#_kdwbu5jtmmti)

[5.3.2. Hibridni hash join sa metodom iskrivljenja (skew)](#_98pw0uufdffz)

[Zaključak](#_kle9ozhi5sog)

[Literatura](#_8ih5jj9fqhd2)

Obrada upita

Kao što je opisano u dokumentaciji PostgreSQL podržava veliki broj mogućnosti koje zadovoljavaju SQL standard iz 2011. Obrada upita je najsloženiji podsistem u PostgreSQL i efikasno obrađuje podržani SQL. Ovaj rad će opisati obradu upita i posebno će se fokusirati na optimizaciju upita.

Rad će se sastojati iz 3 dela:

* Deo koji prikazuje obradu upita u PostgreSQL
* Deo koji opisuje korake koji slede da bi se dobio optimalan plan upita za jednu tabelu. Najpre su objašnjeni procesi procene troškova i kreiranje stabla plana, a zatim je opisan rad te izvršne jedinice.
* Deo koji objašnjava postupak dobijanja optimalnog plana upita sa više tabela. Opisane su i tri metode spajanja: ugnježdena petlja, spajanje i heš pridruživanje.

## 1.Pregled

U PostgreSQL, iako je paralelni upit implementiran u verziji 9.6. koristi se više pozadinskih procesa za obradu, bekend proces u osnovi obrađuju sve upite koje je izdao taj povezani klijent. Taj bekend sastoji se od 5 podsistema, kao što je prikazano u nastavku:

1. Parser (Raščlanjivač)

Parser generiše stablo rasčlanjivanja iz SQL izraza u običan tekst.

1. Analyzer / Analzser (Analizator)

Analzyer vrši semantičku analizu stabla raščlanjivanja i generiše stablo upita.

1. Rewriter (Prepisivač)

Rewriter transformiše stablo upita koristeći pravila koja se nalaze u sistemu pravila, ako takva pravila postoje.

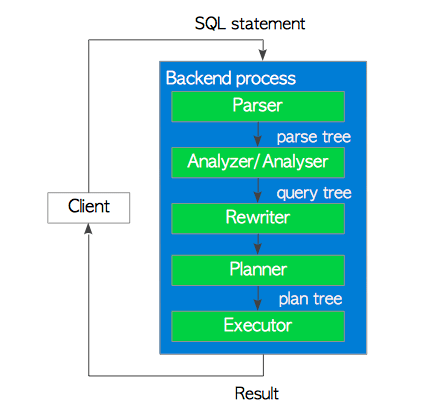
1. Planner (Planer)

Planer generiše stablo plana koje se najefikasnije može izvršiti iz tog upita.

1. Executor (Izvršilac)

Executor izvršava upit pristupajući tabelama i indeksima redosledom koji propisuje stablo plana.

U narednom poglavnju dat je pregled svih ovih podsistema. Zbog činjenice da su planner i executor veoma komplikovani, detaljno objašnjenje ovih podsistema biće dato u narednim odeljcima.



Slika 1., Podsistemi pozadinskog procesa

### 1.1. Parser

Parser generiše stablo raščlanjivanja (parsovanja) koje mogu pročitati naredni podsistemi, iz SQL izraza transformiše ga u običan tekst. U nastavku će biti prikazan konkretan primer bez detaljnog opisa.

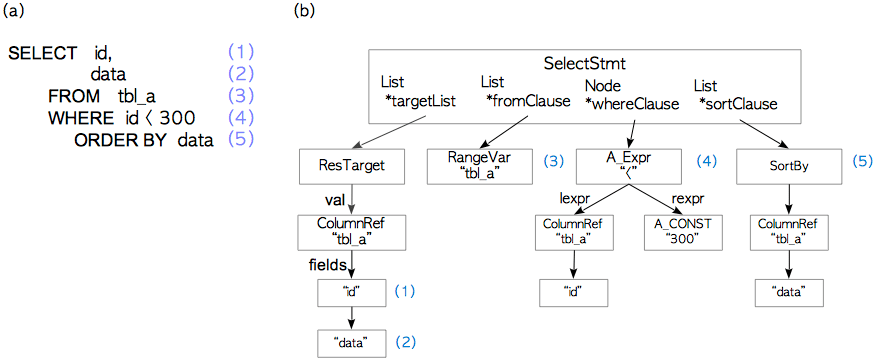
Razmotrimo upit prikazan u nastavku:

*testdb=# SELECT id, data FROM tbl\_a WHERE id < 300 ORDER BY data;*

Stablo raščlanjivanja je stablo čiji root čvor je Select izraz definisan u određenom obliku.

Elementi Select upita i odgovarajući elementi stabla parsovanja su označeni istim brojevima (na slici 2.), na primer stavka (1) je stavka prve liste u stablu i kolona “id” u tabeli itd.

Parser proverava samo sintaksu ulaza i prilikom generisanja stabla raščlanjivanja, vraća se greška samo u koliko je u pitanju sintaksna greška u upitu. Parser ne proverava semantiku upita. Na primer, čak i ako upit sadrži ime tabele koje ne postoji neće se vratiti greška tada. Semantičke greške proveravaće analyzer.

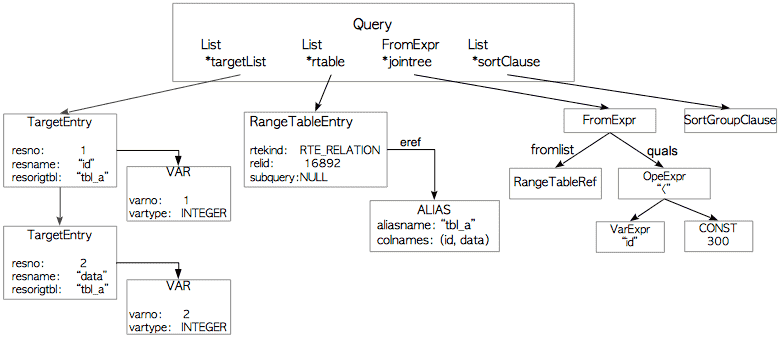


Slika 2., Primer stabla parsitanja (raščlanjivanja)

### 1.2. Alalyzer

Analyer pokreće semantičku analizu stabla raščlanjivanja koje generiše parser i nakon toga generiše stablo upita.

Koren stabla upita je određena query struktura, ova struktura sadrži metapodatke odgovarajućeg upita kao što je na primer tip naredbe (SELECT, INSERT...) i listove, svaki list formira listu ili stablo i sadrži podatke određene pojedinačne stavke. Slika 3. prikazuje stablu upita sa slike 2 a).



Slika 3., Primer stabla upita

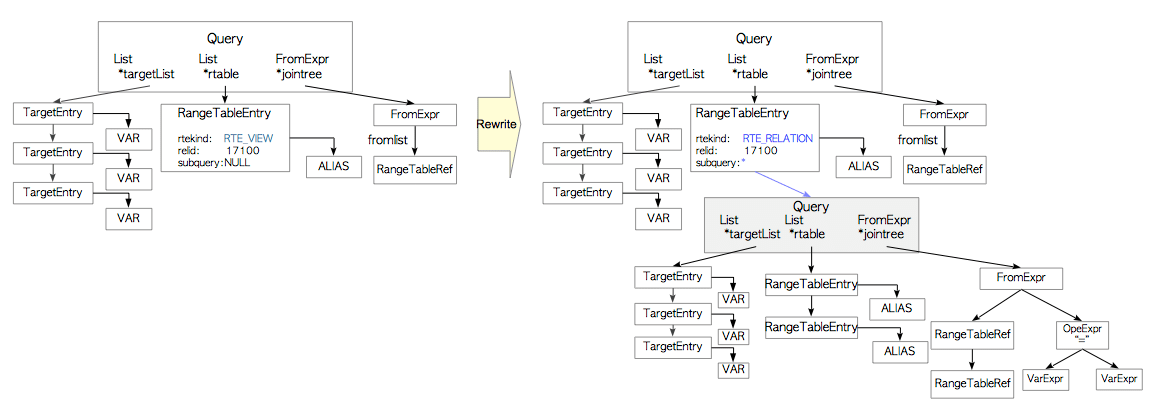
Ukratko opisano stablo upita:

* ***targetList*** je lista kolona koje su rezultat ovog upita. U ovom primeru lista se sastoji od dve kolone “id” i “data”. Ako upit sadrži “\*”, analizator ce eksplicitno to zameniti sa svim kolonama.
* ***rtable*** je lista veza koje se koriste u ovom upitu. Na primer, ova tabela sadrži podatke tabele “tbl\_a”, kao što su identifikatori, naziv itd.
* ***joinTree*** čuva informacije FROM ili WHERE klauzulama.
* ***sortClausule*** je lista SortGroupClausule koje će se koristiti kasnije za sortitanje.

### 1.3. Rewriter (Prepisivač)

Rewriter je deo koji realizuje sistem pravila i po potrebi transformiše stablo upita u skladu sa pravilima sačuvanim u katalogu **pg\_rules** sistema. Pogledi (VIEW) u PostgreSQL se realizuju korišćenjem sistema pravila. Kada je pogled definisan naredbom CREATE VIEW, odgovarajuće pravilo automatski se generiše i čuva u katalogu.

Pretpostavimo da je sledeći pogled definisan i da je odgovarajuće pravilo smešteno u sistemskom **pg\_rules** katalogu.



Slika 4. Primer rewriter faze

sampledb=# CREATE VIEW employees\_list

sampledb-# AS SELECT e.id, e.name, d.name AS department

sampledb-# FROM employees AS e, departments AS d WHERE e.department\_id = d.id;

Kada se izvrši upit koji sadrži pogled prikazan u nastavku kreiraće se stablo analize kao što je prikazano na slici 4.

sampledb=# SELECT \* FROM employees\_list;

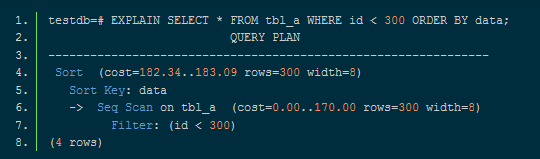
U ovoj fazi, rewriter će dodati stablo pod-upita u **RangeTable** čvor, koji je odgovarajući VIEW i nalazi se u ***pg\_rules*** delu.

### 1.4. Planner i executor (planer i izvršilac)

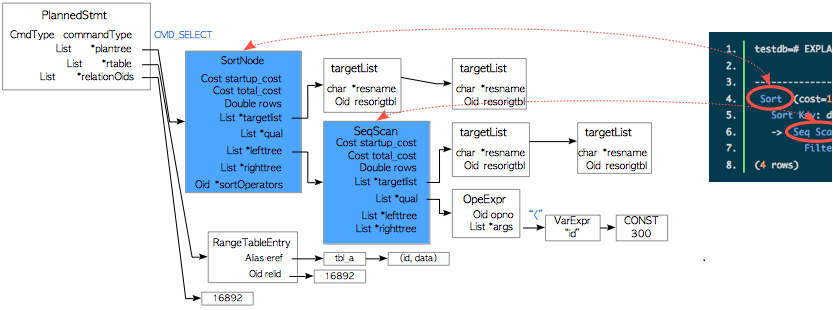
Planner prima stablo upita od rewritera i generiše stablo (upita) plana koje executor može najefikasnije obraditi.

Planner je u PostgreSQL realizovan na čistoj optimizaciji na osnovu troškova, a ne podržava optimizaciju i nagoveštaje zasnovane na pravilima. Ovaj planer je najsloženiji podsistem u RDBMS i detaljniji pregled plannera će biti dat u nekom narednom odeljku.

Kao i ostali RDBMS, EXPLAIN naredba u PostgreSQL prikazuje stablo plana. Primer je prikazan u nastavku.



Struktura rezultata ove naredbe prikazana je na slici 5 i predstavlja stablo plana.



Slika 5., Stablo plana i odnos između stabla plana i rezultat naradebe EXPLAIN

Svaki čvor plana ima informacije koje su executoru potrebne za obradu, a executor obrađuje čvorove počevši od kraja stabla plana do korena u slučaju upita u jednoj tabeli.

Na primer, stablo plana prikazano na slici 5. je lista od dva čvora, jedan za sortiranje i drugi za sekvencialno skeniranje (obradu) čvorova, na taj način executor obrađuje tabelu ***tbl\_a*** najpre sekvencijalnim skeniranjem, a zatim sortira dobijeni rezultat.

## 2. Cena troškova kod upita nad jednom tabelom

PostgreSQL optimizacija upita je zasnovana na troškovima. Troškovi su bezdimenzione vrednisti i nisu apsolutni pokazatelji performansi, već su vezani za upoređivanje relativnih performansi operacija. Svaka funkcija ima definisanu funkciju vezanu za troškove, npr. troškovih sekvencijalnih i indeksnih obrada procenjuju se sa cost\_seqscan() i cost\_index() redom.

Postoje tri vrste troškova: *start-up*, *run* i *total*. Ukupan (*total*) tošak je zbir *start-up* i *run* troškova. *Start-up* i *run* troškovi se nezavisno procenjuju.

Početni trošak (*start-up*) je trošak potrošen pre nego što se prvi red u tabeli tj. stavka prikupi. Na primer početni trošak čvora za skeniranje indeksa je trošak za čitanje indeksnih stranica kako bi se pristupilo prvom redu u ciljanoj tabeli.

## 3. Izrada stabla plana kod upita nad jednom tabelom

Poštoje obrada planera veoma složena u ovom delu biće opisan najjednostavniji postupak, odnosno kako se stvara stablo plana upita nad jednom tabelom.

Planer u PostgreSQL izvodi tri koraka, koji su prikazani u nastavku.

1. Izvršava preprocesiranje,
2. Nabavlja najjeftiniji put za pristup sa najmanjim mogućim troškovima,
3. Pravi stablo plana sa najjeftinijim putem.

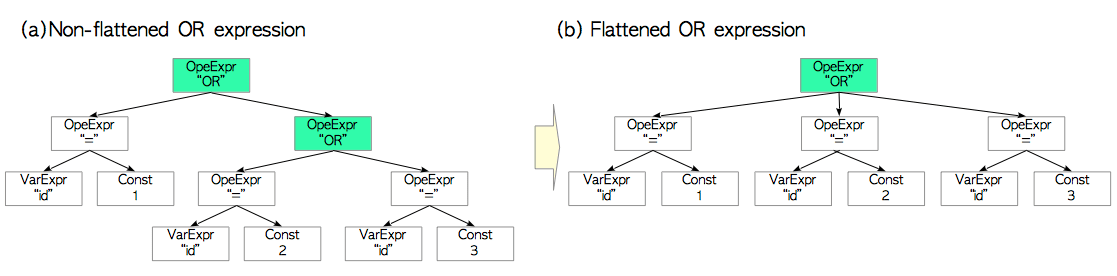
Pristupna putanja (access path) je jedinica kod procesiranja za procenu troškova, npr. sortiranje, sekvencijalna obrada i razne operacije imaju svoje odgovarajuće pristupne putanje, ove putanje se koriste samo kod planera za kreiranje stabla plana.

U ovom delu opisano je kako se kreira stablo plana na osnovu stabla upita pomoću posebnih primera.

### 3.1. Preprocesiranje

Pre kreiranja stabla plana, potrebno je da planer izvrši različita procesiranja nad stablom upita, to uključuje mnogo koraka u ovom delu će se govoriti samo o glavnoj obradi, ostale će biti opisane u kasnijem nastavku.

1. Pojednostavljene ciljanih listi, ograničenja i tako dalje, npr. ako postoji “2+2” biće prepisano u 4.
2. Normalizacija logičkih izraza, npr “NOT(NOT a)” se prebacuje u “a”.
3. Izravnanje AND / OR izraza. AND i OR su u SQL standardni binarni operatori, međutim u PostgreSQL unutrašnjosti su n-arni operatori i planer uvek izravna ugnježdene AND i OR izraze, primer se nalazi na slici 6.



Slika 6. Izravnanje AND/OR izraza

### 3.2. Dobijanje najjjeftinije pristupne putanje

Da bi se dobila najjeftinija pristupna putanja planer procenjuje troškove svih mogućih pristupnih putanja i bira najjeftiniju. Tačnije obavlja sledeće operacije:

1. Kreira ***RelOptInfo*** strukturu da bi u njoj čuvao pristupne putanje i odgovarajuće troškove. ***RelOptInfo*** struktura je kreirana funkcijom *make\_one\_rel()* i nalazi se u ***simple\_rel\_array*** unutar ***PlannerInfo*** strukture, što se može videti na slici 7.

U ovom početnom stanju***RelOptInfo*** sadrži ***baserestrictinfo*** i ***indexlist*** ako postoje povezani indeksi,***baserestrictinfo***čuva WHERE klauzulu upita, a ***indexlist***čuva indekse ciljane tabele.

1. Procenjuje troškove svih mogućih pristupnih putanja i dodaje pristupne staze u ***RelOptInfo*** strukturu.

Detalji ove obrade su sledeći:

[1]. Kreira se putanja, vrši se procena i troškovi se pridodaju putanji. Zatim se putanja dodaje listi putanja strukture ***RelOptInfo***.

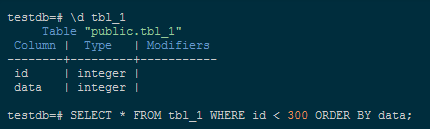
[2]. Ako postoje indeksi koji se odnose na ciljanu tabelu, stvaraju se indeksiranja pristupnih putanja, svi troškovi skeniranja indeksa procenjuju se i procenjeni troškovi se dodaju u listu putanja. Zatim se i indeksirane putanje dodaju u listu puteva

[3]. Takođe postoji i određena bitmap obrada, ako se može obaviti i ovi troškovi se procenjuju i dodaju za putanju.

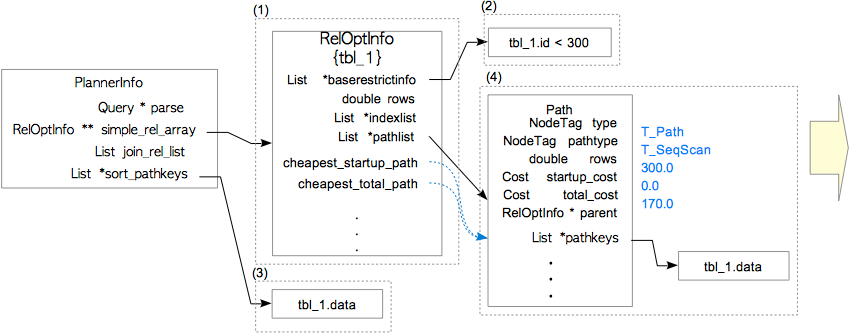
[4]. Pribavlja se najjeftinija pristupna putanja iz liste putanja.

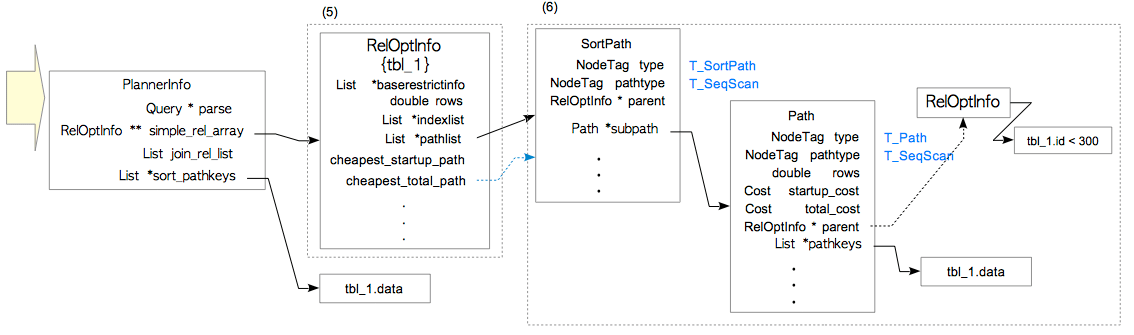
[5]. Procenjuju se LIMIT, ORDER BY I ARREGSFDD troškovi ako je to potrebno.

Primer 1:



Traženje najjeftinije pristupne putanje iz primera 1. je opisano u nastavku i skicirano na slici 7.





Slika 7. Kako dobiti najjeftiniji put

1. Kreiranje ***RelOptInfo*** strukture sa ***simple\_rel\_array*** nizom ***PlannerInfo*** elemenata.
2. Dodavanje WHERE klauzule u ***baserestrictinfo***.
3. Dodavanje pathkey u ***sort\_pathkeys*** strukturu u ***PlannerInfo*** klasi. ***Pathkey*** je struktura podataka koja predstavlja redosled sortiranja, u ovom primeru kolona “data” je dodata u ***sort\_pathkeys***, kao ***pathkey*** zato što upit sadrži klauzulu ORDER BY.
4. Zatim se kreira ***Path*** struktura i dodaje u ***pathlist*** da bi se izračunali troškovi sekvencijalne obrade koji će tu biti zabeleženi.
5. Nakon toga se kreira nova ***RelOptInfo*** struktura koja treba da se odnosi na ORDER BY funkcionalnost.
6. Kreira se putanja za sortiranje i dodaje u nov ***RelOptInfo*** deo, zatim se ***path*** za sekvencijlnu obradu povezuje sa delom ***subpath*** sa delom za sortiranje.

Na osnovu najjeftinije pristupne putanje dobijene ovde, generiše se stablo plana.

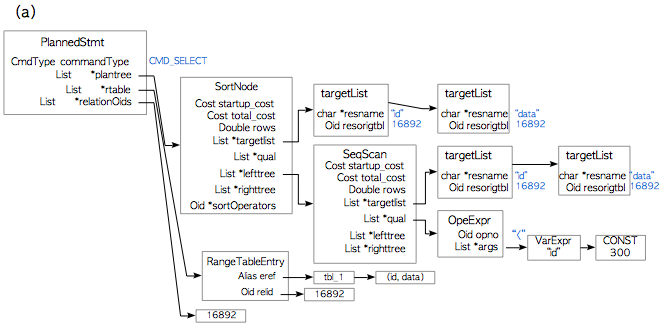
### 3.3. Kreiranje stabla plana

U poslednjoj fazi planer generiše stablo plana iz najjeftinije putanje. Koren stabla plana je PlannedStmt struktura. Iako se sastoji od devetnaest polja, ovde će biti prikzana četiri najbitnija.

* ***commandType***tip operacije npr. SELECT, UPDATE ili INSERT,
* ***rtable***lista *RangeTableEntry* elemenara,
* ***relationOids***čuva *oid-e* povezanih tabela za ovaj upit,
* ***plantree***čuva stablo plana koje se sastoji od čvorova plana, pri čemu svaki čvor odgovara određenoj operaciji, kao što je sekvencijalna obrada, sortiranje ili obrada indeksa.

Kao što je spomenuto stablo plana (plantree) sastoji se od različitih čvorova plana. Struktura *PlanNode* je osnovni čvor, a ostali čvorovi ga uvek sadrže. *PlanNode* sadrži četrnaest polja, a opisana su u nastavku sedam najvažnijih.

* ***startup\_cost***i ***total\_cost***procenjeni troškovi operacije,
* ***rows***je broj redova za skeniranje procenjenih od strane planera,
* ***targetlist***čuva stavke ciljane liste koji se nalaze u stablu upita,
* ***qual***lista u kojoj se čuvaju qual uslovi,
* ***lefttree***i ***righttree***su čvorovi za dodavanje čvorova dece.



Slika 8.Stablo plana

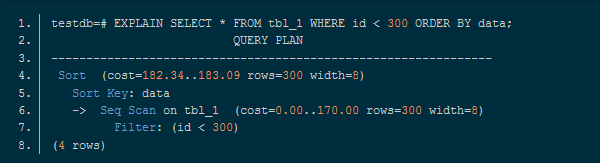
Stablo plana koje će biti generisano od najjeftinijih putanja prikazanih u primerima iz prethodnog dela dato je na slici 8.

## 4. Kako izvršilac (executor) radi

Kod upita nad jednom tabelom izvršilac preuzima čvorove iz stabla plana redosledom od kraja stabla plana do korena i zatim poziva funkcije koje obavljaju obradu odgovarajućih čvorova.

Svaki čvor plana ima funkcije koje su namenjene za izvršenje odgovarajuće opcije, nalaze se u direktorijumu src/backend/executor/directory.

Najbolji način da shvatite kako izvršilac radi je pročitati rezultat naredbe EXPLAIN, jer PostgreSQL EXPLAIN prikazuje stablo plana skoro onakvo kakvo jeste. Objašnjenje je dato u narednom primeru.

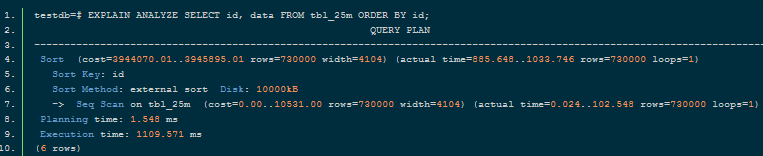


Da bi pregledali rad izvršioca potrebno je pročitati rezultat naredbe EXPLAIN iz treba čitati naredbe odozdo na gore.

Linija 6: Prvo izvršilac izvršava operaciju sekvencijalne obrade.

Linija 4: Zatim izvršilac sortira rezultat operacije sekvencijalne obrade.

Koristeći opciju ANALYZE, naredba EXPLAIN izvršava upit koji će prikazati prave brojeve redova, pravo vreme izvršenja i stvarnu potrošnju memorije. Konkretan primer je prikazan u nastavku:



Na liniji 6 naredba EXPLAIN pokazuje da je izvršilac koristio privremenu datoteku čija je veličuna 10000kB.

## 5. Join (pridruživanje) operacije

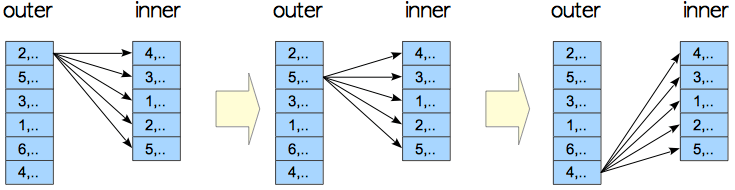
PostgreSQL podržava tri join operacije: join sa ugnježdenom petljom(nested loop join), merge (spajanje) join i hash join. Join sa ugnježdenom petljom i merge join u PostgreSQL imaju nekoliko varijacija. Unastavku se pretpostavlja da čitalac upoznat sa osnovnim ponašanjem ove tri join operacije.

### 5.1. Join sa ugnjezdenom petljom

Join sa ugnježdenom petljom je najosnovnija join operacija i može se koristiti u bilo kojim uslovima spajanja. PostgreSQL podržava više varijacija ovog joina.

#### 5.1.1. Osnovni join sa ugnježdenom petljom

Kod osnovnog joina sa ugnježdenom petljom ne postoji nikakva obrada na početku. Osnovni trošak joina sa ugnježdenom petljom proporcionalan je proizvodu outer i inner tabele.



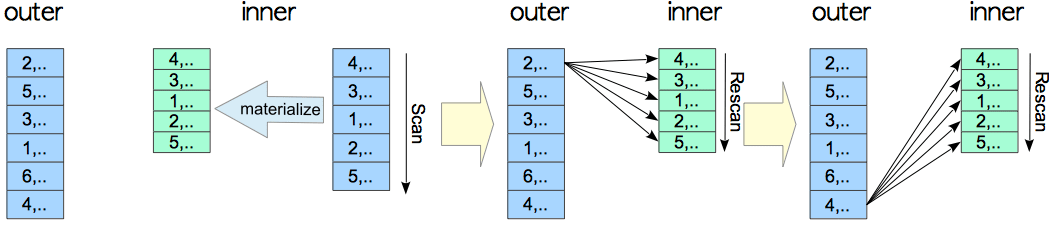
Slika 9. Primer osnovne join operacije sa ugnježdenom petljom

Trošak opisane join operacije se uvek procenjuje, ali se ova join operacija retko koristi jer se obično koriste efikasnije varijacije koje su opisane u nastavku (slika 9.).

#### 5.1.2. Materijalizovani join sa ugnježdenom petljom

Način obrade opisan iznad mora da obiđe sve elemente inner tabele nakon što pročita jedan element outer tabele. Pošto je ovakva obrada veoma skupa PostgreSQL poržava materijalizovani join sa ugnježdenom petljom da bi smanjio ukupni trošak obrade unutrašnje tabele.

Pre pokretanje operacije kod ovog tipa join operacije executor upisuje redove iz inner tabele u radnu memoriju ili privremeni fajl. Tako može efikasno obraditi sve kolone inner tablice koristeći bafer, posebno ako su svi svi redovi upisani u radnu memoriju.

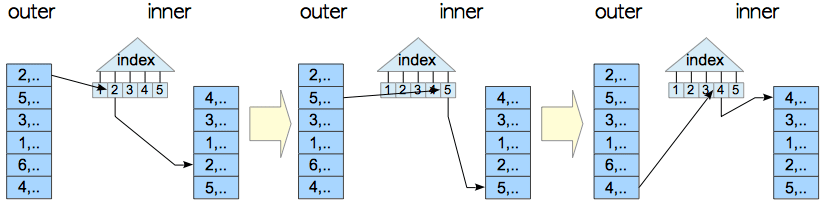


Slika 10. Primer materijalizovane join operacije sa ugnježdenom petljom

#### 5.1.3. Indeksirani join sa ugnježdenom petljom

Ako postoji indeks inner tabele i ovaj indeks može biti od pomoći da se lakše pronađu stavke koje ispunjavaju uslove join operacije, planer uzima u obzir da to da može da koristi ovaj indeks za direktno pretraživanje podataka unutar tabele umesto sekvencijalnog skeniranja, ova varijacija se naziva indeksirani join sa ugnježdenom petljom.

Uprkos tome što ga oni nazivaju join sa ugnježdenom petljom, ovaj algoritam se može izvršiti pomoću jedne petlje inner tabele, join operacija sa ovim može da se obavlja vrlo efikasno.

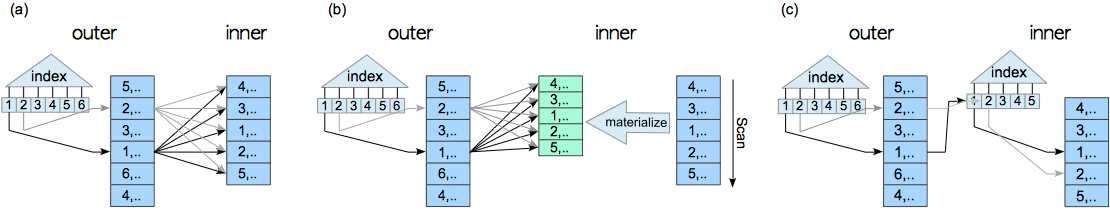


Slika 11. Primer indeksirane join operacije sa ugnježdenom petljom

#### 5.1.4. Ostale varijante

Ako postoji indeks outer tabele i njegovi atributi su uključeni u operaciju join, on se može koristiti za prolazak kroz tabelu pomoću indeksa umesto za sekvencijalne obrade outer tabele. Konkretno ako postoji indeks čiji atribut se može iskoristriti kod WHERE opcije, opseg pretraživanja oputer tabele se sužava, prema tome troškovi spajanja ugnježdene petlje se mogu drastično smanjiti.

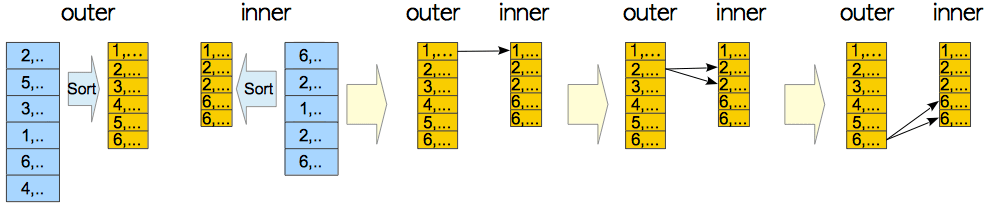
PostgreSQL podržava tri varijacije spajanja ugnježdene petlje sa outer indeksom za skeniranje. Varijante join operacije sa ugnježdenom petljom pomoću outer indeksa za skeniranje su prikazane na slici 12.



Slika 12. Join sa ugnježdenom petljom pomoću outer indeksa

### 5.2. Megre Join

Za razliku od prethodnog joina sa ugnježdenom petljom, merge join se može samo koristiti kod natural join i equi-join slučajeva. Merge join ima više varijacija.



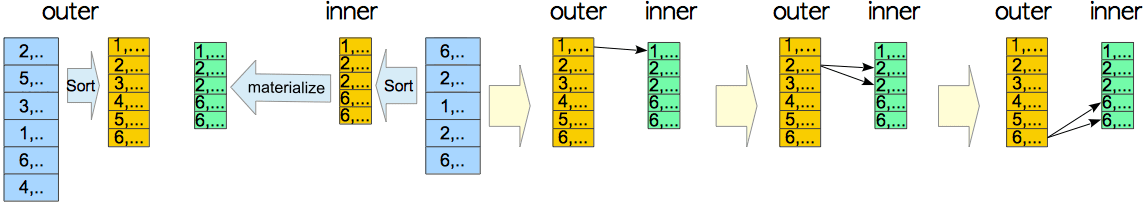
Slika 13. Merge join

#### 5.2.1. Osnovni merge join

Primer merge join operacije prikazan je na slici 13., ako svi delovi tabele mogu stati u memoriju, operacija sortiranja će se sprovesti u samoj memoriji, u suprom koristiće se privremene datoteke.

#### 5.2.2. Materijalizovani merge join

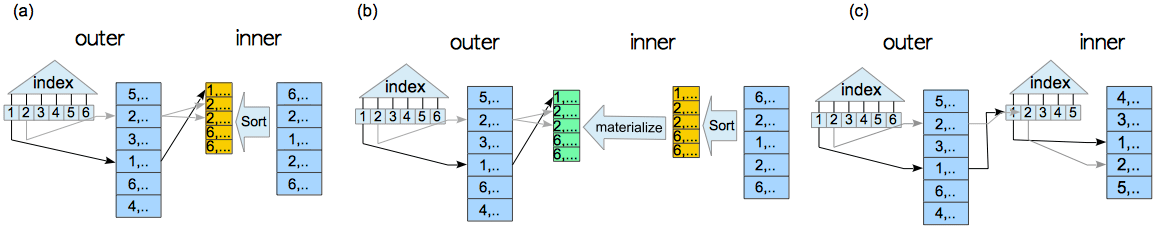
Kao kod join operacije sa ugnježdenom petljom, merge join takođe podržava materijalizovani merge join kako bi materijalizovao inner tabelu i omogućio da obrada nad inner tabelom bude efikasna, primer je prikazan na slici 14.



Slika 14. Materijalizovani merge join

#### 5.2.3. Ostale varijacije

Slično kao kod joina sa ugnježdenom petljom, merge join u PostgreSQL ima varijacije na osnovu kojih može sprovesti indeksiranu obradu outer tabele. Tri varijacije merge join operacije sa outer indeksom obradom prikazane su na slici 15.



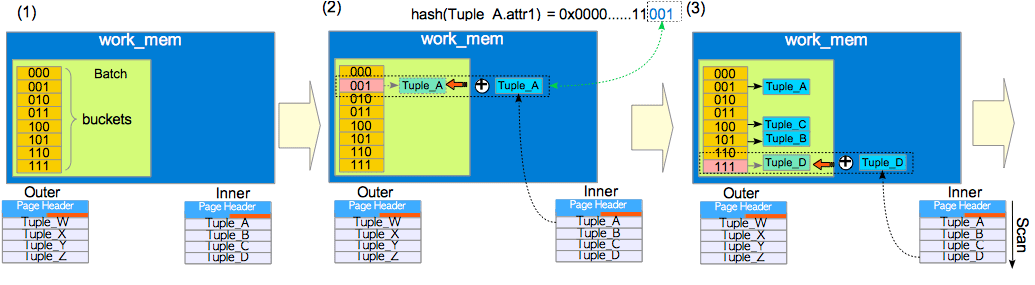
Slika 15. Varijacije megre join operacije

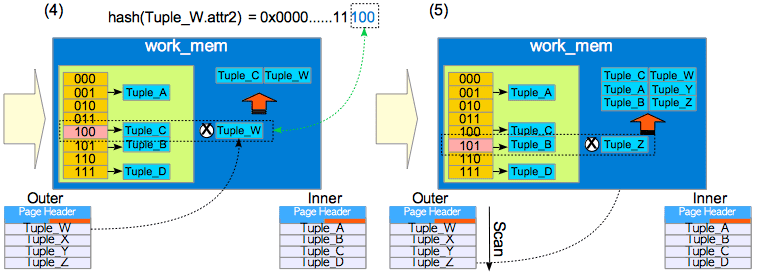
### 5.3. Heš Join

Slično kao merge join, heš join se može samo koristiti kod natural join i equi-join. Heš join u PostgreSQL ponša se različito u zavisnosti od veličine tabele. Ako je ciljana tabela dovoljno mala (veličina unutrašnje tabele je 25% od radne memorije), to će biti jednostavan heš join u memoriji, u suprotnom koristiće se hibridni heš join sa skew metodom. Grubo rečeno start-up i run troškovi su O(Nouter+Ninner) ako nema komflikta prilikom pretraživanja i umetanja u heš tabelu.

#### 5.3.1. In memory heš join

In-memory heš join ima dve faze, build fazu i probe fazu. U build fazi dodaju se sve kolone iz inner tabele se dodaju u batch deo, odnosno deo koji predstavlja hash tablicu. U probe fazi svaka stavka iz outer tabele se poredi sa stavkama iz inner tabele, koje se nalaze u batch delu, i povezuju se ako je uslov povezivanja zadovoljen. Primer prikazan na slikama 16.





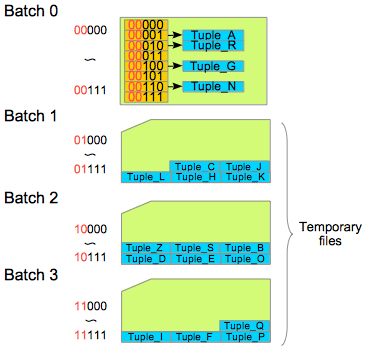
Slika 17.Build i probe faza za in-memory hash join

1. Kreira se batch u radnoj memoriji.
2. Izračunava se hash prve stavke inner tabele i dodaje se u batch.
3. Dodaju se ostale stavke.
4. Proverava se prva stavka outer tabele i pridružuje se stavki iz haš tabele koja ispunjava uslov.
5. Ispituju se ostale stavke iz outer tabele i pridružuju se odgovarajućim stavkama iz heš tabele.

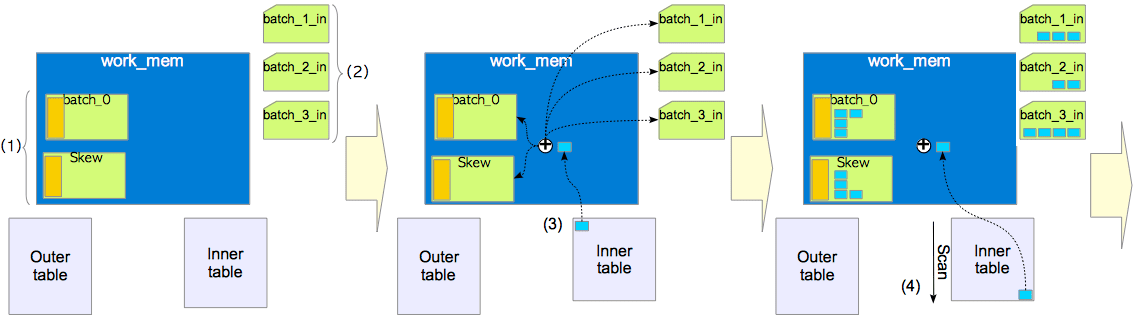
#### 5.3.2. Hibridni heš join sa metodom skew

Ovaj način se koristi kada sve stavke iz inner tabele ne mogu stati u batch deo radne memorije. Ovde se samo deo stavki iz tabele čuvaju u radnom memoriji, ostale stavke se čuvaju u privremenim datotekama. Na slici 18. prikazano je na koji način se čuvaju stavke iz fajlova. Ovde PostgreSQL priprema poseban skew deo koji će pomoći da se obrade mnogo više stavki što efikasnije u prvoj rundi.

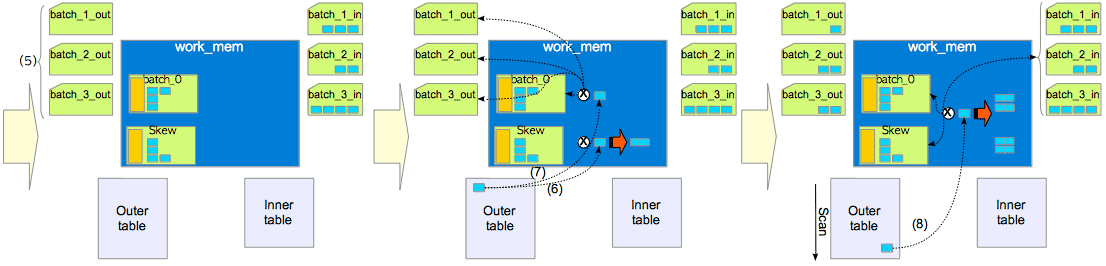
Skew deo bi trebao da sadrži one stavke iz inner tabele koje su povezani sa najvećim brojem stavki iz outer tabele. Ovo je moguće razumeti na primeru kupaca i kupovine u određenoj prodavnici, pretpostavlja se da 10 % kupaca obavlja 70 % kupovina u toj prodavnici. Samim tim koristeći skew heš deo moguće je, ako odmah dodamo stavke sa najviše veza, u prvoj fazi obavi veliki deo posla.



Slika 18. Način čuvanja savki iz tabele kod hibridnog hash join



Slika 19.



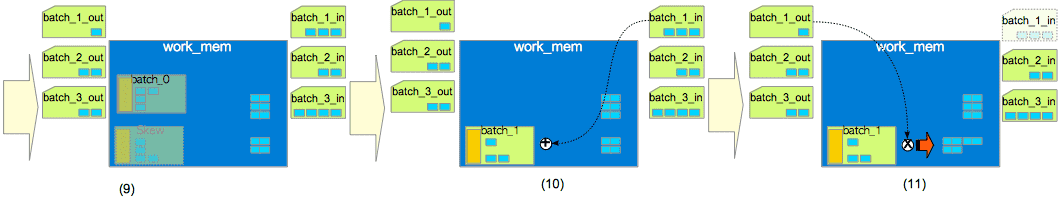
Slika 20.

U nastavku je prikazan slikoviti opis ove metode uspomoć slika 19., 20., 21. i 22.

1. Kreiraju se batch i skew u radnoj memoriji.
2. Kreiraju se privremene batch datoteke za čuvanje stavki inner tabele.
3. i (4) Započinje se build operacija, ako je stavka inner tabele među najpopularnijim, ubacuje se u skew deo, ako nije, dodaje se u neki od ostalih batch delova.

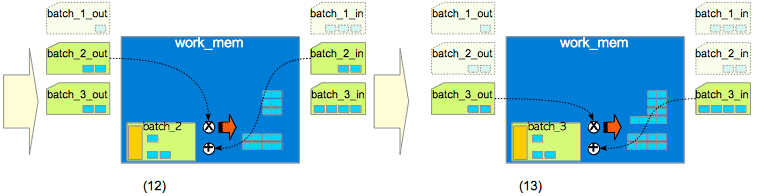
(5) Kreiraju se privremene batch datoteke za čuvanje stavki outer tabele.

(6), (7), (8) Započeće obrada outer tabele, stavke će se povezati sa stavkom iz skew dela ili batch\_0 dela u koji se trentno nalazi radnoj memoriji ili će se upisati u odgovarajuću privremenu batch datoteku i čekati neki naredni batch deo koji mu je odgovarajući.



Slika 21.

(9) Kada se isprazne skew deo i batch deo u koji se nalaze u radnoj memoriji dodavaće se redom podaci iz privremenih inner batch fajlova jedan po jedan (10), zatim će se nastaviti obrada tako što će se čitati stavke iz outer batch privremenih fajlova dok se sve ne povezu sve odgovarajuće stavke do kraja (11),(12),(13).



Slika 22.

## Zaključak

Obrada upita kod PostgreSQL baze podataka je veoma komplikovan proces, i tema o kojoj može dosta da se govori. Zato je autor ovog rada zeleo da samo ukratko prikaže delove tog procesa i da slikovitim promerima približi čitaocu obradu upita kod ove baze podataka.

PostgreSQL baza podataka, iako je besplatan i open source projekat, ona poseduje veoma kvalitetan sistem za obradu i optimizaciju upita koj je takođe i odlično dokumentovan. Velika prednost ove baze je što se vrlo lako pomoću dokumentacije može saznati o detaljima obrade upita, a svakako je velikom broju korisnika važno da razumeju šta tačno koriste u svom projektu, što čini njenu veliku prednost.

## 

## Literatura

[1] Query processing http://www.interdb.jp/pg/pgsql03.html

[2] Documentation Appendix D. SQL Conformance https://www.postgresql.org/docs/current/features.html

[3] http://etutorials.org/SQL/Postgresql/Part+I+General+PostgreSQL+Use/Cha

pter+4.+Performance/Understanding+How+PostgreSQL+Executes+a+Query/