UNIVERZITET U NISU

ELEKTRONSKI FAKULTET

**OPTIMIZACIJA UPITA KOD POSTGRESQL BAZE PODATAKA**

Seminarski rad

Studijski program: Racunarstvo i informatika

Modul: Softversko inzenjerstvo

Mentor: Student:

Prof. dr Aleksandar Stanimirovic Zeljko Vasic, br.ind. 1808

**Sadrzaj**

[1. Osnovni koncepti optimizacije upita 3](#_Toc10246)

[1.1. Osnovne metrike performansi SQL upita 4](#_Toc19811)

[1.2. Anti-pattern u pisanju SQL upita 5](#_Toc1153)

[2. Query execution plan 5](#_Toc23477)

[3. Indeksi i njihova uloga u optimizaciji upita 8](#_Toc18917)

[3.1. Sta je indeks i kako funkcionise 8](#_Toc10141)

[3.2. Partial i expression indeksi 11](#_Toc19083)

[4. Koriscenje spojeva i podupita na optimalan nacin 12](#_Toc20141)

[4.1. EXISTS vs IN 12](#_Toc22515)

[4.2. Optimizacija visestrukih JOIN-ova 13](#_Toc9681)

[5. Optimizacija agregacija i grupisanja 15](#_Toc16153)

[6. Materialized views i Query rewriting 19](#_Toc27336)

[6.1. Kreiranje i odrzavanje Materialized views 19](#_Toc16528)

[6.2. Query Rewriting 20](#_Toc13931)

[7. Particionisanje tabela 21](#_Toc10805)

[7.1. Range partitioning ( particionisanje po opsegu vrednosti) 21](#_Toc24056)

[7.2. List partitioning (particionisanje po listi vrednosti) 22](#_Toc10169)

[7.3. Hash Partitioning (Particionisanje po hash vrednosti) 23](#_Toc9222)

[8. Zakljucak 24](#_Toc18526)

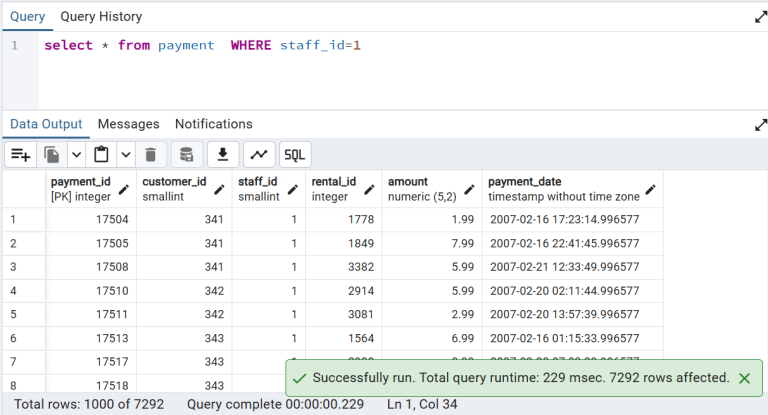
# Osnovni koncepti optimizacije upita

Optimizacija upita je proces poboljsanja performasi upita tako da se potrose sto manje resursa a da se dobije zeljeni rezultat upita. Efikasni upiti trose manje CPU i memorije na serveru, samim tim se smanjuje opterecenje servera i omogucava skaliranje baze podataka bez dodatnog hardvera (ili sa manje u poredjenju sa neoptimizovanim upitima). Manje servera i memorije znaci smanjenje troskova.

Bez optimizacije upita, kako baza raste neoptimizovani upiti bi postali prespori i ometali bi normalan rad aplikacije.

Efikasan upit nije isto sto i brz upit. Brz upit moze trenutno da vraca rezultat brzo ali to ne znaci da je optimalan za sistem. Upit moze raditi brzo na maloj bazi ali sa povecanjem podataka moze postati spor.

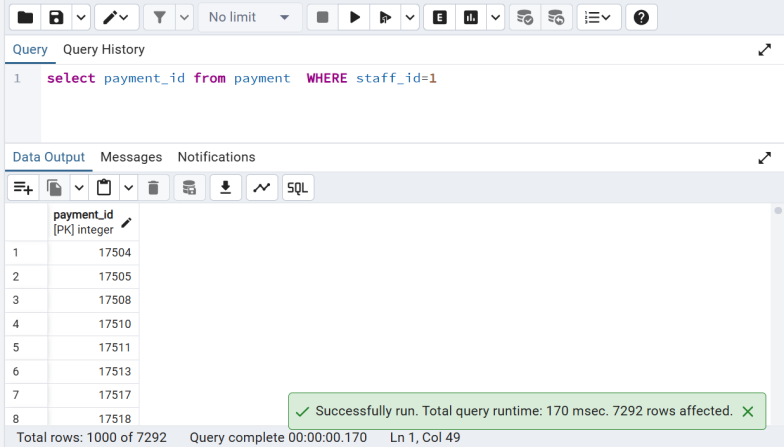
SELECT \* FROM payment WHERE staff\_id = 1;



Ovaj upit moze da radi brzo ako se u tabeli payment nalazi mali broj redova kojima je staff\_id=1, ali ukoliko se vremenom doda mnogo redova kojima je staff\_id=1, taj upit moze da vrati mnogo redova i moze postati spor.

Efikasan upit je upit koji je se izvrsava brzo nad trenutnom bazom, vec ce ostati brz i kada raste baza, a nece se naglo usporiti.

Jednostavnom promenom da umesto koriscenja SELECT \* gde uzimamo sve kolone iz tabele, uzimamo samo kolone koje su nam potrebne mozemo da ubrzamo upit, pogotovo ako tabela ima mnogo kolona. To je osnovna i najcesce primenjivana operacija za optimizaciju upita, ali je to samo vrh brega cele optimizacije upita.



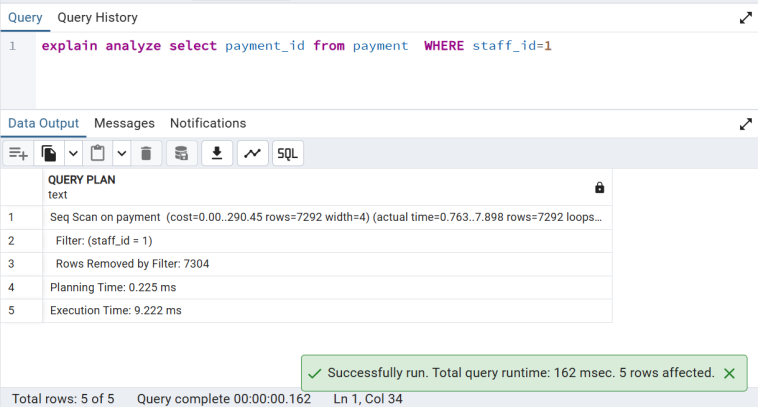
Vazno je napomenuti da je za optimalne performanse kljucno imati i dobar dizajn baze podataka. Ako je baza lose dizajnirana nijedan upit nece biti efikasan u poredjenju sa tim upitom napisanim nad dobro dizajniranom bazom. Takodje, ako je upit lose napisan prednosti dobro dizajna baze nece doci do izrazaja.

## Osnovne metrike performansi SQL upita

1. **Vreme izvrsavanja (Execution Time)**

Najcesce koriscena metrika za merenje performansi SQL upita. Predstavlja vreme koje je potrebno serveru da se upit izvrsi.

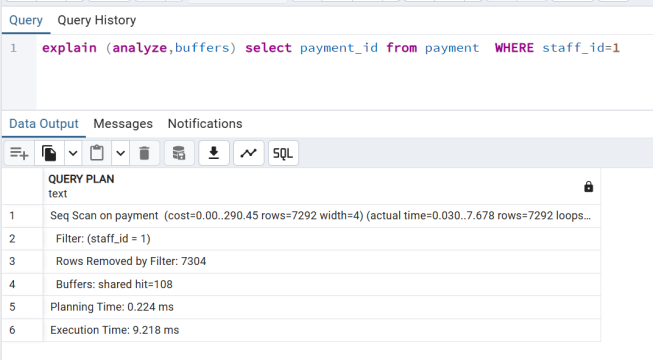
Execution time je razlicitim pojam od total query time. Execution time je samo deo total query time i predstavlja vreme koje je potrebno serveru da izvrsi upit. Execution time je uvek manji od total query runtime.



U PostgreSQL-u se koriste komande EXPLAIN i EXPLAIN ANALYZE (detaljnije obradjeno u 2. poglavlju) za merenje stvarnog vremena izvrsavanja upita. Na vreme izvrsavanja upita uticu velicina dataseta, prisustvo indeksa, vrsta spoja (joina), filtiranje i agregacija.

1. **Broj citanja**

Broj citanja prikazuje koliko puta je baza podataka morala da procita stranice iz memorije ili sa diska da bi izvrsila odredjeni upit. PostgreSQL cuva deo podataka u memoriji da ne mora svaki put da cita sa diska. Citanja iz memorije(shared hit) su znacajno brza od citanja sa diska(read).



Ovde vidimo da su 108 citanja bila shared hit, odnosno da se podaci citaju direktno iz memorije. Nije bilo citanja sa diska. Da je bilo mnogo citanja sa diska, upit bi bio sporiji.

Broj citanja prikazuje koliko se efikasno pristupa trazenim podacima. Veliki broj citanja usporava izvrsavanje. Koriscenje indeksa i particionisanja moze znacajno smanjiti broj citanja a samim tim i poboljsati performanse.

1. **Iskoriscenost CPU-a i memorije**

Kontrola koriscenja hardvera je posebno znacajna u okruzenjima sa velikim brojem korisnika, gde se mnogo upita izvrsava istovremeno, odnosno vise upita koristi zajednicke hardverske resurse. Slozeni spojevi i agregacije mogu izazvati veliko opterecenje procesora. Ukoliko je upitima potrebno vise memorije nego sto je dostupno desava se pisanje privremenih fajlova na disk (*spill to disk*), a kasnije i citanje sa diska, sto je naravno mnogo sporije nego iz memorije i znacajno narusava performanse.

## Anti-pattern u pisanju SQL upita

Anti-pattern je losa praksa pisanja upita. Upiti se mogu ciniti funkcionalno ispravnim kada se izvrsavaju na malim datasetovima, ali kada tabela raste (ili kada se upit umesto lokalne baze sada izvrsava na produkcijskoj bazi koja je uglavnom veca) znacajno raste i vreme izvrsavanja zato sto su upiti neefikasni.

Primeri najcescih SQL anti-patterna

1. SELECT \* - vracanje svih kolona iz tabele iako su potrebne samo neke od njih.
2. Neefikasni spojevi - spajanje velikih tabela bez odgovarajucih indeksa. Previse spojeva koji mogu biti efikasnije struktuirani i kombinovani.
3. Neadekvatno indeksiranje - nedostatak indeksa na koloni koja se cesto upotrebljava ili koriscenje previse indeksa koji usporavaju UPDATE i INSERT.
4. Koriscenje IN kada je moguce koristiti EXISTS - (pojasnjenje u 2.2.)
5. Koriscenje DISTINCT da bi se uklonili duplikati umesto da se upit optimizuje
6. Nepotrebni podupiti u SELECT klauzuli - izvrsavanje podupita za svaki red tabele umesto da se koristi agregacija ili JOIN

Dodavanje indeksa na kolone, selektovanje samo potrebnih kolona, koriscenje filtera i optimizacija JOIN-ova su neki od standarnih nacina za redukovanje kolicine hardvera koju upit zahteva.

# Query execution plan

Query Execution Plan (QEP) je detaljan prikaz kako SQL engine planira da izvrsi jedan upit. Prikazuje korake koje baza preduzima kao sto su redosled operacija, metode za pristup tabelama, koriscenje indeksa, vrste joinova, filtiranja i agregacije. Provera se koje tabele se indeksiraju, koliko redova se obradjuje, da li se kolone indeksiraju. Pomaze u pronalasku takozvanih uskih grla.

EXPLAIN je komanda u PostgreSQL-u koja omogucava programeru da pregleda plan izvrsenja upita (QEP), ali ne vraca stvarne podatke jer se taj upit ne izvrsava nad bazom. Korisno je u situacijama kada se samo ispituju performanse baze, bez da se izvrsavaju nepotrebni upiti nad tom bazom.

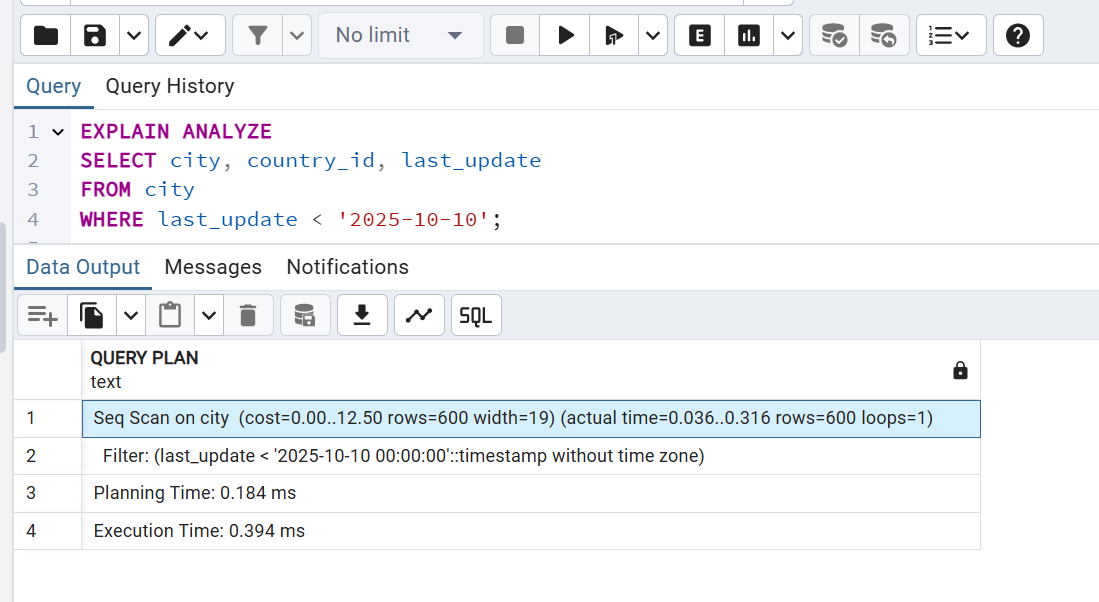
EXPLAIN ANALYZE je komanda u PostgreSQL-u koji prikazuje plan izvrsenja i izvrsava sam upit. Prikazuje vreme citanja svake tabele, broj obradjenih redova, ukupni trosak operacija.

Kljucni elementi u izlazu EXPLAIN i EXPLAIN ANALYZE su:

1. Access Methods (nacin na koji se vrsi pristup podacima)
   1. Seq Scan - citanje cele tabele
   2. Index Scan - citanje tabele pomocu indeksa
   3. Index Only Scan - koristi indeks i ne cita tabelu ako su sve potrebne kolone u indeksu
   4. Bitmap Index Scan - kreira bitmapu relevantnih redova iz indeksa, svaki blok se oznava sa 1 ako treba da se procita i sa 0 ako ne treba
   5. Bitmap Heap Scan - cita samo blokove oznacene sa 1 u bitmapi
2. Join Methods (nacin spajanja tabela)
   1. Nested loop join - dobar za male setove podataka i kada postoje indeksi
   2. Merge join - efikasan ako su obe tabele vec sortirane
   3. Hash join - efikasan kod velikih nesortiranih tabela
3. Filter

Prikazuje uslove za WHERE i HAVING komande koje se primenjuju na podatke u toj operaciji.

1. Agregacije - nacin na koji se racunaju SUM, AVG, COUNT …
2. Operacije sortiranja - operacije sortiranje rezultata, ORDER BY
3. Planning time
4. Execution Time
5. Cost



Seq Scan - znaci da baza cita celu tabelu city red po red.

U prvoj zagradi se nalaze vrednosti koje procenjuje optimizator:

Cost=0.00..12.50 - znaci da je startup cost 0.00 a total cost 12.50. Ove jedinice su interne jedinice PostgreSQL-a koje sluze samo za uporedjivanje upita sa ostalim PostgreSQL upita.

Rows=600 - optimizer procenjuje da ce biti vraceno 600 redova kao rezultat upita.

Widht = 19 - optimizer procenjuje da je prosecna velicina jednog reda 19 bajta.

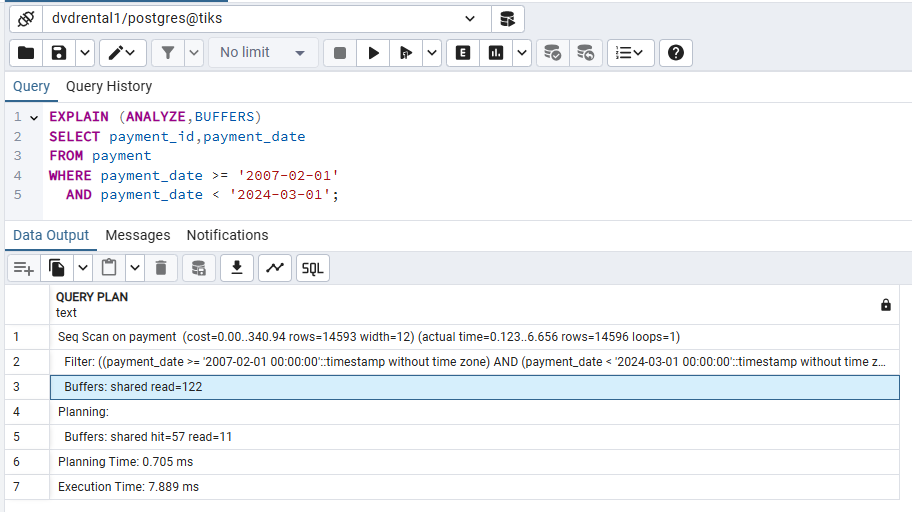
U drugoj zagradi se nalaze stvarne vrednosti nakon izvrsavanja upita:

Actual time - 0.036.. 0.316 - znaci da je posle 0.036ms pocelo citanje redova a da je posle 0.316ms zavrseno citanje redova.

Rows=600 - broj vracenih redova u rezultatu upita

Loops=1 - operacija je izvrsena jednom.

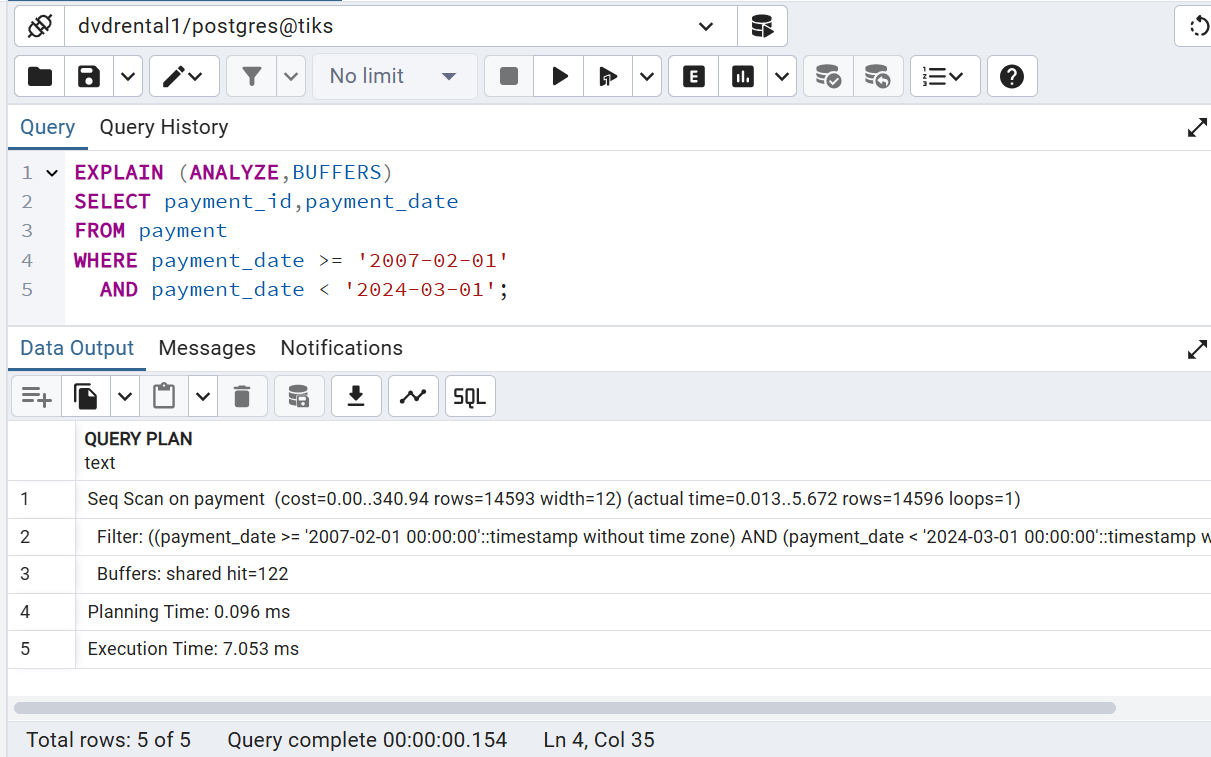
Citanje iz memorije vs citanje sa diska



U delu Planning: Buffers se prikazuje koliko stranica je PostgreSQL procitao tokom planiranja upita, shared hit=57 a shared read=11. U toj fazi se citaju metapodaci, indeksi i particije nad tabelom. 57 stranica je bilo vec u memoriji a 11 je procitano sa diska.

U oznacenom plavom delu Buffers se prikazuje koliko stranica je PostreSQL procitao tokom samog izvrsavanja upita, odnosno stranica u kojima su smesteni stvarni podaci iz tabele. Vidimo da su sve stranice procitane direktno sa diska, shared read = 122.

Ako taj isti upit izvrsimo jos jednom sve stranice ce vec biti ucitane u memoriji:



Primecujemo da je sada shared hit = 122, odnosno sve stranice koje upit koristi su ucitane iz memorije sto bi trebalo da bude brze.

Medjutim, moze se primetiti da je Execution time priblizno jednak u oba slucaja, malo preko 7ms za oba slucaja. To se desava zato sto je ukupan broj redova u tabeli 14593. To je relativno mali broj redova i ne moze se primetiti razlika u brzini izmedju citanja sa diska i citanja iz memorije.

Za potrebe uporedjivanja performansi izvrsava se upit koji duplira broj redova tabele:

INSERT INTO payment (customer\_id, staff\_id, rental\_id, amount, payment\_date)

SELECT customer\_id, staff\_id, rental\_id, amount, payment\_date

FROM payment;

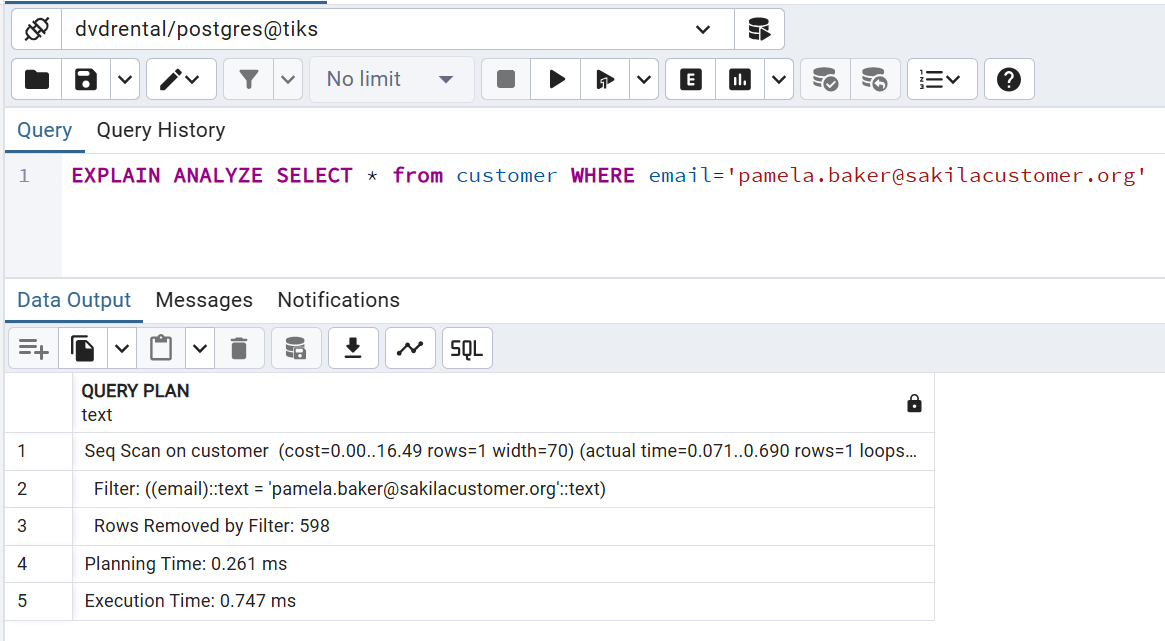
Nakon visestrukog izvrsavanja ovog upita, tabela ima veliki broj redova, i razlika izmedju citanja sa diska i rama moze biti osetna. Ipak, PostgreSQL koristi tuples streaming, sto znaci da ne ubacuje sve rezultate upita u memoriju odjednom. Redovi se u memoriju smestanu u batchevima. Upravo zbog slucaja kada postoje ogromne tabele sa milionima redova. Kada bi sve redove pokusao da ubaci u memoriju, nestalo bi mesta u memoriji, cela memorija bi bila zauzeta redovima iz tabele sto bi uzorkovalo probleme.

# Indeksi i njihova uloga u optimizaciji upita

## Sta je indeks i kako funkcionise

Indeks u SQL bazi podataka je posebna **fizicka** struktura podataka koje omogucava brze pronalazenje redova bez potrebe da se pregledava cela tabela red po red. Ta struktura podataka sadrzi kopije vrednosti iz indeksiranih kolona, zajedno sa pointerima ka redovima u tabeli. Podrazumevana struktura je B-tree (balansirano stablo). Indeks se cesto poredi sa indeksom na kraju knjige, umesto da se cita cela knjiga, odmah moze da pronadje stranica na kojoj se nalazi odredjena rec.

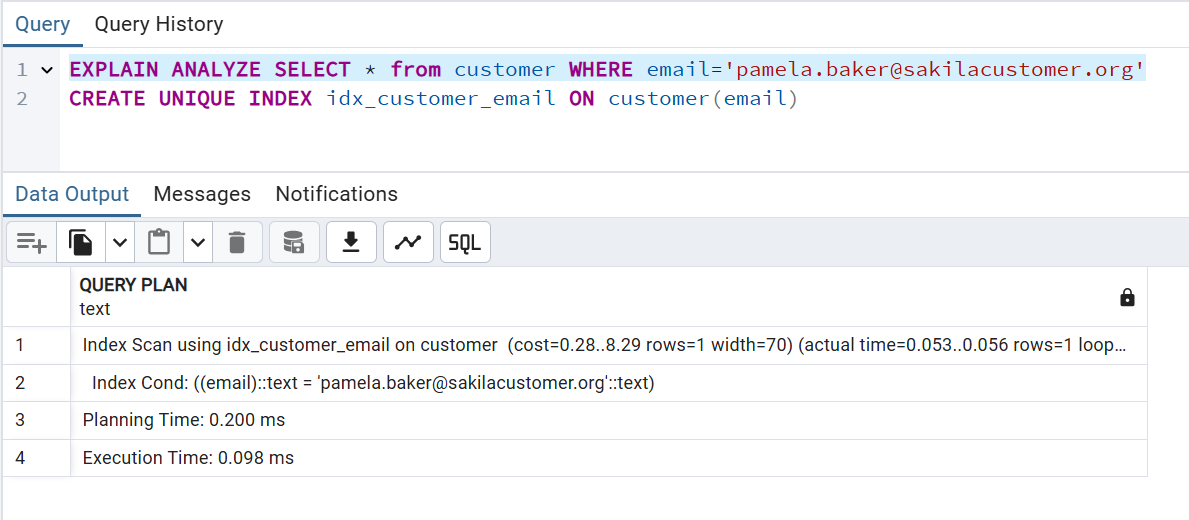
SELECT \* from customer WHERE [email='pamela.baker@sakilacustomer.org'](mailto:email='pamela.baker@sakilacustomer.org')



Indeks na koloni email tabele custom trenutno ne postoji. Tabela se pretrazuje sekvencijalno (Seq Scan iz query plana). Indeks se moze kreirati komandom

CREATE UNIQUE INDEX idx\_customer\_email ON customer(email);

Kada postoji indeks na koloni email, baza nece pregledati sve redove u tabeli vec ce vrsiti pretragu u sortiranom stablu i brze pristupiti redovima koji odgovaraju uslovu. Slozenost trazenja sa indeksom je O(log n) dok bi bez indeksa slozenost bila O(n), sto znaci da sto vise redova ima u tabeli to je veca razlika u performansama u korist indeksa. U ovom slucaju posto je email jedinstven za tabelu, pretraga ce odmah pronaci tacno jedan red sa datom email adresom, ukoliko postoji.



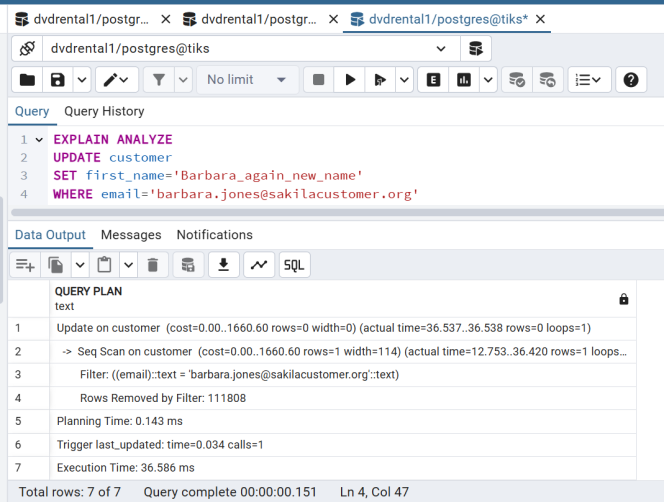
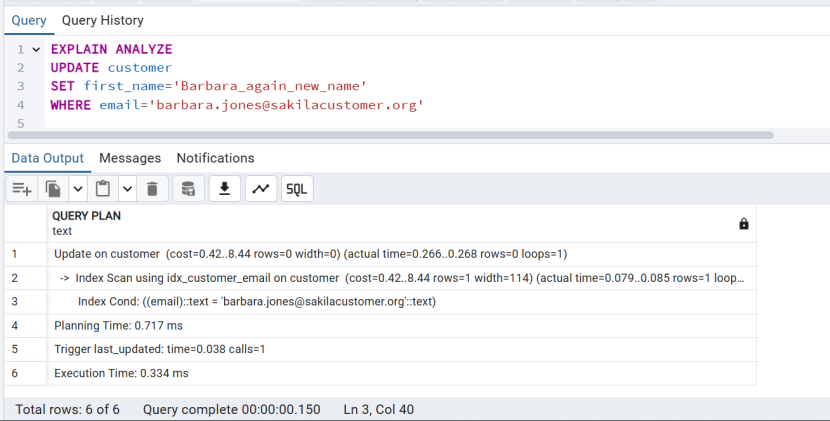
Izvrsavanjem oznacenog upita , vidimo da je Execution Time 0.098ms, sto je u poredjenju sa 0.75ms iz prethodnog primera kada se isti ovaj upit izvrsio bez indeksa, ogromna razlika. Upit se skoro 10 puta brze izvrsava.

Indeksi se najcesce koriste kada se kolona cesto koristi u WHERE uslovima, za brze pronalazenje redova koji zadovoljaju uslov. Koriste se i kod JOIN operacija, kada se spajaju dve tabele putem primarnog i stranog kljuca. U tim slucajevima indeks na odgovarajucim kolonama omogucava efikasno uparivanje redova, bez potrebe za potpunim skeniranjem tabela. Kada se koriste klauzule ORDER BY i GROUP BY a kolona po kojoj se vrsti sortiranje ili grupisanje vec postoji u indeksu( u pravilnom redosledu), baza moze da iskoristi indeks i izbegne dodatno sortiranje, sto znatno poboljsava performanse.

Vazno je napomenuti da indeksi imaju i svoje negativne strane. Indeksi omogucavaju brze citanje ali pisanje moze biti sporije kod bulk upisa zato sto se pri svakom kreiranju, azuriranju ili brisanju podataka moraju dodatno azurirati relevantni indeksi. Previse indeksa usporava sistem i zbog toga ih treba praviti samo na kolonama koje se cesto koriste za filtriranje (WHERE), spajanje (JOIN) ili sortiranje (ORDER BY, GROUP BY). Dobra je praksa koristiti ih kada se neki podaci cesto citaju ali se ne menjaju cesto. Indeksi zauzimaju prostor na disku sto je jos jedna mana indeksa. Pravilno koriscenje indeksa je kljuc za balans izmedju brzog citanja podataka i efikasnost upisa podataka.

Ne mora da znaci da ukoliko postoji indeks nad kolonom da ce upis biti sporiji nego da nema indeksa, pogotovo kod pojedinacnih upita. Moze se desiti:

Index Scan time + Index update time < Seq Scan time



U primeru iznad vidimo da je azuriranje malog broja redova nad kolonom sa indeksom mnogo brze nego azuriranje reda bez indeksa.

Indeks je fizicki smesten u stranicama na disku. Svaka stranica ima organicenu velicinu i u njoj se nalaze kljucevi indeksa i pokazivaci ka odgovarajucim redovima u heapu (tabeli).

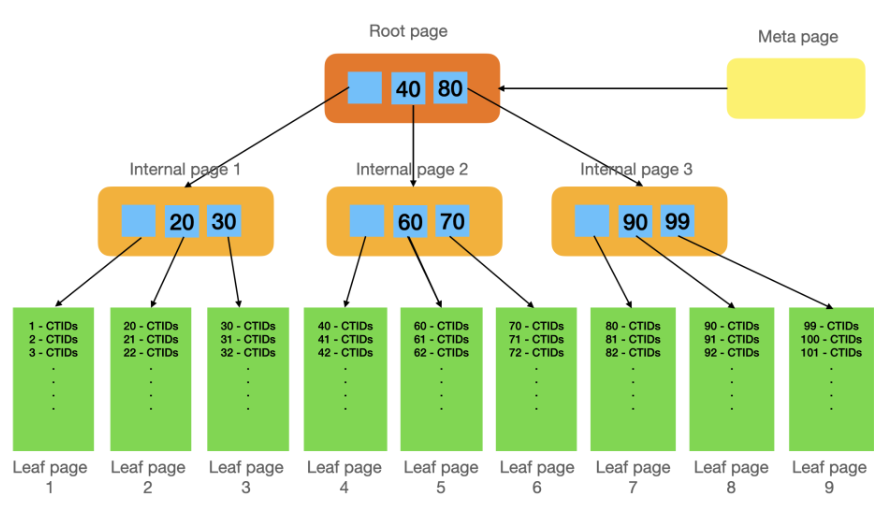
U PostgreSQL-u B-tree indeks se sastoji od 4 tipa stranica organizovanih u hijerarhijsku strukturu:

1. Meta stranica
2. Root stranica
3. Internal stranica
4. Leaf stranica

Meta stranica je prva stranica indeksa i ona sadrzi samo meta-podatke i pokazuje na root stranicu.

Root stranica je prva stranica koja pokazuje na internal stranice i leaf stranice. Root stranica je ista kao i internal stranica, jedina razlika je sto je root stranica u samom korenu stabla i postoji samo jedna, dok je internal stranica vise.

Leaf stranice se nalaze u poslednjem nivou stabla i ne mogu se prosirivati dalje. One zapravo sadrze informacije koje su nam potrebne odnosno pokazivace na redove tabele gde se nalazi odredjena vrednost.



Kada se ubacuje novi red, baza treba da ga stavi u list page koji odgovara vrednosti kolone po kojoj je indeksiran. Ako na leaf stranici nema vise slobodnog mesta desava se **page split**. To znaci da se stranica deli na dve stranice, polovina kljuceva ostaje na staroj stranici a polovina se premesta u novu stranicu. Roditeljski cvor se azurira da pokazuje i na novu stranicu.

Ukoliko se cesto vrsi INSERT, UPDATE ili DELETE operacije nad podacima koji su deo indeksa moze nastati **fragmentacija indeksa**. Fragmentacija je posledica mnogobrojnih page splitova, brisanja i dodavanja podataka. Indeks se fizicki skladisiti na disku, a fragmentacija indeksa znaci da su stranice na kojima se indeks skladisti nepovezane ili nepotpuno popunjene. To za rezultat ima da citanje indeksa postaje sporije. Ako baza koristi B-tree indeks, listovi sadrze pokazivace na redove u tabeli(heap). Svaki list ima ogranicenu velicinu, odnosno ograniceni broj pointera na jednom listu.

Ako se menja kolona koja je deo indeksa stari kljuc se brise iz lista. Odnosno pokazivac ka tom redu u tabeli se uklanja iz indeksa a dodaje se novi kljuc. Posto je novi kljuc razliciti od prethodnog dodaje se na nekom drugom listu, u zavisnosti od vrednosti kljuca. To moze ostaviti puno praznog prostora na listovima ukoliko su azuriranja cesta. Ako se mnogo kljuceva dodaje istom listu, desava se ista situacija kao prethodno opisana, stranica se deli i nastaje page split.

Ukoliko se desavaju mnogo brisanja podataka, nastaju dosta praznih mesta na listovima, i indeks vise nije optimalno popunjen. Zbog toga je pretrazivanje po indeksu sporije nego sto bi moglo biti da indeks ima manje stranica.

U PostgreSQL-u se komandom REINDEX moze izvrsiti restruktuiranje indeksa. Indeks ce se rekreirati i ponovo poredjati stranice da idu redom kako bi se efikasnije pristupalo indeksu.

REINDEX INDEX idx\_customer\_email

Podela indeksa po logickoj svrsi:

* Primarni indeks (PRIMARY KEY) - automatski se kreira za primarni kljuc
* Jedinstveni indeks (UNIQUE) - obezbedjuje da nema duplih vrednosti
* Slozeni indeks (COMPOSITE) - indeks nad vise kolona
* Neklasterovani indeks - cuva se odvojeno od podataka a ukazuje na lokaciju reda
* Klasterovani indeks - obezbedjuje redosled fizickog cuvanja redova u tabeli

Podela indeksa po tipu strukture:

* B-tree - standardni upiti sa poredjenjem i sortiranjem
* Hash - brza pretraga po jednakosti
* GIN - full text search za json i array tipove
* GIST - prostorni podaci, geometrija
* SP-GIST - nepravilno rasopredjeni podaci (quadtrees, k-trees)
* BRIN - velike tabele sa sekvencijalnim redovima
* Expression - funkcije i izrazi nad kolonom
* Partial - Samo deo tabele u zavisnosti od uslova

## Partial i expression indeksi

**Partial indeks** se pravi na koloni ali samo za one redove koji zadovoljavaju odredjeni uslov.Time se smanjuje prostor na disku koji indeks zauzima.

Na primer ukoliko se uglavnom u cita email ali samo za one musterije koje su aktivne, indeks iz prethodnog primera idx\_customer\_email bi mogao da se unapredi:

CREATE index idx\_customer\_email\_active

ON customer(email)

WHERE active=1

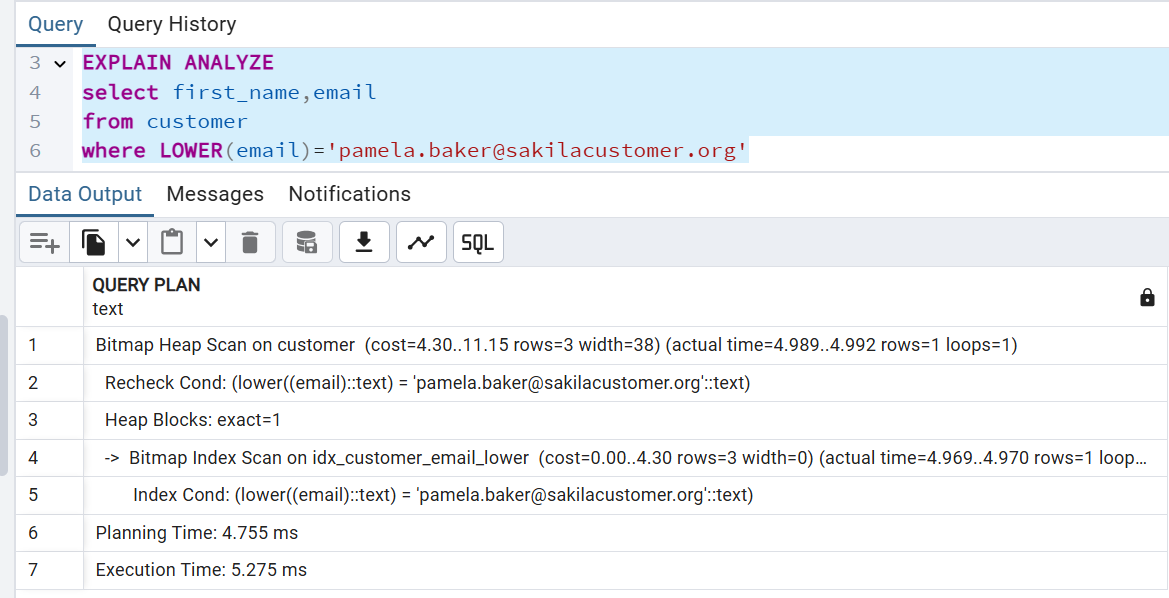
Sada ce indeks biti konkretniji, zauzimace manje prostora na disku i biti jos brzi. Negativna strana je sto ce raditi samo ukoliko se filtira active=1. Bez tog filtriranja indeks se ne bi mogao primeniti i primenilo bi se sekvencijalno trazenje.

**Expression indeks** je indeks koji se pravi na izrazu koji ukljucuje kolonu a ne direktno na samoj koloni. Omogucava optimizaciju upita koji koriste funkcije ili transformacije. Smanjuje potrebu za dodatnom kolonom u tabeli samo radi indeksiranja.

Zamislimo da imamo situaciju u bazi da neke email adrese pocinju velikim pocetnim slovom a neki malim pocetnim slovom. Bilo bi neophodno da se prilikom svakog pretrazivanja po emailu koristi funkcija LOWER() da bismo osigurali poklapanje. Zbog toga se kreira indeks:

CREATE INDEX idx\_customer\_email\_lower ON customer (LOWER(email));

Ovaj indeks omogucava brzo pronalazenje rezultata cak i kada je u upitu primenjena funkcija LOWER()



S obzirom da funkcija LOWER() moze da prevede vise redova na isti rezultat, PostgreSQL mora prvo da napravi bitmapu redova koji zadovoljavaju uslov (Bitmap Index Scan) a onda ucita te redove iz tabele sa stvarnim redovima na disku(Bitmap Heap Scan).

# Koriscenje spojeva i podupita na optimalan nacin

Join je cesto brzi i efikasniji od podupita jer SQL optimizator moze bolje da isplanira izvrsavanje upita kada ima eksplicitne veze izmedju tabela.

Vrste JOIN-ova:

1. Inner join

Najcesce korisceni spoj u praksi. Vraca samo one redove koji imaju podudarajuce vrednosti u obe tabele koje ucestvuju u joinu.

1. Left join

Vraca sve redove iz leve tabele i redove koji imaju podudarajuce vrednosti iz desne tabele. Ako nema podudaranja kolone iz desne tabele su NULL.

1. Right join

Vraca sve redove iz desne tabele i redove koji imaju podudarajuce vrednosti iz leve tabele. Ako nema podudaranja kolone iz leve tabele su NULL.

1. Full join

Kombinuje left i right join. Vraca sve redove iz obe tabele cak i ako nema podudaranja.

1. Cross join - Kartezijanski proizvod. Vraca sve moguce kombinacije redova iz obe tabele.

## EXISTS vs IN

Oba operatora se koriste za filtiranje podataka pomocu podupita. Osnovna razlika izmedju ova dva operatora je ta da EXISTS proverava da li podupit vraca barem jedan odgovarajuci red i prestaje proveru cim nadje prvi odgovarajuci red a operator IN prvo izvrsava podupit i vraca sve vrednosti a nakon toga poredi svaki vrednost glavnog upita sa tim vrednostima.

Operator EXISTS podrzava rad samo sa podupitima,

SELECT \*

FROM Company c

WHERE EXISTS (

SELECT 1

FROM Employees e

WHERE e.CompanyID = c.CompanyId

AND e.Country = ‘Serbia’)

Operator EXISTS u ovom primeru radi tako sto uzima prvu kompaniju iz tabele Company, zatim proverava da li postoji red u tabeli Emplyoees takav da je CompanyID iz tog reda jednak sa CompanyID kompanije uzete iz tabele Company i da je polje Country tog reda jednako sa ‘Serbia’. Ukoliko ne ne postoji takav red u tabeli Employees, uzima se sledeci red iz tabele Company i pocinje se proveravanje u tabeli Employees ispocetka. Ukoliko se nadje jedan red koji ispunjava te uslove, izvrsavanje se prekida i funkcija EXISTS vraca true, ukoliko se ne pronadje nijedan odgovarajuci red, funkcije vraca false.

Operator IN podrzava rad i sa listama i sa podupitima.

SELECT \*

FROM Company

WHERE CompanyCode IN (‘SU’, ‘CE’, ‘ME’);

SELECT \*

FROM Company

WHERE CompanyID IN

(SELECT CompanyID

FROM Employees

WHERE Country= ‘Serbia’).

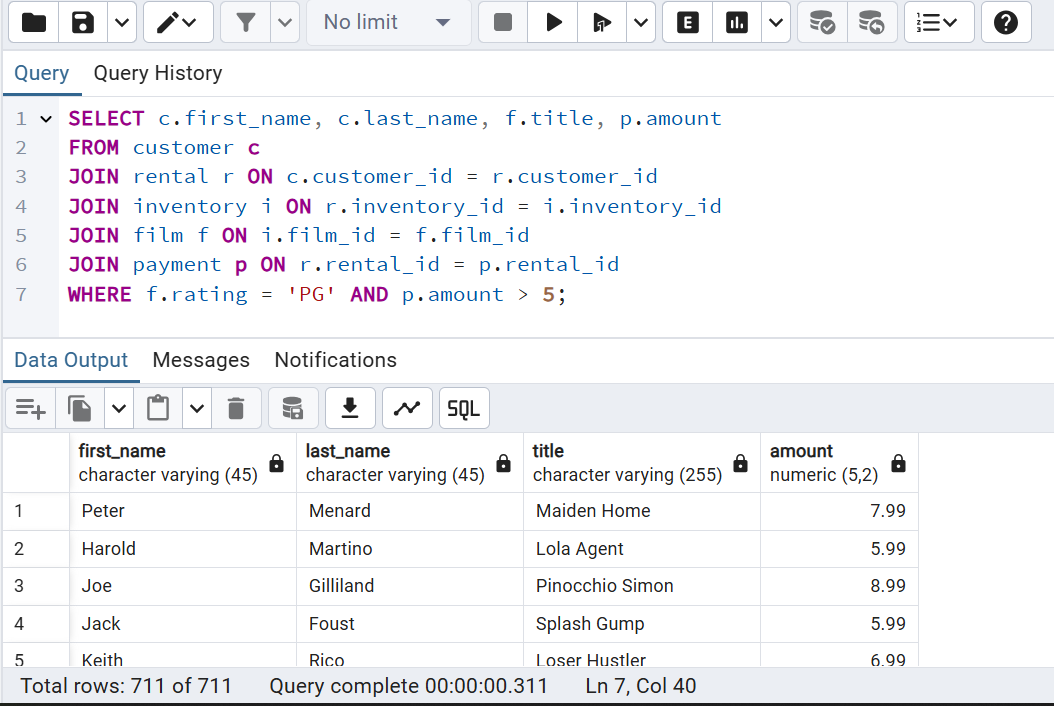
Operator IN radi tako sto se prvo izvrsi podupit i vrate se sve vrednosti koje zadovoljavaju uslove tog podupita. Nakon toga se poredi svaki red iz tabele Company po CompanyID sa vrednostima dobijenih iz podupita, i redovi koji se poklapaju se vracaju kao rezultat upita.

Exists je efikasniji za velike podupite, kada podupit vraca mnogo redova, zato sto se izvrsavanje upita prekida cim se pronadje jedan red koji odgovara uslovu. Operator IN je efikasan za liste sa malim brojem vrednosti.

## Optimizacija visestrukih JOIN-ova

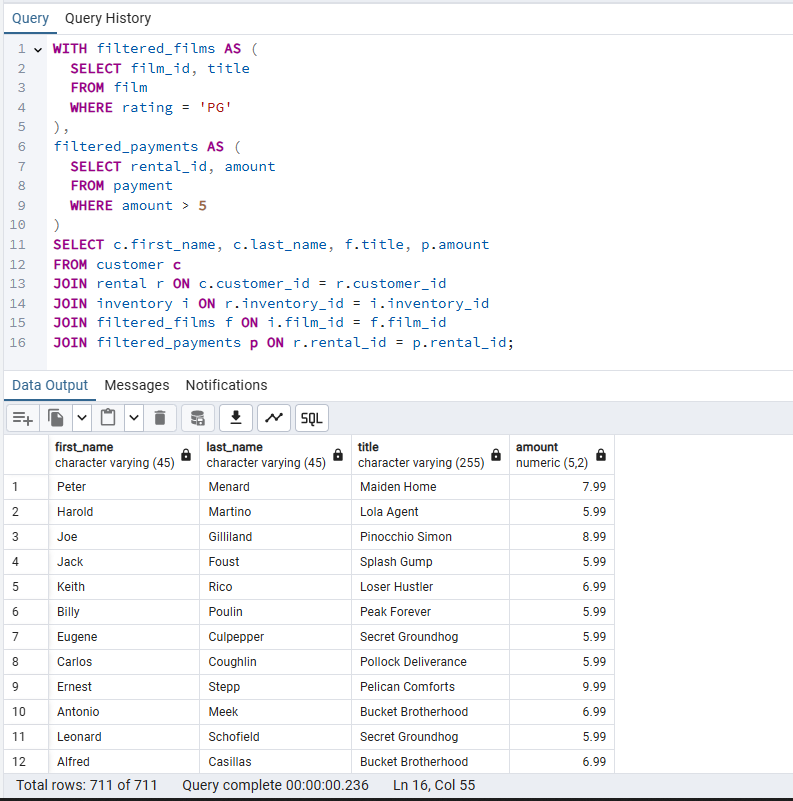
Kada upit sadrzi vise JOIN operacija, baza mora da odluci kojim redosledom se spajaju tabele i kojim mehanizmom ce se izvrsiti spajanje ( hash join, merge join, nested loop). Ukoliko upit nije dobro optimizovan onda broj kombinacija i medjurezultata moze biti veoma veliki sto ce usporiti izvrsavanje upita.

Preporuka je da se podaci filtriraju sto je moguce vise pre JOIN operacije kako bi se smanjila velicina medjurezultata i poboljsale performanse upita.



U primeru iznad se filtiranje vrsi tek nakon sto se velike tabele spoje sto daje nepotreban a veliki medjurezultat koji se kasnije filtrira. To znaci da je baza prosla kroz veliki broj redova koji su se nakon toga filtrirali, odnosno nisu svi potrebni u krajnjem rezultatu.

Mnogo bolja praksa je da se ranije izvrsi filtiranje kako bi se upit izvrsavao nad manjim brojem podataka cime se ubrzava izvrsavanje.



U primeru iznad vidimo da se koristi kljucna rec WITH koja u SQL-u oznava CTE(Common Table Expression), odnosno neku vrstu privremene tabele koja postoji samo tokom izvrsavanja upita. Te tabele predstavljaju filtrirane podatke iz tabele i mogu da se koriste u izvrsavanju glavnog upita.

Koriscenjem privremenih tabela u kojima su smesteni filtrirani podaci, JOIN-i rade sa tabelama koje imaju manji broj podataka, odnosno sadrze samo potrebne podatke i vreme izvrsavanja upita je krace.

Kolone koje se cesto koriste u ON izrazu prilikom spajanja tabela treba da budu indeksirane. Primarni kljuc tabele se obicno automatski indeksira, bez potrebe da ga programer eksplicitno indeksira.

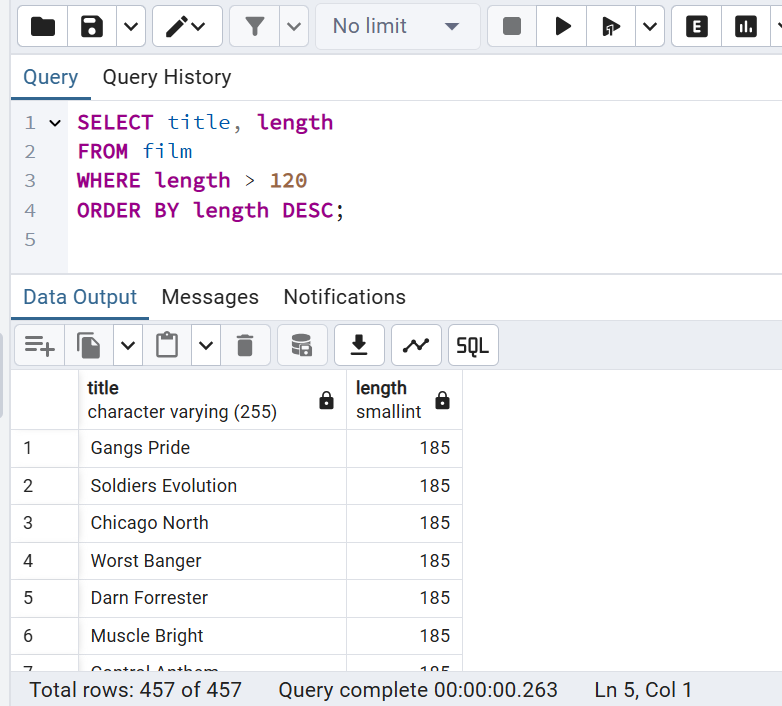
S obzirom da se spajanje tabela najcesce vrsi preko stranih kljuceva, strani kljucevi su kolone koje se u praksi najcesce eksplicitno indeksiraju.

Uloga indeksa u razlicitim JOIN algoritmima:

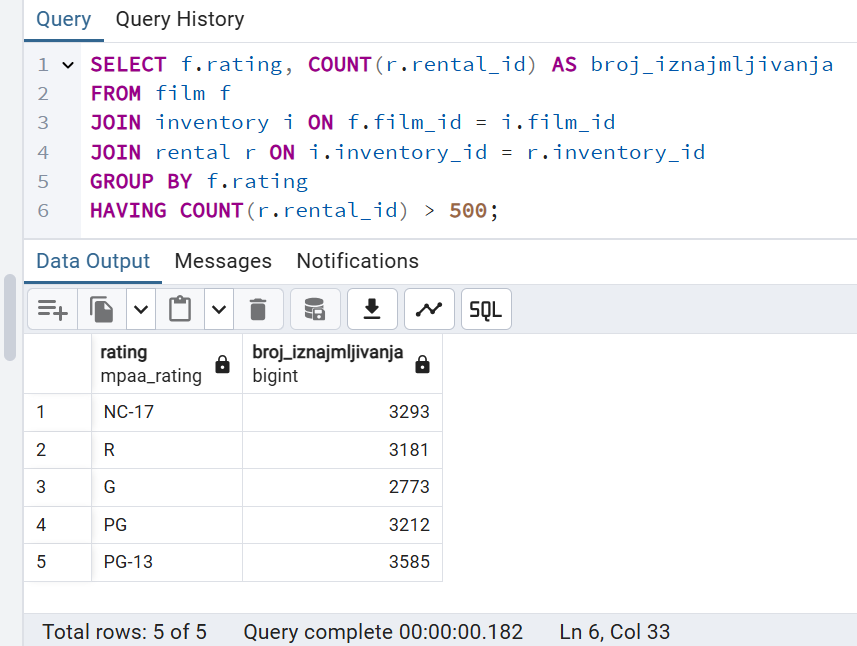
* Nested Loop JOIN - u ovom slucaju je indeksiranje najvaznije. Za svaki red iz prve tabele baza trazi odgovarajuce redove u drugoj tabeli. Funkcionise kao ugnjezdene for petlje kao sto mu i naziv kaze. Ako postoji indeks slozenost pretrage je O(log n), sto znaci da se broj koraka potrebnih za pronalazenje nekog reda u tabeli povecava logaritamski sa velicinom tabele a ne linearno kao kada indeks ne postoji. Kada indeks ne postoji proverava se svih n redova, odnosno O(n). S obzirom da logaritamska funkcija raste mnogo sporije od linearne, sto su vece tabele i broj redova u tabeli razlika u brzini izvrsavanja je vidljivija.
* Hash JOIN - Kreira se hash tabela iz jedne tabele a za svaki red iz druge tabele se provereva da li taj red postoji u hash tabeli. U ovom slucaju indeks nije obavezan jer se pravi hash tabela u memoriji, ali indeks ipak moze pomoci da se podaci brze filtriraju pre ulaska u samog ulaska podataka u JOIN.
* Merge JOIN - Baze sortiraju obe tabele po koloni na osnovu koje se spajaju. Merge JOIN najbolje radi kada su obe tabele vec sortirane po kolonama po kojima se spajaju. Ako postoji indeks koji podrzava sortiranje baza moze koristititi indeksiran pristup umesto dodatnog sortiranja.

# Optimizacija agregacija i grupisanja

Za filtriranje podataka pre nego sto se primeni agregacija koristi se WHERE klauzula. Ovo smanjuje broj redova koji se obradjuju i poboljsava perfomanse upita. Klauzula WHERE se koristi za filtriranje pojedinacnih redova pre grupisanja.



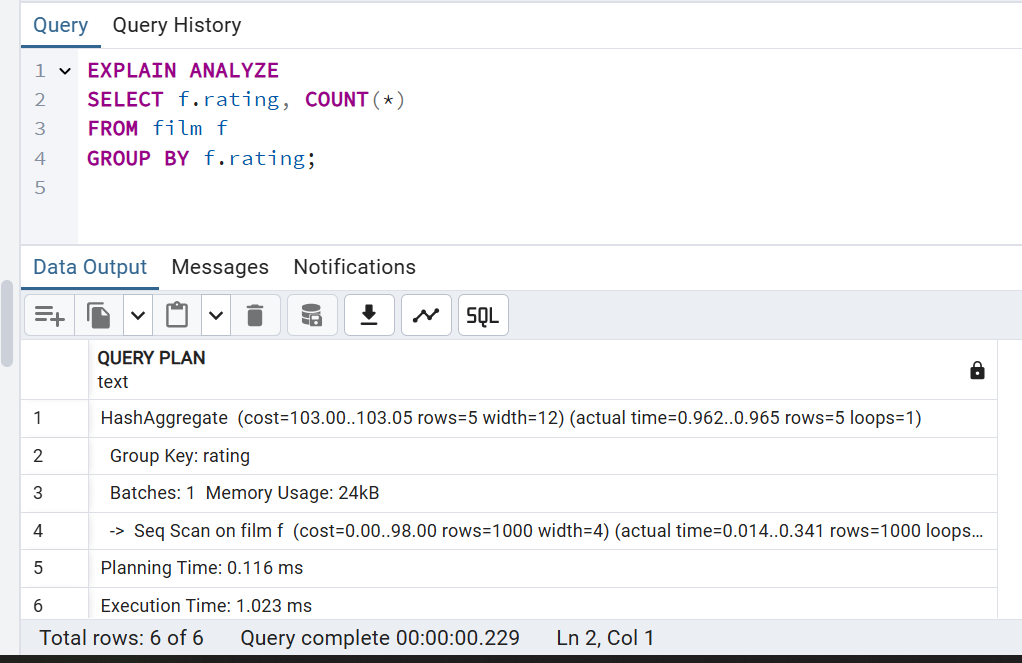
Za filtriranje podataka nakon grupisanja se koristi HAVING klauzula. Ova klauzula se primenjuje na agregacione funkcije kao sto su SUM, AVG, COUNT…



Ovaj upit nije moguce napisati sa WHERE klauzulom zato sto je neophodno prvo izracunati COUNT(r.rental\_id) a tek onda izvrsiti filtriranje na osnovu rezultata funkcije count.

Pozeljno je koristiti WHERE klauzulu umesto HAVING kada je to moguce, s obzirom da je optimalnije da se podaci sto ranije filtriraju kako bi bilo manje podataka za obradu.

GROUP BY zahteva sortiranje po koloni po kojoj se grupise. Ako vec postoji indeks na toj koloni, PostgreSQL moze da iskoristi taj indeks i na taj nacin izbegne dodatno sortiranje.

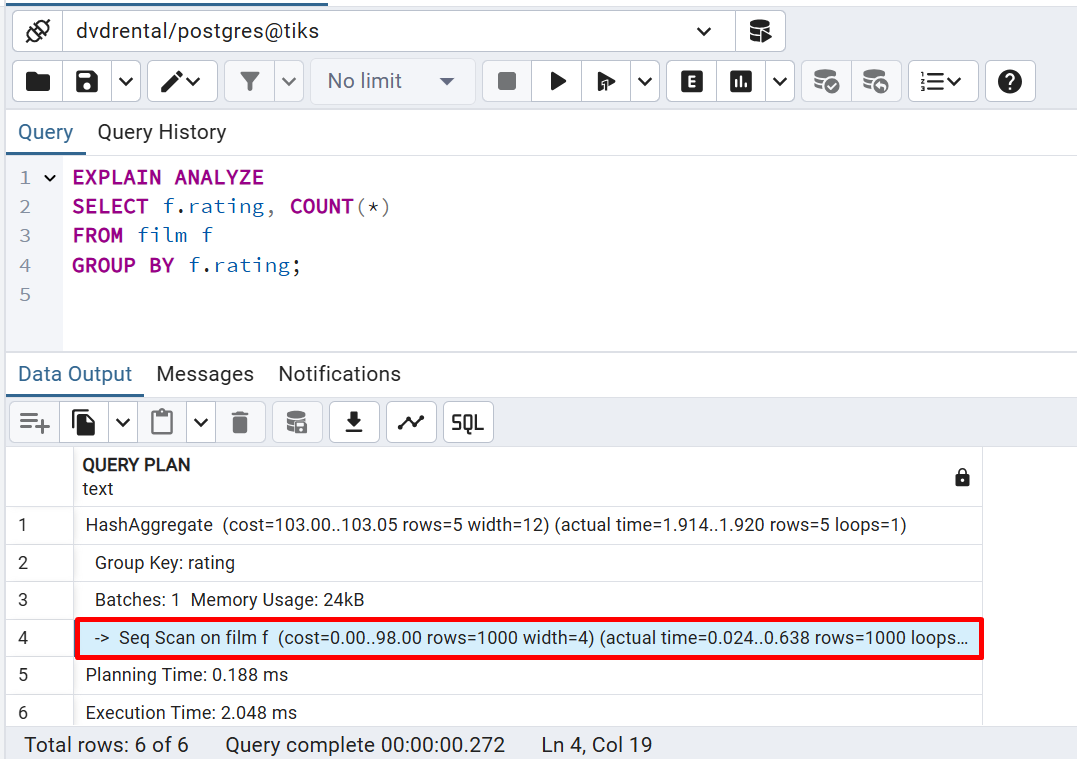




Kreiranjem indeksa na koloni film pravi se indeks i podaci su automatski sortiranu u okviru tog indeksa.

Sada kada se opet izvrsi primer iznad izvrsenje bi trebalo biti efikasnije s obzirom da se ne trosi vreme na sortiranje kolone nego se samo iskoriste podaci iz indeksa kao sto je prikazano u dosadasnjim primerima.

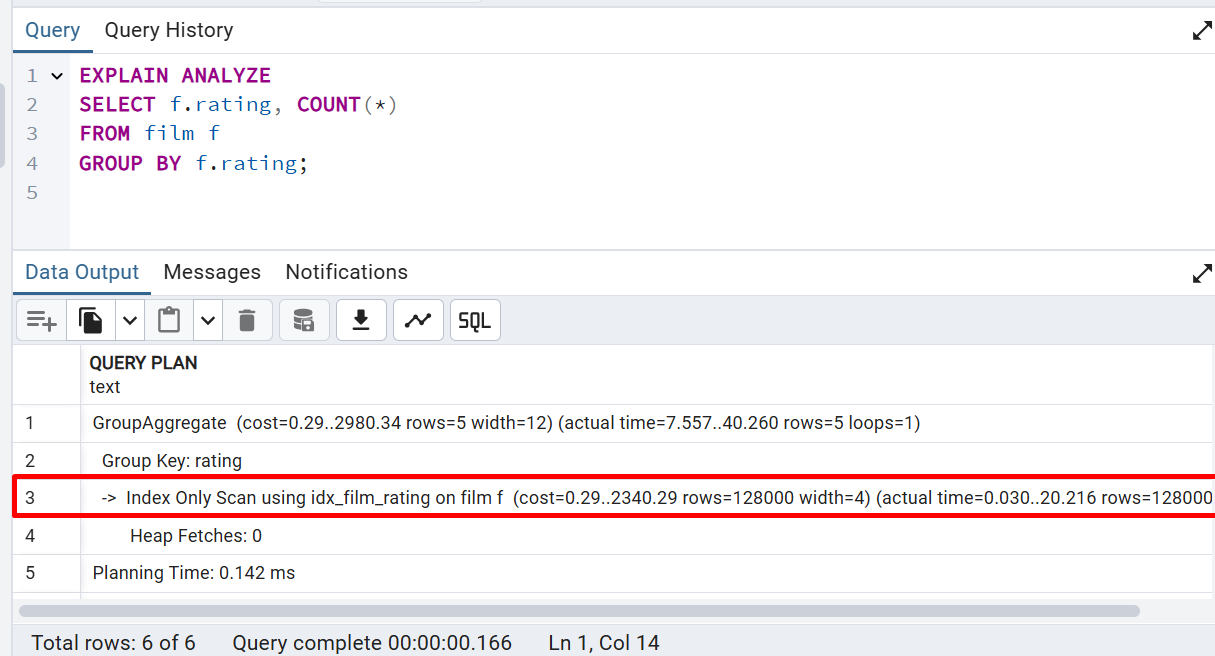
Ipak, ovaj primer je specifican zato sto tabela na kojoj je napravljen indeks ima 1000 redova, sto se smatra malom tabelom i podaci mogu da stanu na svega par stranica na disku. PostgreSQL je procenio da je brze da izvrsi sekvencijalno citanje podataka pa cak i ako postoji indeks nad kolonom on ga ne koristi jer ce to zapravo usporiti izvrsavanje.



Vestacko uvecavanje podataka u tabeli film visestrukim izvrsavanjem komande koja duplira broj redova u tabeli, samo da bi se dobili mock podaci i uvecala tabela za potrebe primera



Sada tabela ima 128000 redova, vise nije mala tabela, i kada se izvrsi grupisanje po koloni rating, PostgreSQL ce koristiti postojeci indeks nad tom kolonom zato sto je u ovom slucaju optimalnije koristiti ga.



# Materialized views i Query rewriting

Kod klasicnog view-a se uvek prikazuju najnoviji podaci ali to u nekim slucajevima moze biti izuzetno sporo a nepotrebno. Zbog toga se koriste Materializied views.

Materialized views je predefinisana tabela koja cuva rezultat vec izvrsenog upita. Umesto da se upit izvrsi nad celom tabelom svaki put, materialized view vec sadrzi rezultat upita u posebnoj tabeli. Baza ne mora svaki put da obradjuje sve redove tabele sto drasticno smanjuje vreme izvrsenja. Sva slozena filtriranja, agregacije i spojevi koji bi uticali na vreme izvrsenja da nije iskoriscen materiazlied view su vec izvrsena i opterecenje procesora je znacajno smanjeno.

Za razliku od obicnog view-a koji se osvezava svaki put kada se upit izvrsava, materiazlied view je fizicki sacuvan i moze se periodicno osvezavati. Time se smanjuje vreme izvrsenja upita jer se ne obradjuju svi redovi iz osnovnih tabela svaki put. Moze sadrzati agregacije, spojeve i filtriranja koja se cesto koriste.

## Kreiranje i odrzavanje Materialized views

Za kreiranje Materiazlied view-a se se u PostgreSQL-u koristi komanda CREATE MATERIALIZED VIEW



Rezultat ovog upita se cuva u bazi kao tabela film\_revenue\_summary.

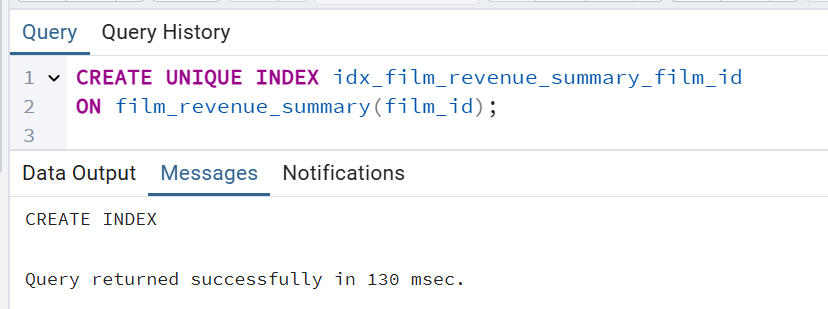
Da bi se osvezio sadrzaj materialized view-a koristi se komanda

REFRESH MATERIALIZED VIEW

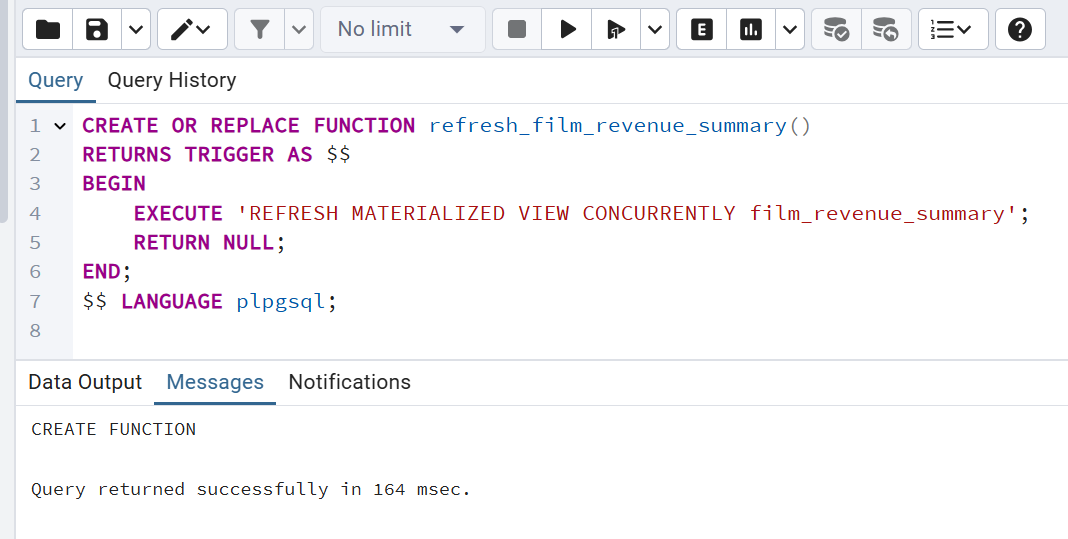
Ova komanda brise stare podatke iz materialized view-a, izvrsava upit na osnovu koga se prave podaci u tom materialized view-u i rezultat upita smesta u njega.

Najcesca praksa je da se definise odredjeni vremenski interval nakon kojeg se upit izvrsava i azurira se materialized view. Ukoliko je potrebno da se materilized view automatski azurira kada se doda red u tabeli to se moze uraditi pomocu trigera. Ovu opciju nije dobro koristiti kada se cesto dodaju redovi jer se onda gubi svrha materialized view-a, vec je pozeljno koristiti je ukoliko se redovi dodaju retko i samo tada je potrebno azurirati materialized view.

Najpre je neophodno kreirati indeks na tom materialized view-u da bi se omogucilo citanje bez blokiranja, odnosno konkurentno citanje.



Nakon toga je potrebno napraviti PostgreSQL funkciju koja ce pozivati triger svaki put kada se u tabeli payment doda novi red.



Ukoliko programer zna da postoji materialized view sa zeljenim rezultatom, moze odmah pisati upit da koristi materialized view umesto osnovne tabele. Ako postoji materialized view film\_revenue\_summary iz prethodnog primera, podaci mogu biti procitani odatle:

SELECT title, total\_revenue

FROM film\_revenue\_summary

WHERE total\_revenue > 100;

Koriscenje materiazlized view-a omogucava brze vracanje rezultata ali zauzima dodatni prostor na disku.

Materialized view treba koristiti kada su upiti veoma spori zbog velike kolicine podataka ili kompleksnih agregacija. Takodje, treba se koristiti kada se dataset ne menja cesto i rezultati upita ne moraju biti konstantno azurirani u realnom vremenu vec mogu biti osvezeni periodicno.

## Query Rewriting

Query Rewriting je tehnika optimizacije originalnog upita u ekvivalentan ali efikasniji upit.

PostgreSQL optimizator moze automatski preformulisati upit interno kako bi redukovao duplikate, promenio redosled JOIN-ova, kombinovao WHERE uslove.

SELECT store\_id

FROM customer c

WHERE EXISTS (

SELECT 1

FROM store s

WHERE c.store\_id = s.store\_id

);

Optimizer moze samostalno izmeniti ovaj upit, ako proceni da je izmenjena verzija brza, u na primer:

SELECT DISTINCT c.store\_id

FROM customer c

JOIN store s ON c.store\_id = s.store\_id;

Dakle iako je programer napisao jedan upit, optimizator je procenio da moze optimizovati taj upit i ,,ispod haube” se izvrsava optimizovana verzija upita. Ova verzija upita moze biti brza jer spajanjem tabela preko store\_id moze da se iskoristi indeks nad kolonom store\_id sto eliminise potrebu da se proverava tabela red po red pomocu EXISTS kao u prvobitnom upitu i ubrzava izvrsenje.

# Particionisanje tabela

Particionisanje tabele predstavlja razbijanje jedne velike tabele na vise povezanih delova, koji se zovu particije. Svaka particija sadrzi podskup podataka iz glavne tabele, pri cemu se delovi podataka dodeljuju particijama na osnovu odredjenih kriterijuma. Ti kriterijumi mogu biti opseg vrednosti, neke kategorije ili hash funkcije. Na ovaj nacin upit umesto da obradjuje celu tabelu, moze ciljano da pristupi samo relevantim particijama, sto znacajno poboljsava brzinu izvrsenja i efikasnost.

Particionisanje tabela olaksava odrzavanje baze podataka. Moguce je obrisati podatke iz jedne particije bez uticaja na ostale particije.

Vrste particionisanja u PostgreSQL-u:

1. Range partitioning (particionisanje po opsegu vrednosti)
2. List partitioning (particionisanje po listi vrednosti)
3. Hash partitioning (particionisanje po hash vrednosti)

Particionisanje je odlicna opcija za poboljsavanje performansi kada je tabela vrlo velika (milioni redova), kada je potrebno da se stari podaci cesto brisu (brisanje celih particija odjednom),ili se upiti cesto filtriraju po particionom kljucu.

Za manje tabele particionisanje moze vise da uspori performanse nego sto ce pomoci zato sto se pretragom skace sa particije na particiju a brze bi bilo da se svi podaci nalaze na jednoj particiji.

Takodje, ne znaci nuzno da ako je tabela velika da treba da se izvrsi particionisanje. Ukoliko neki upit ne koristi particioni kljuc u WHERE klauzuli PostgreSQL mora da pretrazi sve particije te tabele, sto je neefikasnije od skeniranja jedne velike tabele.

## Range partitioning ( particionisanje po opsegu vrednosti)

Najcesce koriscena metoda particionisanja tabela. Podaci u tabeli se dele u razlicite particije u zavisnosti od opsega vrednosti neke kolone. Najcesce su to kolone sa datumima ili brojevima.

Na primer ukoliko u tabeli postoji kolona koja predstavlja godinu kreiranja, pozeljno je napraviti particije tako da se podaci za svaku godinu nalaze u posebnoj particiji. Kada se dodaje nov red u tabelu, baza ce sama znati u koju tacno particiju da ga smesti na osnovu vrednosti te kolone. Najveca prednost se vidi kada se izvrsavaju upiti za tacno odredjenu godinu u ovom slucaju, i vreme izvrsenja je znatno manje. Moguce je lako brisati podatke iz prethnodih godina ukoliko vise ne postoji potreba da se oni cuvaju. Omogucava ravnomerniju distribuciju podataka i lakse skaliranje baza koje sadrze velike kolicine podataka.

Mana je sto ako se veliki deo upita odnosi na jednu particiju, moze doci do preopterecenja te particije.

## List partitioning (particionisanje po listi vrednosti)

Redovi se rasporedjuju u particije na osnovu tacnih vrednosti jedne ili vise kolona. Svaka particija sadrzi redove cija vrednost particionog kljuca odgovara nekoj unapred definisanoj listi vrednosti.

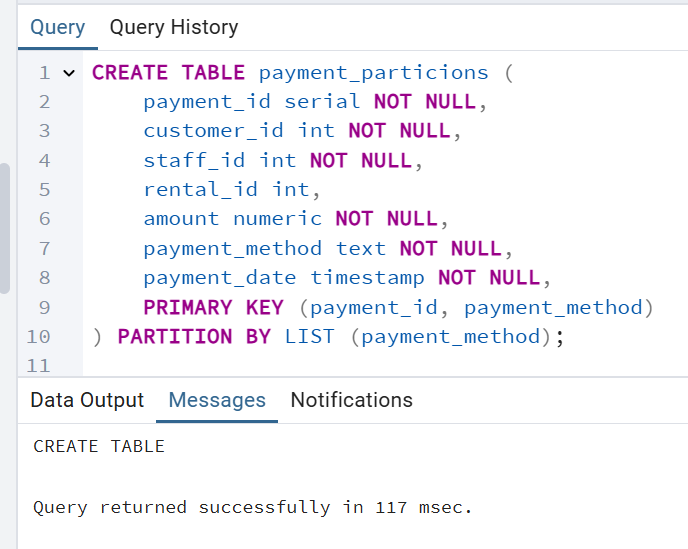
List partitioning deli podatke po konkretnim vrednostima.

Particioni kljuc je kolona ili skup kolona u tabeli koje se koriste za odredjivanje u koju ce particiju svaki red biti smesten prilikom particionisanja.

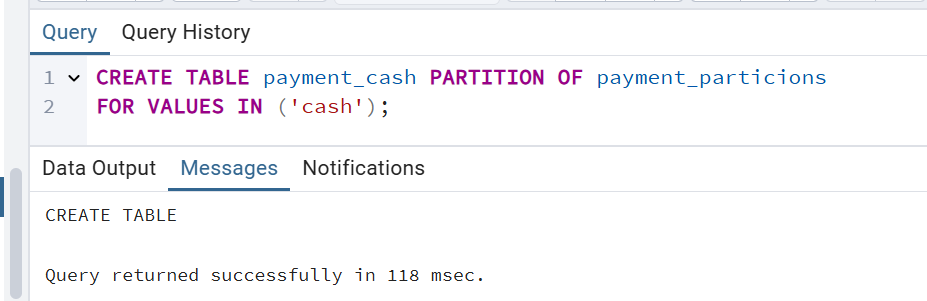
Za svaku particiju se definise tacna lista vrednosti koje ona sadrzi. Kada se ubacuje novi red u tabelu, PostgreSQL proverava vrednost particionog kljuca i smesta taj red u odgovarajucu particiju na osnovu definisane liste.

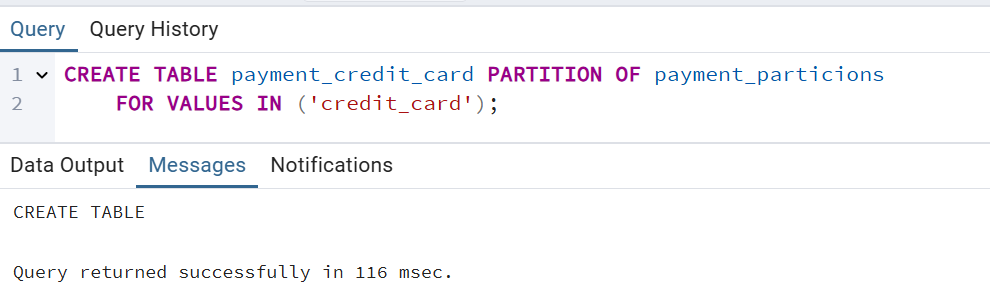
Podaci se grupisu po tacno definisanim vrednostima, sto moze biti korisno kada podaci nemaju odredjene opsege ali imaju skup nekih vrednosti za neku kolonu. Na primer u tabeli Payment imamo kolonu payment\_method i tu se ponavljaju jedne iste vrednosti kao sto su ‘cash’ , ‘credit\_card’ i ‘voucher’.

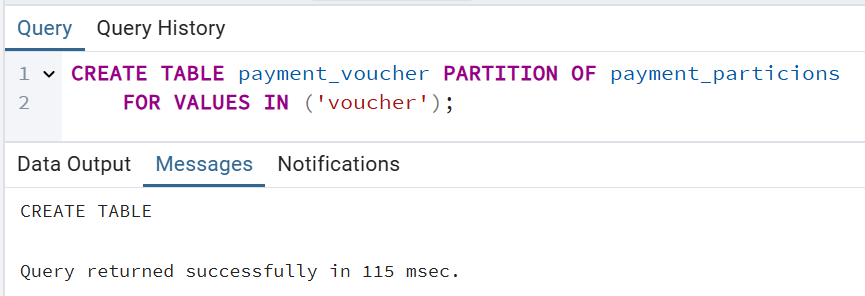
Najpre je neophodno da se particionise zeljena tabela. Posto u bazi podataka vec postoji tabela payment koja nije kreirana sa PARTITION BY, sto je neophodno da bi se tabela mogla particionisati, pravi se nova tabelu koja ce biti particionisana.



Kreiranje particija tabele payment\_particions:







Ukoliko se sada izvrsi upit

SELECT payment\_id, amount, payment\_date

FROM payment\_particions

WHERE payment\_method = 'cash';

Ovaj upit ce sada skenirati samo particiju payment\_cash, cime se znacajno smanjuje broj obradjenih redova.

Nedostatak je sto sve moguce vrednosti koje ce se koristiti u particionisanju moraju da budu unapred poznate, sto moze biti problem ako se vrednosti cesto menjaju ili dodaju. Takodje, ako se ubaci vrednost koja nije obuhvacena nijednom particijom baza ce baciti gresku, osim ako se ne definise posebna particija za ostale vrednosti.

Ako postoje vise jedinstvenih vrednosti moze postati komplikovano i tesko odrzivo imati veliku kolicinu particija.

## Hash Partitioning (Particionisanje po hash vrednosti)

Hash partitioning je metoda particionisanja tabela u kojoj se podaci rasporedjuju u particije na osnovu rezultata hash funkcije primenjene na particioni kljuc. Koristi hash algoritam da ravnomerno rasporedi redove u odredjeni broj particija.

Kada se ubacuje novi red, vrednost particionog kljuca prolazi kroz hash funkciju, i rezultat te funkcije odredjuje u koju particiju ce red biti smesten. Cilj hash particionisanja je da podaci budu sto ravnomernije rasporedjeni, cime se smanjuje rizik da ce neka particija postati previse opterecena. Omogucava lako horizontalno skaliranje baze jer se broj particija moze povecati kako raste kolicina podataka.

Hash particionisanje se koristi kada nemamo prirodan kriterijum za particionisanje a zelimo da particionisemo tabelu na ravnomerne particije. I tamo gde je potrebna skalabilnost i efikasna distribucija podataka. Nedostatak je sto nije lako odrediti u kojoj tacno particiji se nalazi odredjeni podatak, vec je potrebno koristiti hash funkciju.

# Zakljucak

Optimizacija je kljucan mehanizam za efikasan i stabilan rad baze podataja. Baze podataka u savremenim aplikacijama sadrze veliki broj podataka i bez optimizacije vecina baza bi bila prakticno neupotrebljiva. Izvrsavanje upita bi trajalo neuporedivo duze sto je u savremenim aplikacijama nedopustivo. Optimizacija moze da donese drasticne razlike u performansama, upiti se mogu ubrzati i nekoliko redova velicine. Takva pobosljanja osim sto stede na resursima, poboljsavaju korisnicko iskustvo. Optimizacija upita nije jednostavan ni jednokratan proces. Kako baza raste i razvija se, moguce je da dodje do promena u planu optimizacije izvrsavanja upita. Neko resenje koje je bilo najbolje u jednom trenutku, evolucijom baze moze da postane neoptimalno. Zbog toga je neophodno kontinuirano pratiti performanse baze i blagovremeno reagovati na promene. Samo takvim pristupom je moguce postici balans izmedju performansi upita, koriscenja resursa, tacnosti izvrsavanja upita i adektatnog korisnickog iskustva.