

Interna struktura i organizacija skladiša podataka u PostgreSQL bazi

Studijski program: Računarstvo i informatika

Modul: Softversko inženjerstvo

Predmet: Sistemi za upravljanje bazama podataka

|  |  |
| --- | --- |
| Student: | Profesor: |
|  |  |
| Jovan Vukadinović,  broj indeksa 1654 | Prof. dr Aleksandar Stanimirović |

Niš, april 2024. godina

Sadržaj

[Uvod 3](#_Toc164616721)

[Baze podataka 4](#_Toc164616722)

[Sistemski Katalog 5](#_Toc164616723)

[Šeme 5](#_Toc164616724)

[Tabele prostora (Tablespace) 6](#_Toc164616725)

[Relacije 8](#_Toc164616726)

[Fajlovi i grane 8](#_Toc164616727)

[Glavna grana 9](#_Toc164616728)

[Inicijalizaciona grana 10](#_Toc164616729)

[Mapa slobodnog prostora 11](#_Toc164616730)

[Mape vidljivosti 11](#_Toc164616731)

[TOAST 12](#_Toc164616732)

[Vacuum (usisavanje) 16](#_Toc164616733)

[Faze usisavanja 17](#_Toc164616734)

[Heap Scan 17](#_Toc164616735)

[Index Vacuuming 17](#_Toc164616736)

[Heap Vacuuming 18](#_Toc164616737)

[Heap Truncation 18](#_Toc164616738)

[Pages i Tuples 19](#_Toc164616739)

[Stranice 19](#_Toc164616740)

[Struktura strane 19](#_Toc164616741)

[Page Header 19](#_Toc164616742)

[Special Space (poseban prostor) 20](#_Toc164616743)

[Tuples 20](#_Toc164616744)

[Item Pointers 21](#_Toc164616745)

[Free Space 21](#_Toc164616746)

[Row Version Layout 21](#_Toc164616747)

[Operacije nad Tuple-ovima 23](#_Toc164616748)

[Zaključak 24](#_Toc164616749)

[Literatura 25](#_Toc164616750)

# Uvod

PostgreSQL je napredna objektno-relaciona baza podataka koja se ističe svojom specifičnom internom strukturom i organizacijom skladišta podataka, čime se razlikuje od drugih popularnih sistema za upravljanje bazama podataka kao što su MySQL ili Oracle.

U osnovi, skladište podataka u PostgreSQL bazi organizovano je na nekoliko nivoa. Osnovna jedinica skladištenja podataka je stranica, fiksne veličine od 8KB. Stranice se grupišu u segmente koji predstavljaju logičke delove baze podataka kao što su tabele, indeksi i transakcioni logovi. Unutar ovih segmenata, podaci su organizovani u tabele, koje se mogu podeliti na više delova radi optimizacije performansi, dok se indeksi koriste za brže pretraživanje i pristupanje podacima.

Što se tiče podrške za različite tipove podataka, PostgreSQL se ističe svojom fleksibilnošću. Osim standardnih SQL tipova podataka, PostgreSQL podržava i kompleksne tipove kao što su JSON, ARRAY i različiti geometrijski tipovi. Ova fleksibilnost omogućava razvoj i upravljanje različitim vrstama podataka unutar iste baze, što je čini veoma prilagodljivom za različite vrste aplikacija.

Ona takođe nudi efikasne mehanizme za održavanje i optimizaciju performansi baze podataka, kao što je proces "vacuuming". Proces "vacuuming" je esencijalni deo održavanja PostgreSQL baze podataka koji se koristi za uklanjanje mrtvih redova i optimizaciju skladištenja. Kada se vrši mnogo izmena u bazi podataka, kao što su brisanja i ažuriranja, može doći do akumulacije "mrtvih" redova koji zauzimaju nepotrebni prostor. Vacuum proces analizira bazu podataka i oslobađa ovaj nepotrebni prostor, čime poboljšava performanse upita i smanjuje veličinu baze podataka na disku.

Dodatno, PostgreSQL pruža podršku za različite proceduralne jezike kao što su PL/pgSQL, PL/Python i PL/Perl. Ovo omogućava razvoj kompleksnih funkcionalnosti direktno unutar baze podataka, što može značajno olakšati i ubrzati razvoj aplikacija.

# Baze podataka

PostgreSQL je program koji pripada klasi sistema za upravljanje bazama podataka. Kada ovaj program radi, nazivamo ga PostgreSQL serverom ili instancom. Podaci upravljani PostgreSQL-om čuvaju se u bazama podataka. Jedna PostgreSQL instanca može istovremeno služiti nekoliko baza podataka, a zajedno se nazivaju klaster baze podataka.

Da bi se mogao koristiti klaster, prvo se mora inicijalizovati (kreirati). Direktorijum koji sadrži sve datoteke povezane sa klasterom obično se naziva PGDATA, po imenu promenljive okruženja koja pokazuje na ovaj direktorijum.

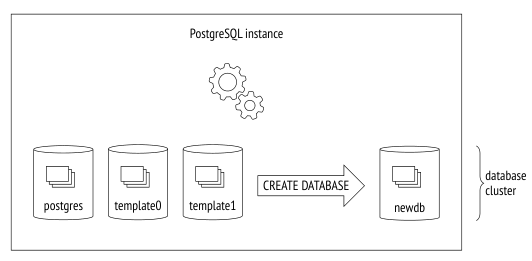
Instalacije iz prethodno izgrađenih paketa mogu dodati svoje "slojeve apstrakcije" preko redovnog PostgreSQL mehanizma eksplicitnim podešavanjem svih parametara potrebnih za alate. U ovom slučaju, serverska baza podataka radi kao servis operativnog sistema, i možda nikada nećete direktno naići na promenljivu PGDATA. Ali, sam termin je dobro uspostavljen, pa će biti korišćen nadalje.

Nakon inicijalizacije klastera, PGDATA sadrži tri identične baze podataka:

**template0** se koristi za slučajeve kao što su obnova podataka iz logičke rezerve ili kreiranje baze podataka sa drugačijim kodiranjem, i nikada se ne sme menjati.

**template1** služi kao šablon za sve druge baze podataka koje korisnik može kreirati u klasteru.

**Postgres** je standardna baza podataka koja se može koristiti po želji.



Slika 1. PostgreSQL instanca

# Sistemski Katalog

Metapodaci svih objekata klastera (kao što su tabele, indeksi, tipovi podataka ili funkcije) čuvaju se u tabelama koje pripadaju sistemskom katalogu. Svaka baza podataka ima svoj skup tabela (i pogleda) koji opisuju objekte ove baze podataka. Neke sistemsko kataloške tabele su zajedničke za ceo klaster, i one ne pripadaju ni jednoj određenoj bazi podataka (tehnički, koristi se lažna baza podataka sa nultim ID-jem), ali im se može pristupiti iz svih njih.

Sistemski katalog može se pregledati korišćenjem redovnih SQL upita, dok se sve modifikacije u njemu vrše DDL naredbama. Klijent psql takođe nudi čitav niz naredbi koje prikazuju sadržaj sistemskog kataloga.

Imena svih tabela sistemskog kataloga počinju sa pg\_, kao što je pg\_database. Imena kolona počinju sa tročlanim prefiksom koji obično odgovara imenu tabele, kao što je datname.

U svim tabelama sistemskog kataloga, kolona deklarisana kao primarni ključ naziva se oid (objektni identifikator), njen tip, koji se takođe naziva oid, je 32-bitni ceo broj.

Implementacija oid objektnih identifikatora je gotovo ista kao i kod sekvenci, ali se pojavila u PostgreSQL-u mnogo ranije. Ono što je čini posebnom je to što generisani jedinstveni ID-jevi izdati od zajedničkog brojača služe u različitim tabelama sistemskog kataloga. Kada dodeljeni ID pređe maksimalnu vrednost, brojač se resetuje. Da bi se osiguralo da su sve vrednosti u određenoj tabeli jedinstvene, sledeći izdati oid se proverava preko jedinstvenog indeksa, a ako je već korišćen u ovoj tabeli, brojač se povećava, i provera se ponavlja.

# Šeme

Šeme su imenski prostori koji čuvaju sve objekte jedne baze podataka. Osim korisničkih šema, PostgreSQL nudi nekoliko unapred definisanih:

* **public** je podrazumevana šema za korisničke objekte osim ako nisu specificirana druga podešavanja.
* **pg\_catalog** se koristi za tabele sistemskog kataloga.
* **information\_schema** pruža alternativni prikaz sistemskog kataloga definisan standardom SQL.
* **pg\_toast** se koristi za objekte povezane sa TOAST.
* **pg\_temp** obuhvata privremene tabele. Iako različiti korisnici kreiraju privremene tabele u različitim šemama nazvanim pg\_temp\_N, svi se pozivaju na svoje objekte koristeći alias pg\_temp.

Svaka šema je ograničena na određenu bazu podataka, i svi objekti baze podataka pripadaju nekoj od ovih šema.

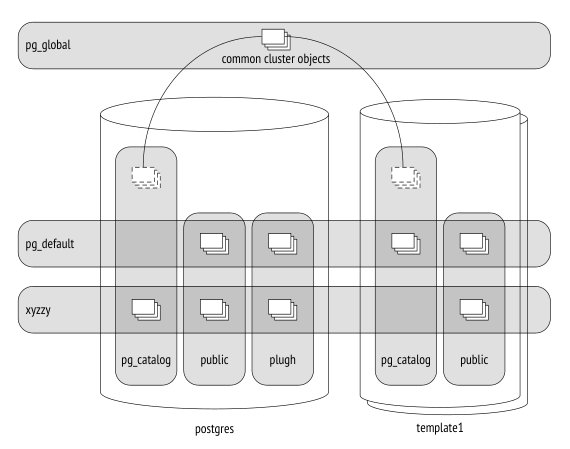
Ako šema nije eksplicitno specificirana prilikom pristupa objektu, PostgreSQL bira prvu odgovarajuću šemu iz putanje pretrage. Putanja pretrage se zasniva na vrednosti parametra search\_path, koji se implicitno proširuje sa pg\_catalog i (ako je potrebno) pg\_temp šemama. To znači da različite šeme mogu sadržati objekte sa istim imenima.

# Tabele prostora (Tablespace)

Za razliku od baza podataka i šema koje određuju logičku distribuciju objekata, tabelarni prostori definišu fizičku raspodelu podataka. Tabelarni prostor je praktično direktorijum u fajl sistemu. Mogu distribuirati svoje podatke između tabelarnih prostora tako da se arhivski podaci čuvaju na sporim diskovima, dok podaci koji se aktivno ažuriraju idu na brze diskove.

Jedan te isti tabelarni prostor može koristiti više baza podataka, i svaka baza podataka može čuvati podatke u nekoliko tabelarnih prostora. To znači da logička struktura i fizička raspodela podataka ne zavise jedna od druge.

Svaka baza podataka ima takozvani podrazumevani tabelarni prostor. Svi objekti baze podataka kreiraju se u ovom tabelarnom prostoru osim ako nije specificirana druga lokacija. Objekti sistemskog kataloga povezani sa ovom bazom podataka takođe se čuvaju tamo.



Slika 2. Tabela prostora

Prilikom inicijalizacije klastera, kreiraju se dva tabelarna prostora:

* **pg\_default** se nalazi u direktorijumu PGDATA/base, koristi se kao podrazumevani tabelarni prostor osim ako nije eksplicitno izabran neki drugi tabelarni prostor za tu svrhu.
* **pg\_global** se nalazi u direktorijumu PGDATA/global, a čuva objekte sistemskog kataloga koji su zajednički za ceo klaster.

Prilikom kreiranja prilagođenog tabelarnog prostora, može se specificirati bilo koji direktorijum, a PostgreSQL će kreirati simboličku vezu ka ovoj lokaciji u direktorijumu PGDATA/pg\_tblspc. Zapravo, sve putanje koje koristi PostgreSQL su relativne u odnosu na direktorijum PGDATA, što omogućava da se premesti na drugu lokaciju (uz pretpostavku da je server unapred zauzet, naravno).

Ilustracija na prethodnoj slici kombinuje baze podataka, šeme i tabelarne prostore. Ovde baza podataka postgres koristi tabelarni prostor xyzzy kao podrazumevani, dok baza podataka template1 koristi pg\_default.

Različiti objekti baze podataka prikazani su na preseku tabelarnih prostora i šema.

# Relacije

Uprkos svim njihovim razlikama, tabele i indeksi kao najvažniji objekti baza podataka, imaju nešto zajedničko: sastoje se od redova. Ovaj pojam je prilično očigledan kada razmišljamo o tabelama, ali je podjednako tačan i za čvorove B-stabla, koji sadrže indeksirane vrednosti i reference na druge čvorove ili redove tabela.

Neki drugi objekti takođe imaju istu strukturu, na primer, “sequences” (praktično tabele sa jednim redom) i “materialized views” (koji se mogu smatrati tabelama koje “čuvaju” odgovarajuće upite). Pored toga, postoje i redovni pogledi, koji ne čuvaju nikakve podatke, ali su inače vrlo slični tabelama.

U PostgreSQL-u, svi ovi objekti se nazivaju generičkim terminom relacija. To može dovesti do zabuna sa pravim relacijama izmedju tabela, zbog čega je kasnije počeo da se koristi naziv uređena relacija.

Sistemska kataloška tabela za relacije prvobitno se zvala pg\_relation, ali prateći trend objektno-orijentisanog programiranja, ubrzo je preimenovana u pg\_class, što je sada svima poznato. Ipak, njene kolone još uvek imaju prefiks REL.

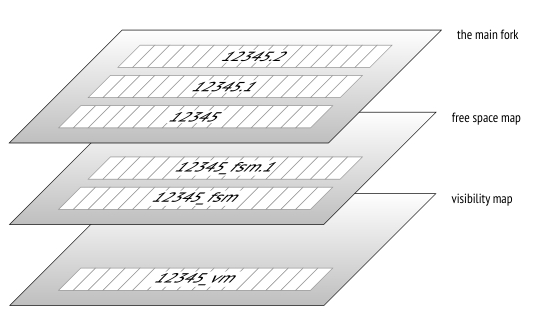
# Fajlovi i grane

Sva informacija povezana sa relacijom čuva se u nekoliko različitih grana, pri čemu svaka grana sadrži podatke određenog tipa.

Na početku, grana je predstavljena jednim fajlom. Ime fajla sastoji se od numeričkog ID-ja (oid), koji može biti proširen sufiksom koji odgovara tipu grane.

Fajl raste tokom vremena, i kada njegova veličina dostigne 1GB, kreira se još jedan fajl ove grane (takvi fajlovi se ponekad nazivaju segmenti). Redni broj segmenta dodaje se na kraj njegovog imena fajla.

Ograničenje veličine fajla od 1GB istorijski je uspostavljeno da podržava različite fajl sisteme koji nisu mogli da rukuju sa velikim fajlovima. Ovo ograničenje se može promeniti prilikom izgradnje PostgreSQL-a (./configure --with-segsize).



Slika 3. Raspored fajlova u skladištenju

Tako, jedna relacija na disku predstavljena je sa nekoliko fajlova. Čak i mala tabela bez indeksa imaće najmanje tri fajla, prema broju obaveznih grana.

Svaki direktorijum tabelarnog prostora (osim pg\_global) sadrži posebne poddirektorijume za određene baze podataka. Svi fajlovi objekata koji pripadaju istom tabelarnom prostoru i bazi podataka smešteni su u istom poddirektorijumu. Ovo se mora uzeti u obzir, jer previše fajlova u jednom direktorijumu možda neće biti dobro rukovano od strane fajl sistema.

Postoji nekoliko standardnih tipova grana i to su glavna grana, inicijalizaciona grana, mapa slobodnog prostora i mape vidljivosti.

## Glavna grana

Glavna grana predstavlja stvarne podatke, redove tabela ili indeksne redove. Ova grana je dostupna za sve relacije (osim za poglede, koji ne sadrže podatke).

Fajlovi glavne grane imenuju se prema svojim numeričkim ID-jevima, koji su sačuvani kao vrednosti relfilenode u tabeli pg\_class.

Pogledajmo putanju do fajla koji pripada tabeli kreiranoj u tabelarnom prostoru pg\_default:

=> **CREATE UNLOGGED TABLE** t(

a integer,

b numeric,

c text,

d json

);

=> **INSERT INTO** t **VALUES** (1, 2.0, 'foo', '{}');

=> **SELECT** pg\_relation\_filepath('t');

pg\_relation\_filepath

−−−−−−−−−−−−−−−−−−−−−−

base/16384/16385

(1 row)

Osnovni direktorijum odgovara tabelarnom prostoru pg\_default, sledeći poddirektorijum koristi se za bazu podataka, i upravo tu nalazimo fajl koji tražimo:

=> **SELECT** oid **FROM** pg\_database **WHERE** datname = 'internals';

oid

−−−−−−−

16384

(1 row)

=> **SELECT** relfilenode **FROM** pg\_class **WHERE** relname = 't';

relfilenode

−−−−−−−−−−−−−

16385

(1 row)

Evo odgovarajućeg fajla u fajl sistemu:

=> **SELECT** size

**FROM** pg\_stat\_file('/usr/local/pgsql/data/base/16384/16385');

size

−−−−−−

8192

(1 row)

## Inicijalizaciona grana

Inicijalizaciona grana je dostupna samo za neregistrovane tabele (kreirane sa UNLOGGED klauzulom) i njihove indekse. Takvi objekti su isti kao i redovni, osim što se sve akcije koje se na njima izvršavaju ne upisuju u zapisnik unapred napisanog reda (write-ahead log).

To znatno ubrzava ove operacije, ali je onemogućeno obnavljanje konzistentnih podataka u slučaju kvara. Zbog toga PostgreSQL jednostavno briše sve grane ovakvih objekata tokom oporavka i prebriše glavnu granu inicijalizacionom granom, čime se stvara lažni fajl.

Tabela t je kreirana kao neregistrovana, pa je inicijalizaciona grana prisutna. Ima isto ime kao glavna grana, ali sa sufiksom \_init:

=> **SELECT** size

**FROM** pg\_stat\_file('/usr/local/pgsql/data/base/16384/16385\_init');

size

−−−−−−

0

(1 row)

## Mapa slobodnog prostora

Mapa slobodnog prostora prati dostupan prostor unutar stranica. Njena zapremina se stalno menja, raste nakon usisavanja (vacuuming) i smanjuje kada se pojave nove verzije redova. Mapa slobodnog prostora koristi se za brzo pronalaženje stranice koja može da primi nove podatke koji se ubacuju.

Svi fajlovi koji se odnose na mapu slobodnog prostora imaju sufiks \_fsm. Inicijalno, takvi fajlovi nisu kreirani, a pojavljuju se samo kada je to potrebno. Najlakši način da se dobiju je usisavanjem (vacuum) tabele:

=> **VACUUM** t;

=> **SELECT** size

FROM pg\_stat\_file('/usr/local/pgsql/data/base/16384/16385\_fsm');

size

−−−−−−−

24576

(1 row)

Da bi se ubrzala pretraga, mapa slobodnog prostora je organizovana kao drvo, za to su potrebne najmanje tri stranice (otuda njegova veličina fajla za gotovo praznu tabelu).

Mapa slobodnog prostora je obezbeđena kako za tabele tako i za indekse. Međutim, pošto se indeksni red ne može dodati u proizvoljnu stranicu (na primer, B-stabla definišu mesto ubacivanja prema redosledu sortiranja), PostgreSQL prati samo one stranice koje su potpuno ispražnjene i mogu se ponovo koristiti u strukturi indeksa.

## Mape vidljivosti

Mapa vidljivosti može brzo pokazati da li je stranica potrebno usisati ili zamrznuti. U tu svrhu, pruža dva bita za svaku stranicu tabele.

Prvi bit je podešen za stranice koje sadrže samo ažurirane verzije redova. Usisavanje preskače takve stranice jer nema šta da se očisti. Osim toga, kada transakcija pokušava da pročita red sa takve stranice, nema svrhe proveravati njegovu vidljivost, tako da se može koristiti samo indeksno pretraživanje.

Drugi bit je podešen za stranice koje sadrže samo zamrznute verzije redova.

Fajlovi mape vidljivosti imaju sufiks \_vm. Obično su najmanji:

=> SELECT size

FROM pg\_stat\_file('/usr/local/pgsql/data/base/16384/16385\_vm');

size

−−−−−−

8192

(1 row)

Mapa vidljivosti je obezbeđena za tabele, ali ne i za indekse.

# TOAST

Svaki red mora da se uklopi u jednu stranicu, nema načina da se red nastavi na sledećoj stranici. Da bi čuvali dugački redovi, PostgreSQL koristi poseban mehanizam nazvan TOAST (The Oversized Attributes Storage Technique - Tehnika za čuvanje prevelikih atributa).

TOAST podrazumeva nekoliko strategija. Dugačke vrednosti atributa mogu se premestiti u posebnu servisnu tabelu, tako što će se iseći na manje "toaste". Druga opcija je kompresija dugih vrednosti tako da red odgovara stranici. Ili se može uraditi i jedno i drugo: prvo kompresija vrednost, a zatim odsecanje kompresovane vrednost i njeno premeštanje.

Ako glavna tabela sadrži potencijalno dugačke atribute, odmah se kreira posebna TOAST tabela za nju, jedna za sve atribute. Na primer, ako tabela ima kolonu numeričkog ili tekstualnog tipa, TOAST tabela će biti kreirana čak i ako ova kolona nikada neće čuvati dugačke vrednosti.

Za indekse, mehanizam TOAST-a može ponuditi samo kompresiju, premestiti dugačke atribute u posebnu tabelu nije podržano. To ograničava veličinu ključeva koji se mogu indeksirati (stvarna implementacija zavisi od određene klase operatora).

Podrazumevano, strategija TOAST-a se bira na osnovu tipa podataka kolone. Najjednostavniji način da se pregledaju korišćene strategije je da pokrenete \d+ komandu u psql-u, u ovom primeru će se upitati sistemski katalog kako bi se dobio čistiji izlaz:

=> **SELECT** attname, atttypid::regtype,

**CASE** attstorage

**WHEN** 'p' **THEN** 'plain'

**WHEN** 'e' **THEN** 'external'

**WHEN** 'm' **THEN** 'main'

**WHEN** 'x' **THEN** 'extended'

**END AS** storage

**FROM** pg\_attribute

**WHERE** attrelid = 't'::regclass **AND** attnum > 0;

attname | atttypid | storage

−−−−−−−−-+−−−−−−−−−−+−−−−−−−−−−

A | integer | plain

B | numeric | main

C | text | extended

D | json | extended

(4 rows)

PostgreSQL podržava sledeće strategije:

* **plain** znači da se TOAST ne koristi (ova strategija se primenjuje na tipove podataka za koje se zna da su "kratki", kao što je tip integer).
* **extended** omogućava i kompresiju atributa i njihovo čuvanje u posebnoj TOAST tabeli.
* **external** podrazumeva da se dugački atributi čuvaju u TOAST tabeli u nekompresovanom stanju.
* **main** zahteva da se dugački atributi prvo kompresuju, i biće premešteni u TOAST tabelu samo ako kompresija nije pomogla.

U opštim terminima, algoritam izgleda ovako. PostgreSQL cilja na to da ima najmanje četiri reda u stranici. Dakle, ako veličina reda prelazi jednu četvrtinu stranice, isključujući zaglavlje (za stranicu standardne veličine to je oko 2000 bajtova), moramo primeniti TOAST mehanizam na neke od vrednosti. Prateći opisani radni tok, zaustavljamo se čim dužina reda više ne prelazi prag:

1. Prvo prolazimo kroz atribute sa eksternim i proširenim strategijama, počevši od najdužih. Prošireni atributi se kompresuju, i ako rezultujuća vrednost (sama po sebi, bez uzimanja drugih atributa u obzir) prelazi jednu četvrtinu stranice, odmah se premješta u TOAST tabelu. Eksterni atributi se obrađuju na isti način, osim što se preskače faza kompresije.
2. Ako red i dalje ne stane u stranicu nakon prvog prolaza, premeštaju se preostali atributi koji koriste eksternu ili proširenu strategiju u TOAST tabelu, jedan po jedan.
3. Ako ni to nije pomoglo, pokušava se kompresuja atributa koji koriste glavnu strategiju, čuvajući ih u stranici tabele.
4. Ako red i dalje nije dovoljno kratak, glavni atributi se premještaju u TOAST tabelu.

Vrednost praga je 2000 bajtova, ali je moguće redefinisati je na nivou tabele koristeći parametar skladištenja toast\_tuple\_target.

Ponekad može biti korisno promeniti podrazumevanu strategiju za neke od kolona. Ako se unapred zna da podaci u određenoj koloni ne mogu biti kompresovani (na primer, kolona čuva JPEG slike), može se postaviti eksterna strategiju za ovu kolonu, a to omogućava da se izbegne uzaludno pokušaje kompresije podataka. Strategija se može promeniti na sledeći način:

=> **ALTER TABLE** t **ALTER COLUMN** d **SET STORAGE** external;

Ako ponovimo upit, dobićemo sledeće rezultate:

attname | atttypid | storage

−−−−−−−−+−−−−−−−−−−+−−−−−−−−−−

A | integer | plain

B | numeric | main

C | text | extended

D | json | external

(4 rows)

TOAST tabele se nalaze u posebnoj šemi nazvanoj pg\_toast, i nije uključena u putanju pretrage, tako da su TOAST tabele obično skrivene. Za privremene tabele, koriste se šeme pg\_toast\_temp\_N, po analogiji sa pg\_temp\_N. Pogledajmo unutrašnju mehaniku procesa. Pretpostavimo da tabela t sadrži tri potencijalno dugačka atributa, to znači da mora postojati odgovarajuća TOAST tabela. Primer:

=> **SELECT** relnamespace::regnamespace, relname

**FROM** pg\_class

**WHERE** oid = (

**SELECT** reltoastrelid

**FROM** pg\_class **WHERE** relname = 't'

);

relnamespace | relname

−−−−−−−−−−−−−−+−−−−−−−−−−−−−−−−

pg\_toast | pg\_toast\_16385

(1 row)

=> \d+ pg\_toast.pg\_toast\_16385

TOAST table "pg\_toast.pg\_toast\_16385"

Column | Type | Storage

−−−−−−−−−−−−+−−−−−−−−-−+−−−−−−−−−

chunk\_id | oid | plain

chunk\_seq | integer | plain

chunk\_data | bytea | plain

Owning table: "public.t"

Indexes:

"pg\_toast\_16385\_index" PRIMARY KEY, btree (chunk\_id, chunk\_seq)

Access method: heap

Logično je da rezultujući delovi tostiranog reda koriste plain strategiju: nema TOAST na drugom nivou.

Pored same TOAST tabele, PostgreSQL kreira odgovarajući indeks u istoj šemi. Ovaj indeks se uvek koristi za pristup TOAST delovima. Naziv indeksa je prikazan u izlazu, ali se možete pogledati i izvršavanjem sledećeg upita:

=> **SELECT** indexrelid::regclass **FROM** pg\_index

**WHERE** indrelid = (

**SELECT** oid

**FROM** pg\_class **WHERE** relname = 'pg\_toast\_16385'

);

indexrelid

−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−

pg\_toast.pg\_toast\_16385\_index

(1 row)

=> \d pg\_toast.pg\_toast\_16385\_index

Unlogged index "pg\_toast.pg\_toast\_16385\_index"

Column | Type | Key? | Definition

−−−−−−−−-−+−−−−−−−−−−+−−−−−−+−−−−−−−−−−−−

chunk\_id | oid | yes | chunk\_id

chunk\_seq | integer | yes | chunk\_seq

primary key, btree, for table "pg\_toast.pg\_toast\_16385"

Dakle, TOAST tabela povećava minimalni broj fork fajlova koje tabela koristi do osam: tri za glavnu tabelu, tri za TOAST tabelu i dva za TOAST indeks.

Kolona c koristi proširenu strategiju, pa će njene vrednosti biti kompresovane:

=> **UPDATE** t **SET** c = repeat('A',5000);

=> **SELECT** \* **FROM** pg\_toast.pg\_toast\_16385;

chunk\_id | chunk\_seq | chunk\_data

−−−−−−−−−+-−−−−−−−−−−+−−−−−−−−−−−−

(0 rows)

TOAST tabela je prazna, ponavljajući simboli su kompresovani LZ algoritmom, tako da vrednost odgovara stranici tabele.

A sada da konstruišemo ovu vrednost nasumičnih simbola:

=> **UPDATE** t **SET** c = (

**SELECT** string\_agg( chr(trunc(65+random()\*26)::integer), '')

**FROM** generate\_series(1,5000)

)

**RETURNING** left(c,10) || '...' || right(c,10);

?column?

−−−−−−−−−−−−−−−−−−−−−−−−−

YEYNNDTSZR...JPKYUGMLDX

(1 row)

UPDATE 1

Ova sekvenca ne može da se kompresuje, pa se stavlja u TOAST tabelu:

=> **SELECT** chunk\_id,

chunk\_seq,

length(chunk\_data),

left(encode(chunk\_data,'escape')::text, 10) || '...' ||

right(encode(chunk\_data,'escape')::text, 10)

**FROM** pg\_toast.pg\_toast\_16385;

chunk\_id |chunk\_seq | length | ?column?

−−−−−−−−−−-−−+−−−−−-−−−−--+−−−−−−−−−−−−+−−−−−−−

16390 | 0 | 1996 | YEYNNDTSZR...TXLNDZOXMY

16390 | 1 | 1996 | EWEACUJGZD...GDBWMUWTJY

16390 | 2 | 1008 | GSGDYSWTKF...JPKYUGMLDX

(3 rows)

Može se videti da su znakovi isečeni u delove. Veličina dela je odabrana tako da stranica TOAST tabele može da smesti četiri reda. Ova vrednost se malo menja iz verzije u verziju u zavisnosti od veličine zaglavlja stranice.

Kada se pristupa dugom atributu, PostgreSQL automatski obnavlja originalnu vrednost i vraća je klijentu, a sve se to dešava bez ikakvog problema za aplikaciju. Ako dugi atributi ne učestvuju u upitu, TOAST tabela uopšte neće biti čitana. To je jedan od razloga zašto treba izbegavati korišćenje zvezdice u proizvodnim rešenjima.

Ako klijent traži jedan od prvih delova duge vrednosti, PostgreSQL će čitati samo potrebne delove, čak i ako je vrednost kompresovana.

Ipak, kompresija podataka i sečenje zahtevaju puno resursa, isto važi i za obnavljanje originalnih vrednosti. Zato nije dobra ideja čuvati obimne podatke u PostgreSQL-u, posebno ako se ti podaci aktivno koriste i ne zahtevaju transakcionu logiku (kao što su skenirani računovodstveni dokumenti). Potencijalno bolja alternativa je da se takvi podaci čuvaju u fajl sistemu, čuvajući u bazi podataka samo imena odgovarajućih fajlova. Ali tada sistem baze podataka ne može garantovati konzistentnost podataka.

# Vacuum (usisavanje)

Rutinsko usisavanje, koje je glavna procedura usisavanja, izvodi se pomoću VACUUM komande. Ona procesira celu tabelu i eliminiše kako zastarele heap tuple-ove, tako i sve odgovarajuće unose u indeksa.

Usisavanje se izvodi paralelno sa ostalim procesima u bazi podataka. Dok se usisava, tabele i indeksi mogu se koristiti na uobičajeni način, kako za čitanje tako i za pisanje (ali istovremeno izvršavanje komandi kao što su CREATE INDEX, ALTER TABLE i neke druge nije dozvoljeno).

Da bi izbegli skeniranje dodatnih stranica, PostgreSQL koristi mapu vidljivosti. Stranice praćene u ovoj mapi preskaču se, jer je sigurno da sadrže samo trenutne tuple-ove, pa će stranica biti usisana samo ako se ne pojavi u ovoj mapi. Ako su svi tuple-ovi koje ostaju na stranici nakon usisavanja izvan horizonta baze podataka, mapa vidljivosti se ažurira da uključi ovu stranicu.

Mapa slobodnog prostora takođe se ažurira da odražava prostor koji je oslobođen.

## Faze usisavanja

Mehanizam usisavanja deluje prilično jednostavno, ali to baš i nije slučaj. Na kraju krajeva, i tabele i indeksi moraju biti obrađeni istovremeno, bez blokiranja drugih procesa. Da bi omogućili takvu operaciju, usisavanje svake tabele sprovodi se u nekoliko faza.

Sve počinje skeniranjem tabele u potrazi za mrtvim tuple-ovima. Ako se pronađu, prvo se uklanjaju iz indeksa, a zatim iz same tabele. Ako treba usisati previše mrtvih tuple-ova odjednom, ovaj proces se ponavlja. Na kraju, može se izvesti i skraćivanje heap-a.

### Heap Scan

U prvoj fazi izvodi se skeniranje heap-a. Proces skeniranja uzima u obzir mapu vidljivosti, sve stranice praćene u ovoj mapi preskaču se jer je sigurno da ne sadrže zastarele tuple-ove. Ako je tuple izvan horizonta i više nije potrebna, njegov ID se dodaje u poseban niz tid. Takvi tuple-ovi još uvek ne mogu biti uklonjeni, jer ih indeksi možda još uvek referenciraju.

Niz tid-a se nalazi u lokalnoj memoriji VACUUM procesa, veličina dodele memorije definisana je parametrom maintenance\_work\_mem. Čitav blok memorije se dodeljuje odjednom, a ne po potrebi. Međutim, dodeljena memorija nikada ne prelazi zapreminu potrebnu u najgorem scenariju, pa ako je tabela mala, usisavanje može koristiti manje memorije nego što je navedeno u ovom parametru.

### Index Vacuuming

Prva faza može imati dva ishoda: ili se cela tabela skenira u potpunosti, ili se memorija dodeljena za niz tid popuni pre nego što se ovaj proces završi. U svakom slučaju, počinje se sa usisavanjem indeksa. U ovoj fazi, svaki od indeksa kreiranih na tabeli potpuno se skenira kako bi se pronašli svi unosi koji se odnose na tuple-ove registrovane u nizu tid. Ovi unosi se uklanjaju sa stranica indeksa.

Indeks vam može pomoći da brzo pristupite tuple-u heap-a putem njenog indeksnog ključa, ali ne postoji način da brzo pronađete unos u indeksu prema odgovarajućem ID-u tuple-a.

Ako postoji nekoliko indeksa većih od vrednosti min\_parallel\_index\_scan\_size, oni mogu biti usisani od strane pozadinskih radnika koji rade paralelno. Osim ako nivo paralelizma nije eksplicitno definisan klauzulom parallel N, VACUUM pokreće jednog radnika po odgovarajućem indeksu (unutar opštih ograničenja postavljenih na broj pozadinskih radnika). Jedan indeks ne može biti obrađen od strane više radnika.

Tokom faze usisavanja indeksa, PostgreSQL ažurira mapu slobodnog prostora i izračunava statistike o usisavanju. Međutim, ova faza se preskače ako su samo uneti redovi (i nisu ni obrisani ni ažurirani), jer tabela u ovom slučaju ne sadrži mrtve tuple-ove. Tada će skeniranje indeksa biti izvršeno samo jednom na samom kraju, kao deo posebne faze čišćenja indeksa.

Faza usisavanja indeksa ne ostavlja reference na zastarele tuple-ove heap-a u indeksima, ali sami tuple-ovi i dalje postoje u tabeli. To je potpuno normalno, skeniranja indeksa ne mogu pronaći nikakve mrtve tuple-ove, dok se sekvencijalna skeniranja tabele oslanjaju na pravila vidljivosti da bi ih filtrirala.

### Heap Vacuuming

Zatim počinje faza usisavanja heap-a. Tabela se ponovo skenira kako bi se uklonili tuple-ovi registrovani u nizu tid i oslobodili odgovarajući pokazivači. Sada kada su sve povezane reference u indeksima uklonjene, to se može bezbedno uraditi.

Prostor koji je VACUUM oslobodio odražava se na mapi slobodnog prostora, dok su stranice koje sada sadrže samo trenutne tuple-ove vidljive u svim snapshot-ovima označenim na mapi vidljivosti. Ako tabela nije u potpunosti pročitana tokom faze skeniranja heap-a, niz tid se briše i skeniranje heap-a se nastavlja od mesta gde je prethodno stalo.

### Heap Truncation

Stranice heap-a koje su usisane sadrže neki slobodan prostor, povremeno možete imati sreće i očistiti celu stranicu. Ako dobijete nekoliko praznih stranica na kraju fajla, usisavanje može "odseći" ovaj rep i vratiti oslobođeni prostor operativnom sistemu. Ovo se dešava tokom “Heap Truncation”, koje je poslednja faza usisavanja.

Odsecanje heap-a zahteva kratku ekskluzivnu blokadu nad tabelom. Kako bi se izbeglo zadržavanje drugih procesa predugo, pokušaji da se dobije blokada ne traju duže od pet sekundi.

Pošto tabela mora da bude zaključana, odsecanje se vrši samo ako prazni rep zauzima barem tabele ili je dostigao dužinu od 1000 stranica. Ovi pragovi su unapred definisani i ne mogu se konfigurisati.

Ako, uprkos svim ovim merama opreza, blokade tabela i dalje prouzrokuju bilo kakve probleme, odsecanje se može potpuno onemogućiti koristeći parametre skladištenja vacuum\_truncate i toast.vacuum\_truncate.

# Pages i Tuples

## Stranice

Da bi se olakšao I/O, sve datoteke su logički podeljene na stranice (ili blokove), koje predstavljaju minimalnu količinu podataka koja može da se čita ili piše. Stoga su mnogi interni algoritmi PostgreSQL-a podešeni za obradu stranica.

Veličina stranice je obično 8 kB. Može se neznatno konfigurisati (do 32kB), ali samo prilikom izgradnje (./configure --with-blocksize), i retko ko to obično radi. Jednom izgrađen i pokrenut, instanca može raditi samo sa stranicama iste veličine, nemoguće je kreirati tabelarne prostore koji podržavaju različite veličine stranica.

Bez obzira na granu kojoj pripadaju, sve datoteke server obrađuje na otprilike isti način. Stranice se prvo premeštaju u keš memoriju bafera (gde ih mogu čitati i ažurirati procesi), a zatim se po potrebi ponovo upisuju na disk.

## Struktura strane

Svaka stranica ima određeni unutrašnji raspored koji obično uključuje sledeće delove:

* zaglavlje stranice
* niz pokazivača na stavke (item pointers)
* slobodan prostor
* stavke (verzije redova)
* poseban prostor

### Page Header

Zaglavlje stranice se nalazi na najnižim adresama i ima fiksnu veličinu. Čuva različite informacije o stranici, kao što su njen checksum i veličine svih ostalih delova stranice.

Ove veličine se lako mogu prikazati koristeći pageinspect ekstenziju. Pogledamo prvu stranicu tabele (numeracija stranica je bazirana na nuli):

=> **CREATE EXTENSION** pageinspect;

=> **SELECT** lower, upper, special, pagesize

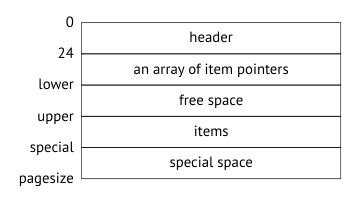
**FROM** page\_header(get\_raw\_page('accounts',0));

lower | upper | special | pagesize

−−−−−−−+−−−−−−−+−−−−−−−−−+−−−−−−−−−−

152 |6904 |8192 |8192

(1 row)



### Special Space (poseban prostor)

Poseban prostor se nalazi na suprotnom delu stranice, zauzimajući njene najviše adrese. Koristi se u nekim indeksima za čuvanje pomoćnih informacija, a u drugim indeksima i tabelarnim stranicama ovaj prostor je nulte veličine.

Opšti raspored stranica indeksa je prilično raznolik, njihov sadržaj uglavnom zavisi od određenog tipa indeksa. Čak i isti indeks može imati različite vrste stranica: na primer, B-stabla imaju metapodatak stranice specifične strukture (stranica nula) i redovne stranice koje su veoma slične tabelarnim stranicama.

### Tuples

Redovi sadrže stvarne podatke koji se čuvaju u bazi podataka, zajedno sa nekim dodatnim informacijama. Nalaze se neposredno pre specijalnog prostora.

U slučaju tabela, moramo se baviti verzijama redova umesto redovima, jer kontrola konkurentnosti s više verzija podrazumeva postojanje nekoliko verzija istog reda. Indeksi ne koriste ovaj mehanizam MVCC. Umesto toga, oni moraju da se referenciraju na sve dostupne verzije redova, oslanjajući se na pravila vidljivosti da odaberu odgovarajuće.

Verzije redova u tabelama i unosi u indeksima često se nazivaju tuple-ovima. Ovaj termin je preuzet iz teorije relacija - to je još jedno nasleđe akademskog porekla PostgreSQL-a.

### Item Pointers

Niz pokazivača na tuple-ove služi kao sadržaj stranice ili "table of contents". Nalazi se odmah posle zaglavlja.

Unosi u indeksima moraju nekako da se odnose na određene tuple-ve u glavnoj tabeli. Za tu svrhu, PostgreSQL koristi šestobajtni identifikator tuple-a (TID). Svaki TID se sastoji od broja stranice glavne strukture i reference na određenu verziju reda koja se nalazi na ovoj stranici.

Teorijski, tuple-vi bi mogli da se odnose na svoj položaj od početka stranice. Ali, tada bi bilo nemoguće pomeriti tuple-ove unutar stranica bez narušavanja ovih referenci, što bi dalje dovelo do fragmentacije stranica i drugih neočekivanih posledica.

Zbog toga PostgreSQL koristi indirektno adresiranje. Identifikator tuple-a se odnosi na odgovarajući broj pokazivača, a ovaj pokazivač specificira trenutni položaj tuple-a. Ako se tuple pomeri unutar stranice, njegov TID ostaje isti. Dovoljno je modifikovati pokazivač koji se takođe nalazi na ovoj stranici.

Svaki pokazivač zauzima tačno četiri bajta i sadrži sledeće podatke:

* položaj tuple-a od početka stranice
* dužina tuple-a
* nekoliko bitova koji definišu status tuple-a

### Free Space

Stranice mogu imati neki slobodan prostor između pokazivača i tuple-ova (što je prikazano u mapi slobodnog prostora). Nema fragmentacije stranica, sav dostupan slobodan prostor uvek je agregiran u jedan blok.

## Row Version Layout

Svaka verzija reda sadrži zaglavlje praćeno stvarnim podacima. Zaglavlje se sastoji od više polja, uključujući sledeće:

* **xmin, xmax** predstavljaju identifikatore transakcija, i koriste se za razlikovanje ove verzije od drugih verzija istog reda.
* **infomask** pruža skup informacionih bitova koji definišu svojstva verzije.
* **ctid** je pokazivač na sledeću ažuriranu verziju istog reda.
* **null bitmap** je niz bitova koji označava kolone koje mogu sadržati NULL vrednosti.

Kao rezultat, zaglavlje ispadne prilično veliko. Potrebno je najmanje 23 bajta za svaki tuple, a ovaj iznos često prelazi zbog null bitmap i obaveznog poravnavanja podataka. U "uskoj" tabeli, veličina različitih metapodataka lako može prevazići veličinu stvarnih podataka koji su sačuvani.

Raspored podataka na disku u potpunosti se poklapa sa predstavljanjem podataka u RAM-u. Stranica zajedno sa svojim tuple-ovima učitava se u keš memoriju onakva kakva jeste, bez ikakvih transformacija. Zbog toga su datoteke sa podacima nekompatibilne između različitih platformi.

Jedan od izvora nekompatibilnosti je redosled bajtova. Na primer, x86 arhitektura je little-endian, z/Architecture je big-endian, a ARM ima konfigurabilan redosled bajtova.

Drugi razlog je poravnavanje podataka prema granicama mašinskih reči, što je potrebno za mnoge arhitekture. Na primer, u 32-bitnom x86 sistemu, celi brojevi (tip integer, zauzima četiri bajta) poravnavaju se prema granici četvorobajtnih reči, baš kao i brojevi dvostruke preciznosti s plutajućim zarezom (tip double precision, osam bajta). Ali u 64-bitnom sistemu, dvostruke vrednosti poravnavaju se prema granici osmobajtnih reči.

Poravnavanje podataka čini veličinu tuple-a zavisnom od redosleda polja u tabeli. Ovaj efekat je obično zanemarljiv, ali u nekim slučajevima može dovesti do značajnog povećanja veličine. Pogledajmo primer:

=> **CREATE TABLE** padding(

b1 boolean,

i1 integer,

b2 boolean,

i2 integer

);

=> **INSERT INTO** padding **VALUES** (**true**,1,**false**,2);

=> **SELECT** lp\_len **FROM** heap\_page\_items(get\_raw\_page('padding', 0));

lp\_len

−−−−−−−−

40

(1 row)

Korišćena je funkcija heap\_page\_items ekstenzije pageinspect da bismo prikazali detalje o pokazivačima i tuple-ovima.

U PostgreSQL, tabele se često nazivaju heap-ovima. To je još jedan nejasan termin koji ukazuje na sličnost između alokacije prostora za tuple-ove i dinamičke alokacije memorije. Neka analogija svakako postoji, ali tabele upravlja potpuno drugačijim algoritmima. Možemo tumačiti ovaj termin u smislu da je "sve gomilano u heap", za razliku od uređenih indeksa.

Veličina reda je 40 bajta. Njegovo zaglavlje zauzima 24 bajta, kolona tipa integer zauzima 4 bajta, a kolone tipa boolean zauzimaju 1 bajta svaka. To čini 34 bajta, i 6 bajta se gubi na poravnanje kolona tipa integer na granicu od četiri bajta.

Ako ponovo izgradimo tabelu, prostor će se efikasnije koristiti:

=> **DROP TABLE** padding;

=> **CREATE TABLE** padding(

i1 integer,

i2 integer,

b1 boolean,

b2 boolean

);

=> **INSERT INTO** padding **VALUES** (1,2,**true**,**false**);

=> **SELECT** lp\_len **FROM** heap\_page\_items(get\_raw\_page('padding', 0));

lp\_len

−−−−−−−−

34

(1 row)

Još jedna moguća mikro-optimizacija je da se tabela počne sa kolonama fiksne dužine koje ne mogu sadržati NULL vrednosti. Pristup ovakvim kolonama će biti efikasniji, jer je moguće keširati njihov offset unutar tuple-a.

## Operacije nad Tuple-ovima

Da bi se identifikovale različite verzije istog reda, PostgreSQL obeležava svaku od njih sa dve vrednosti: xmin i xmax. Ove vrednosti definišu "vreme važnosti" svake verzije reda, ali umesto stvarnog vremena, oslanjaju se na rastuće ID transakcija.

Kada se red kreira, njegova xmin vrednost se postavlja na ID transakcije INSERT naredbe.

Kada se red obriše, xmax vrednost njegove trenutne verzije se postavlja na ID transakcije DELETE naredbe.

Sa određenim stepenom apstrakcije, UPDATE naredbu možemo posmatrati kao dve odvojene operacije: DELETE i INSERT. Prvo se xmax vrednost trenutne verzije reda postavlja na ID transakcije UPDATE naredbe. Zatim se kreira nova verzija ovog reda, a njena xmin vrednost će biti ista kao xmax vrednost prethodne verzije.

# Zaključak

U tekstu koji prethodi bila je prikazana složena arhitektura i mehanizmi koji stoje iza ovog sistema za upravljanje bazama podataka. PostgreSQL se ističe svojom sposobnošću da efikasno upravlja i organizuje velike količine podataka, pružajući istovremeno fleksibilnost i skalabilnost koje su neophodne za savremene poslovne aplikacije i analitičke sisteme.

Jedna od ključnih komponenti interne strukture PostgreSQL-a je sistemski katalog, koji omogućava centralizovanu kontrolu i upravljanje metapodacima. Ova funkcionalnost je esencijalna za organizaciju podataka i omogućava različite metode pristupa i manipulacije podacima. Takođe, šeme unutar PostgreSQL-a pružaju mogućnost da se podaci grupišu i organizuju u logičke celine, što olakšava upite i operacije nad bazom.

Tabelarni prostori i relacije su osnovne strukturne jedinice PostgreSQL baze podataka, koje omogućavaju strukturiranu i povezanu organizaciju podataka. Ove komponente omogućavaju izgradnju kompleksnih baza podataka sa jasnim i konzistentnim strukturama, čime se olakšava izvršavanje kompleksnih upita i optimizacija performansi.

Pored toga bila je prikazana i napredna tehnika kao što je TOAST za efikasno upravljanje velikim podacima. Ova tehnika automatski optimizuje skladištenje velikih atributa, pružajući bolju iskorišćenost prostora i poboljšane performanse.

PostgreSQL omogućava korisnicima da optimizuju performanse sistema prema svojim specifičnim potrebama. Ova fleksibilnost je ključna za pružanje optimalne performanse u različitim scenarijima korišćenja, što čini PostgreSQL pouzdanim izborom za širok spektar aplikacija i poslovnih potreba.

Interna struktura i organizacija skladišta podataka u PostgreSQL bazi su pažljivo dizajnirane da pruže visoke performanse, pouzdanost i fleksibilnost. Sa kontinuiranim unapređenjem i podrškom široke zajednice, PostgreSQL ostaje jedan od vodećih sistema za upravljanje bazama podataka koji postavlja standarde u industriji.

# Literatura

[1] PostgreSQL dokumentacija - <https://www.postgresql.org/files/documentation/pdf/16/postgresql-16-A4.pdf>

[2] Ibrar Ahmed, Gregory Smith, Enrico Pirozzi - PostgreSQL 10 High Performance-Packt Publishing (2018)

[3] Jayadevan Maymala - PostgreSQL for Data Architects-Packt Publishing (2015)

[4] Joshua D. Drake, John C. Worsley - Practical PostgreSQL

[5] Baji Shaik, Avinash Vallarapu - Beginning PostgreSQL on the Cloud\_ Simplifying Database as a Service on Cloud Platforms-Apress (2018)