



**UNIVERZITET U NIŠU  
ELEKTRONSKI FAKULTET**



**Fizičko Projektovanje PostgreSQL Baze Podataka i Optimizacija  
Podataka**  
-Seminarski rad-

**Mentor:**

Prof. dr. Aleksandar Stanimirović

**Student:**

Sena Savić, br ind. 1570

Niš, 2023.

## SAŽETAK

U seminarskom radu na temu „Fizičko projektovanje PostgreSQL baze podataka i optimizacija podataka“, obrađene su celine koje se odnose na upoznavanje sa terminima PostgreSQL baze podataka i fizičkog projektovanja, njihovim osnovnim konceptima i osobinama.

Centralni deo rada odnosi se na razmatranje indeksiranja kao elementa fizičkog projektovanja PostgreSQL baze podataka i bavi se izučavanjem i opisom pojmova indeksiranja, indeksa i strukture indeksa, opštih vrsti indeksa i indeksnih vrsti kod PostgreSQL baze podataka. Takođe se fokusira na prikaz praktične realizacije svojstava indeksnih vrsti kod PostgreSQL baze podataka, pokazujući na taj način u kojim situacijama je u toku fizičkog projektovanja, neophodno izabrati određenu indeksnu strukturu i u čemu je značaj korišćenja indeksiranja kod fizičkog projektovanja.

Važan aspekt rada je i proces vakumiranja kao jedan od načina optimizacije tabela i indeksnih struktura u PostgreSQL bazi podataka. Pored opisa procesa vakumiranja i njegovog uticaja, dat je i pregled osnovnih parametara i njihovih efekata na elemente PostgreSQL baze podataka i praktični primer načina funkcionisanja vakum procesa.

## SADRŽAJ

1. UVOD .....	4
2. POSTGRESQL BAZA PODATAKA .....	5
2.1 Kratak pregled istorijskog razvoja PostgreSQL baze podataka .....	5
2.2 Arhitektura PostgreSQL baze podataka .....	6
2.3 PostgreSQL Feature-i.....	8
2.4 Alati koji se koriste uz PostgreSQL bazu podataka .....	8
2.5 Prednosti i mane PostgreSQL baze podataka.....	8
3. FIZIČKO PROJEKTOVANJE POSTGRESQL BAZE PODATAKA.....	9
3.1 Proces fizičkog projektovanja .....	9
3.2 Ključni elementi koji se razmatraju u toku procesa fizičkog projektovanja .....	10
3.3 Izazovi u toku procesa fizičkog projektovanja.....	11
3.4 Značaj fizičkog projektovanja baze podataka .....	12
4. INDEKSIRANJE KAO DEO FIZIČKOG PROJEKTOVANJA POSTGRESQL BAZE PODATAKA.....	12
4.1 Definicije pojmova indeksiranja i indeksa .....	12
4.2 Vrste indeksa.....	13
4.3 Indeksi kod PostgreSQL baze podataka.....	19
4.4 Značaj indeksa za fizičko projektovanje PostgreSQL baze podataka.....	32
5. OPTIMIZACIJA PODATAKA POSTGRESQL BAZE PODATAKA.....	33
5.1 Fizičko mapiranje tabela i indeksa kod PostgreSQL baze podataka.....	33
5.2 Vacuuming tehnika kao metoda optimizacije kod PostgreSQL baze podataka .....	34
6. ZAKLJUČAK .....	37
7. SPISAK KORIŠĆENE LITERATURE.....	38

## 1. UVOD

Pojam tehnologija baza podataka predstavlja koncepte, sisteme, metode i alate koji podržavaju modelovanje, konstrukciju i sam rad baza, čineći ih na taj način neizostavnim delom softverskih sistema. Takođe, zbog svega prethodno navedenog, ali i zbog specifičnih svojstava modela baza podataka i njihovih pojedinačnih karakteristika i mogućnosti, imaju veliku primenu u mnogim naučnim oblastima, profesionalnim i privatnim delatnostima.

Sam pojam baze podataka predstavlja kolekciju međusobno povezanih podataka, tabelarno organizovanih ili u vidu neke druge strukture podataka, koja se koristi za jednu ili više aplikacija. Podaci u bazama mogu biti istog tipa ili raznorodni( numerički, tekstualni, slike, video, itd.), a samom bazom upravlja sistem za upravljanje bazama podataka(DBMS). Podaci skladišteni u bazi, zajedno sa DBMS-om i programima koji koriste podatke iz same baze, zajedničkim imenom nazivaju se sistemom baze podataka.

DBMS (eng. Database Management System) predstavlja softverski sistem koji korisnicima omogućava da stvaraju, upravljaju i koriste baze podataka. Takođe im pruža i različite vrste funkcionalnosti, poput kreiranja, brisanja i izmene zapisa i upita koji se vrše nad podacima skladištenim u bazi, upravljanja pristupom i sigurnosti podataka, kao i pružanjem podrške za transakcije. Primeri DBMS-a uključuju MySQL, Oracle, Microsoft SQL Server, PostgreSQL, MongoDB i mnoge druge.

PostgreSQL je objektno-relacioni sistem za upravljanje bazama podataka, koji predstavlja open-source rešenje i koristi prošireni SQL upitni jezik. Nastao je 1986. godine, a od 2001. godine je ACID kompatibilan, odnosno podržava validne transakcije u okviru baze podataka. Kao open-source potomak originalnog Berkeley-ovog koda, PostgreSQL nudi mnoga moderna svojstva poput: formiranja složenih upita, mehanizama stranih ključeva, transakcionog integriteta. Zbog liberalnih uslova licenciranja, PostgreSQL može biti korišćen i modifikovan besplatno od strane korisnika u privatne, komercijalne ili akademske svrhe [1].

Fizički dizajn baze podataka, odnosno fizičko projektovanje baze podataka razmatra tzv. „radna opterećenja“ (eng. workloads) koja baza mora da podrži i odnosi se na usavršavanje fizičkog dizajna baze, kako bi se osiguralo ispunjenje željenih kriterijuma i performansi [2]. Optimizacija performansi baza podataka u toku procesa fizičkog dizajna, odnosi se na organizaciju same baze na fizičkom nivou u vidu izbora tipova podataka, ali i dodavanja karakteristika fizičkog dizajna, kao što su indeksiranje, particionisanje, materijalizovani prikazi [3], grupisanje pojedinih tabela ili suštinski redizajn delova šeme baze podataka, dobijene u ranijim fazama projektovanja [2].

Optimizacija podataka, takođe utiče na poboljšanje performansi baze podataka i to najčešće kroz smanjenje vremena odziva prilikom izvršavanja upita nad bazom. Takođe, može uključivati i optimizaciju same baze podataka, na primer kroz izbor odgovarajućeg indeksa, smanjenja broja join operacija, ali i optimizacije fizičkog dizajna.

Cilj seminarskog rada je proučavanje i upoznavanje sa tehnikama fizičkog projektovanja PostgreSQL baze podataka, sa akcentom na indeksiranju, kao jednom od segmenata fizičkog projektovanja (šta se podrazumeva pod terminom indeks, kako indeksi izgledaju, itd.), uključujući i proučavanje optimizacije podataka. Pored toga, u radu će biti reči i o samoj PostgreSQL bazi podataka, pregledu njenog istorijskog razvoja, ali i o njenim najznačajnijim svojstvima i karakteristikama.

## 2. POSTGRESQL BAZA PODATAKA

Objektno-relacioni sistem za upravljanje bazama podataka, danas poznat kao PostgreSQL, izveden je iz POSTGRES paketa napisanog na Univerzitetu Kalifornije u Berkliju. Nakon dve decenije razvoja, PostgreSQL je danas jedna od najpoznatijih i najnaprednijih baza podataka otvorenog koda [4], koja se koristi u privatne, akademske ili komercijalne svrhe.

PostgreSQL koristi prošireni SQL upitni jezik, ACID je kompatibilan i nudi mnoga moderna svojstva, poput mehanizama kreiranja kompleksnih upita, principa stranih ključeva, trigera, transakcionog integriteta i updatable pogleda, kao i mnogih drugih svojstava. Takođe, PostgreSQL može biti i proširen od strane korisnika, na mnogo načina, dodavanjem novih tipova podataka, funkcija i operatora, agregacionih funkcija i indeksnih metoda, ali i proceduralnih jezika [1].

Još jedna od značajnih karakterista PostgreSQL baze podataka je konstantan rad na unapređenju postojećih verzija i prevazilaženju njihovih nedostataka, tako da korisnici uvek imaju najažurnije verzije, spremne za korišćenje. Trenutno aktuelna i stabilna verzija je PostgreSQL 15, objavljenja oktobra 2022.godine, koja je februara meseca 2023. godine, unapređena uvođenjem verzija PostgreSQL 15.1 i 15.2. Za prikaz praktično realizovanih teorijskih elemenata, obrađenih u seminarskom radu, korišćene su najnovije verzije.

### 2.1 Kratak pregled istorijskog razvoja PostgreSQL baze podataka

Implementacija projekta „POSTGRES“ započela je 1986.godine pod rukovodstvom profesora Majkla Stounbrejkera ( eng. Michael Stonebraker), i bila je sponzorisana od strane DARPA (eng. Defense Advanced Research Projects Agency), ARO-a (eng. Army Research Office), NSF-a (eng. National Science Foundation) i ESL-a. Tada su postavljeni temelji za definiciju početnog modela podataka, koncepata i dizajna sistema, ali i obrazloženja arhitekture skladišta [4]. Prvi „demoware“ sistem postao je operativan 1987. godine, a javnosti je predstavljen 1988. godine na ACM-SIGMOD konferenciji, da bi tek 1989. godine bio dat na korišćenje nekolicini korisnika, nakon čega se nastavilo sa razvojem projekta i prevazilaženjem nedostataka kroz kreiranje novijih verzija sistema. U trećoj verziji, objavljenoj 1991.godine, dodata je podrška za višestruke storage manager-e, poboljšano je izvršenje upita, ali su i redizajnirana pravila razvoja sistema [4]. Do pojave Postgres95 verzije, razvojni timovi su bili fokusirani na obezbeđenje prenosivosti i pouzdanosti sistema, a sam sistem je korišćen za implementaciju različitih aplikacija, poput sistema za analizu finansijskih podataka, baze podataka za praćenje asteroida, baze podataka za medicinske i geografsko-informacione sisteme, ali se koristio i kao obrazovno sredstvo na nekoliko univerziteta.

Zvanično, originalni Berklijev projekat „POSTGRES“, završio je sa razvojem nakon objavljivanja verzije 4.2.

Nakon okončanja razvoja „POSTGRES“ projekta, Endru Ju (eng. Andrew Yu) i Džoli Čen ( eng. Jolly Chen), kreirali su 1994. godine open-source rešenje kao potomka originalnog projekta, obogativši original prevodiocem za SQL upitni jezik. Novokreirani sistem je nazvan Postgres95, celokupan kod je bio ANSI C i u odnosu na originalni sistem, veličina je smanjena za 25%, a brzina rada povećana za oko 30%-50% u odnosu na poslednju POSTGRES verziju 4.2. Glavne novine u Postgres95 sistemu, bile su [4]:

- Zamena PostQUEL upitnog jezika SQL upitnim jezikom;
- Omogućavanje novog programa za korišćenje interaktivnih upita (psql);
- Dodavanje nove front-end biblioteke (libpq);
- Uklanjanje sistema inverzije datoteka;
- Uklanjanje instance-level pravila sistema;

- Prevazilaženje problema pojave duplikata.

1996. godine, autori Postgres95 sistema, promenili su mu ime u PostgreSQL, ime koje je globalno prihvaćeno i koje se koristi i dan danas. Ovo ime je izabrano kako bi simbolično označilo odnos između originalnog POSTGRES-a i novijih verzija sa dodacima SQL upitnog jezika. Takođe je vraćena i numeracija verzija u prethodno uspostavljen redosled, koji je započeo sa razvojem originalnog Berklijevog projekta. Trenutno aktuelne verzije, podržane od strane zvanične globalne grupe zadužene za razvoj PostgreSQL-a su verzije 15, 14, 13, 12 i 11. Za razliku od Postgres95 sistema, gde je akcenat bio na identifikaciji, razumevanju i razrešenju postojećih mana i problema u samom kodu servera, novije verzije PostgreSQL-a, okreću se ka poboljšanju karakteristika i mogućnosti samog sistema za upravljanje bazama podataka (npr. poboljšanje performansi prilikom izvršenja upita, vođenje računa o ACID svojstvima, itd) iako i dalje prate prethodno započete koncepte razvoja [4].

Pregledom faza istorijskog razvoja PostgreSQL-a, može se zaključiti da se konstantnim unapređivanjem funkcionalnosti, korisnicima omogućava da ostanu konkurentni na tržištu i da budu u toku sa najnovijim tehnologijama. Samim tim, često izdavanje novih verzija PostgreSQL-a utiče na bolju sigurnost i poboljšanje performansi, ali i na unapređenje SQL standarda što direktno ukazuje na visok nivo interoperabilnosti sa drugim bazama podataka i alatima. Takođe, sa pojavom svake nove verzije, deluje se i na poboljšanje tipova podataka, povećanje skalabilnosti sistema, ali i mogućnosti proširenja korišćenjem raznih plug-in-ova, kako bi se izvršila efikasna integracija sa drugim aplikacijama i alatima. Iz svih ovih razloga, PostgreSQL predstavlja jednu od izuzetno naprednih i popularnih baza podataka.

## 2.2 Arhitektura PostgreSQL baze podataka

Komponente na kojima se zasniva arhitektura PostgreSQL baze podataka, podrazumevaju osnovnu (eng. basic) arhitekturu sistema, fizičku strukturu storage-a, pretraživanje podataka (eng. querying), sigurnost i skalabilnost.

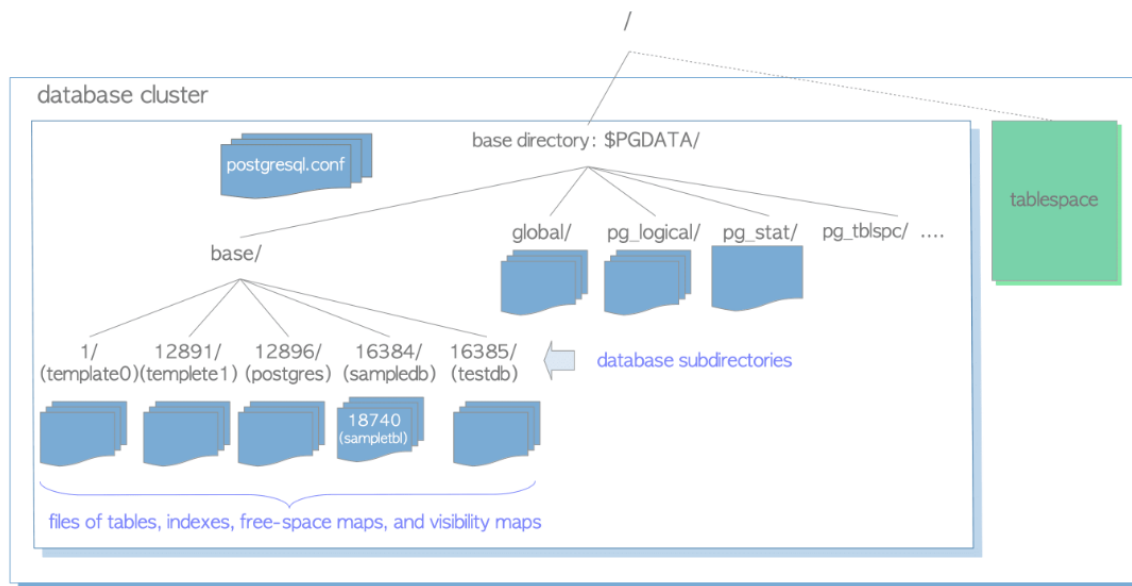
Osnovna arhitektura PostgreSQL baze podataka koristi klijent-server model za pružanje usluga korisnicima, što znači da se PostgreSQL sesija sastoji od međusobno povezanih procesa [5]:

- Server-proces- upravlja podacima u bazi podataka, prihvata i omogućava konekciju između krajnjeg korisnika i baze. Program koji je vezan za ovaj proces se naziva postgres.
- Klijent-proces- uglavnom front-end deo aplikacija, preko kog klijenti zahtevaju izvršenje određenih operacija nad bazom podataka. Za realizaciju prikaza ove vrste procesa mogu se koristiti različiti alati, kao što su karakter interfejs alati, grafički interfejs ili web serveri koji prikazuju web stranice koje imaju pristup bazi podataka.

PostgreSQL server ima mogućnost upravljanja višestrukim korisničkim zahtevima, tako što podržava mehanizam „fork-ovanja“, tj kreiranja novih procesa od postojećeg roditeljskog, kako bi uspešno i nezavisno jedan od drugog, obradio sve korisničke zahteve.

Podaci koji se skladište u PostgreSQL bazi podataka, inicijalno se čuvaju u okviru direktorijuma, koji se po default-u kreira sa kreiranjem relacione baze podataka, odnosno svaka kreirana baza u sistemu ima svoje nezavisne datoteke u okviru glavnog direktorijuma (database cluster). Važno je napomenuti da od PostgreSQL verzije 8, postoji dodatna oblast za skladištenje podataka van glavnog direktorijuma (eng. tablespace) [6]. Pored samih podataka, u direktorijumu se nalaze i informacije o konkretnoj šemi relacione baze kojoj oni pripadaju i elementima koji se u njoj nalaze (informacije o tabelama, indeksima, sekvencama, itd). Svi ovi elementi su objekti

baze podataka, kojima se interno upravlja preko njihovih jedinstvenih identifikatora ( eng. object identifier- OID). OID je neoznačen 4 byte-ni integer, koji je smešten u pg\_class sistemu tabele baze podataka [5].



Slika 1- Primer clustera PostgreSQL baze podataka<sup>1</sup>

Query








Query History

1SELECT oid, relname, relkind FROM pg\_class

Data Output

Messages

Notifications



	oid [PK] oid	relname name	relkind "char" (1)
1	16439	pk_parent...	i
2	16441	pk_keywo...	i
3	16443	pk_digital...	i
4	16445	pk_histor...	i
5	16447	pk_placef...	i
6	16449	pk_court...	i
7	16451	pk_digital...	i
8	16453	pk_person	i
9	16455	pk_courtr...	i
10	16412	parentdir	r
11	16415	keywords	r
12	16424	placefrom	r
13	16421	historyofv...	r
14	16427	courtcase...	r
15	16430	digitalcri...	r
16	16433	person	r
17	16436	courtroom	r
18	16418	digitalevi...	r

Total rows: 429 of 429    Query complete 00:00:00.163

Slika 2- Prikaz kreiranja upita za pribavljanjem OID-a iz pg\_class sistema i rezultata izvršenja

<sup>1</sup> Izvor slike 1- <https://www.interdb.jp/pg/pgsql01.html>

Konkretno govoreći o fizičkoj strukturi storage-a, PostgreSQL podatke skladišti u vidu fajlova koji se nalaze u ranije pomenutom direktorijumu, pri čemu se ti fajlovi nazivaju „relfiles“ ( skraćenica za „relational database files“), gde svaki takav fajl predstavlja tabelu ili indeks iz baze podataka. Tabele predstavljaju najveće logičke jedinice u storage sistemu, jer se svi podaci unešeni u bazu podataka, skladište upravo u tabelama. Kod PostgreSQL-a, posmatrajući organizaciju na najnižem nivou, struktura fajlova tabela, predstavlja blok podataka zapamćen na disku i naziva se stranicom podataka, veličine 8KB. Ukoliko se govori o bloku stranica, uskladištenom u memoriji, on se naziva baferom, dok su tabele i indeksi jednim imenom nazvani relacijama, a sami redovi u tabelama taplovima [5]. Na višim nivoima, a u zavisnosti od veličine tabela, govori se o većem broju stranica, koje se organizuju u hijerarhijsku strukturu nazvanu „B-tree“, koja omogućava brzu pretragu podataka. Pored ovakve pretrage, uvek je moguće izvršiti i sekvencijalnu pretragu, koja se kod PostgreSQL-a naziva sekvencijalno skeniranje i podrazumeva skeniranje svih pointera postojećih stranica [5].

Na kraju, fizička struktura storage-a kod PostgreSQL-a je dizajnirana tako da obezbedi efikasan i pouzdan pristup podacima, ali i da garantuje mehanizme za obezbeđenje integriteta podataka i oporavka u slučaju kvara sistema.

### **2.3 PostgreSQL Feature-i**

PostgreSQL je jedna od najpopularnijih baza koja podržava JSON i SQL upite (ne relacione i relacione upite). Takođe, podržava širok asortiman naprednih tipova podataka i robusnih setova podataka, čime direktno utiče na mogućnost proširivosti sistema, pouzdanosti i garancije integriteta podataka. Pored nabrojanih feature-a, PostgreSQL karakterišu još i kompatibilnost sa različitim operativnim sistemima i programskim jezicima, podrška za razne SQL feature (MVCC, SQL sub selekcija, itd), kompatibilnost sa različitim tipovima podataka, ali i mogućnost besplatnog download-a i instalacija sistema ( open-source rešenje) [6].

### **2.4 Alati koji se koriste uz PostgreSQL bazu podataka**

PostgreSQL administracija za održavanje baze podataka, omogućava i podržava rad PostgreSQL-a sa velikim brojem open-source i komercijalnih alata. Za izradu seminarskog rada korišćeni su [7] :

- psql- front-end terminal alat komandne linije, koji omogućava direktno unošenje i izvršenje SQL upita, uključujući i prikaz alata. Takođe nudi i veliki broj metanardebi;
- pgAdmin- najpoznatiji alat, tj platforma otvorenog koda za upravljanje i razvoj PostgreSQL-a. Kompatibilan je sa velikim brojem OS-a.

### **2.5 Prednosti i mane PostgreSQL baze podataka**

Naposletku i PostgreSQL baza podataka, kao i ostale baze podataka, obiluje brojnim prednostima, ali i nedostacima.

Neke od prednosti korišćenja PostgreSQL baze podataka su [7]:

- Jednostavnost korišćenja;
- Zahteva nizak nivo upravljanja enterprise i embeded usluga;
- Upravljanje velikom količinom podataka u okviru relacione baze je efikasno;
- Otpornost na rizike.

Neki od nedostataka korišćenja PostgreSQL-a su [7]:

- Manja brzina performansi u odnosu na neke druge baze podataka (npr u odnosu na MySQL);



- Proces kreiranja replikacija je složen;
- Smanjen broj podrške za razne open-source aplikacije u odnosu na druge baze (npr. MySQL) i neintuitivan proces instalacije sistema;

### 3. FIZIČKO PROJEKTOVANJE POSTGRESQL BAZE PODATAKA

Čin generisanja fizičke strukture podataka iz ranije kreiranog modela logičke šeme podataka i implementacija sistema za upravljanjem bazom podataka predstavlja fizičko projektovanje baze podataka [8].

Fizičko projektovanje baze podataka je proces dizajniranja i implementacije baze podataka na fizičkom nivou. Pod ovim se podrazumeva razmatranje načina skladištenja podataka na disku, izbor vrste indeksa i ključeva koji će se koristiti za pretraživanje podataka, ali i kako će podaci biti raspoređeni na fizički različitim uređajima u cilju povećanja performansi. Glavna svrha fizičkog projektovanja je transliranje logički opisanog skupa podataka u tzv. „tehničku specifikaciju“ za skladištenje i pribavljanje podataka [9].

Fizički dobro isprojektovana baza podataka treba da sadrži specifikaciju svih tabela, vrsti i kolona u tabelama, ali treba da podrži i prepoznavanje relacionih odnosa između elemenata uskladištenih u bazi podataka, jer upravo pomenuti aspekti odvajaju fizički model od logičkog modela. Iz ovoga se zaključuje da je logički model nacrt po kome se vrši fizičko projektovanje baze podataka [8].

Ključni elementi koji se razmatraju u toku procesa fizičkog projektovanja su definisanje šema baze podataka, identifikacija ključeva, izbor vrste podataka, particionisanje, kreiranje i korišćenje materijalizovanih pogleda, strategije čuvanja podataka, klastering i indeksiranje, o kome će detaljno biti reči u narednom poglavlju seminarskog rada.

Cilj fizičkog projektovanja je formiranje skladištenog sistema, koji će pored uloge čuvanja podataka, garantovati i integritet baze podataka, bezbednost i mogućnost oporavka sačuvanih podataka [9].

#### 3.1 Proces fizičkog projektovanja

Proces fizičkog projektovanja PostgreSQL baze podataka, kao i bilo kog drugog sistema za upravljanjem bazama podataka, obuhvata niz koraka koje je neophodno pažljivo razmotriti i izvršiti, kako bi isprojektovana baza bila pouzdana. Glavni koraci u toku fizičkog projektovanja su:

- Definisanje šema baze podataka – označava proces definisanja strukture baze podataka (koje će tabele biti kreirane, polja i veze među njima);
- Identifikacija ključeva – identifikacija primernih i stranih ključeva;
- Indeksiranje- kreiranje odgovarajućih indeksnih struktura, radi postizanja efikasne pretrage podataka;
- Izbor vrste podataka – izbor vrsti podataka koji će se nalaziti u poljima tabela, na osnovu potreba sistema za koji se baza projektuje;
- Normalizacija- uklanjanje redundanse;
- Particionisanje- podela velikih tabela na manje celine;
- Optimizacija performansi- podešavanje projektovanog sistema za poboljšanje izvršenja upita. Podešavanje može uključivati podešavanje radne memorije, parametara konfiguracije baze podataka i izbor odgovarajućih hardverskih resursa;

- Testiranje i podešavanje- isprojektovan sistem je potrebno testirati i podesiti, kako bi se postigla što veća pouzdanost i optimalnost;

### 3.2 Ključni elementi koji se razmatraju u toku procesa fizičkog projektovanja

U toku procesa fizičkog projektovanja razmatraju se i analiziraju mnogi faktori, poput definisanja strukture podataka i veza među njima, identifikacije primarnih i stranih ključeva za efikasno pretraživanje, normalizacije koja služi kako bi se uklonila redundantnost, ali i optimizacije i testiranja performansi projektovanog sistema ( ovi faktori su pomenuti nešto ranije u seminarskom radu, u okviru ovog poglavlja).

Ključni među nabrojanim faktorima su ipak načini čuvanja podataka na fizičkim lokacijama, particionisanje, klastering i indeksiranje.

Nakon što se na logičkom nivou izvrši normalizacija, posmatrana logička šema se prevodi u fizički dizajn baze podataka. Jedno od važnih pitanja u tom trenutku projektovanja jeste i način na koji baza vrši fizičko skladištenje podataka. Arhitektura i neki od načina čuvanja podataka na fizičkim lokacijama na sistemu PostgreSQL baze podataka, opisani su u prethodnom poglavlju, ali predstavljaju ključni element u procesu fizičkog projektovanja. Vrlo je važno da u fizičkom dizajnu, svi tabelarni prostori budu različiti, tj. na različitim lokacijama, kako ne bi došlo do međusobnog preklapanja. To znači da tabele i njihovi indeksi treba da imaju različitu lokaciju u okviru generalnog direktorijuma ( ukoliko se govori o basic direktorijumu, ali i o tablespace prostorima, koji se nalaze van basic direktorijuma).

Fizičko projektovanje se bavi i dizajnom polja. Polje predstavlja najmanju jedinicu aplikativnih podataka koju prepoznaje sistemski softver. Polja su povezana sa kolonama u tabeli baze podataka i oslanjanju se na izbor tipa podataka, tehnike kodiranja, integritet podataka, ali i na rukovanje null unosima za vrednost nekog podatka. Dizajn polja u procesu fizičkog projektovanja je značajan, jer ukoliko se ova faza razvoja izvrši pravilno, postiže se visok stepen efikasnosti sistema baze podataka [10].

**Particionisanje** kao jedna od tehnika fizičkog projektovanja, predstavlja podelu podataka uskladištenih u velikim tabelama, na taj način što se na osnovu nekog kriterijuma velika količina podataka deli i razmešta u manje tabele, što direktno utiče na efikasnije upravljanje i brže pretraživanje podataka [8].

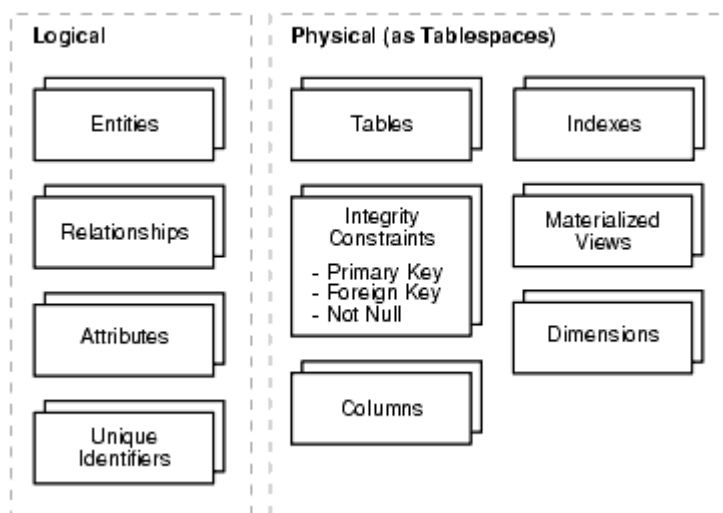
PostgreSQL nudi nekoliko vrsti particionisanja:

- Particionisanje po opsegu- podela tabela na particije na osnovu opsega vrednosti u određenom polju. Particionisanje po opsegu je dobro primeniti kada tabele u sebi imaju prirodni raspored vrednosti ( npr. datumi, cene i tome slično);
- Particionisanje po listama- kreiranje particija na osnovu lista diskretnih vrednosti u određenom polju. Dobro je primeniti ovu vrstu particionisanja za one tabele koje imaju diskretne vrednosti u svojim poljima;
- Hibridno particionisanje- kombinuje prethodne dve vrste particionisanja, na taj način što se particionisanje u jednom polju može izvršiti na osnovu opsega vrednosti, a u drugom na osnovu lista diskretnih vrednosti. Ova vrsta particionisanja je dobra kada je potrebno precizno kontrolisati raspored particija kreiranih na osnovu više kriterijuma.
- Hash particionisanje – tabela se particioniše specificiranjem modula za particionisanje i ostatka za svaku particiju. Heš vrednost ključa particije podeljena sa specificiranim modulom, uzrokuje pojavu tog ostatka particije, pri čemu tada svaka particija sadrži redove iz originalne tabele [11].

Sve vrste particionisanja se oslanjaju na šeme za particionisanje, koje su karakteristične za PostgreSQL, jer upravo ove šeme definišu način na koji se određena tabela particioniše, ali i kako se particije raspoređuju.

**Klastering** baze podataka je jedan od elemenata na koje posebno treba obratiti pažnju, jer predstavlja zadatak objedinjenja više instanci povezanih sa jednom bazom podataka. Ovaj pristup je koristan kada jedan server nije dovoljan za upravljanje kompletnom količinom generisanih zahteva za podacima. Glavna prednost klasteringa je što se njegovim uvođenjem u toku projektovanja postiže balansiranje opterećenja na server, povećava dostupnost podacima i postiže bolje praćenje istih [8].

**Indeksiranje** predstavlja ključan faktor prilikom fizičkog projektovanja, jer je sam indeks struktura, koja u mnogome ubrzava pretragu podataka, kao i izvršenje upita nad njima [8]. Samim tim, indeksne strukture utiču na poboljšanje performansi baze podataka, ali u celini povećavaju i opterećenje sistema baze podataka, tako da prilikom projektovanja baze i o tome treba voditi računa [12]. Detalji o principu indeksiranja i vrstama indeksnih struktura kod PostgreSQL baze podataka, obrađeni su u narednom poglavlju seminarskog rada, koje predstavlja i suštinu istraživačkog rada.



Slika 3- Prikaz poređenja logičkog modela i fizičkog modela sistema baza podataka<sup>2</sup>

### 3.3 Izazovi u toku procesa fizičkog projektovanja

Proces fizičkog projektovanja baze podataka, može naići na mnoge izazove.

Prvi izazov na koji se nailazi je odluka o načinu skladištenja svakog atributa. Izbor ispravnog načina grupisanja atributa za formiranje fizičkih zapisa i odabir tačnih struktura za povezivanje datoteka, koje se koriste za efikasno pronalaženje podataka, mogu biti usko grlo u procesu fizičkog projektovanja.

Takođe, još jedan od izazova u toku ovog procesa, jeste razmatranje smanjenja vremena odziva u komunikaciji korisnik-sistem. Ova stavka se povezuje sa izborom formata skladištenja, jer i on sam treba da bude optimalan u smislu brzine i prostora. Tačnije, skladišteni format, mora da bude formiran tako da uspešno skladišti sve neophodne podatke, pritom smanjujući neophodni fizički prostor i poboljšavajući integritet podataka [8].

<sup>2</sup> Izvor slike 3- [https://docs.oracle.com/cd/B13789\\_01/server.101/b10736/physical.htm](https://docs.oracle.com/cd/B13789_01/server.101/b10736/physical.htm)

### 3.4 Značaj fizičkog projektovanja baze podataka

Fizičko projektovanje baze podataka predstavlja izuzetno složen i važan proces u toku kreiranja sistema baze podataka, jer upravo od njega zavisi da li će implementacija čitavog sistema biti ispravna i da li će na pravi način predstaviti prethodno formiran logički nacrt. Značaj fizičkog projektovanja je i u poboljšanju performansi i očuvanju integriteta podataka, što se postiže pravilnim rasporedom podataka na fizičkim lokacijama, optimizacijom podataka i sprečavanjem redundantnosti istih [8].

## 4. INDEKSIRANJE KAO DEO FIZIČKOG PROJEKTOVANJA POSTGRESQL BAZE PODATAKA

U prethodnim delovima seminarskog rada, bilo je reči o mehanizmu indeksiranja, kao jednom od elemenata fizičkog projektovanja.

Indeksiranje ima veoma važnu ulogu u fizičkom projektovanju, jer se strukture indeksa koriste za poboljšanje performansi baza podataka, na taj način što omogućavaju brže pretraživanje i filtriranje podataka.

Ovo poglavlje predstavlja ključni deo seminarskog rada i posvećeno je definiciji pojma indeksa, njegovom značaju u procesu fizičkog projektovanja, ali i vrstama indeksa koje podržava PostgreSQL baza podataka.

### 4.1 Definicije pojmova indeksiranja i indeksa

Opšte je poznato da se podaci u bazama podataka čuvaju u vidu zapisa. Svaki od zapisa ima polje koje je definisano kao ključ polje, na osnovu koga se zapis jedinstveno identifikuje i prepoznaje.

Indeksiranje je tehnika preuzimanja zapisa iz datoteka baze podataka, na osnovu atributa nad kojima su kreirane indeksne strukture, odnosno indeksiranje je definisano na osnovu njegovih indeksnih atributa [13].

Indeksiranje u procesu fizičkog projektovanja ima veliki značaj, jer se koristi za optimizaciju performansi baze podataka, smanjenjem broja pristupa podacima na disku prilikom obrade upita [14].

Indeks predstavlja strukturu podataka, koja omogućava pretragu ID slogova, tako što pronalazi slogove koji imaju zadatu vrednost u poljima- ključevima indeksa. Takođe se koristi i za brzo lociranje i pristup podacima uskladištenim u tabelama.

Jedna od mogućih struktura indeksa prikazana je na slici:

Search key	Data Reference
------------	----------------

Slika 4- Struktura indeksa <sup>3</sup>

Prva od kolona na slici je ključ pretrage, koji sadrži kopiju primarnog ključa ili ključa kandidata za tabelu. Vrednosti primarnog ključa se čuvaju u sortiranom redosledu tako da se

<sup>3</sup> Izvor slike 4- <https://www.javatpoint.com/indexing-in-dbms>

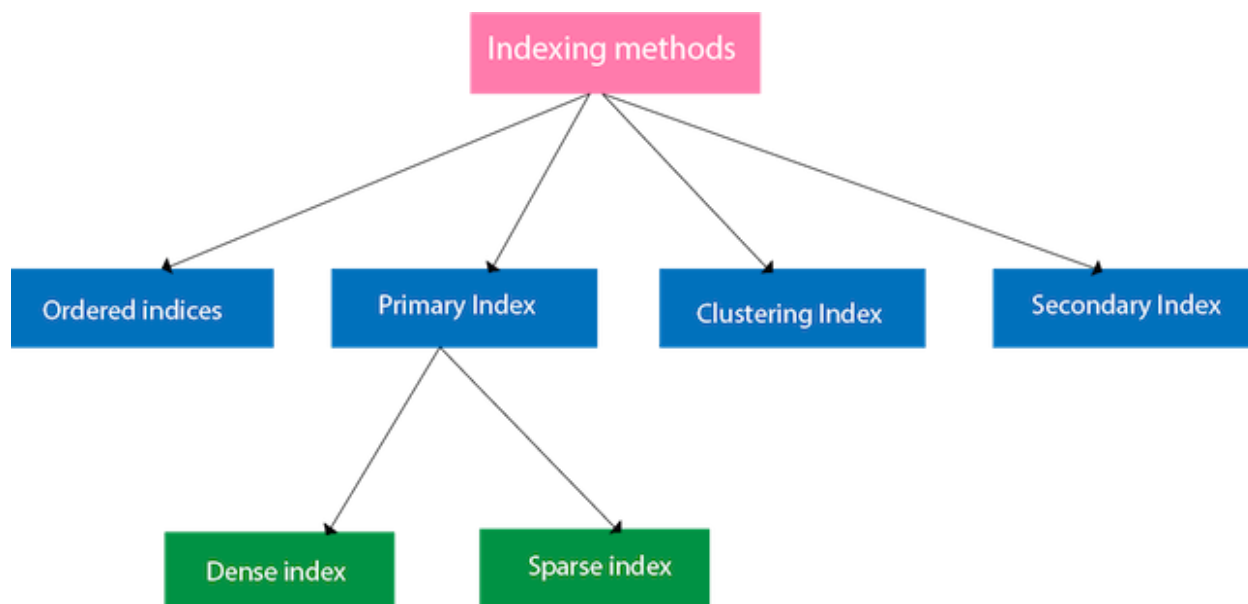
odgovarajućim podacima pristupa lako. Druga kolona je referenca na podatke. Ona sadrži skup pokazivača koji sadrže adresu bloka diska gde se nalazi vrednost određenog ključa [14]. Prikazana indeksna struktura je najgeneralniji opis indeksa, međutim postoje brojne vrste indeksnih struktura, o kojima će biti više reči u nastavku.

#### 4.2 Vrste indeksa

U uvodnom delu priče o indeksnim strukturama, rečeno je da indeksi služe za ubrzanu pretragu zapisa, predstavljajući time odgovor na određene uslove pretraživanja. Pored toga, indeksne strukture su dodatne datoteke na disku koje obezbeđuju sekundarni i alternativni način pristupa zapisima, bez uticaja na promenu fizičke lokacije primarnog fajla na disku, u kome se nalaze zapisi [15].

Za kreiranje indeksa može se koristiti bilo koje polje datoteke, a takođe se indeks može kreirati i nad većim brojem polja (više-nivovski indeksi). Samim tim, dolazi se do zaključka da postoji dosta različitih vrsti indeksnih struktura, a najzastupljeniji među njima su indeksi zasnovani na uređenim datotekama (uređeni jedno-nivovski indeksi), višenivovski indeksi i indeksi zasnovani na strukturi stabla, a postoje i hash indeksne strukture i bitmap indeksi [15].

Kod **uređenih jedno-nivovskih indeksa**, za zadatu datoteku sa strukturom zapisa, koja se sastoji od nekoliko polja (atributa), indeksna struktura se obično definiše nad jednim poljem datoteke i naziva se atributom indeksiranja [15]. Indeks čuva vrednosti indeksnih polja zajedno sa listom pokazivača na blokove koji sadrže zapise sa određenim vrednostima polja i to tako da se nalaze u uređenom redosledu, pa su nad njima efikasne i sekvencijalna, ali i binarna pretraga.



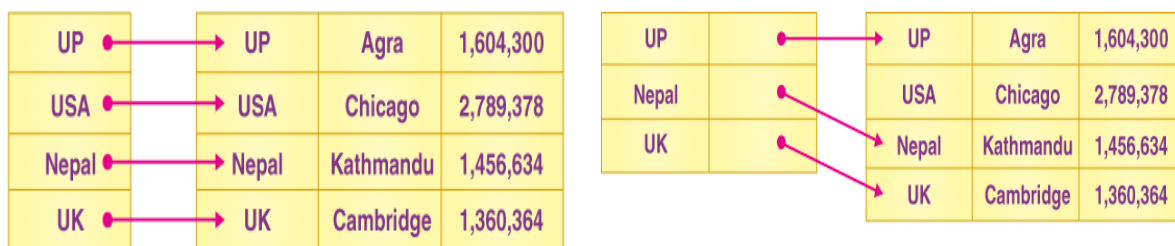
Slika 5- Primer vrsti uređenih jedno-nivovskih indeksa <sup>4</sup>

Jedno-nivovski indeksi se mogu podeliti na:

- **Klasično uređene indekse**- predstavljaju vrstu uređenih/sortiranih indeksa, kako bi pretraga bila lakša;

<sup>4</sup> Izvor slike 5- <https://www.javatpoint.com/indexing-in-dbms>

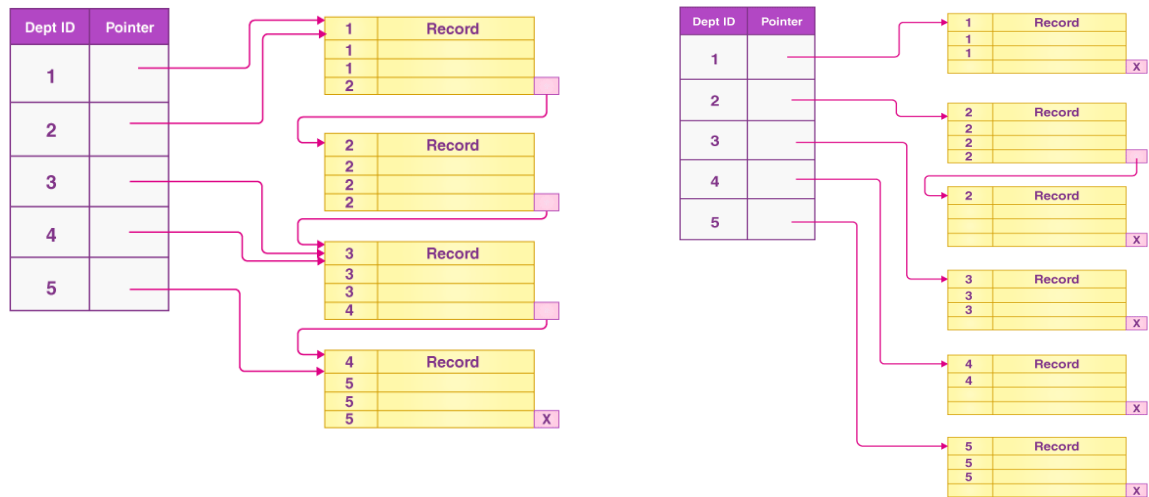
- **Primarne indekse**- kreiraju se na osnovu primarnog ključa tabela i specifični su za svaki zapis posebno ( uspostavljaju odnos 1:1). Primarni indeksi su sortirani fajlovi, fiksne dužine, koji se sastoje od dva polja. Prvo polje je polje koje je po tipu identično datoteci sa podacima, a drugo polje je pokazivač na blok diska. Primarni indeksi mogu biti retki ili gusti. Gusti indeksi za svaku vrednost zapisa imaju ključ pretrage u datoteci podataka (uključeni su pokazivač na stvarni zapis na disku i ključ pretrage), što znači da je pretraga izuzetno brza, ali je neophodan dodatni prostor. Retki indeksi podrazumevaju da samo nekoliko stavki u datoteci podataka imaju indeksne zapise, odnosno svaka stavka ukazuje na određeni blok. Iako se kaže da su retki i gusti indeksi vrste primarnih indeksnih struktura, zapravo su primarni indeksi po svom tipu bliži retkim indeksima, jer podrazumevaju pokazivače na svaki blok diska, a ne za svaku vrednost primarnog ključa. Problem kod ove vrste indeksa jesu kompleksne operacija brisanja i umetanja zapisa.



Slika 6- Primer gustog i retkog indeksa <sup>5</sup>

- **Clustering indekse**- uređene datoteke podataka se mogu označiti kao indeksne strukture i tada se definišu kao klasterizovani indeksi. Karakteristični su po tome što nastaju grupisanjem kolona koje nisu kolone primarnog ključa i koje mogu, ali i ne moraju biti jedinstvene za svaki zapis. Spajanjem više kolona u jedinstvenu vrednost, generiše se indeks, a ovaj metod je poznat kao klasterizacija indeksne strukture. Kao i jedno-nivovski indeks i klastering indeks je uređen fajl od dva polja, u kome je prvo polje istog tipa kao polje po kome je izvršeno grupisanje, a drugo polje je pokazivač na blok diska. I dalje su i kod ove vrste indeksa problem operacije brisanja i umetanja zapisa, jer se klasterizacijom postiže fizičko grupisanje zapisa. Međutim, česta praksa koja olakšava ove operacije jeste rezervisanje celog bloka svake od vrednosti polja po kojima se vrši grupisanje. Zapisi koji imaju istu vrednost se smeštaju u jedan blok/klaster ( ovo upućuje na to da se odvojeni klasteri nalaze na različitim blokovima diska). Primeri klastering indeksa dati su na narednoj slici.

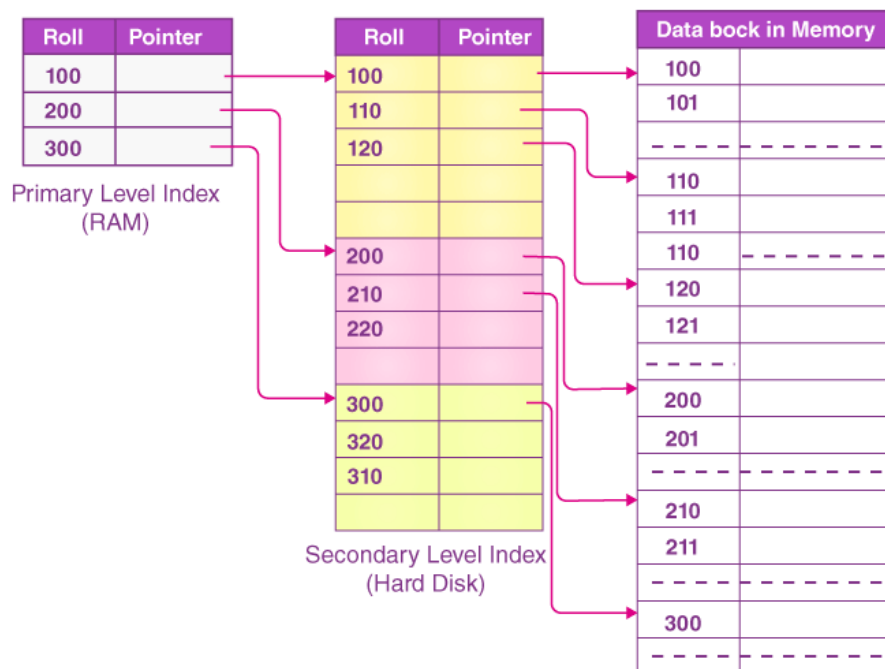
<sup>5</sup> Izvor slike 6 – <https://byjus.com/gate/indexing-in-dbms-notes/>



Slika 7- Primer klasičnog klaster indeksa i klaster indeksa sa posebnim blokovima <sup>6</sup>

- Sekundarne indekse** – U slučajevima kada se koristi retko indeksiranje, veličina mapiranja raste u skladu sa rastom tabele, jer se podrazumeva postojanje pokazivača na svaki blok diska. Takva mapiranja tj. preslikavanja se čuvaju u primarnoj memoriji, kako bi se ubrzalo sa pretragom prilikom pronalaženja adresa bloka podataka. Nakon toga, sekundarna memorija vrši pretragu za stvarnim podacima, koristeći adresu dobijenu mapiranjem. Međutim, na taj način proces preuzimanja adresa postaje sporiji, sa uslozljavanjem veličine mapiranja. Samim tim, retki indeks postaje neefikasan za pretraživanje, pa se uvodi pojam sekundarnog indeksa. Proces kreiranja sekundarnog indeksa, zahteva malo drugačiji pristup prilikom kreiranja indeksne strukture u odnosu na dosad opisane vrste. Kreiranje sekundarnog indeksa podrazumeva da već prethodno postoji primarna indeksna struktura. Iz primarne indeksne strukture, iz ogromnog opsega kolona, bira se manji opseg i već se na tom prvom nivu vrši smanjenje veličine mapiranja. Svaki naredni opseg se deli na manje grupe. Ovo je vrlo važno prilikom fizičkog projektovanja baze. Mapiranje se čuva u primarnoj memoriji (pribavljanje adresa je brže), a stvarni podaci se nalaze na sekundarnoj memoriji (hard disku). Sekundarni indeks se može kreirati nad poljem koje predstavlja kandidata za ključ i pritom ima jedinstvenu vrednost u svakom zapisu, ili nad poljem koje nije ključ, ali sadrži vrednosti koje se ponavljaju. Sekundarni indeks je sortirani fajl sa dva polja. Prvo polje je po tipu identično poljima koja nisu uređena u datoteci podataka i predstavlja polje po kome se vrši indeksiranje. Drugo polje je pokazivač na blok ili na zapis. Ukoliko je sekundarni indeks nastao nad poljem koje je kandidat za ključ, takav indeks se naziva tzv. „sekundarnim ključem“ i u relacionom modelu odgovara atributu primarnog ključa. Sekundarni indeks može biti kreiran i nad poljem koje nije kandidat za ključ. Sekundarni indeks, zbog većeg broja zapisa zahteva više prostora za skladištenje u odnosu na primarni indeks, ali ubrzava vreme izvršenja upita zbog toga što utiče na ubrzanje pretrage (nema potrebe za linearnom pretragom indeksne strukture).

<sup>6</sup> Izvor slike 7- <https://byjus.com/gate/indexing-in-dbms-notes/>



Slika 8 – Primer sekundarnog indeksa <sup>7</sup>

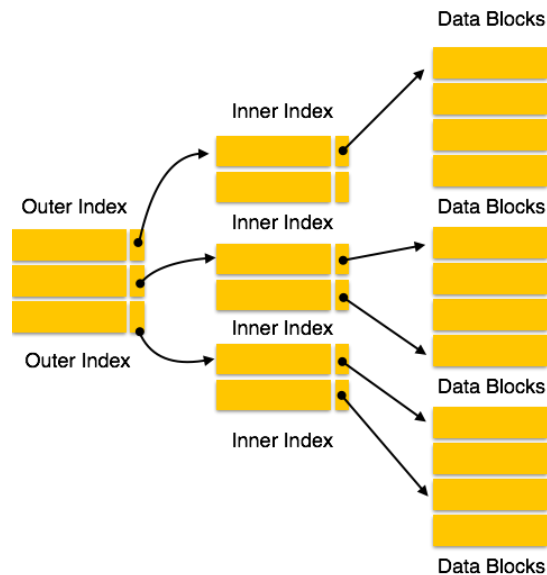
Opisane tehnike i šeme indeksiranja, odnosile su se na uređene indeksne strukture, gde indeksirani zapisi sadrže vrednosti ključeva za pretragu i pokazivače na podatke. Sa porastom veličine baze podataka, raste i veličina indeksa, a kako bi operacije pretraživanja bile što efikasnije (pogotovu binarno pretraživanje), poželjno je da se indeksni zapis nalazi u glavnoj memoriji. Ukoliko se u ovakvoj situaciji koriste jedno-nivovski indeksi, oni se ne mogu kompletno čuvati u primarnoj memoriji, što dovodi do velikog broja obraćanja disku. Iz tog razloga se koriste indeksi u više nivoa, jer se smanjuje vreme pristupa bloku podataka.

**Indeks na više nivoa (više-nivovski indeks)** se skladišti na disku zajedno sa stvarnim datotekama baze podataka. Više-nivovski indeksi imaju strukturu u vidu zapisa fajlova organizovanih u nivoe. Osnovni nivo je prvi nivo od kog se kreće sa metodom indeksiranja i rasporeda slogova. Više-nivovski indeks taj osnovni nivo, odnosno prvi nivo smatra uređenim fajlom sa različitom vrednošću za svaki ključ. Samim tim, taj fajl prvog indeksnog nivoa, predstavlja sortirani fajl sa podacima nad kojim se kreira primarni indeks prvog nivoa. Kreirani indeks prvog nivoa, predstavlja primarni indeks za drugi nivo više-nivovske indeksne strukture [15]. Ovaj isti proces je moguće ponoviti i za drugi nivo i sve naredne nivoe, ali treba voditi računa o blokirajućem faktoru za indekse. Naime, cela suština indeksa raspoređenog u više nivoa, zasniva se na smanjenju indeksa kako bi se ubrzala pretraga. Kako je prvi nivo primarni indeks drugog nivoa, drugi nivo primarni indeks trećeg i tako dalje, blokirajući faktor zapravo predstavlja ograničenje u vidu unosa indeksa za sve nivoe- svi indeksi moraju biti iste veličine i svaki od njih ima tačno 2 polja – vrednost samog polja i adresu bloka.

Veličina više-nivovskog indeksa zavisi od broja zapisa po bloku, odnosno kreiraće se hijerarhijska struktura sve dok svi unosi vezani za trenutno posmatrani nivo ne stanu u tačno jedan blok [15]. Kada se vrši pretraživanje ovakve strukture indeksa, sa svakog nivoa preuzima se po jedan blok diska, što znači da će za n nivoa indeksa biti pristupljeno n blokovima diska.

<sup>7</sup> Izvor slike 8 - <https://byjus.com/gate/indexing-in-dbms-notes/>





Slika 9- Primer više-nivovskog indeksa<sup>8</sup>

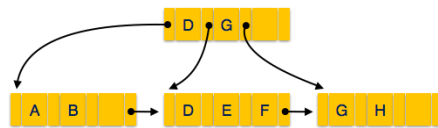
Dinamičke strukture koje prate format indeksa na više nivoa, su B stablo i njegova varijacija B+ stablo.

**B stabla i B+ stabla** su posebni slučajevi strukture podataka koja se koristi za pretragu, poznate kao binarno stablo pretrage. Opšte je poznato da se struktura stabla sastoji od čvorova, gde su neki čvorovi označeni kao roditeljski, neki kao sledbenici, a jedan od njih je koren ( nema svog prethodnika). Čvorovi koji nemaju svoje sledbenike, predstavljaju listove stabla.

B i B+ stablo su balansirna/uravnotežena binarna stabla pretrage koja se koriste da vode računa o pretrazi zapisa na osnovu jednog polja tog zapisa [15]. Više-nivovski indeksi mogu biti smatrani varijacijom stabla pretrage, gde su vrednosti jednog nivoa, zapravo vrednosti koje se nalaze u čvorovima stabla, a čvorovi potomci su elementi narednog nivoa, sve dok se ne dođe do konkretnog bloka datoteke podataka. Pokazivač postoji i u ovakvoj strukturi, i svaki nivo više-nivovskog indeksa je zapravo podstablo koje definiše upravo pokazivač. Operacije brisanja i dodavanja u ovakve indeksne strukture su prilično efikasne.

B stabla kao indeksne strukture, kreću od korena ( nulti nivo kod više-nivovskih indeksa) i kada se taj osnovni čvor napuni vrednostima ključeva za pretragu, koren se deli na još dva čvora. Vrednosti ključeva se ravnomerno dele između novokreiranih čvorova, a u korenu ostaje samo srednja vrednost. Svaka vrednost polja za pretragu i pokazivač, se kod B stabla, javljaju samo jednom na nekom nivou. Kod B+ stabla, pokazivači na podatke se nalaze samo u listovima stabla, što direktno utiče na razliku između listova i ostalih čvorova [15]. Pokazivači koji ne ukazuju na listove u B+ stablu, služe da ukažu na unutrašnje čvorove ( u poređenju sa više-nivovskim indeksima, to su pokazivači na naredni nivo). Korišćenjem B+ stabla kao indeksne strukture, moguće je uskladištiti veći broj zapisa u odnosu na klasično B stablo, jer se u čvorovima B+ stabla pamte i vrednost pretrage ali i pokazivač na podstabla, bez pokazivača na konkretne podatke ( već je pomenuto da se oni pamte u listovima B+ stabla).

<sup>8</sup> Izvor slike 9- [https://www.tutorialspoint.com/dbms/dbms\\_indexing.htm](https://www.tutorialspoint.com/dbms/dbms_indexing.htm)

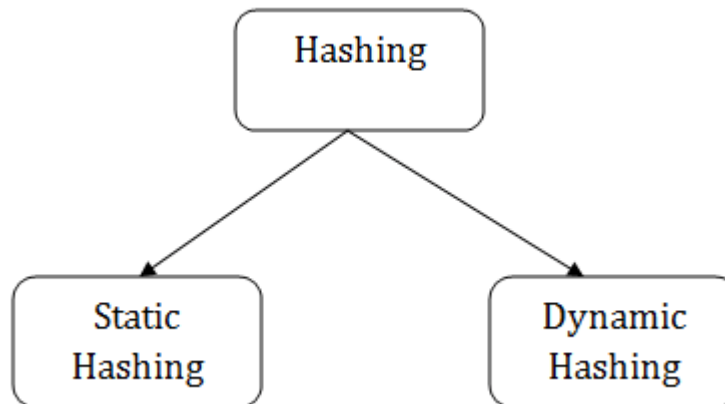


Slika 10- Struktura B stabla<sup>9</sup>

**Hash indeksne strukture** predstavljaju strukture indeksa koja se zasnivaju na heširanju. Ovako kreiran indeks je sekundarna struktura za pristup fajlovima. Nastao je primenom heširanja nad ključem koji nije iskorišćen za organizaciju primarnog fajla podataka. Prema tome heš funkcija, koja je jednostavna matematička funkcija, može izabrati bilo koju vrednost kolone u tabeli za generisanje adrese bloka podataka [16]. I sam primarni ključ, može se smatrati adresom bloka podataka, što znači da će svaki red u tabeli, čija je adresa ista kao primarni ključ, biti uskladištena u istom bloku podataka. Traženje po ovoj strukturi se vrši na taj način što se pretražuje zapis, heširan algoritmom heširanja i predstavljen u obliku ključa. Kada se on pronađe, koristi se pokazivač za lociranje konkretnog zapisa u fajlu podataka.

Postoje dve vrste heširanja:

- Statičko heširanje- rezultujuća adresa podataka je uvek ista. To znači da je broj blokova podataka u memoriji, konstantan tokom čitavog perioda.
- Dinamičko heširanje- prevazilazi probleme statičkog heširanja ( eng. bucket overflow). Segmenti podataka se menjanju sa promenom zapisa (eng. extendable hashing method). Operacije brisanja i umetanja moguće su bez gubitaka.



Slika 11 – Prikaz primera vrsti heširanja<sup>10</sup>

**Bitmap indeksi** su indeksi koji se koriste kod relacija sa velikim brojem redova. Indeks koji je ovako formiran, može da obuhvati jednu ili više kolona, pri čemu se svaka od vrednosti ili ceo opseg vrednosti indeksira. Uglavnom se koristi za indeksiranje kolona koje imaju mali broj jedinstvenih vrednosti. Zapisi u relaciji moraju biti numerisani i to počevši od 0 do n, sa ID-jem zapisa/reda koji se može mapirati na fizičku adresu bloka, kako bi se kreirao indeks bitmape. Ovako kreiran indeks je izgrađen na vrednosti određenog polja i predstavlja niz bitova.

Pored predstavljenih vrsti indeksa, važno je napomenuti da postoji još dosta njih, poput indeksa uređenih po većem broju atributa, particionisanog heširanja itd., ali su opisane vrste

<sup>9</sup> Izvor slike 10- [https://www.tutorialspoint.com/dbms/dbms\\_indexing.htm](https://www.tutorialspoint.com/dbms/dbms_indexing.htm)

<sup>10</sup> Izvor slike 11- <https://www.javatpoint.com/dbms-hashing>

najčešće korišćene i najčešće razmatrane u toku procesa fizičkog projektovanja sistema baze podataka. Koja od tehnika indeksiranja će biti izabrana za kreiranje indeksa, zavisi od potreba i namena sistema za koji se baza projektuje, kako bi efikasnost i optimalnost pribavljanja podataka bile što više.

### 4.3 Indeksi kod PostgreSQL baze podataka

Svi indeksi u PostgreSQL bazi podataka su sekundarni indeksi, što u tehničkom smislu znači da su fizički odvojeni od datoteke tabele koju opisuju. Svaki od indeksa se čuva na posebnoj fizičkoj lokaciji i kao takvi su opisani u pg\_class katalogu. Sadržaj indeksa je u potpunosti pod kontrolom indeksne metode kojoj indeks pripada ( opisane u prethodnom delu poglavlja). U praksi, sve indeksne metode, dele indeks na stranice standardne veličine, kako bi mogli da koriste regularne storage i buffer menadžere za pristup sadržaju indeksa [17].

PostgreSQL baza podataka podržava nekoliko vrsti indeksa:

1. B-tree indeks- rangiranje i sortiranje podataka, kao i brzo pretraživanje na osnovu jednakosti i raspona vrednosti;
2. Hash indeks – brzo pretraživanje podataka zasnovano na jednakosti među vrednostima;
3. GiST indeks – pretraživanje i upoređivanje geometrijskih i tekstualnih podataka;
4. SP-GiST- pretraživanje podataka koji se modeluju u vidu strukture stabla ili grafa;
5. GIN indeks- pretraživanje podataka koji sadrže velike skupove ključeva;
6. BRIN- pretraživanje velikih tabela organizovanih po blokovima.

#### 1. B-tree indeks i primer

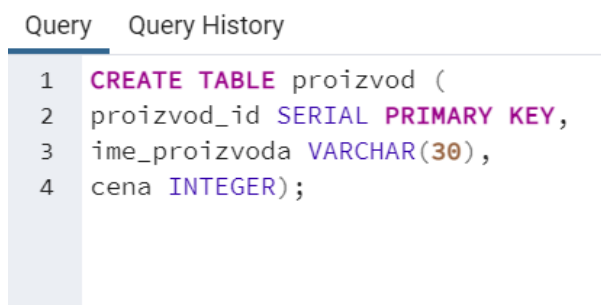
B-tree indeksi su najčešća vrsta indeksa kod PostgreSQL baze podataka (po default-u komandom za kreiranje indeksa, kreira se upravo ovaj tip indeksa) i najkorisnija struktura podataka u sistemima za upravljanjem bazama podataka [18]. Ovaj tip indeksa, implementiran je metodom indeksiranja koja se zasniva na strukturi B stabla i B+ stabla, što znači da je pogodan za podatke koji se mogu rangirati i sortirati. Pored toga, omogućava i efikasnu pretragu zapisa, umesto da se vrši klasično skeniranje cele tabele. Indeksni redovi B-tree-a su zapakovani kao stranice. Jedna od stranica se čuva na fiksnoj poziciji na početku datoteke prvog segmenta, dok ostale stranice predstavljaju listove stabla ili interne stranice. Listovi stranica, sadrže redove koji imaju podatke koje treba indeksirati i reference na redove tabele. Kod internih stranica, svaki red upućuje na podređenu stranicu indeksa (naredni red tabele) i na toj stranici sadrži minimalnu veličinu indeksa. Ovakva struktura B-tree indeksa ukazuje na to da on predstavlja B+ stablo. To znači da je balansiran, pa traženje bilo koje vrednosti traje isto. Kako svaka interna stranica sadrži referencu na naredne redove u tabeli, dubina B-tree-a nije prevelika. Podaci se u indeksu sortiraju u neopadajućem redosledu ( i između stranica i unutar stranice), a stranice su međusobno povezane dvosmernom listom, pa se kretanje može vršiti u oba pravca bez potrebe stalnog obraćanja korenu stabla [19].

B-tree indeksi se najčešće koriste za efikasnu obradu upita koji u sebi sadrže operatore jednakosti i poređenja ( $<$ ,  $>$ ,  $<=$ ,  $>=$ ,  $=$ ), ili imaju neki opseg vrednosti podataka. Ovom vrstom indeksa mogu se indeksirati i upitne konstrukcije koje sadrže BETWEEN, IN, LIKE, ali i konstrukcije koje imaju uslove IS NULL i NOT NULL [20].

U nastavku je dat primer tabele nad čijom kolnom je kreiran B-tree indeks, kao i prikaz benefita korišćenja ove indeksne strukture.

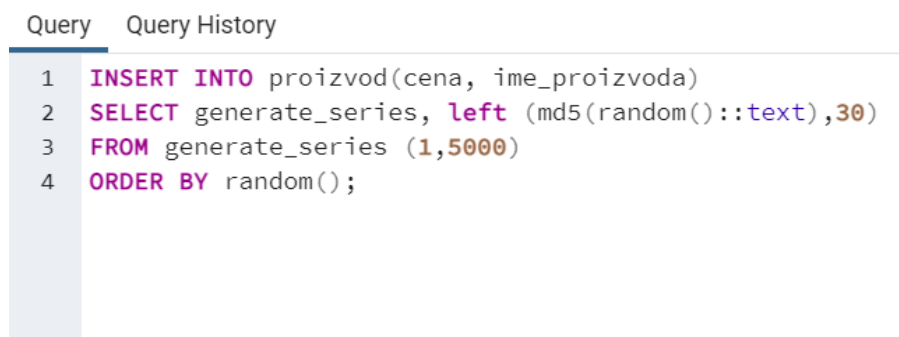
Tabela „proizvod“ je tabela u kojoj su pored id-a svakog proizvoda, sačuvane informacije o imenu proizvoda i ceni proizvoda.

Vrednosti ovih kolona su formirane tako što je za generisanje vrednosti cene i imena proizvoda, iskorišćena PostgreSQL funkcija `generate_series`, kojom je kreirano random 5000 brojnih vrednosti i nasumičnih string vrednosti, koje su potom heširane MD5 kriptografskim postupkom (ovo je urađeno kako bi se kroz seminarski rad pored indeksiranja, pokazale još neke mogućnosti PostgreSQL baze podataka).



```
Query  Query History
1  CREATE TABLE proizvod (
2  proizvod_id SERIAL PRIMARY KEY,
3  ime_proizvoda VARCHAR(30),
4  cena INTEGER);
```

Slika 12 – Kreiranje tabele za testiranje B-tree indeksa u PostgreSQL bazi podataka



```
Query  Query History
1  INSERT INTO proizvod(cena, ime_proizvoda)
2  SELECT generate_series, left(md5(random()::text),30)
3  FROM generate_series (1,5000)
4  ORDER BY random();
```

Slika 13- Popunjavanje tabele „proizvod“ korišćenjem `generate_series` funkcije za generisanje random vrednosti

Važno je uočiti, da su random generisane vrednosti, odmah inicijalno sortirane u random poretku, što ukazuje na to da će podaci biti grupisani u nasumičnom redosledu. Sekvencijalna pretraga ovakvih podataka zahteva nešto više vremena za izvršenje. Rezultat `SELECT` naredbe dat je na slici ispod (napomena: random je raspored cena vezanih za proizvode, nisu sortirani prema vrednosti cene). Sortiranje prema vrednosti cene, može se postići kreiranjem B-tree indeksne strukture, koja po default-u sortira indekse u rastućem redosledu (moguće je izabrati i opadajući redosled navođenjem `desc` klasifikatora pri kreiranju). Nakon kreiranja indeksa, moguće je izvršiti i klasterizaciju po njemu, čime se ne postiže samo sortiranje vrednosti prema polju nad kojim je indeks kreiran, već i fizičko uređenje redosleda sadržaja tabele. Nakon što se izvrši kreiranje indeksa, zatim klasterizacija, običnom `SELECT` naredbom mogu se dobiti podaci koji su uređeni prema indeksnoj strukturi polja nad kojim su kreirani, a takođe su i fizički uređeni na isti način u tabeli.

Na narednim slikama, prikazani su rezultati izvršenja `select` naredbe nad random uređenim podacima, kreiranja indeksa i klastera, kao i rezultat izvršenja `select` naredbe nad indeksno uređenim podacima.

Query	Query History
1	<b>CREATE INDEX</b> idx_cena <b>on</b> proizvod (cena);

Query	Query History
1	<b>CLUSTER</b> proizvod <b>USING</b> idx_cena ;

Slika 14 – Prikaz kreiranja B-tree indeksa nad kolonom cena i klasterizacija tabele po kreiranom Indeksu

	proizvod_id [PK] integer	ime_proizvoda character varying (30)	cena integer
1	2	07b97342ee4f54b83...	1474
2	3	d752ecb8157e34bec...	1337
3	4	0bb99d7733ff862fff...	3453
4	5	5cb68f210e34a827a9...	2801
5	6	4d91940c5f01dec9d5...	965
6	7	4d053d83da30a67e4...	813
7	8	b55f3f4b615a6c9ff0b...	907
8	9	25a35a965ff63e6757...	3309
9	10	701596201d2132e7d...	3036
10	11	893489e8fbd5afbd84...	507
11	12	b4a589876ad21d7d...	1105
12	13	880b6e1c0818c86ce...	146
13	14	47e4ebd21f1fe837af...	1548
14	15	e36a7ff8dab84d6e7b...	2172
15	16	87aae1968dcd4a8ee...	2454
16	17	31b3d35064fc762ef9...	365
17	18	e52f358aa2238753a...	3903
18	19	4a4fa30446e507b3b...	3414
Total rows: 5000 of 5000		Query complete 00:00:00.316	

	proizvod_id [PK] integer	ime_proizvoda character varying (30)	cena integer
1	1861	1e395921c00c6dade...	1
2	538	b42d17ca91e62cc51...	2
3	2537	46e68c13a95c9033c...	3
4	1153	5467d582d29cff8938...	4
5	3117	7ac3b61dd5d6709fc5...	5
6	1543	34b0276b2ee5dc28b...	6
7	870	a9f445c8d18fb55b03...	7
8	1683	dda9fe1d85cbc07b0c...	8
9	2427	437b308805269e8de...	9
10	4669	b72f8418503e38166...	10
11	2734	f630d1d8554a8b9f7d...	11
12	1349	c649d2942fe553e2c7...	12
13	4294	573961896e2238608...	13
14	1351	9a12b9366aed41a3b...	14
15	867	c52643baac054d50df...	15
16	1439	49e98a21489fb097c7...	16
17	1324	f3c2459e3339d42f06...	17
18	3144	8a876ce2c44e8f5f3c...	18
Total rows: 1000 of 5000		Query complete 00:00:00.106	

Slika 15- Prikaz rezultata izvršenja select naredbe nad random uređenim poljima tabele i nad indeksiranim poljima

Na slici iznad, jasno se može uočiti, da je i čak i izvršenje jednostavne naredbe, kakva je SELECT nardeba, čak tri puta kraće kada se koristi indeks.

Međutim, prava svrha uvođenje B-tree indeksa, vidi se prilikom izvršenja i analize upita koji pretražuje cene proizvoda u opsegu od 2500 do 5000.

Query	Query History
1	<b>EXPLAIN</b> (ANALYZE, BUFFERS)
2	<b>SELECT</b> cena
3	<b>FROM</b> proizvod
4	<b>WHERE</b> cena <b>BETWEEN</b> 2500 <b>and</b> 5000;

Data Output	Messages	Notifications
<div> <div>QUERY PLAN</div> <div>text</div> </div>		
1	Seq Scan on proizvod (cost=0.00..117.00 rows=2500 width=4) (actual time=0.009..0.504 rows=2501 loops=1)	
2	Filter: ((cena >= 2500) AND (cena <= 5000))	
3	Rows Removed by Filter: 2499	
4	Buffers: shared hit=42	
5	Planning:	
6	Buffers: shared hit=6	
7	Planning Time: 0.115 ms	
8	Execution Time: 0.586 ms	

Slika 16 – Prikaz rezultata izvršenja upita i njegova analiza bez korišćenja indeksa

Query		Query History
1	EXPLAIN (ANALYZE, BUFFERS)	
2	SELECT cena	
3	FROM proizvod	
4	WHERE cena BETWEEN 2500 and 5000	

Data Output		Messages	Notifications
<div> <div></div> <div></div> <div></div> <div></div> <div></div> <div></div> <div></div> <div></div> </div>			
<div> <div>QUERY PLAN</div> <div>text</div> <div></div> </div>			
1	Index Only Scan using idx_cena on proizvod (cost=0.28..86.30 rows=2501 width=4) (actual time=0.032..0....		
2	Index Cond: ((cena >= 2500) AND (cena <= 5000))		
3	Heap Fetches: 0		
4	Buffers: shared hit=3 read=7		
5	Planning:		
6	Buffers: shared hit=17 read=3 dirtied=2		
7	Planning Time: 1.985 ms		
8	Execution Time: 0.481 ms		

Slika 17- Prikaz rezultata izvršenja upita i njegova analiza sa korišćenjem indeksne strukture

Prethodne dve slike jasno pokazuju vrednost korišćenja B-tree indeksa. Kako bi se pronašle vrednosti u zadatom opsegu, sekvencijalnom pretragom, potrošeno je priližno 5 puta više vremena u odnosu na planirano vreme izvršenja, dok je pretraga po B-tree indeksu, izvršila pretragu vrednosti u zadatom opsegu, 4 puta brže u odnosu na planirano vreme izvršenja upita i 2 puta brže u odnosu na sekvencijalnu pretragu. Uvid u query plan izvršenja upita korišćenjem sekvencijalne i indeksirane pretrage, omogućen je korišćenjem konstrukcija EXPLAIN, ANALYZE i BUFFERS. EXPLAIN konstrukcija se kod PostgreSQL baze podataka, koristi za pregled plana izvršenja upita ili bilo koje sql naredbe i izuzetno je korisna komanda, pogotovu u procesu razmatranja performansi kod fizičkog projektovanja. Ključna reč ANALYZE govori PostgreSQL-u da izvrši naredbu koja sledi iza nje, ali tako da uključi broj deljenih bafera, koji su pretraženi kako bi se dobili podaci. PostgreSQL odvaja veliki deo memorije za skladištenje podatka i indeksiranih stranica, ali se one moraju prikupiti sa diska u skup deljene memorije, pre nego se formira rezultat upita. Zato je ključna reč BUFFERS u ovom primeru iskorišćena kako bi se dobio uvid u to koliko je deljenih bafera (stranica sa podacima u memoriji) uzeto u razmatranje za formiranje datog rezultata. Što je manje deljenih bafera potrebno pretražiti, to je upit brži. Sa slika se jasno vidi, da je kod sekvencijalne pretrage, broj deljenih bafera 42, a kod indeksirane 3, što je daleko optimalnije. Primer izvršenja još jednog upita u kome se koristi operator jednakosti, i u kome je takođe broj deljenih bafera 3, dat je na slici ispod:

Query		Query History
1	EXPLAIN (ANALYZE, BUFFERS)	
2	SELECT cena	
3	FROM proizvod	
4	WHERE cena = 2500;	

Data Output		Messages	Notifications
<div> <div></div> <div></div> <div></div> <div></div> <div></div> <div></div> <div></div> <div></div> </div>			
<div> <div>QUERY PLAN</div> <div>text</div> <div></div> </div>			
1	Index Only Scan using idx_cena on proizvod (cost=0.28..8.30 rows=1 width=4) (actual time=0.014..0.015 rows=1 loop...		
2	Index Cond: (cena = 2500)		
3	Heap Fetches: 1		
4	Buffers: shared hit=3		
5	Planning Time: 0.108 ms		
6	Execution Time: 0.027 ms		

Slika 18 – Primer izvršenja upita sa operatorom jednakosti i njegova analiza

## 2. Hash indeks i primer

Hash indeks, kod PostgreSQL baze podataka, implementira varijaciju strukture podataka heš tabele, gde konkretna heš funkcija za indeks uzima vrednost polja (kolonu, može biti čak i ključ iskorišćen za kreiranje indeksa), kako bi formirala heš ključ koji predstavlja 4-byte-nu označenu integer vrednost (32-bita) i skladišti heširane podatke, zajedno sa pokazivačima na postojeće zapise ( redove u tabeli) [18].

Pre PostgreSQL verzije 10, heš indeksi se nisu dobro pokazivali u radu pod velikim opterećenjem, a takođe se nisu pridržavali PostgreSQL Write-Ahead protokola evidentiranja, što znači da su vrlo često bili oštećeni, kada bi došlo do pada sistema i oporavka istog. Međutim, sa novijim verzijama, ti problemi su prevaziđeni, pa heš indeksi mogu i nadmašiti B-tree indekse [18]. Heš indeksi su vrlo korisni kada se vrše klasični upiti poređenja, jer od relacionih operatora podržavaju rad samo sa operatorom jednakosti. Najbolje su optimizovani kada se radi o SELECT i UPDATE upitima, pogotovu u odnosu na B-tree indekse, koji bi za skeniranje jednakosti vrednosti datih upitom, prolazili kroz strukturu stabla dok ne pronađu listu stranica. Ovakav pristup B-tree indeksa, predstavlja skupu operaciju. Heš indeks, za razliku od B-tree indeksa, ima direktan pristup stranicama segmenata ( liste stranica kod B-tree indeksa), što utiče na smanjenje vremena pristupa indeksu kod tabela sa velikom količinom podataka. Međutim, u nekim situacijama korišćenje heš indeksa može biti daleko neefikasno. Takve su situacije kod WHERE klauzola, jer ukoliko se u njima ne koristi operator jednakosti, neće biti moguće iskoristiti benefite ove indeksne strukture. Takođe, heš indeksi, su dizajnirani da dobro podnose neujednačenu distribuciju heš vrednosti, koja podrazumeva kreiranje dodatnih stranica segmenata, kada se ravnomerno napunjenje stranice prepune, pri čemu se novokreirane stranice vezuju za originalnu stranicu segmenta. Međutim, prilikom skeniranja heš indeksa pri obradi upita, skeniraju se i sve te dodatne stranice, čime se povećava broj potrebnih pristupa bloku podataka, što heš indeksiranje čini manje pogodnim u odnosu na B-tree indeksiranje. Prilikom fizičkog projektovanja, treba obratiti pažnju na još jednu stvar kada su u pitanju heš indeksi, a to je razmatranje INSERT naredbe. Ukoliko se ona jako često koristi, odnosno, ukoliko tabele konstantno povećavaju svoje skladištene kapacitete, mapiranje heš indeksa u obliku ključ-vrednost, takođe će se postepeno proširivati, pri čemu će kreiranje novog segmenta uticati na podelu postojećeg segmenta, kako bi se torke rasporedile između segmenata. Ovaj scenario, utiče na povećanje vremena izvršenja korisničkih upita, ali ne utiče na veličinu samog indeksa, s obzirom da heš indeksi ne čuvaju stvarnu vrednost ključa, već njegovu heširanu vrednost. Iz tog ugla, heš indeksu u odnosu na B-tree indekse nemaju ograničenja u pogledu veličine i veličina indeksa ne utiče na selektivnu sposobnost heš ključa.

Prednosti korišćenja heš indeksa kod PostgreSQL baze podataka, prikazane su narednim primerom.

Kreirana je tabela „Korisnik“ koja sadrži kolone u kojima se nalaze podaci o imenu, prezimenu i mejlu nekog korisnika. Tabela je popunjena na taj način što je iskorišćena random() funkcija koja se nalazi u skupu funkcija PostgreSQL baze podataka. Pored nje je korišćena i MD5 hash funkcija koja je primenjena na nasumično generisanim tekstualnim vrednostima, a funkcija left() uzima prvih 10 karaktera iz dobijenog MD5 heša, kako bi se kreirao podatak duže 10 karaktera. Sve kolone u tabeli su kreirane na opisani način, s tim što je kod kolone „email“ podatak formiran kombinovanjem MD5 heš vrednosti podataka iz kolona „ime“ i „prezime“, koje su međusobno razdvojene znakom „\*“ i upotpunjene stringom „@example.com“, kako bi se dobila klasična struktura mejla. Kao i u prethodnom, primeru i ovde je iskorišćena funkcija

PostgreSQL pgAdmin alata, generate\_series(), kojom je kreirano 10000 redova u tabeli „Korisnik“.

Query	Query History
1	CREATE TABLE korisnik (
2	korisnik_id SERIAL PRIMARY KEY,
3	ime VARCHAR(50),
4	prezime VARCHAR(50),
5	email VARCHAR(50)
6	);
7	
8	INSERT INTO korisnik (ime, prezime, email)
9	SELECT
10	concat(left(md5(random()::text), 10)) as ime,
11	left(md5(random()::text), 10) as prezime,
12	concat(left(md5(random()::text), 10), '.', left(md5(random()::text), 10), '@example.com') as email
13	FROM generate_series(1, 10000)
14	ORDER BY random();

	korisnik_id [PK] integer	ime character varying (50)	prezime character varying (50)	email character varying (50)
1	1	51bb1c4ccb	51bb1c4ccb	51bb1c4ccb.51bb1c4ccb@example.com
2	2	4ee5d8f817	4ee5d8f817	4ee5d8f817.4ee5d8f817@example.com
3	3	138894421d	138894421d	138894421d.138894421d@example.com
4	4	739a7c2a90	739a7c2a90	739a7c2a90.739a7c2a90@example.com
5	5	cce7d4da79	cce7d4da79	cce7d4da79.cce7d4da79@example.com
6	6	5bc8b1e4e6	5bc8b1e4e6	5bc8b1e4e6.5bc8b1e4e6@example.com
7	7	6e0185f387	6e0185f387	6e0185f387.6e0185f387@example.com
8	8	ee3701f76c	ee3701f76c	ee3701f76c.ee3701f76c@example.com
9	9	03ee8f5919	03ee8f5919	03ee8f5919.03ee8f5919@example.com
10	10	7fd863d42f	7fd863d42f	7fd863d42f.7fd863d42f@example.com
11	11	3c80ec0310	3c80ec0310	3c80ec0310.3c80ec0310@example.com
12	12	95f39f9cd6	95f39f9cd6	95f39f9cd6.95f39f9cd6@example.com
13	13	47b0dd1f9e	47b0dd1f9e	47b0dd1f9e.47b0dd1f9e@example.com
14	14	c7c0bca421	c7c0bca421	c7c0bca421.c7c0bca421@example.com
15	15	1c5abd25ca	1c5abd25ca	1c5abd25ca.1c5abd25ca@example.com
16	16	d70b359eaa	d70b359eaa	d70b359eaa.d70b359eaa@example.com
17	17	429e299aed	429e299aed	429e299aed.429e299aed@example.com
18	18	f2ad37ad3a	f2ad37ad3a	f2ad37ad3a.f2ad37ad3a@example.com

Slika 19 – Kreiranje tabele „Korisnik“ i prikaz podataka koji su kreirane i zapamćeni u tabeli

Nakon što je kreirana tabela, izvršen je upit u kome se zahtevaju informacije o korisniku sa datim imenom. Na narednoj slici dat je prikaz izvršenja upita i rezultata analize sekvencijalnog traženja podataka.

Query	Query History
1	EXPLAIN (ANALYZE)
2	SELECT *
3	FROM korisnik
4	WHERE email = 'abb63accf3.abb63accf3@example.com';

Data Output	Messages	Notifications
<div>QUERY PLAN</div> <div>text</div>		
1	Seq Scan on korisnik (cost=0.00..239.00 rows=1 width=60) (actual time=0.030..1.368 rows=1 loops=1)	
2	Filter: ((email)::text = 'abb63accf3.abb63accf3@example.com')::text	
3	Rows Removed by Filter: 9999	
4	Planning Time: 1.118 ms	
5	Execution Time: 1.383 ms	

Slika 20- Izvršenje upita bez indeksa



Naredbom datoj na slici 21, kreiran je heš indeks nad kolnom email i potom je ponovljen isti upit kao ranije. Nakon izvršenja upita, gde je za pretragu korišćen heš indeks, može se videti da je vreme traženja skraćeno, a samim tim performanse su povećane, što je važno za koncepte fizičkog projektovanja.

Query
Query History

```

1 CREATE INDEX korisnik_email_idx ON korisnik USING HASH(email);

1 EXPLAIN (ANALYZE)
2 SELECT *
3 FROM korisnik
4 WHERE email = 'abb63accf3.abb63accf3@example.com';

```

Data Output
Messages
Notifications

+

📄

▼

📋

🗑️

🗄️

⬇️

📈

QUERY PLAN

text

🔒

1	Index Scan using korisnik_email_idx on korisnik (cost=0.00..8.02 rows=1 width=60) (actual time=0.028..0.028 rows=1 loops=1)
2	Index Cond: ((email)::text = 'abb63accf3.abb63accf3@example.com'::text)
3	Planning Time: 1.482 ms
4	Execution Time: 0.047 ms

Slika 21- Prikaz kreiranja heš indeksa nad poljem „email“ i analiza rezultata izvršenja upita

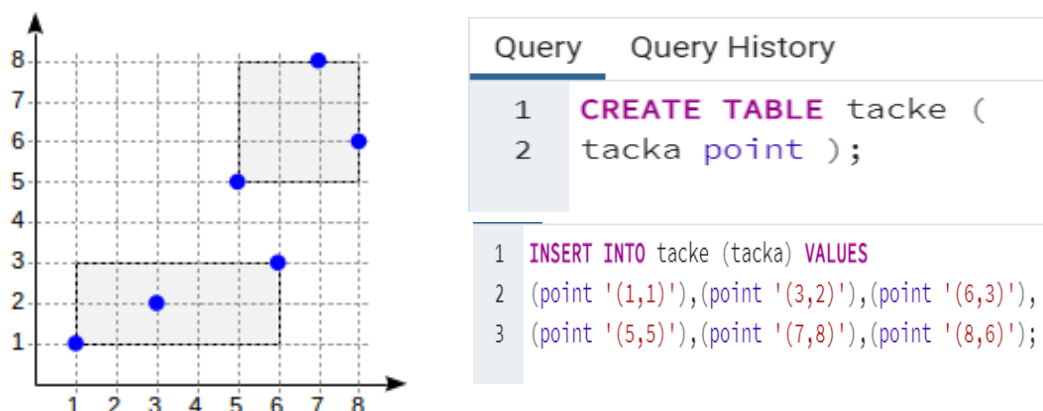
### 3. GiST indeks i primer

GiST je predstavlja tzv. „generalizovano stablo pretrage“, što znači da je kao i B-stablo, izbalansirano stablo, ali za razliku od B-stabla, koje izuzetno efikasno pruža podršku za rad sa upitima koji sadrže relacione i operatore poređenja, GiST ima mogućnost definisanja pravila za distribuciju podataka proizvoljnog tipa (geopodaci, txt dokumenti, slike..), što B-stablo ne omogućava [19]. Korišćenjem GiST-a, moguće je iskoristiti pomenute operatore, koji se nalaze u sklopu funkcija operatora klasa, koje PostgreSQL baza podataka podržava za GiST.

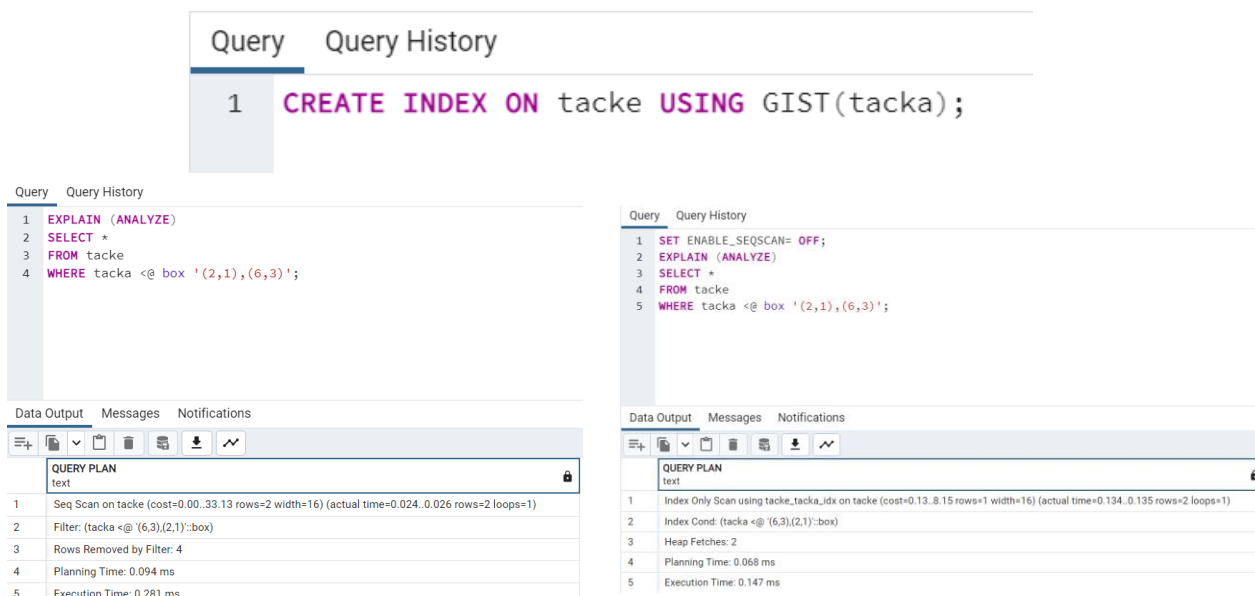
GiST indeks je izuzetno proširiv, pa se može koristiti kod sistema sa velikim domenom i može se posmatrati kao okvir za izgradnju novih metoda pristupa podacima.

Kako je po strukturi, GiST generalizovano i visoko izbalansirano stablo pretrage, sastoji se od čvorova, koji su uskladišteni kao stranice i u njima se nalaze indeksni redovi. Svaki od redova koji predstavljaju lisni čvor, sadrže neki logički izraz koji se naziva prediktom i referencu na red tabele (TID), pri čemu indeksirani podaci moraju ispuniti taj predikat. Unutrašnji redovi koji se indeksiraju, imaju istu strukturu i takođe moraju ispuniti taj predikat i po tome se struktura GiST-a usložnjava u odnosu na strukturu B-stabla. Pretraga podataka indeksiranih ovom metodom, kreće od korenog čvora i koristi funkciju konzistentnosti, koja vrši eliminaciju nepotrebnih čvorova indeksne strukture i vraća podatak ukoliko je on list nekog čvora. Pretraga se vrši po dubini, a umetanje nove vrednosti u indeksnu strukturu, postiže se što manjim proširivanjem predikata roditeljskih redova, ali kod brisanja ovo nije slučaj, tj roditeljski redovi se ne smanjuju, već se dele na dva dela ili se indeks kreira od nule (full vacuum). Iz tog razloga, GiST indeks nije najbolje rešenje za podatke koji su često podložni promenama.

GiST indeks ima dobru primenu kada je potrebno uskladištiti geometrijske entitete, podatke sa velikim brojem intervala i ograničenja ili podatke koji zahtevaju full-search pretragu. GiST indeks kombinuje strukture B-stabla i R-stabla (koristi se da podeli velike regione u manje celine i pretraži ih), što će biti prikazano u nastavku.



Slika 22 – Prikaz kreiranja tabele koja modeluje predstavljene tačke koordinatnog sistema. Svrha korišćenja GiST indeksa u ovom primeru je da efikasno pronađe sve sve označene tačke u prvom pravougaoniku (između tački (2,1) i (6,3)). Sekvencijalna pretraga bi u ovom slučaju dala takođe rezultate, ali na račun usporjenja izvršenja upita. GiST kreće od korena stabla i prateći strukturu pokazivača na interne stranice, traži rezultat odbacujući sve nerelevantne rezultate.



Slika 23- Kreiranje GiST indeksa nad kolonom tacka i poređenje rezultata izvršenja upita. Na slici se može uočiti da se u upitu koristi operator @ koji pripada familiji points\_ops operatora PostgreSQL baze (sadrži operatore tipa &&, <<, >>, itd) i znači da se element sadrži u pravougaoniku označenom datim tačkama. Samim tim, dolazi se do priče o operatorima i ograničenjima, koja se vrlo često koriste kod ove vrste indeksa, pogotovu kada se koriste nad geopodacima. Operatori koji su prikazani pripadaju grupi tzv. operatora traženja, jer oni ukazuju na način pretrage u samoj strukturi indeksa. Važno je pomenuti i operatore uređenja, koji se koriste za specifikaciju redosleda sortiranja u klauzolama ORDER BY i limit. Vrlo često se sa

njima koristi operator  $< - >$  koji određuje distancu jednog operatora u odnosu na drugi operator. Primer ovih operatora, dat je kroz određivanje razdaljine između zadatih lokacija u tabeli „Lokacija“, pri čemu su zadate koordinate lokacija i traže se sve one koje su na udaljenosti unutar 10 metara od zadate lokacije. Dati su rezultati izvršenja upita sekvencijalno i korišćenjem strukture GiST indeksa.

Query Query History

```

1 CREATE TABLE lokacija (
2   lokacija_id SERIAL PRIMARY KEY,
3   ime_lokacije VARCHAR(50),
4   lokacija_geom GEOMETRY (POINT, 4326)
5 );
6
7 INSERT INTO lokacija (ime_lokacije, lokacija_geom)
8 SELECT
9   'Naziv lokacije ' || generate_series AS ime_lokacije,
10  ST_SetSRID(ST_MakePoint(
11    random() * 10 + 44,
12    random() * 10 + 20
13  ), 4326) AS lokacija_geom
14 FROM generate_series(1, 1000);

```

Slika 24- Tabela lokacija

Query Query History

```

1 CREATE INDEX lokacija_geom_gist_idx ON lokacija USING GIST (lokacija_geom);

```

EXPLAIN (ANALYZE)

```

SELECT ime_lokacije
FROM lokacija
WHERE ST_Distance(lokacija_geom, ST_SetSRID(ST_MakePoint(40.789, 20.451), 4326)) <= 10;

```

EXPLAIN (ANALYZE)

```

SELECT ime_lokacije
FROM lokacija
WHERE ST_Distance(lokacija_geom, ST_SetSRID(ST_MakePoint(40.789, 20.451), 4326)) <= 10
ORDER BY lokacija_geom <-> ST_SetSRID(ST_MakePoint(40.789, 20.451), 4326)
LIMIT 10;

```

Output Messages Notifications

QUERY PLAN

Seq Scan on lokacija (cost=0.00..25023.50 rows=333 width=18) (actual time=0.026..0.469 rows=507 loops=1)

Filter: (st\_distance(lokacija\_geom, '0101000020E6100000A245B6F3FD644440FA7E6ABC74733440::geometry') <= '10':double precision)

Rows Removed by Filter: 493

Planning Time: 0.116 ms

Execution Time: 0.503 ms

Output Messages Notifications

QUERY PLAN

Limit (cost=0.14..766.04 rows=10 width=26) (actual time=0.130..0.164 rows=10 loops=1)

-> Index Scan using lokacija\_geom\_gist\_idx on lokacija (cost=0.14..25504.39 rows=333 width=26) (actual time=0.129..0.162 rows=10 loops=1)

Order By: (lokacija\_geom <-> '0101000020E6100000A245B6F3FD644440FA7E6ABC74733440::geometry')

Filter: (st\_distance(lokacija\_geom, '0101000020E6100000A245B6F3FD644440FA7E6ABC74733440::geometry') <= '10':double precision)

Planning Time: 0.092 ms

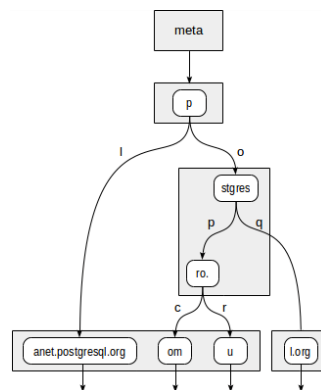
Execution Time: 0.211 ms

Slika 25- Kreiranje GiST indeksa nad kolonom lokacija\_geom i poređenje rezultata izvršenja upita

#### 4.SP-GiST indeks i primer

SP-GiST je kao i GiST generalizovano stablo pretrage, koje pruža okvire za izgradnju različitih metoda pristupa. SP-GiST je skraćenica za space-partitioned GiST, što znači da podržava pretragu particionisanih stabala, koja modeluju dvodimenzioni prostor na kakav smo navikli. SP-GiST je pogodan za strukture podataka, gde se prostor može rekurzivno podeliti na oblasti koje se međusobno ne ukrštaju, kao što su k-d stabla, kvad-stabla i radix stabla. Ove strukture podataka, dele prostor koji se indeksira i pretražuje na particije koje ne moraju biti jednake veličine, ali je sistem pretrage dosta brz [19]. SP-GiST indeksne strukture, inicijalno su razvijane za skladištenje u glavnoj memoriji, kao skup dinamički dodeljenih čvorova sa pokazivačima, kako bi se izbeglo višestruko pristupanje blokovima podataka na disku. Međutim,

čvorovi indeksa se pakuju u stranice, koje se čuvaju na disku i iako je struktura nekog mapiranja takva, da je potrebno proći kroz veliki broj stranica čvorova i pokazivača, te više puta pristupiti blokovima stranica na disku, SP-GiST se trudi da optimizuje pristup stranicama na disku, kako bi se dobili rezultati pretrage, a ujedno i zadržale dobre performanse. Za demonstraciju karakteristika SP-GiST-a, iskorišćena je tabela nazvana „postgreSajtovi“ u koju je upisano 5000 URL-a sajtova sa domenima: .ru, .com i .org. URL vrednosti su upisane u vidu stringova nad kojima se kreira SP-GiST indeks, tako da predstavi implementaciju redix stabla. Ideja redix stabla je da uskladišteni string u lisnom čvoru, ne skladišti iz jednog dela, već se jedinstveni zapis dobija prolaskom i spajanjem vrednosti uskladištenih u čvorovima povezanim sa lisnim čvorom ( to su interni elementi indeksa i nalaze se u stablu iznad lisnog čvora, a put ide od korena ka lisnom čvoru). Zamislamo sada, da je potrebno u kreiranoj tabeli „postgreSajtovi“, pronaći sve url-ove koji u sebi sadrže postgresp kao deo url-a i za domen imaju .ru. To znači da je potrebno pretražiti tabelu i izdvojiti sve url-ove, koji imaju veću ili jednaku vrednost od postgresp, ali nikako manju i pripadaju domenu .ru. Kao logično rešenje, s obzirom da su neophodni operatori poređenja klase operatora text\_ops PostgreSQL-a, nameće se korišćenje B-stabla. Međutim klase ovih operatora u PostgreSQL radije manipulišu bajtovima nego samim karakteristikama. Iz tog razloga, nad kolonom url\_sajta koja pamti informacije o url-ovima, kreiran je SP-GiST indeks koji vrši pretragu po zadatom kriterijumu, i to struktura redix stabla, data na narednoj slici.



Slika 26 – Prikaz strukture SP-GiST indeksa za primer tabele „postgreSajtovi“

Filter koji je zadat upitom koji je ranije pomenut, funkcijom konzistencije kreće od korena ( u ovom slučaju koren je p) i omogućava da indeks prati pokazivače na čvorove dok ne dođe do lisnih čvorova u kojima je rezultat zapisa, nakon čega filter bira odgovarajući rezultat. I pokazivači se obeležavaju simboličnim nazivima i oni se dodatno skladište na disku. Rezultati sekvencijalnog i indeksiranog izvršenja upita, dati su na narednim slikama.

```

1 CREATE TABLE postgreSajtovi (
2   url_sajta text);
3
4 INSERT INTO postgreSajtovi (url_sajta)
5 SELECT
6   'http://www.' ||
7   CASE
8     WHEN generate_series % 4 = 0 THEN 'postgrespro.ru'
9     WHEN generate_series % 4 = 1 THEN 'postgrespro.com'
10    WHEN generate_series % 4 = 2 THEN 'postgresql.org'
11    ELSE 'planet.postgresql.org'
12  END
13 FROM generate_series(1,5000);

```

Slika 27 – Prikaz kreiranja i popunjavanja tabele „postgreSajtovi“

QueryQuery History

1CREATE INDEX url\_sajta\_spgist\_idx ON postgresajtovi USING SPGIST (url\_sajta);

1EXPLAIN (ANALYZE)

2SELECT \*

3FROM postgresajtovi

4WHERE url\_sajta LIKE 'http://www.postgrespřru';

1EXPLAIN (ANALYZE)

2SELECT \*

3FROM postgresajtovi

4WHERE url\_sajta LIKE 'http://www.postgrespřru';

Data OutputMessagesNotifications

QUERY PLAN

text

1Seq Scan on postgresajtovi (cost=0.00..101.50 rows=1250 width=28) (actual time=0.010..0.584 rows=1250 loops=1)

2Filter: (url\_sajta ~\* http://www.postgrespřru):text)

3Rows Removed by Filter: 3750

4Planning Time: 0.039 ms

5Execution Time: 0.618 ms

Data OutputMessagesNotifications

QUERY PLAN

text

1Index Only Scan using url\_sajta\_spgist\_idx on postgresajtovi (cost=0.15..98.15 rows=1250 width=28) (actual time=0.232..0.526 rows=1250 loops=1)

2Index Cond: (url\_sajta ~\* http://www.postgrespřru):text)

3Filter: (url\_sajta ~\* http://www.postgrespřru):text)

4Rows Removed by Filter: 1250

5Heap Fetches: 0

6Planning Time: 0.081 ms

7Execution Time: 0.563 ms

Slika 28- Prikaz kreiranja SP-GiST indeksa nad kolonom url\_sajta i sekvencijalno(levo) i indeksirano(desno) izvršenje upita i rezultati poređenja

Sa stanovišta performansi, možda je jednostavnije koristiti B-stablo za indeksiranu pretragu, ali sa stanovišta optimalnosti traženja u smislu provere različitih nastavaka koji slede iza postgresp korena reči, lakše je koristiti SP-GiST indeks koji vrši podelu jedinstvenog stringa na manje celine i njih pretražuje kao elemente stabla. SP-GiSt, za razliku od GiST-a ne može podržati jedinstvena ograničenja ili sortiranje po rezultatima pretrage, kao ni mehanizam kreiranja indeksa na većem broju kolona, ali je zato moguća klasterizacija [19], pa i ovo treba imati u vidu prilikom fizičkog projektovanja.

## 5.GIN indeks i primer

GIN (Generalizovani Invertovani Indeks) je jedna od vrsti indeksa kod PostgreSQL baze podataka i koristi se za brzu pretragu polja sa vrednostima koje nisu atomične, već predstavljaju kolekciju vrednosti, poput nizova ili JSON objekata. GIN indeks funkcioniše tako što pravi listu svih vrednosti koje se nalaze u kolekciji, sortira ih i zatim gradi stablo pretrage koje omogućava brzo pretraživanje vrednosti u kolekciji. Takođe, pomenuta lista vrednosti, sadrži i ključeve koji su za njih vezani i važno je napomenuti da se indeksiraju vrednosti tih kolekcija, ali GIN indeks uvek traži ključ koji je vezan za zapamćenu vrednost, a ne traži samu tu vrednost [19]. Još jedna od karakteristika GIN indeksa je što on vrednost jednog ključa čuva samo jednom, pa je samim tim ovaj indeks koristan u slučajevima kada se u nekom upitu, više puta zahteva ta vrednost ključa. Struktura GIN indeksa je takva da zapravo sadrži B-stablo, ali nadređeni skup referenci na redove tabele koji sadrže složene vrednosti imaju veze sa svakim od elemenata, dok je za pronalaženje podataka uređenost nevažna. Sačuvani elementi se nikada ne brišu iz GIN indeksa, što značajno pojednostavljuje algortime za rad sa više procesa nad istim indeksom.

U odnosu na druge vrste indeksa kod PostgreSQL baze podataka, GIN indeks može imati nešto sporije performanse, pogotovu kada se koristi za vrlo disperzivne vrednosti u odnosu na ukupan skup vrednosti. Takođe i izgradnja GIN indeksa zahteva značajnu količinu memorije i resursa, pa treba voditi računa kada se projektuju sistemi sa velikim kolekcijama podataka.

Primena GIN indeksa je u seminarskom radu, prikazana kreiranjem tabele „dokument“ u kojoj se nalazi kolekcija tekstualnih dokumenata, gde svakom od dokumenata iz kolekcije odgovara element iz kolone dokument\_tsv, koja ja po tipu TSVECTOR. Ovakvo mapiranje dokumenata na TSVECTOR tip je izvršeno, jer je nad kolonom dokument\_tsv kreiran GIN indeks, kako bi se omogućilo brzo pretraživanje koje obezbeđuje PostgreSQL baza. Naime, ona

tekstualni dokument ovim mapiranjem pretvara u TSVECTOR, i zahvaljujući tome što je indeks kreiran nad ovim tipom podatka, može da pretraži tražene delove dokumenta u bilo kom redosledu. Kombinacija TSVECTOR tipa i GIN indeksa u primeru pretrage tabele „dokument“ radi nalaženja svih dokumenata u kojima se javljaju lekseme „many“ and „slitter“, dala je najbolje rezultate u pogledu efikasnosti i fleksibilnosti. Rezultati izvršenja upita koji traži pomenute lekseme i poređenja sekvencijalnog i indeksiranog traženja, dati su na slikama u nastavku.

```
1 CREATE TABLE dokumenti (
2 dokument_id SERIAL PRIMARY KEY,
3 dokument TEXT,
4 dokument_tsv TSVECTOR
5 );
```

```
1 INSERT INTO dokumenti (dokument) VALUES
2 ('Can a sheet slitter slit sheets?'),
3 ('How many sheets could a sheet slitter slit?'),
4 ('I slit a sheet, a sheet I slit.'),
5 ('Upon a slitted sheet I sit.'),
6 ('Whoever slit the sheets is a good sheet slitter.'),
7 ('I am a sheet slitter.'),
8 ('I slit sheets.'),
9 ('I am the sleekest sheet slitter that ever slit sheets.'),
10 ('She slits the sheet she sits on.');
```

```
1 UPDATE dokumenti SET dokument_tsv= to_tsvector(dokument);
```

Slika 29 – Primer kreiranja i popunjavanja tabele „dokument“ i mapiranje kolone document na kolonu dokument\_tsv

```
1 CREATE INDEX ON dokumenti USING GIN (dokument_tsv);
```

Slika 30 – Kreiranje GIN indeksa nad kolonom dokument\_tsv

```
EXPLAIN (ANALYZE)
SELECT dokument
FROM dokumenti
WHERE dokument_tsv @@ to_tsquery('many & slitter');
```

Output Messages Notifications

QUERY PLAN  
text

Seq Scan on dokumenti (cost=0.00..233.13 rows=1 width=32) (actual time=0.044..0.091 rows=1 loops=1)

Filter: (dokument\_tsv @@ to\_tsquery('many & slitter':text))

Rows Removed by Filter: 8

Planning Time: 5.878 ms

Execution Time: 0.104 ms

```
SET enable_seqscan=OFF;
EXPLAIN (ANALYZE)
SELECT dokument
FROM dokumenti
WHERE dokument_tsv @@ to_tsquery('many & slitter');
```

Output Messages Notifications

QUERY PLAN  
text

Bitmap Heap Scan on dokumenti (cost=12.25..16.51 rows=1 width=32) (actual time=0.037..0.038 rows=1 loops=1)

Recheck Cond: (dokument\_tsv @@ to\_tsquery('many & slitter':text))

Heap Blocks: exact=1

→ Bitmap Index Scan on dokumenti\_dokument\_tsv\_idx (cost=0.00..12.25 rows=1 width=0) (actual time=0.032..0.032 rows=1 loops=1)

Index Cond: (dokument\_tsv @@ to\_tsquery('many & slitter':text))

Planning Time: 0.086 ms

Execution Time: 0.059 ms

Slika 31- Sekvencijalno i indeksirano izvršenje upita za pronalaz leksema „many“ i „slitter“ u dokumentima i prikaz rezultata izvršenja upita

Sa slike se može videti da je izvršenje upita korišćenjem GIN indeksa 2 puta brže, nego izvršenje upita korišćenjem sekvencijalne pretrage.

GIN indeks se u slučaju gore prikazanog upita, vrlo efikasno može koristiti i prilikom parcijalnog pretraživanja, ukoliko je recimo potrebno pretražiti samo dokumente u kojima se polovično javlja leksema „slit“, jer daje dobre rezultate i kod takvih parcijalnih pretraga.



## 6. BRIN indeks i primer

BRIN ( Block Range Index) je indeksni mehanizam PostgreSQL baze podataka koji se koristi za pretraživanje tabela u kojima su podaci uskladišteni u fizički uzastopnim opsezima blokova (npr. vremenski serijalizovani podaci). Iz tog razloga, ovakav tip indeksa je najefikasniji za kolone čije su vrednosti u dobroj korelaciji sa fizičkim redosledom redova tabele[20].

Indeksiranje se vrši tako, što se ne indeksiraju sve pojedinačne vrednosti, već se indeks formira na temelju statističkih informacija o vrednostima u blokovima podataka. Ideja BRIN indeksa je da se njegovim korišćenjem, eliminišu definitivno neodgovarajući redovi, a ne da se odmah pronađu oni koji se podudaraju sa zadatim uslovom upita. Zato je ovaj indeks, koristan kada se radi sa velikom količinom podataka, jer ovime smanjuje broj onih podataka koje je potrebno indeksirati, pa se ne preporučuje za rad sa malim tabelama, jer pokazuje slabije performanse u poređenju sa ostalim vrstama indeksa.

BRIN indeks radi tako što čuva zbirne informacije o podacima u svakom opsegu blokova stranica ( podrazumeva se da su podaci tabele podeljeni na blokove stranica). Po pravilu, te informacije koje indeks čuva su minimalna i maksimalna vrednost, ali u praksi je drugačije. U praksi, kada se izvrši upit koji u sebi ima neki uslov za neku kolonu, ako tražene vrednosti ne ulaze u opseg blokova, ceo opseg se preskače, međutim, ukoliko se pronađe podudaranje sa makar jednom vrednošću, moraju se pregledati svi redovi u svim blokovima opsega, kako bi se pronašli odgovarajući. Zbog ovakvog načina rada, BRIN indeks mnogi nazivaju akceleratorom sekvencijalnog skeniranja ili alternativom „virtuelnog particionisanja“ [19].

Što se strukture indeksa tiče, prva od stranica u opsegu blokova, sadrži metapodatke. Nakon nje, na određenom fizičkom rastojanju, nalaze se stranice sa sažetim informacijama, a svaki indeksni red u tim stranicama, sadrži zbirne informacije o jednom opsegu. Između stranice metapodataka i stranica sažetih podataka, sadrže se stranice sa mapom obrnutih opsega tj. invertovanih opsega, pri čemu je to zapravo niz pokazivača na odgovarajuće redove indeksa. Primer korišćenja BRIN indeksa dat je u nastavku.

```
1 CREATE TABLE narudzbina (  
2   narudzbina_id SERIAL PRIMARY KEY,  
3   datum_narucivanja DATE,  
4   ime_narudzbine VARCHAR (50));  
5  
6 INSERT INTO narudzbina (datum_narucivanja, ime_narudzbine)  
7 SELECT  
8   date_trunc('day', current_date - random() * interval '365 days') AS datum_narucivanja,  
9   left(md5(random()::text), 10) AS ime_narudzbine  
10 FROM generate_series(1, 5000);
```

Slika 32- Kreiranje i popunjavanje tabele „narudzbina“

Za demonstraciju rada BRIN indeksa, kreirana je tabela „narudzbina“ koja sadrži informacije o broju narudžbinem imenu i datumu naručivanja iste. Datum naručivanja, kreiran je korišćenjem date\_trunc funkcije koja uzima trenutni datum i od njega oduzima radnom izgenerisanu vrednost koju množi sa intervalom od 365 dana, kako bi kreirala opseg datuma. S obzirom da se kreira opseg vrednosti datuma, BRIN indeks će biti kreiran nad kolonom datum\_naručivanja.

```
1 CREATE INDEX narudzbina_brin_idx ON narudzbina USING BRIN (datum_narucivanja);
```

Slika 33- Kreiranje BRIN indeksa nad kolonom datum\_naručivanja

Sada je potrebno izvršiti upit koji zahteva informacije o svim narudžbinama koje su izvršene u periodu od 12.11.2022. do 11.04.2023. godine. Ovako definisan upit, predstavlja opseg vrednosti i sekvencijalnim skeniranjem mora se proći kroz sve redove tabele kako bi se pronašli

odgovarajući elementi. Korišćenjem indeksa, zahvaljujući njegovoj strukturi u vidu opsega blokova koji sadrže pokazivače na redove tabele, svi oni elementi iz opsega koji ne odgovaraju, biće eliminisani, pa je pretraga brža. Rezultati izvršenja ova dva pristupa izvršenju upita, prikazani su na narednoj slici.

<pre> 1 EXPLAIN (ANALYZE) 2 SELECT * FROM narudzbina 3 WHERE datum_narucivanja BETWEEN '2022-11-12' AND '2023-04-12'; </pre>	<pre> 1 SET enable_seqscan = OFF; 2 EXPLAIN (ANALYZE) 3 SELECT * FROM narudzbina 4 WHERE datum_narucivanja BETWEEN '2022-11-12' AND '2023-04-12'; </pre>
<p>Data Output Messages Notifications</p> <p>QUERY PLAN text</p> <p>1 Seq Scan on narudzbina (cost=0.00..107.00 rows=2103 width=19) (actual time=0.029..0.472 rows=2097 loops=1)</p> <p>2 Filter: ((datum_narucivanja &gt;= '2022-11-12':date) AND (datum_narucivanja &lt;= '2023-04-12':date))</p> <p>3 Rows Removed by Filter: 2903</p> <p>4 Planning Time: 1.793 ms</p> <p>5 Execution Time: 0.523 ms</p>	<p>Data Output Messages Notifications</p> <p>QUERY PLAN text</p> <p>1 Bitmap Heap Scan on narudzbina (cost=12.56..119.56 rows=2103 width=19) (actual time=0.016..0.461 rows=2097 loops=1)</p> <p>2 Recheck Cond: ((datum_narucivanja &gt;= '2022-11-12':date) AND (datum_narucivanja &lt;= '2023-04-12':date))</p> <p>3 Rows Removed by Index Recheck: 2903</p> <p>4 Heap Blocks: losty=32</p> <p>5 → Bitmap Index Scan on narudzbina_brin_idx (cost=0.00..12.03 rows=5000 width=0) (actual time=0.009..0.009 rows=320 loops=1)</p> <p>6 Index Cond: ((datum_narucivanja &gt;= '2022-11-12':date) AND (datum_narucivanja &lt;= '2023-04-12':date))</p> <p>7 Planning Time: 0.065 ms</p> <p>8 Execution Time: 0.517 ms</p>

Slika 34 – Prikaz poređenja sekvencijalnog i indeksiranog izvršenja upita

U toku procesa fizičkog projektovanja, BRIN indeks se može činiti kao primamljivo rešenje zbog sposobnosti definisanja opsega indeksiranih vrednosti, nalik virtuelnom particionisanju tabela, zbog veoma malo prostora koje ovakva struktura fizički zauzima na računarima, ali treba stalno imati na umu da ovaj indeksni mehanizam za jako male tabele, daje manje efikasne rezultate u odnosu na druge indekse. Tada je bolje iskoristiti B-stablo kao indeksnu strukturu, jer iako zauzima dvostruko više mesta, performanse izvršenja korisničkih upita su daleko efikasnije.

<pre> 1 SELECT pg_size_pretty(pg_total_relation_size('narudzbina_brin_idx')); </pre>	<pre> 1 SELECT pg_size_pretty(pg_total_relation_size('datum_narudzbine_bttree')); </pre>
<p>Data Output Messages Notifications</p> <p>pg_size_pretty text</p> <p>1 48 kB</p>	<p>Data Output Messages Notifications</p> <p>pg_size_pretty text</p> <p>1 64 kB</p>

Slika 35- Poređenje veličina BRIN (levo) i B-stablo (desno) indeksa

#### 4.4 Značaj indeksa za fizičko projektovanje PostgreSQL baze podataka

Pregledom prikazanih vrsti indeksnih struktura kod PostgreSQL baze podataka, može se zaključiti da indeks predstavlja ključan element u procesu fizičkog projektovanja. To je zato što indeks omogućava efikasno pretraživanje i sortiranje podataka u bazi (kao što je prikazano kroz pregled praktičnih primera seminarskog rada), utiče na spajanje podataka u bazi, a sve to zajedno rezultuje brzim odgovorima baze podataka na korisničke upite i zahteve. Indeksi se mogu primeniti i nad više kolona, pri čemu se upit izvršava uvek od krajnje leve ka desnoj koloni. Indeksiranje kao element fizičkog projektovanja PostgreSQL baze, može imati i negativan uticaj na sistem, jer može usporiti klasične operacije brisanja i dodavanja elemenata zato što isprojektovana baza mora održavati indeksnu strukturu, kao što je ranije pomenuto kroz seminarski rad.

```
CREATE INDEX a_b_c_idx ON brojevi(prvibroj,drugibroj,trecibroj);
```

<pre> 1 EXPLAIN (ANALYZE) 2 SELECT * 3 FROM brojevi 4 WHERE prvibroj = 5 AND drugibroj &gt;= 40 AND trecibroj &lt; 500; </pre>
<p>Data Output Messages Notifications</p> <p>QUERY PLAN text</p> <p>1 Bitmap Heap Scan on brojevi (cost=5.12..11.70 rows=33 width=12) (actual time=0.067..0.074 rows=31 loops=1)</p> <p>2 Recheck Cond: ((prvibroj = 5) AND (drugibroj &gt;= 40) AND (trecibroj &lt; 500))</p> <p>3 Heap Blocks: exact=6</p> <p>4 → Bitmap Index Scan on a_b_c_idx (cost=0.00..5.11 rows=33 width=0) (actual time=0.063..0.063 rows=31 loops=1)</p> <p>5 Index Cond: ((prvibroj = 5) AND (drugibroj &gt;= 40) AND (trecibroj &lt; 500))</p> <p>6 Planning Time: 1.322 ms</p> <p>7 Execution Time: 0.094 ms</p>



U svakom slučaju, indeksiranje je izuzetno važno u toku projektovanja, pre svega kod optimizacije performansi PostgreSQL baze podataka i samim tim zahteva pažljivo planiranje kako bi se uspostavila optimalna ravnoteža između brzine postavljanja upita od strane korisnika i brzine odziva sistema na zahtev.

## 5.OPTIMIZACIJA PODATAKA POSTGRESQL BAZE PODATAKA

Optimizacija podataka, podrazumeva proces poboljšanja performansi baze podataka, najčešće kroz smanjenje vremena odziva prilikom izvršavanja upita nad bazom. Sam proces optimizacije se može odnositi na optimizaciju upita, izbor odgovarajućeg indeksa, smanjenja broja join operacija, korišćenja vakum tehnike koja utiče na optimizaciju fizičkog dizajna baze podataka itd.

Optimizacija podataka kao i sam proces fizičkog projektovanja, predstavlja ključan element u razvoju sistema baze podataka, koji će odgovarati specifičnim potrebama korisnika.

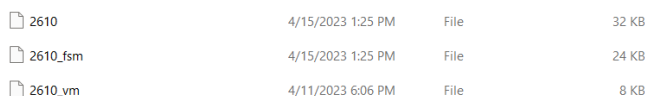
### 5.1 Fizičko mapiranje tabela i indeksa kod PostgreSQL baze podataka

U ranijim poglavljima rada opisan je način čuvanja podataka na fizičkim lokacijama na disku, ali se postavlja pitanje kako se ti podaci, predstavljeni u vidu tabela u bazi, mapiraju na pomenute fizičke lokacije. PostgreSQL baza podataka fizički mapira tabele i kreirane indekse koristeći strukture podatka koje se nazivaju „heap file“ i „index file“.

Heap file je binarna datoteka na disku koja se sastoji od niza stranica podataka u kojima su smešteni redovi tabele. Svaki red tabele zauzima jedan ili više slogova na stranicama podataka. Stranice se popunjavaju podacima kako se tabele popunjavaju, odnosno proces popunjavanja stranica prati proces upisa u tabele. PostgreSQL koristi algoritme za upravljanjem prostorom kako bi održao integritet podataka i optimalnu raspodelu prostora na disku.

Index file je takođe binarna datoteka na disku, koja se sastoji od stranica indeksa u kojima se čuvaju vrednosti indeksiranih kolona povezanih sa odgovarajućim redovima tabele.

Kada se kreira nova tabela u PostgreSQL bazi podataka, heap file za tu tabelu se kreira na disku, a prva stranica se rezerviše za njene podatke. Kako se dodaju redovi u tabelu u PostgreSQL bazi, tako se dodaju i nove stranice u heap file kako bi se podaci i fizički mapirali na disku. Prilikom kreiranja odgovarajuće indeksne strukture, kreira se i index file na disku, pri čemu se prva stranica indeksa rezerviše za njegove podatke. Kada se indeksira nova kolona, PostgreSQL kreira novu stranicu i dodaje sve neophodne informacije za sve postojeće redove.



2610	4/15/2023 1:25 PM	File	32 KB
2610_fsm	4/15/2023 1:25 PM	File	24 KB
2610_vm	4/11/2023 6:06 PM	File	8 KB

Slika 36- Fizički prikaz stranice kreiranog indeksa PostgreSQL baze na disku  
Blokovi stranica u PostgreSQL bazi, mogu biti popunjeni kompletno do kraja ili delimično u zavisnosti od veličine podataka koji se skladište. Međutim, kada se kontinualno vrše izmene nad tim podacima, poput brisanja, ažuriranja ili dodavanja novih podataka, dolazi do restrukturiranja. Restruktuiranje blokova stranica se može izbeći korišćenjem strategije „vacuuming“, o kojoj će biti više reči u nastavku.

Dobra praksa je da se baza podataka dizajnira tako da se u što većoj meri smanji potreba za restrukturiranjem blokova stranica. To se može postići izbegavanjem velikih transakcija, normalizacijom baze kako bi se izbeglo dupliranje podataka i ograničavanjem korišćenja indeksa samo na potrebna polja ( ranije je pokazano da nije potrebo da se indeksi kreiraju nad svim poljima tabele).

Važno je pomenuti da PostgreSQL ima i nekoliko mehanizama automatskog restrukturiranja blokova ( zahtevaju da se uključi njihova konfiguracija u projektovanu bazu) i to su autovacuuming, autoanalyze i autocheckpoint.

## **5.2 Vacuuming tehnika kao metoda optimizacije kod PostgreSQL baze podataka**

Vacuuming tehnika se može iskoristiti kao optimizacioni mehanizam kako bi se umanjila potreba višestrukog restrukturiranja blokova podataka. U PostgreSQL bazi podataka, vacuuming tehnika se postiže korišćenjem naredbe VACUUM. Ovom naredbom se vrši uklanjanje zastarelih redova, ali i oslobađanje prostora na disku koji su prethodno zauzimali ti zastareli redovi. Tehnika vakumiranja je jedan od češće korišćenih mehanizama optimizacije podataka kod PostgreSQL baze podataka, jer se obrisani taplovi ili taplovi koji su izuzetno davno ažurirani, ne uklanjaju fizički sa diska. Oni ostaju prisutni sve dok se ne izvrši VACUUM komanda. Zbog toga je neophodno periodično izvršavati vakumiranje nad tabelama koje se često ažuriraju [21], kako bi performanse baze podataka bile što bolje, a zauzeće prostora na disku što manje.

Vakum proces se može pokrenuti na nivou cele baze ( tako se i pokreće ukoliko se drugačije ne navede) ili na nivou tabele ili kolone u tabeli. Bez liste tabela ili kolona nad kojima je moguće izvršiti vakumiranje, vakum proces se izvršava nas svakom tabelom, materijalizovanim pogledom ili indeksnom strukturom. Ukoliko se navede lista tabela ili kolona, vakumiranje se vrši samo nad njima, a ne nad celom bazom podataka. Iza ključne reči VACUUM u naredbi za izvršenje procesa vakumiranja, mogu se naći parametri, koji zapravo predstavljaju opcije vakum procesa, a to su: FULL;FREEZE;VERBOSE;ANALYZE;INDEX\_CLEANUP;PARALLEL i još mnogi drugi [21].

Običan vakum proces nema ni jedan od parametara kao opciju, osim možda ANALYZE koja služi da pokaže strukturu vakum procesa, i on služi da jednostavno očisti prostor i učini ga ponovno dostupnim za upotrebu. Običan vakum proces može da radi paralelno sa upisom/čitanjem, jer ne definiše pravo ekskluzivnog pristupa, pa se tabela ne zaključava. Međutim dodatni prostor se ne vraća operativnom sistemu nakon vakumiranja ( u većini slučajeva), samo postaje dostupan za korišćenje od strane iste tabele ili baze ( sa stanovišta fizičkog projektovanja baze podataka i optimalnosti podataka, vrlo je efikasno, ali iz ugla OS-a i ne previše). Takođe, običan vakum, omogućava da se iskoristi više CPU-a za obradu indeksa i ova karakteristika je poznata kao paralelno vakumiranje. VACUUM FULL naredbom ceo sadržaj tabele se prepisuje u novu datoteku na disku bez dodatnog prostora, omogućavajući da se neiskorišćeni prostor vrati operativnom sistemu. Ovaj obrazac je daleko sporiji i zahteva ekskluzivno pravo pristupa bazi podataka ili svakoj tabeli koju obrađuje [21]. VACUUM FREEZE vrši takozvano zamrzavanje redova kako bi se izbeglo kasnije ponovno vakumiranje istih redova. Takvi redovi se smatraju stalno važećim i ne tretiraju se kao „dead taplovi“ prilikom izvršenja vakum operacija. Ova operacija se koristi onda kada je važno minimizovati vreme izvršenja vakum operacije. Međutim, i VACUUM FREEZE i VACUUM FULL, treba koristiti sa oprezom, jer one blokiraju pristup tabeli i samim tim direktno utiču na performanse baze podataka.

Što se tiče veze sa indeksima, vakum proces ima značajan uticaj na performanse indeksa u PostgreSQL bazi podataka. Kada se obavlja vakum proces na tabeli, on optimizuje i indeksne strukture koje su povezane sa tabelom. Ovo se dešava jer vakum proces uklanja redundantne podatke iz tabele, pri čemu se smanjuje i broj blokova koji se koriste za indekse, pa je proces pretrage po indeksu brži. Takođe, vakum procesom se ažurira statistika koja se koristi za planiranje optimizovanih upita, što poboljšava performanse upita koji koriste indekse. Inicijalno, vakum proces će preskočiti vakumiranje indeksa, kada u tabeli ima jako malo mrtvih torki.

Opcijom INDEX\_CLEANUP, vakum proces se primorava, da obrađuje i indeksne strukture, onda kada ima više od nula mrtvih torki. Ukoliko se ova opcija postavi na off, vakum proces će apsolutno uvek izbeći indekse, bez obzira na to što postoji dosta umrtvljenih taplova. Ovakvoj strategiji se pribegava, kada je neophodno što brže izvršiti vakumiranje kako bi se izbeglo zaobilaženja ID transakcija [21]. Svakako, treba imati na umu da ukoliko se INDEX\_CLEANUP postavi na off, a vakumiranje vrši kontinualno sa promenama tabele, performanse se mogu narušiti, jer će indeksne strukture zadržati svoje fizičke lokacije i uređenje, ali će sadržati veliki broj pokazivača na elemente koji su vakum procesom odavno uklonjeni i iz tabele i sa fizičkih lokacija. Ovakvi pokazivači će ostati i dalje u strukturi indeksa, bez obzira što je vakumiranjem očišćenja tabela, tačnije ostaće pristupiti sve dok se indeksna struktura ne podvrgne vakumiranju. O ovome ne treba brinuti ukoliko se koristi VACUUM FULL, jer on uklanja apsolutno sve elemente vezane za tabelu u bazi podataka. Praktični primer teorijski datog pregleda vakum procesa, dat je u nastavku.

**DELETE FROM proizvod WHERE cena>2500 AND cena<3000;**

```

1 VACUUM (VERBOSE, ANALYZE) proizvod, idx_cena;

Data Output Messages Notifications

INFO: vacuuming "SistemiBP.public.proizvod"
INFO: finished vacuuming "SistemiBP.public.proizvod": index scans: 1
pages: 0 removed, 42 remain, 42 scanned (100.00% of total)
tuples: 499 removed, 4500 remain, 0 are dead but not yet removable
removable cutoff: 919, which was 0 XIDs old when operation ended
new relfrozenxid: 919, which is 2 XIDs ahead of previous value
index scan needed: 5 pages from table (11.90% of total) had 499 dead item identifiers removed
index "proizvod_pkey": pages: 16 in total, 0 newly deleted, 0 currently deleted, 0 reusable
index "idx_cena": pages: 16 in total, 1 newly deleted, 1 currently deleted, 0 reusable
avg read rate: 0.000 MB/s, avg write rate: 123.274 MB/s
buffer usage: 137 hits, 0 misses, 24 dirtied
WAL usage: 38 records, 21 full page images, 131994 bytes
system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
INFO: analyzing "public.proizvod"
INFO: "proizvod": scanned 42 of 42 pages, containing 4500 live rows and 0 dead rows; 4500 rows in sample, 4500 estimated total rows

```

Slika 37- Prikaz razloga izvršenja vakum procesa i rezultata njegovog rada

Izvršenjem upita sa slike, u tabeli proizvod obrisani su svi redovi koji su sadržali vrednosti u datom opsegu. Međutim, ti redovi su fizički ostali prisutni na disku. Iz tog razloga izvršen je proces vakumiranja nad celom tabelom, ali i indeksnom strukturom kako bi se uklonili i indeksi kreirani u tabeli. Sa slike se može videti da je vakumiranjem uklonjen opseg vrednosti i iz tabele, ali i sa fizičkih lokacija, podrazumevajući i indkesne strukture. Takođe je dat i uvid u način skeniranja indeksa prilikom vakumiranja. Vakum označi mrtvu tokru u zaglavlju stranice koja sadrži tapl. Tog trenutka, on tu torku, ali i set bitova indeksa vezanih za nju tretira kao nestale ( to postiže jer podržava mehanizam kojim pamti poslednji ID transakcije izvršene neposredno pred vakumiranje i tako ustanovi da torka ne postoji u tabeli).Potom skeniranjem indeksa od početka do kraja, uklanja sve referenrence ka nepostojećoj torki ( briše pokazivače na povezane stranice) iz indeksa i vraća se na početni sadržaj stranice da je označi kao ponovo slobodnu stranicu.

```

1 SELECT * FROM pg_stat_all_indexes WHERE indexrelname = 'idx_cena';

relid | indexrelid | schemaname | relname | indexrelname | idx_scan | idx_tup_read | idx_tup_fetch
-----+-----+-----+-----+-----+-----+-----+-----
1 | 16592 | public | proizvod | idx_cena | 16 | 10015 | 5011

1 SELECT pg_size_pretty(pg_total_relation_size('idx_cena'));

pg_size_pretty
text
1 | 128 kB

```

Slika 38- Prikaz strukture indeksa pre operacije vakumiranja

1

SELECT \* FROM pg\_stat\_all\_indexes WHERE indexrelname = 'idx\_cena';

1

2

SELECT pg\_size\_pretty(pg\_total\_relation\_size('idx\_cena'));

relid

indexrelid

schemaname

relname

indexrelname

idx\_scan

idx\_tup\_read

idx\_tup\_fetch

oid

oid

name

name

name

bigint

bigint

bigint

1

16592

16599

public

proizvod

idx\_cena

18

10515

5511

pg\_size\_pretty

text

1

152 kB

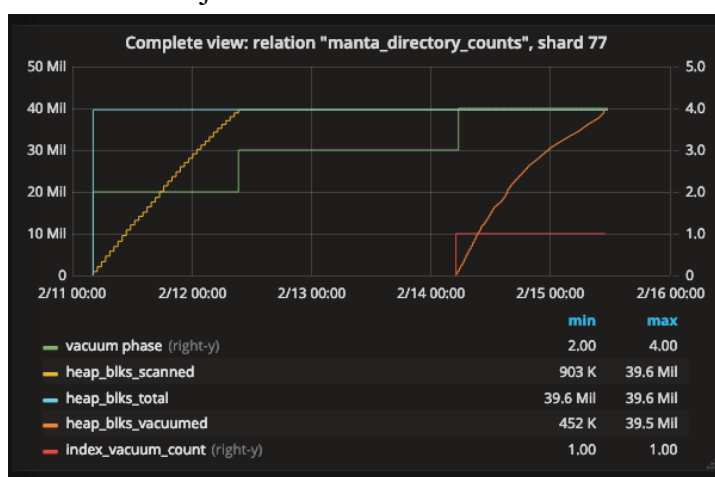
Data Output

Messages

Notifications

Slika 39- Prikaz strukture indeksa nakon vakumiranja

Sa prethodnih slika se vidi da su vrednosti veličine indeksa o polja `idx_scan` i `idx_tup_read` uvećane u odnosu na vrednosti koje su se nalazile u tim poljima pre izvršenja vakumiranja. Vakum proces može, kao u ovom slučaju, da poveća veličinu indeksa, jer ne briše samo zastarele podatke, već vrši i preuređenje sadržaja tabele kako bi se smanjila fragmentacija podataka. Kako su ovde obrisane veće količine podataka, mora se izvršiti reorganizacija preostalih indeksnih struktura kreiranjem novih blokova indeksa da preuzmu novoformirane vrednosti. Veće vrednosti u poljima `idx_scan` i `idx_tup_read`, nastale su kao rezultat skeniranja i čitanja taplova prilikom vakumiranja i kao pokazatelj da je PostgreSQL baza koristila indeksnu strukturu više puta pre izvršenja procesa vakumiranja.



Slika 40 – Grafički prikaz procesa vakumiranja kod PostgreSQL baze<sup>11</sup>

U uvodnom delu priče o vakumiranju, pomenut je i mehanizam autovakumiranja kod PostgreSQL baze podataka. Autovakum predstavlja demona ili background proces PostgreSQL baze i nalazi se u konfiguracionom fajlu, odakle se pokreće i zato ne zahteva manuelno pokretanja tj. vakumiranje. Autovakumiranje, nakon brisanja podataka iz blokova, preostali prostor raspoređuje i obaveštava transakcije koje čekaju o slobodnom prostoru gde mogu smestiti podatke, za razliku od manualnog vakumiranja gde se podaci ubacuju sa istim identifikacionim elementima. Prednosti autovakumiranja su dobro korišćenje skladištenog prostora, smanjeno vreme izvršenja i manje neophodnih resursa. Iako autovakumiranje postoji kao prateći proces, parametri koji se koriste za izvršenje vakumiranja, odgovaraju trenutnoj verziji, a kako bi efikasno bili iskorišćeni kod sistema koji se projektuje, neophodno je izvršiti proporcionalnu konfiguraciju parametara prilikom fizičkog projektovanja, kako bi njegovo korišćenje bilo što efikasnije.

<sup>11</sup> Izvor slike 40 – <https://www.davepacheco.net/blog/2019/visualizing-postgres-vacuum-progress/>

## 6.ZAKLJUČAK

Proces fizičkog projektovanja PostgreSQL baze podataka i razmatranje optimizacije podataka, predstavljaju vrlo važan element kreiranja baze za bilo koji sistem i u bilo koje svrhe, jer upravo od ispravno postavljenog fizičkog dizajna baze i optimizacije podataka u njoj, zavise performanse projektovanog sistema, ali i zadovoljstvo krajnjih korisnika njome.

U seminarskom radu, opisana je PostgreSQL baza, od trenutka razvoja pa do današnjih dana, njene karakteristike i svojstva. Takođe su opisani i osnovni koncepti fizičkog projektovanja baze podataka, ali i značaj koji fizičko projektovanje ima za izgradnju sistema. Posebna pažnja usmerena je ka indeksiranju, kao jednom od ključnih elemenata fizičkog projektovanja, i vrstama indeksa kod PostgreSQL baze podataka, uključujući i prikaz praktičnih primera korišćenja indeksa. Razumevanje strukture indeksnih vrsti kod PostgreSQL baze podataka i prikaz njihovog funkcionisanja, predstavljaju krucijalni faktor za fizičko projektovanje, jer u zavisnosti od izbora indeksne strukture, performanse sistema mogu biti ili poboljšane ili značajno degradirane.

Pored indeksiranja kao dela fizičkog projektovanja, opisan je i proces optimizacije podataka kod PostgreSQL baze podataka. Kao važan element izdvojen je mehanizam vakumiranja i njegov uticaj na tabele, kolone, indeksne strukture i podatke u PostgreSQL bazi podataka. Dat je i kratak opis autovakumiranja kod PostgreSQL-a i poređenje sa vakumiranjem.

Naposletku, pregledom seminarskog rada i poređenjem rezultata izvršenja praktične realizacije opisanih metoda, može se ustanoviti važnost pravilnog izbora indeksa prilikom projektovanja baze podataka i učestalosti izvršenja vakumiranja kao i njegovih efekata na sam sistem.

## 7.SPISAK KORIŠĆENE LITERATURE

- [1] PostgreSQL 15 Documentation, The PostgreSQL Global Development Group, 1996-2023.  
Dostupno: <https://www.postgresql.org/docs/current/intro-what-is.html>
- [2] R.Ramakrishnam, J.Gehrke, "Database Management Systems", Third Edition, McGraw-Hill, International Edition, 2003.
- [3] C. Maier , D.Dash, I.Alagiannis, A. Ailamaki , T. Heinis, " PARINDA: An Interactive Physical Designer for PostgreSQL",Ecole Polytechnique Fédérale de Lausanne, Switzerland, 2010.
- [4] A Brief History of PostgreSQL, The PostgreSQL Global Development Group, 1996-2023.  
Dostupno: <https://www.postgresql.org/docs/15/history.html>
- [5] P. Kushwaha, "Architecture of PostgreSQL DB", 2020.  
Dostupno: <https://medium.com/swlh/architecture-of-postgresql-db-d6b1ac4cc231>
- [6] H. Suzuki, "The Internals of PostgreSQL for database administrators and system developers",2019.  
Dostupno: <https://www.interdb.jp/pg/pgsql01.html>
- [7] PostgreSQL Tutorial,Java T Point  
Dostupno: <https://www.javatpoint.com/postgresql-tutorial>
- [8] A.Chola, "Overview of a Physical Design Database: 5 Critical Aspects",2022.  
Dostupno: <https://hevodata.com/learn/physical-design-database/#Importance>
- [9] Physical Darabase Design  
Dostupno: <https://slideplayer.com/slide/9037414/>
- [10] Physical Database Design and Database Security  
Dostupno: <https://alg.manifoldapp.org/read/introduction-to-database-systems/section/c5a934d1-5720-401e-b90b-e6b01fb0aaa7>
- [11] PostgreSQL 15 Documentation -Table Partitioning, The PostgreSQL Global Development Group, 1996-2023.  
Dostupno: <https://www.postgresql.org/docs/current/ddl-partitioning.html>
- [12] PostgreSQL 15 Documentation -Indexes, The PostgreSQL Global Development Group, 1996-2023.  
Dostupno: <https://www.postgresql.org/docs/15/indexes.html>
- [13] "DBMS - Indexing"  
Dostupno: [https://www.tutorialspoint.com/dbms/dbms\\_indexing.htm](https://www.tutorialspoint.com/dbms/dbms_indexing.htm)
- [14] "Indexing in DBMS"  
Dostupno: <https://www.javatpoint.com/indexing-in-dbms>
- [15] R.Elmasri, S.B. Navathe, "Fundamentals Of Database Systems", Sixth Edition, Addison-Wesley, 2010.
- [16] "Hashing in DBMS"  
Dostupno: <https://www.javatpoint.com/dbms-hashing>
- [17] "Index Access Method Interface Definition"  
Dostupno: <https://www.postgresql.org/docs/current/indexam.html>
- [18] P.S. Randal, "An Introduction to B-Tree and Hash Indexes in PostgreSQL", 2022.  
Dostupno: <https://www.sentryone.com/blog/introduction-to-b-tree-and-hash-indexes-in-postgresql>
- [19] Indexes in PostgreSQL, PostgresPro, 2020.  
Dostupno: <https://postgrespro.com/blog/pgsql/4161516>

[20] Index Types, PostgreSQL 15 Documentation

Dostupno: <https://www.postgresql.org/docs/15/indexes-types.html#INDEXES-TYPES-BTREE>

[21] VACUUM, PostgreSQL 15 Documentation

Dostupno: <https://www.postgresql.org/docs/current/sql-vacuum.html>

[22] PostgreSQL Autovacuum

Dostupno: <https://www.geeksforgeeks.org/postgresql-autovacuum/>