Student: Jovan Mladenović 1277

Profesor: Aleksandar Stanimirović

Interna struktura i organizacija skladišta podataka PostgreSQL

Sadržaj

[Uvod 4](#_Toc101870654)

[1. Struktura - klasteri, baze i tabele 5](#_Toc101870655)

[1.1 Logička struktura klastera 5](#_Toc101870656)

[1.2 Fizička struktura klastera 5](#_Toc101870657)

[1.2.1 Direktorijum baze 6](#_Toc101870658)

[1.2.2 Datoteke tabela i indeksa 6](#_Toc101870659)

[1.2.3 Organizacija Heap Table datoteke 7](#_Toc101870660)

[2. Procesna i memorijska arhitektura 9](#_Toc101870661)

[2.1 Procesna arhitektura 9](#_Toc101870662)

[2.1.1 Postgres Server proces 9](#_Toc101870663)

[2.1.2 Backend proces 9](#_Toc101870664)

[2.1.3 Background proces 10](#_Toc101870665)

[2.2 Memorijska organizacija 10](#_Toc101870666)

[2.2.1 Lokalna memorija 10](#_Toc101870667)

[2.2.2 Deljena memorija 10](#_Toc101870668)

[3. Procesiranje Query-ja 10](#_Toc101870669)

[3.1 Parser 11](#_Toc101870670)

[3.2 Analyzer/Analyser 12](#_Toc101870671)

[3.3 Rewriter 12](#_Toc101870672)

[3.4 Planner i Executor 13](#_Toc101870673)

[3.4.1 Procena cene u single-table upitima 14](#_Toc101870674)

[3.4.2 Kreiranje plan tree-ja za single-table upite 15](#_Toc101870675)

[3.4.3 Kako radi Executor 19](#_Toc101870676)

[4. Kontrola konkurentnosti 20](#_Toc101870677)

[4.1 ID transakcije 21](#_Toc101870678)

[4.2 Struktura tuple-a 21](#_Toc101870679)

[4.2.1 Upis 21](#_Toc101870680)

[4.2.2 Brisanje 21](#_Toc101870681)

[4.2.3 Ažuriranje 21](#_Toc101870682)

[4.3 Commit log 21](#_Toc101870683)

[5. Vakuum procesiranje 20](#_Toc101870684)

[5.1 Visbility map 22](#_Toc101870685)

[6. Linkovi 22](#_Toc101870686)

## Uvod

Napravljen u Univerzitetu u Kaliforniji u odeljenju za kompjuterske nauke PostgreSQL je pionir koncepta koji su postali dostupni u komercijalnim DBMS (Database Managment System) sistemima mnogo kasnije. PostgreSQL je open-source višenamenska baza podataka. Ona je jedan ogroman sistem sačinjen od manjih podsistema, gde svaki ima svoju ulogu i sarađuje sa drugim podsistemima.

Takođe nudi korisniku da proširuje njene funkcionalnosti na mnogo načina kao štoje dodavanje:

* Novih tipova podataka
* Funkcija
* Operacija
* Agregacija funkcija
* Proceduralnih jezika

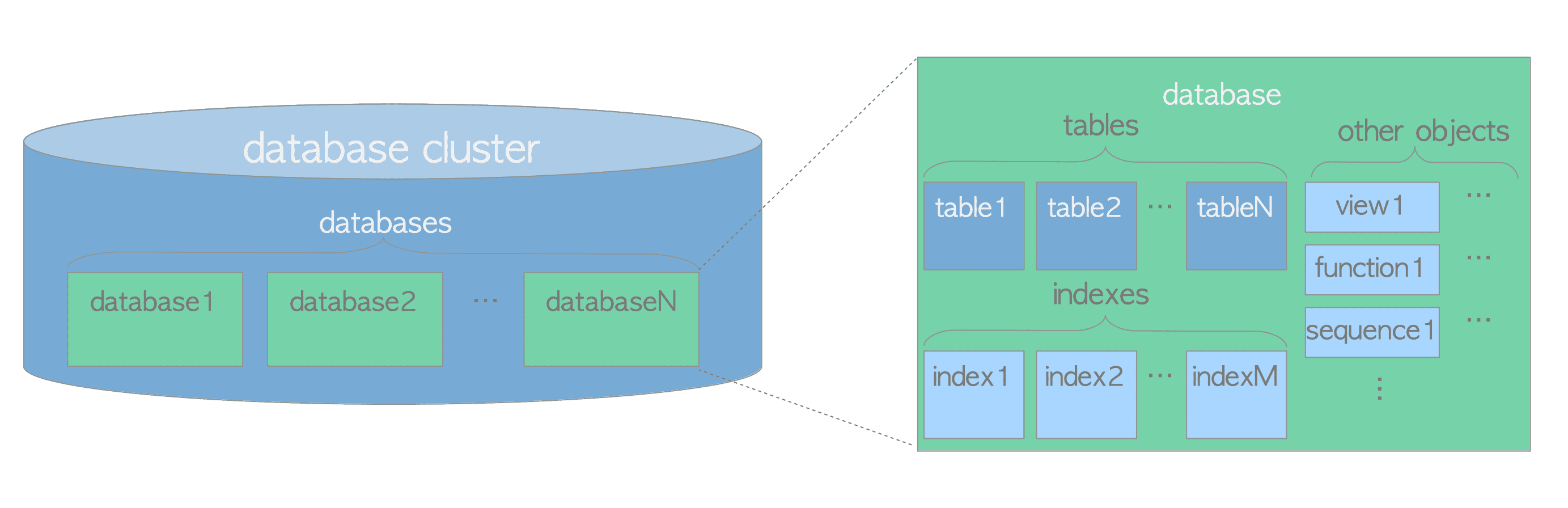
Zahvaljujući svojoj libelarnoj licenci, PostgreSQL može biti korišćen od strane bilo koga za bilo koju namenu bez potrebe za plaćanjem.

# Struktura - klasteri, baze i tabele

### Logička struktura klastera

U PostgreSQL kada se kaže *klaster baza podataka* (eng. Database cluster) ne osnosi se na grupu servera sa bazama podataka, vać PostgreSQL server radi na jednoj mašini i upravlja jednim klasterom baza podataka.

*Database* je kolekcija *database objekata, database object* je struktura podataka koja služi za referenciranje ili čuvanje podataka. Tabela je uobičajan primer ovoga, ali postoje još poput indeksa, sqkvenci, view-a, funkcija itd. U PostgreSQL-u same baze su database object-i i logički su razdvojeni međusobno, svi ostali database object-i (tabele, indeksi, itd.) pripadaju svojim bazama.

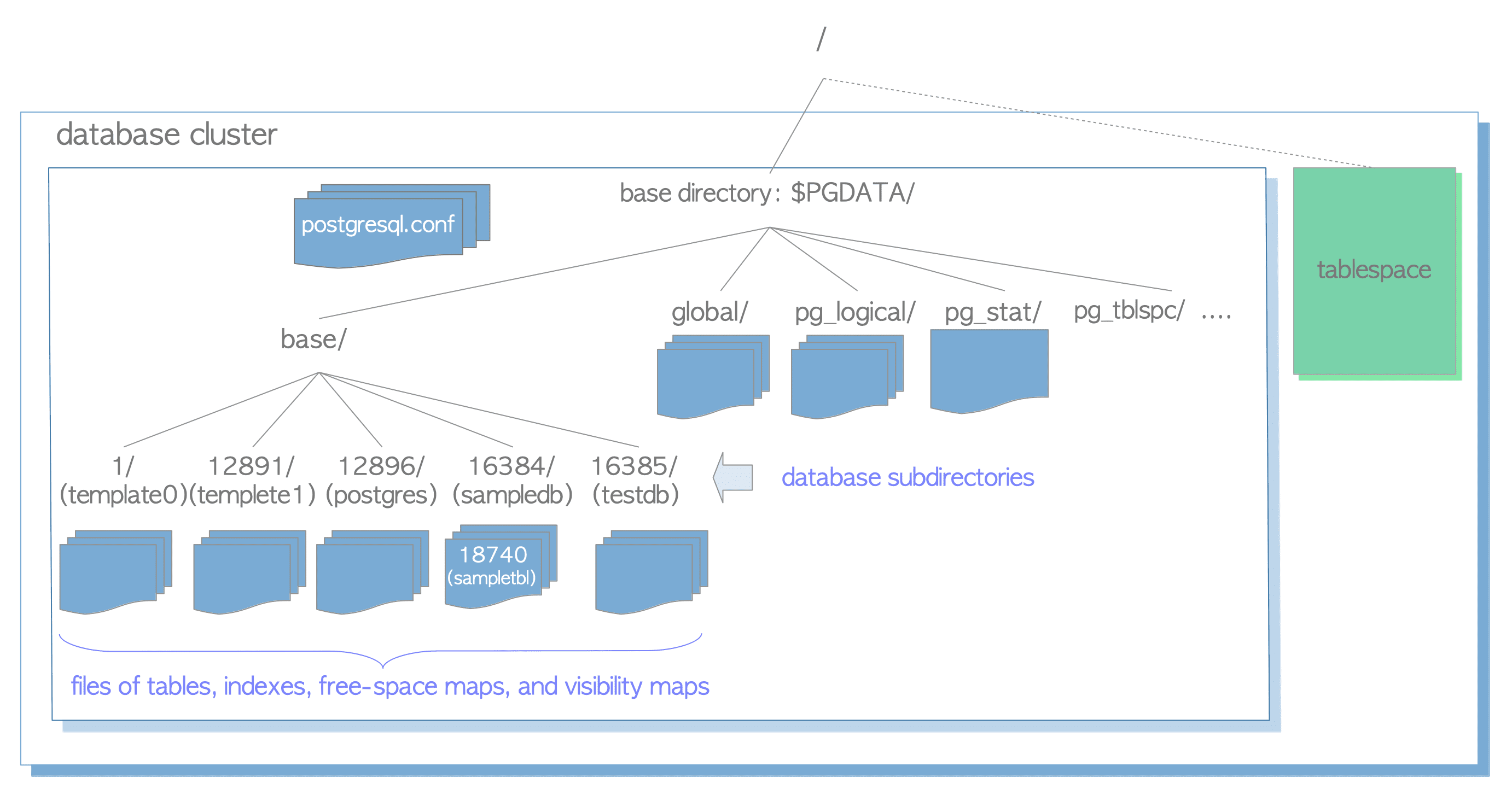


Slika 1.1 – logička struktura klastera

Svi database object-i imaju sopsteveni četvoro bajtni *object identifier* (OID), veza između database object-a i OID-a se pamti u odgovarajućim sistemskim katalozima u zavisnosti koji objekat je u pitanju (npr. OID-i tabela se pamte u *pg\_database,* dok OID-i baza u *pg\_class*).

### Fizička struktura klastera

Fizički klaster se na disku pamti kao jedan direktorijum koji sadrži poddirektorijume i fajlove. Baza podataka je poddirektorijum *BASE* poddirektorijuma i svaka od tabela, indeksa i sl. koji pripadaju bazi su predstavljeni fajlovima unutar poddirektorijuma za datu bazu. Pored toga tu su još poddirektorijuma koji sadrže određene podatke i konfiguracione fajlove. *Table space* u PostgreSQL se odnosi na direktorijum koji sadrži neke podatke van BASE direktorijuma.

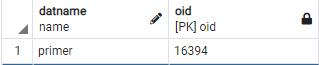


Slika 1.2 – fizička struktura klastera

### 1.2.1 Direktorijum baze

Sledećom komandom možemo otkriti ime direktorijuma naše baze, on se nalazi unutar base direktorijuma.

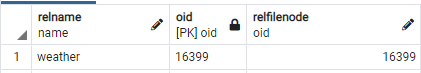
SELECT datname, oid FROM pg\_database WHERE datname = 'primer';



### 1.2.2 Datoteke tabela i indeksa

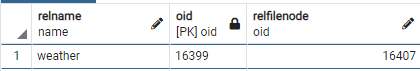
Svi objekti su interno obeleženi OID-om, dok su fajlovi na disku obeleženi variablom *refilenode.* Refilenode i OID vrednosti su često ali ne i uvek jednake.

select relname, oid, relfilenode from pg\_class where relname = 'weather'



Relfilenode vrednosti se menjaju iydavanjem komandi poput TRUNCATE, REINDEX, CLUSTER isl. Ako bi odradili TRUNCATE tabele weather primetićemo da se relfilenode promenio iz 16399 u 16407 dok je iod ostao isti.

reindex table weather

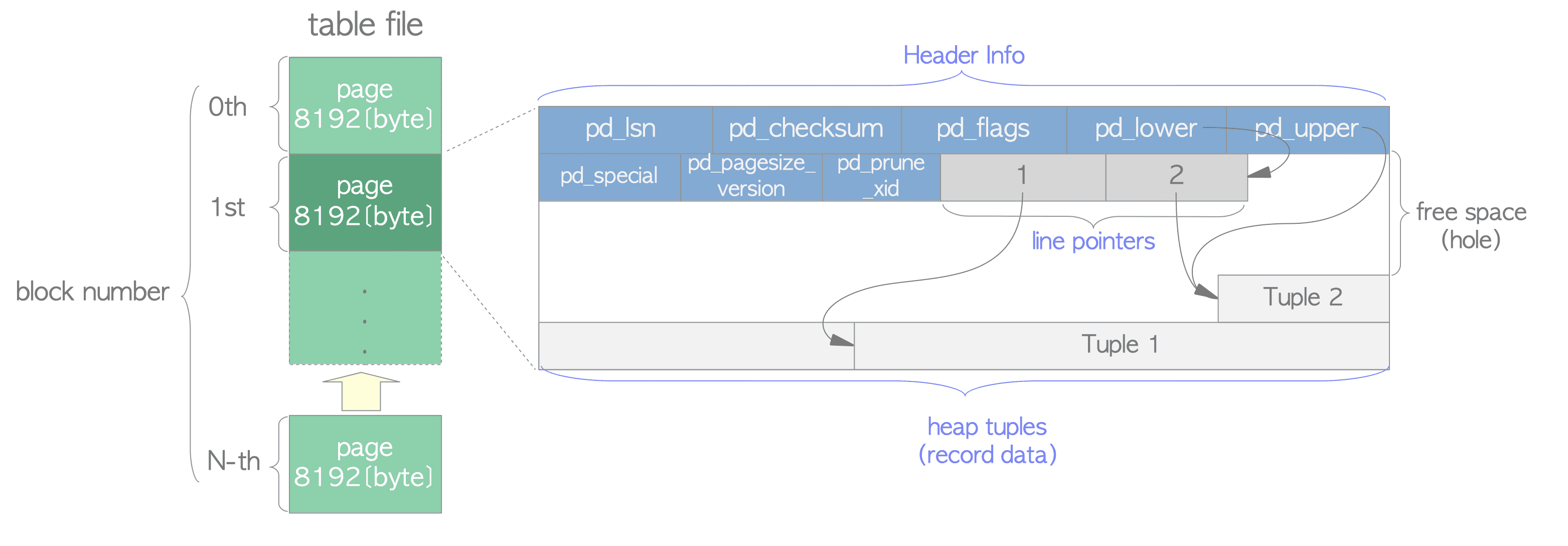


Kada fajl pređe veličinu od 1GB kreira se fajl sa imenom *relfilenode.1* (u prethodnom primeru to bi bilo 16407.1)*,* kada se on popuni pravi se novi pod imenom *relfilenode.2.*

Pored relfilenode fajla postoje i fajlovi sa istim imenom uz sufikse *\_fsm* (free space map) i *\_vm* (visibility map)

### 1.2.3 Organizacija Heap Table datoteke

Unutar data datoteke ona je podeljena u strane (**pages** ili **blocks**) fiksnih veličina, difoltna veličina je 8192 bajza (8 KB). Strane su numerisane sekvencijalno počevši od 0 (**block numbers**). Ako se popuni datoteka, PostgreSQL dodaje novu praznu stranicu kako bi povećao velišinu datoteke. Unutršnja organizacija zavisi od tipa data datoteke.

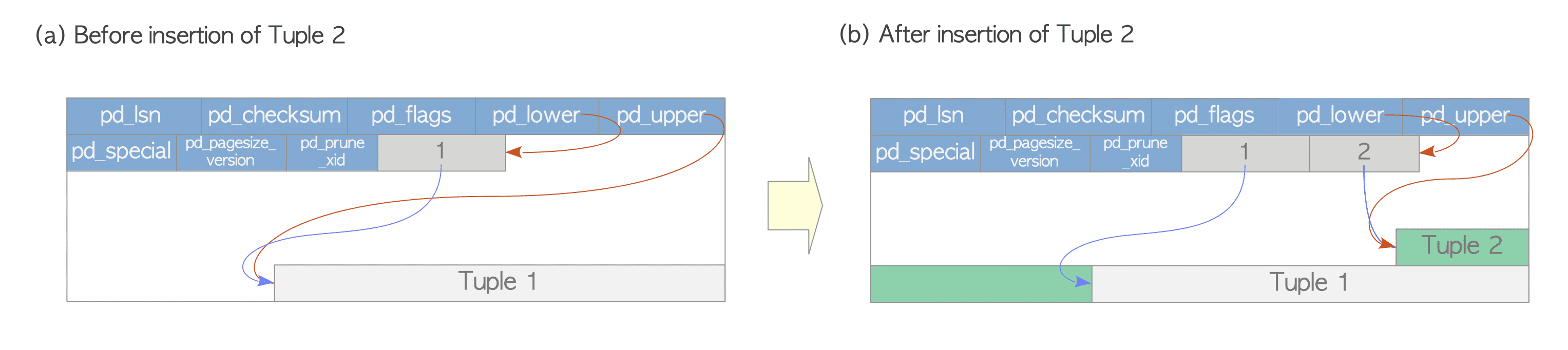


Slika 1.3 – organizacija heap table datoteke

Stranica unutar tabele sadrži tri tipa podataka:

* Heap tuple(s) – sam podatak, ređaju se od dna stranice pre vrhu.
* Line pointer(s) – pokazivač na svaki tuple, takođe poznati kao item pointeri. Line pointeri čine niz koji ima ulogu indeksa, svaki indeks je enumerisan sekvencijalno počevši od 1 (offset number). Kada se doda novi tuple, takođe se dodaje novi pointer koji ukazuje na njega.
* Header data – definisan PageDataHeader strukturom i nalazi se na vrhu stranice i sadrži informacije vezane za stranicu. Glavna polja su:
  + *pd\_checksum*– checksum vrednost za stranicu
  + *pd\_lower* – ukazuje na kraj niza line pointera
  + *pd\_upper* – ukazje na početak najnovijeg tuple-a

Prazan prostor između kraja niza line pointera i najnovijeg tuple-a je poznat kao **slobodan prostor** (**free space**) ili **rupa (hole)**.

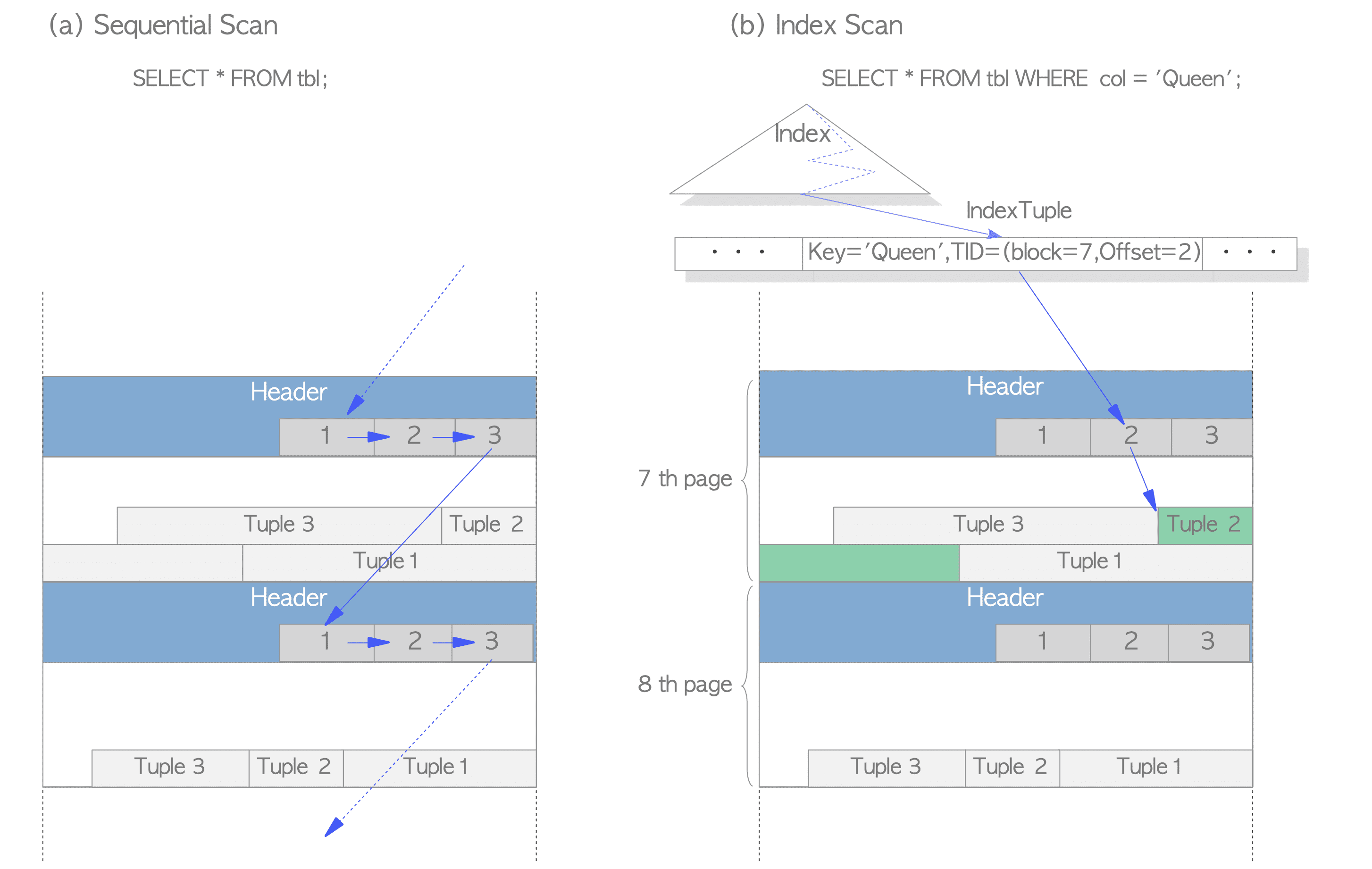


Slika 1.4 – upis novog tuple-a u stranicu

Tipične metode za čitanje tupl-a su

Sekvencujalno skeniranje – svi tuple-ovi su sekvencijalno čitani skeniranjem svih line pointera unutar stranice

B-tree index scan – indeks datoteka sadrži indeks tuple-ove, svaki sastavljen od ideksnog ključa i TID-a koji ukazuje na ciljni heap tuple. Na primeru je dato da je TID vrednost (block = 7, Offset = 2), što znači da je ciljni tuple drugi tuple na sedmoj stranici unutar tabele i PostgreSQL može da pročita traženi tuple bez nepotrebnog skeniranja stranica.



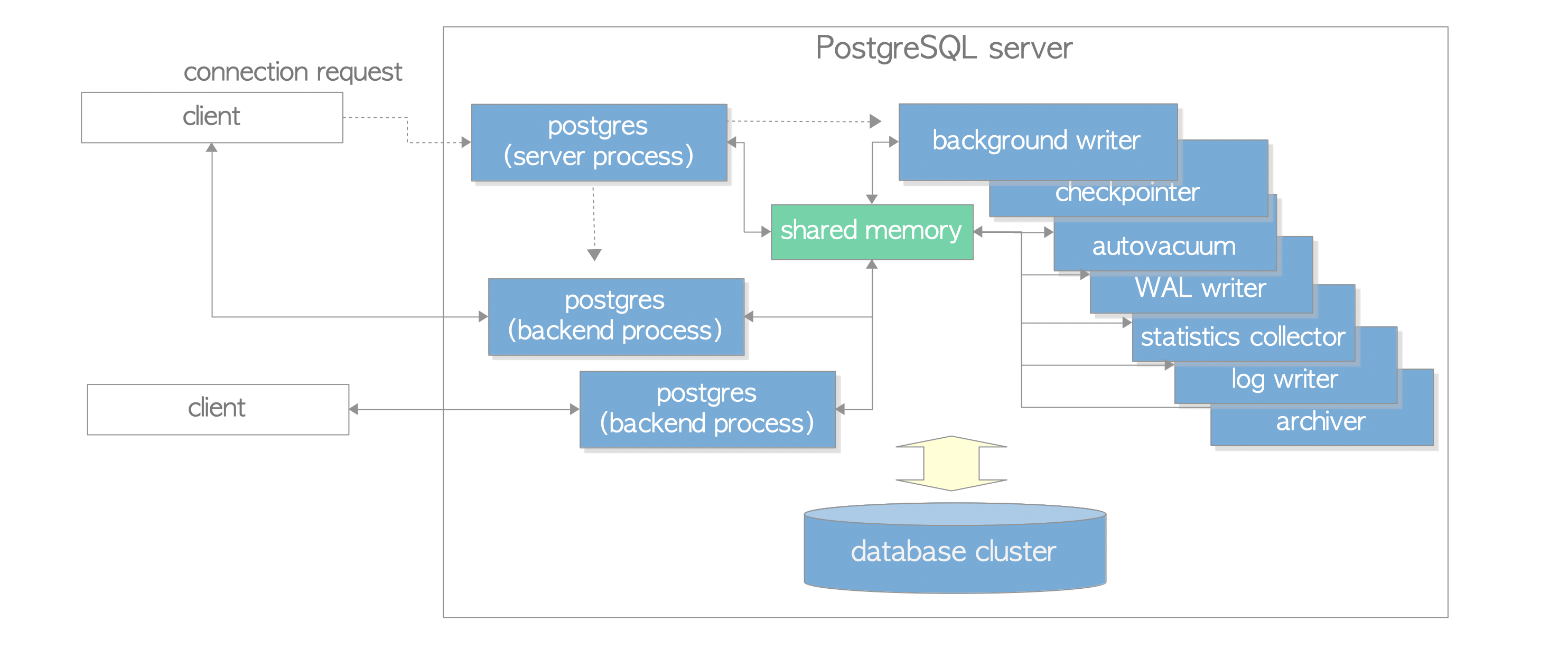
Slika 1.4 – primer čitanja tuple-ova

# Procesna i memorijska arhitektura

## 2.1 Procesna arhitektura

PostgreSQL je multiprocesna arhitektura pokrenuta na jednom host-u. Kolekcija procesa koji se izvršavaju nad jednim klasterom se naziva *PostgreSQL server* i sadrži sledeće procese

* **Postgres server** proces je glavni (parent) proces.
* Svaki od **backend** procesa izvršava zahteve dobijene od konektovanog klijenta.
* Mnoštvo **background** procesa gde svaki izvršava posebnu funkcionalnost
* **Background process worker** izvršava bilo kakvu obradu kreiranu od strane korisnika, preporučuje se velika opreznost jer implmentacija pisana u C-u omogućava neogrničen pristup podacima.



Slika 2.1 – primer procesne arhitekture PostgreSQL-a

### 2.1.1 Postgres Server proces

Pri pokretanju rezerviše memorijski prostor i pokreće potrebne potprocese. Kada se klijent konektuje kreira backend proces koji nadalje rešava zahteve.

Postgres server osluškuje samo jedan port na mašini, po difoltu to je port 5432. Mogu se pokrenuti više postgres servera na jednoj mašini ali svaka mora da sluša različit port.

### 2.1.2 Backend proces

Backend proces poznat kao i *postgres* je kreiran od strane postgres servera i izvršava zahteve jedneg klijenta. Komunikacija sa klijentom je ostvarena TCP konekcijom koja se ukida kada se klijent diskonektuje.

Backend proces može da radi samo nad jednom bazom, koja se mora navesti prilikom uspostavljanja konekcije sa postgres serverom.

PostgresSQL dozvoljava više istovremenih konekcija sa klijentima, difoltna vrednost je 100 konekcija. Kada se javlja mnoštvo uzastopnih konekcija i deiskonekcija kao kada su u pitanju WEB klijenti povećava cenu povezivanja i kreiranje backend procesa, da bi se zaobišli ovi problemi koristi se *middleware* kao što su *pgbouncer* i *pgpool.*

### 2.1.3 Background proces

Background procesi su previse obimni da bi se svi analiyirali u ovom dokumentu, svaki ima zasebnu funkcionalnost koja služi da bi se obezbedio rad PostgreSQL, to su na primer: autovacuum, checkpointer, log writer i sl.

## 2.2 Memorijska organizacija

Memorijska arhitektura se najprostije može klasifikovati u dve grupe

* Lokalna memorija – alocirana od strane pojedinačnog backend procesa
* Deljena memorija – koristi se od strane svih procesa

### 2.2.1 Lokalna memorija

Svaki backend proces alocira zasebni memorijski prostor za izvršenje klijentskih zahteva, podeljena u potprostore kojima je veličina može biti nepromenljiva ili promenljiva.

* work\_mem - koristi se za sortiranje i povezivanje tabela
* maintenence\_work\_mem - operacije poput VACUUM, REINDEX koriste ovaj memorijski prostor
* temp\_buffers – služi za sortiranje privremenih tabela

### 2.2.2 Deljena memorija

Deljena memorija je alocirana od strane postgres servera prilikom pokretanja, ona je podeljena u nekoliko segmenta nepromenljivih veličina.

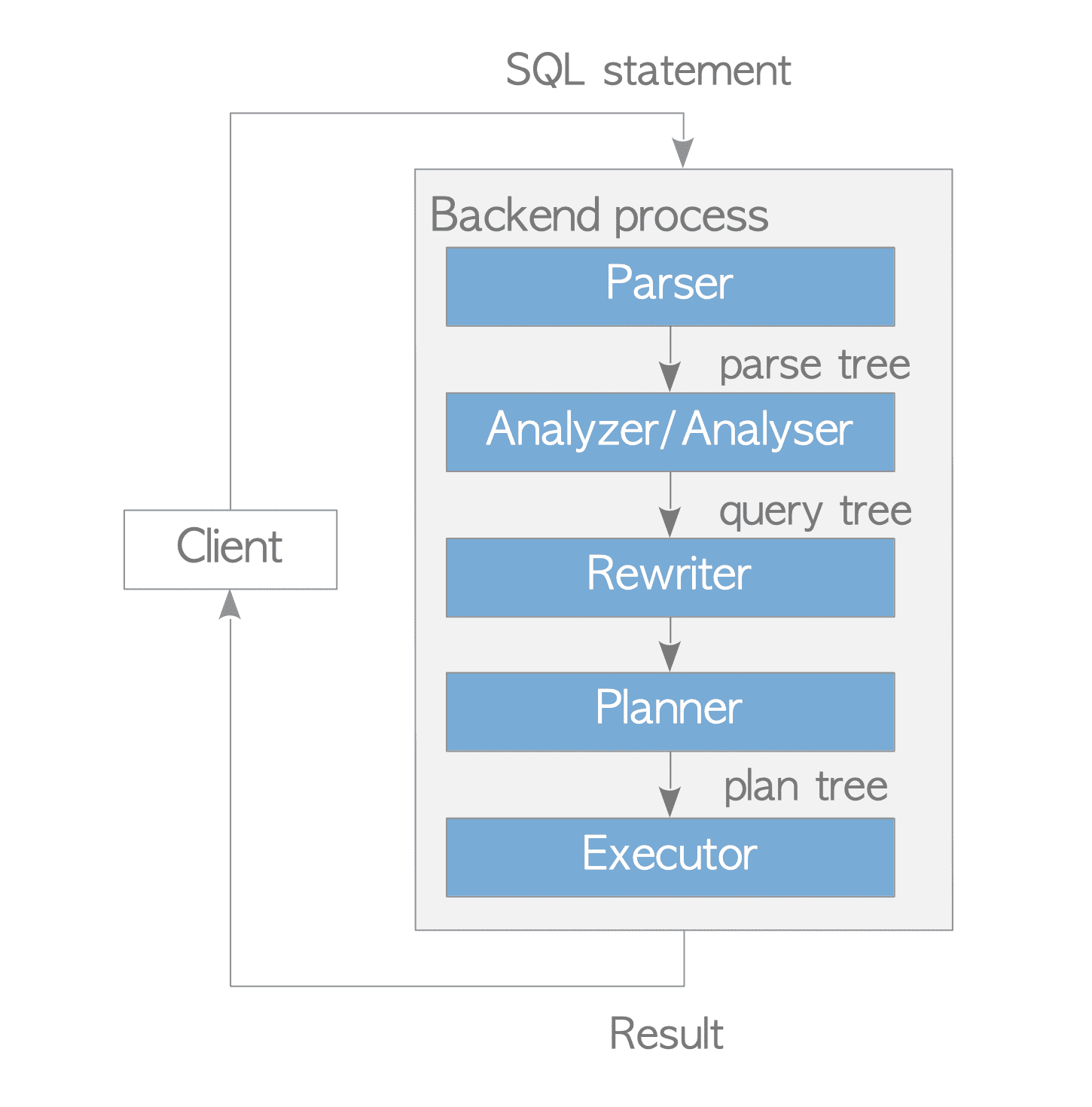
* shared buffer pool – postgreSQL učitava stranice tabela i indeksa ovde
* WAL buffer – kako se ne bi izgubili podaci prilikom otkaza servera postoji *WAL* *mehanizam*. WAL data (takođe poznat i kao *XLOG records*) su transakcioni zapisi koji ide u WAL buffer pre upisa na disk.

# Procesiranje Query-ja

Procesiranje Query’ja je najkomplikovaniji podsistem PostrgresSQL i veoma je efikasno u procesiranju podržanog SQL-a. Ovaj deo dokumenta opisaće korake u obradi query-ja.

I akoje podržano paralelno izvršenje query-ja koje koristi nekoliko background procesa svaki od background procesa se sastoji od pet podsistema:

* Parser – generiše *parse tree* (stablo) od SQL zahteva dobijenom u obliku teksta.
* Analyzer/analzser – vrši nalaizu nad parse tree i generiše query tree.
* Rewriter – vrši transformaciju nad qurty tree baziranu po *rule szstem-u* ako je to potrebno
* Planner – greneriše *plan tree* koji je najefikasnije izvršenje query tree-ja
* Executor – izvršava query pristupanjem tabelama i indeksima po redosledu baziranim na plan tree-ju.

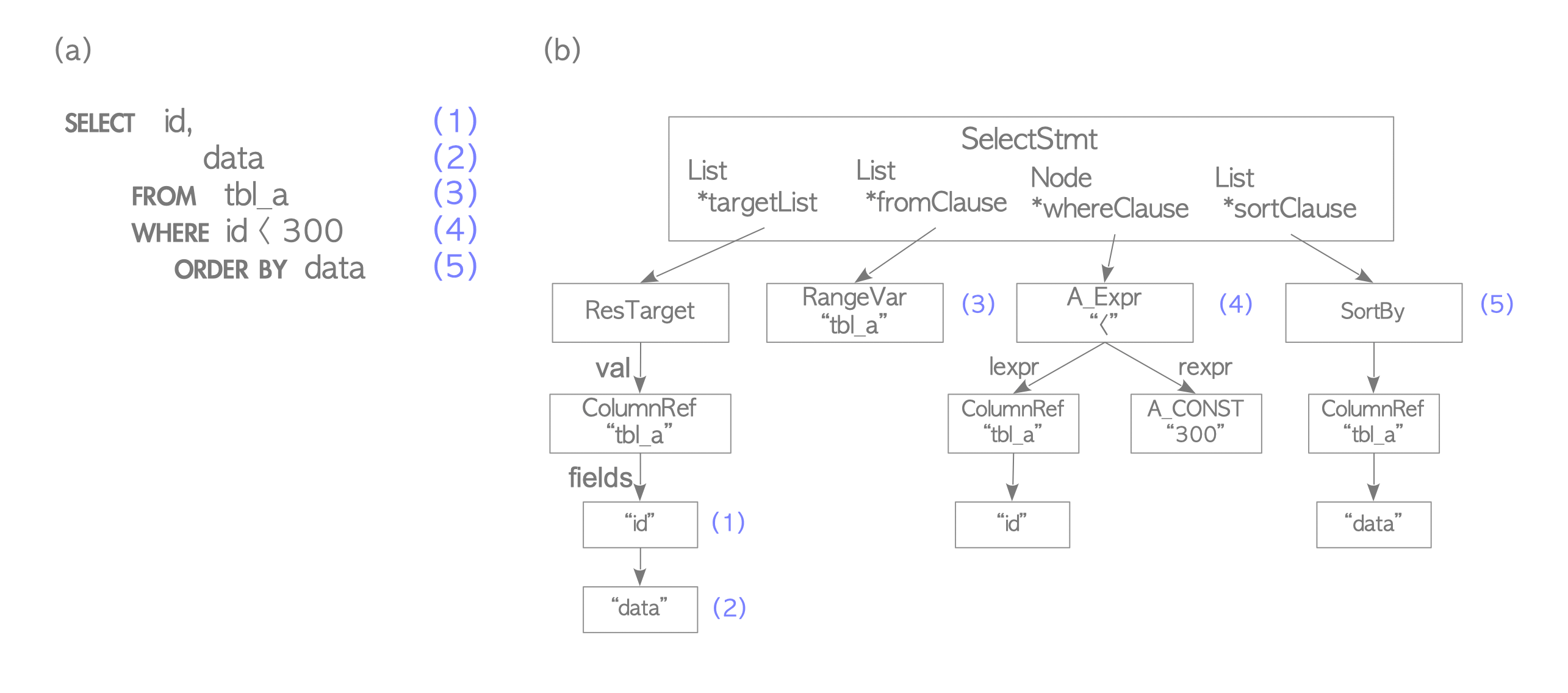


Slika 3.1 – procesiranje Quary-ja

## 3.1 Parser

Parser generiše parse tree koje mogu pročitati sledeći podsistemi na osnovu SQL izraza. Uzmimo sledeći SQL izraz u obzir

SELECT id, data FROM tbl\_a WHERE id < 300 ORDER BY data;



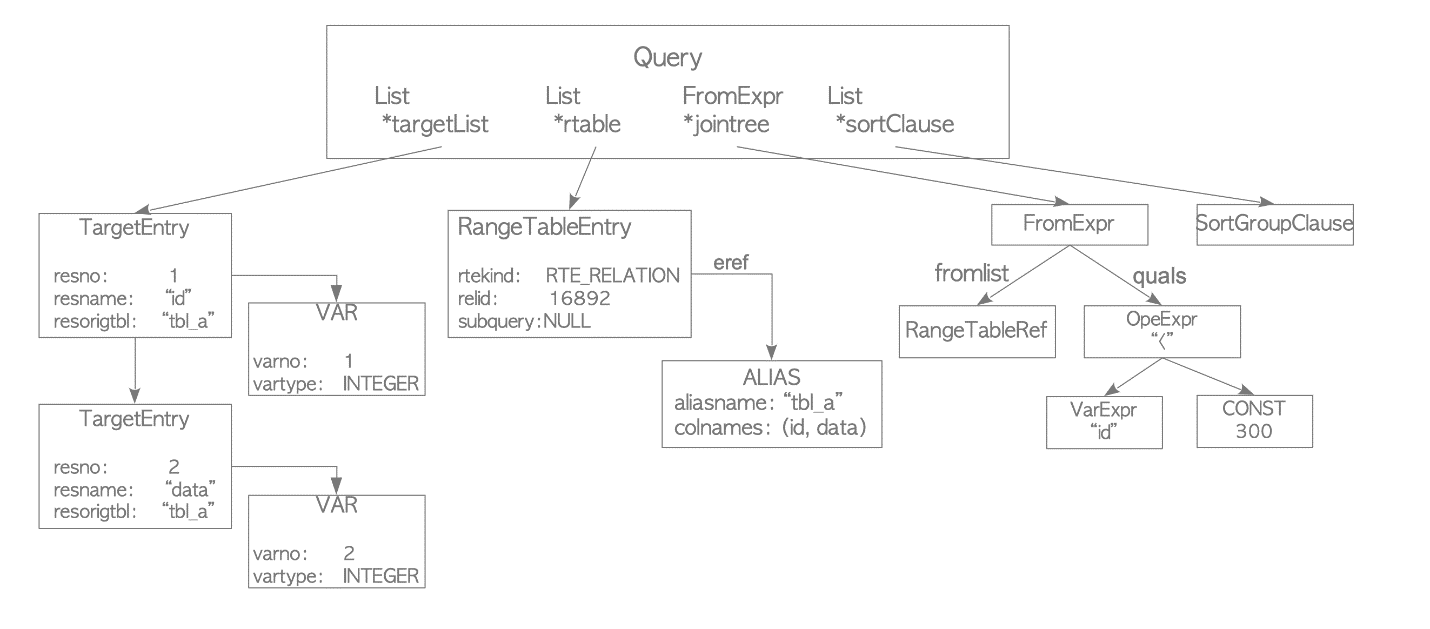
Slika 3.2 – primer parse tree-ja

Parser prilikom generisanja parse tree-ja gleda samo sintaksu, ne uzima u obzir validnost upita kao na primer da li tabela postoji u bazi (ove provere vrši analyzer). On vraća grešku samo u slušaju ako sintaksa upita ne valja.

Parse tree je [SelectStmt](https://github.com/postgres/postgres/blob/42dbbca58e8e87e461bb0a4fe48a450e90e1e932/src/include/nodes/parsenodes.h#L2001) struktura definisana unutar parsenodes.h.

### 3.2 Analyzer/Analyser

Analyzer vrši semantičku analizu nad parse tree-jem i generiše query tree koji je definisan strukturom [Query](https://github.com/postgres/postgres/blob/42dbbca58e8e87e461bb0a4fe48a450e90e1e932/src/include/nodes/parsenodes.h#L118) unutar parsenodes.h.



Slika 3.3 – primer query tree-ja

Struktura querry tree-ja je sledeća:

* **Command type** – nije prikazan na slici, je prosta vrednost koja ukazuje na tip komande (SELECT, INSERT, UPDATE, DELETE)
* **Result relation** – ovo je indeks range tabele u koju idu rezultati komande. Nije prikazan na dijagramu jer SELECT nema ovo polje.
* **Target list** – je spisak kolona koje su rezultat upita. U ovom slušaju to su sve kolone, *id* i *data.* U slučaju da je \* analyzer će je eksplicitno zameniti sa svim kolonama iz tabele. DELETE komanda nema target list.
* **Range table** – spisak relacija potrebnih za upit, drži informacije poput OID-a i imena tabele.
* **Join tree** – je spisak FROM ajtema. Ako je u pitanju *OUTER JOIN* join tree pokazuje strukturu JOIN-a. Restrikcije povezane sa join-ovanjem se pamte kao kvalifikacione ekspresije vezane za join ekspresije, tako da join tree zapravo pamti JOIN i FROM klauzule SELECT-a.
* **Sort clouse** – je spisak klauzula za sortiranje
* **The others** – još klauzula poput sort klauzule, koje neće biti analizirane u ovom dokumentu, takođe *rule system* unosi entitete tokom primenjivanja nekih od pravila.

## 3.3 Rewriter

Revriter je sistem koji primenjuje *rule system*, i transformiše query tree na osnovu pravila upamćenih unutar pg\_rules sistemskog kataloga ako je potrebno. Kako bi sprečili da ovaj deo dokumenta bude predugačak uzećemo slučaj kada je u pitanju komanda na VIEW-om.

Uzmimo u obzir da imamo već izgenerisan view sa komandom

CREATE VIEW employees\_list

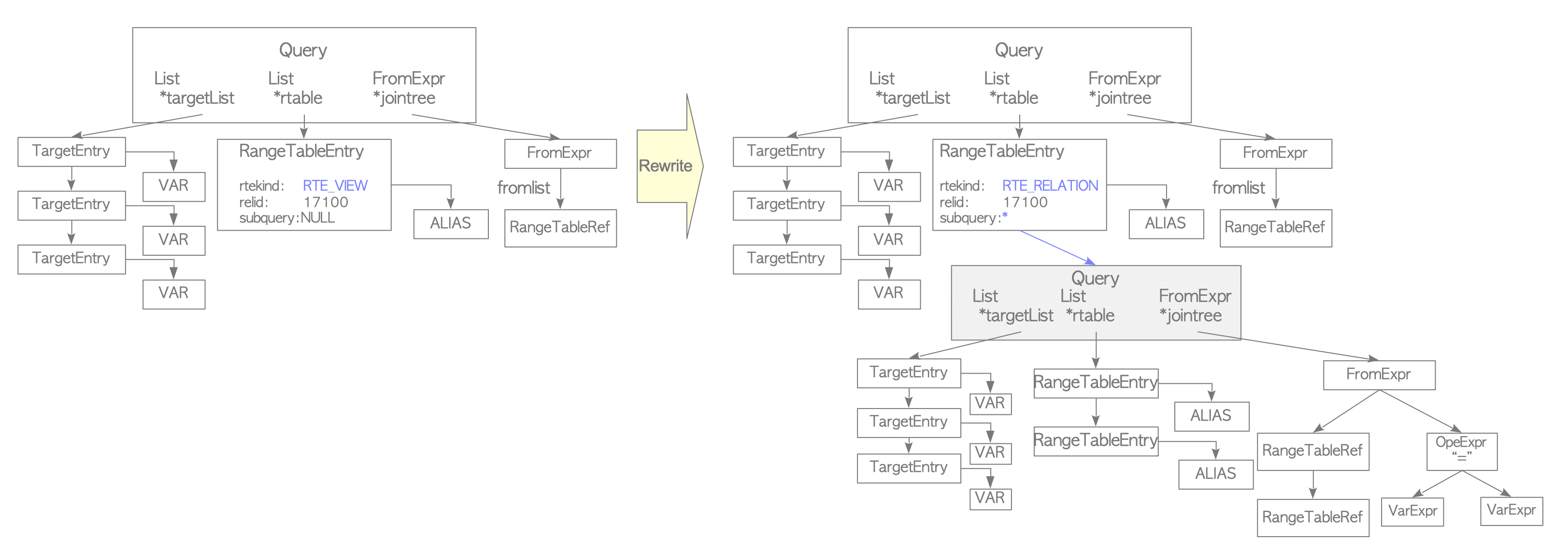
    AS SELECT e.id, e.name, d.name AS department

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

Kada izdamo komandu

SELECT \* FROM employees\_list;

Analyzer će izgenerisati query tree kao prikazan na slici 3.4 (a). Rewriter će procesirati range table čvor koji predstavlja traženi view sa subquery-jem (podupitom), koji generiše traženi view, upamćenom u pg\_rules.



Slika 3.3 – primena pravila nad query tree-jem

Prednost primenjivanja pravila je da se query tree prosiri svim dostupnim informacijama kako bi planner u sledećem koraku mogao da smisli što optimalniji način izvršenja traženog upita.

## 3.4 Planner i Executor

Planer dobija querry tree od rewriter-a i generiše plan tree koji je najoptimalniji executor-u za izvršenje.

Planner u PostgreSQL je baziran isključivo na ceni izvršenja, ne podržava se optimizacija na osnovu pravila ili nagoveštaja. Planner je najkompleksniji podsistem RDBMS-a.

Kao i drugim RDBMS sistemima, EXPLAIN komanda u PostgreSQL prikazuje plan tree.

EXPLAIN SELECT \* FROM tbl\_a WHERE id < 300 ORDER BY data;

                          QUERY PLAN

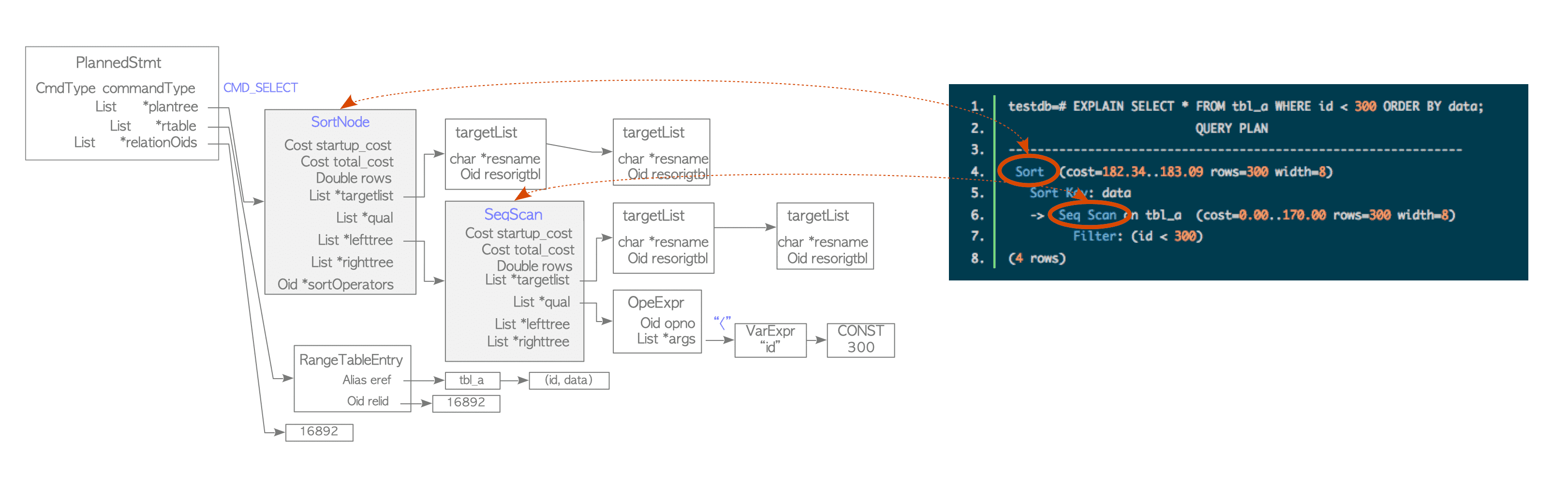
*---------------------------------------------------------------*

 Sort  (cost=182.34..183.09 rows=300 width=8)

   Sort Key: data

   ->  Seq Scan on tbl\_a  (cost=0.00..170.00 rows=300 width=8)

         Filter: (id < 300)



Slika 3.4 – primer plan tree-ja i veza sa EXPLAIN komandom

Plan tree se sastoji od *plan nodes* (čvor)i povezano je sa plantree listom unutar [PlannedStmt](https://github.com/postgres/postgres/blob/42dbbca58e8e87e461bb0a4fe48a450e90e1e932/src/include/nodes/plannodes.h#L43) strukturom definisanom unutar plannode.h . Svaki od čvorova sadrži informacije potrebne za rad executor-a, executor izvršava čvorove or kraja plan tree-ja do *root-a* (korena) u slučaju za upite vezane za samo jednu tabelu.

U primeru sa slike 3.4 se vidi lista čvorova sačinjena od *sort* i *sequental scan* čvorova, značeći da executor skenira tabelu *tbl\_a* pomoću sekvencijalnog skeniranja a zatim sortira dobijeni rezultat.

Pri izvršenju executor koristi memorijske prostore poput *temp\_buffers* i *work\_mem* alociranih unapred i kreira privremene datoteke po potrebi.

Uz to kada izvršava pristup podacima, PostgreSQL koristi *concurrency control mechanism,* o kome će biti reči kasnije u dokumentu, za održavanje konzistentnosti i izolacije pokrenutih transakcija.

### 3.4.1 Procena cene u single-table upitima

PostgreSQL bazira optimizaciju upita na osnovu cene. Cena je bezdimenziona vrednost, računa se na osnovu funkcija defenisanih unutar [costsize.c](https://github.com/postgres/postgres/blob/master/src/backend/optimizer/path/costsize.c) , na primer izračunavanje cene sekvencijalnog skeniranja i indeksnog skeniranja su *cost\_seqscan*() i *cost\_index*() redom.

U PostgreSQL postoje pri tipa cena: **start-up**, **run** i **total**. Gde je total zbir start-up i run cena, samim tim samo one dve se nezavisno estimiraju.

* Start-up cena je cena koja se mora platiti pre nego što se pribave prvi podaci
* Run cena je cena nabavke celog skupa podataka
* Total je zbir start-up i run cena.

EXSPLAIN komanda prikazuje cenu svih čvorova, u narednom primeru se vide informacije za sekvencijalno skeniranje. Postoje dve vrednosti 0.00 i 145.00, to znači da su start-up i run 0.00 i 145.00 redom.

U nastavku analiziraćemo kako se estimira cena sekvencijalnog skeniranja. U primeru se koristi sledeća tabela

CREATE TABLE primer\_tbl (id int PRIMARY KEY, data int);

CREATE INDEX primer\_data\_idx ON primer\_tbl (data);

INSERT INTO primer\_tbl SELECT generate\_series(1,10000),generate\_series(1,10000);

#### 3.4.1.1 Sekvencijalno skeniranje

Cena sekvencijalnog skeniranja se rašuna metodom *cost\_seqscan(),* istražićemokako se estimira cena sekvencijalnog skeniranja na osnovu sledećeg upita

SELECT \* FROM primer\_tbl WHERE id < 8000;

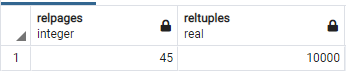
Kod sekvencijalnog skeniranja start-up cena je 0, dok se run cena dobija sledećim izrazom:

‘run cost’ = ‘cpu run cost’ + ‘disk run cost’

= (cpu\_tuple\_cost+cpu\_operator\_cost) × Ntuple + seq\_page\_cost × Npage,

Gde su seq\_page\_cost, cpu\_tuple\_cost i cpu\_operator\_cost podešeni unutar postgresql.conf fajla i difoltne vrednosti su 1.0, 0.01 i 0.0025 redeom, Ntuple i Npage su broj redova i stranica unutar tabele, njihove vrednosti možemo dobiti komadom:

SELECT relpages, reltuples FROM pg\_class WHERE relname = 'primer\_tbl'



Kada zamenimo vrednosti dobijamo

‘run cost’ = (0.01 + 0.0025) x 10000 + 1.0 x 45 = 170.0

‘total cost’ = 0.0 + 170.0 = 170.0

Sto se i poklapa sa rezultatom EXPLAIN komande:



### 3.4.2 Kreiranje plan tree-ja za single-table upite

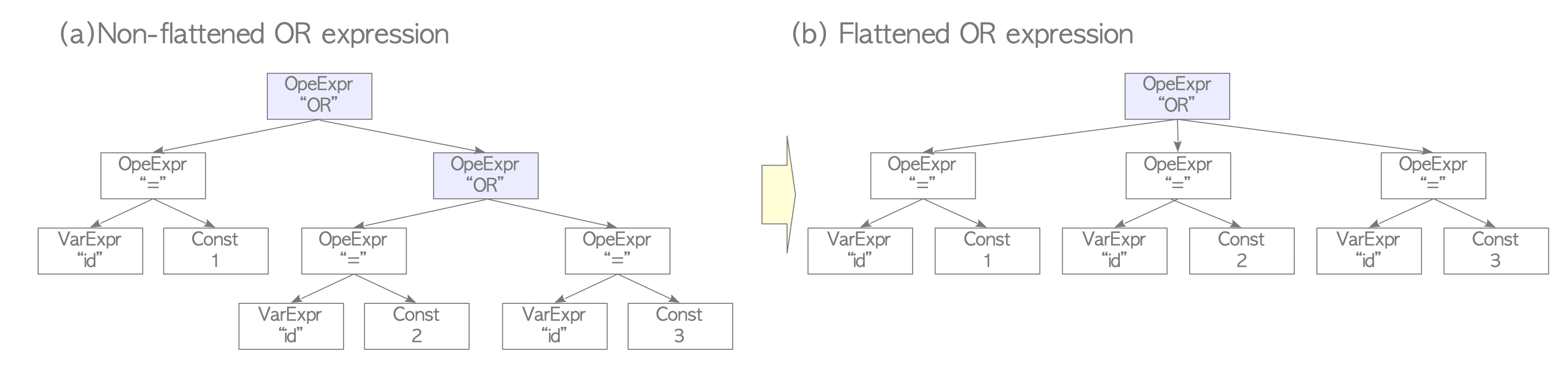
Planner u PostgreSQL prolazi kroz tri faze:

1. Izvrši preprocesiranje.
2. Pronađi najjeftiniji pristup (*access path*) stimiranjem cena svih mogućih pristupa
3. Kreirati plan-tree baziran na pronađenim najjeftinijim pristupom

#### 3.4.2.1 Preprocesiranje

Pre kreiranje plan tree-ja, planner izvršava preprocesiranje query tree-ja, koraci preprocesiranja su:

1. Uprostiti target list, limit klauzule i slično, npr. ‘2 + 2’ se pretvara u ‘4’ od strane *eval\_const\_expressions()*metode
2. Normalizacija *boolean* izraza, npr. ‘NOT (NOT a)’ se pretvara u ‘a’
3. Poravnanje AND/OR izraza. AND i OR operatri su standardni binarni operatori SQL-u, međutim u unutrašnjosti PostgreSQL oni su vektorni i planner podrazumeva da svi ugnježdeni AND-OR operatori se trebaju poravnjati. Sledi specifičan primer izraza *‘(id = 1) OR (id = 2) OR (id = 3)’.* Slika 3. Prikazuje igled stabla pre i posle poravnjanja



Slika 3.5 – poravnjanje and/or izraza

#### 3.4.2.2 Pronalaženje najjeftinijeg pristupa

Kako bi se našao najjeftiniji pristup (access path), planer procenjuje cenu svih mogućih pristupa dodajući pristupa i njegove cene u RelOptInfo strukturu i bira najjeftiniji. Sledi primer rada za upit i tabelu:

 SELECT \* FROM tbl\_2 WHERE id < 240;

Table "public.tbl\_2"

Column | Type | Modifiers

--------+---------+-----------

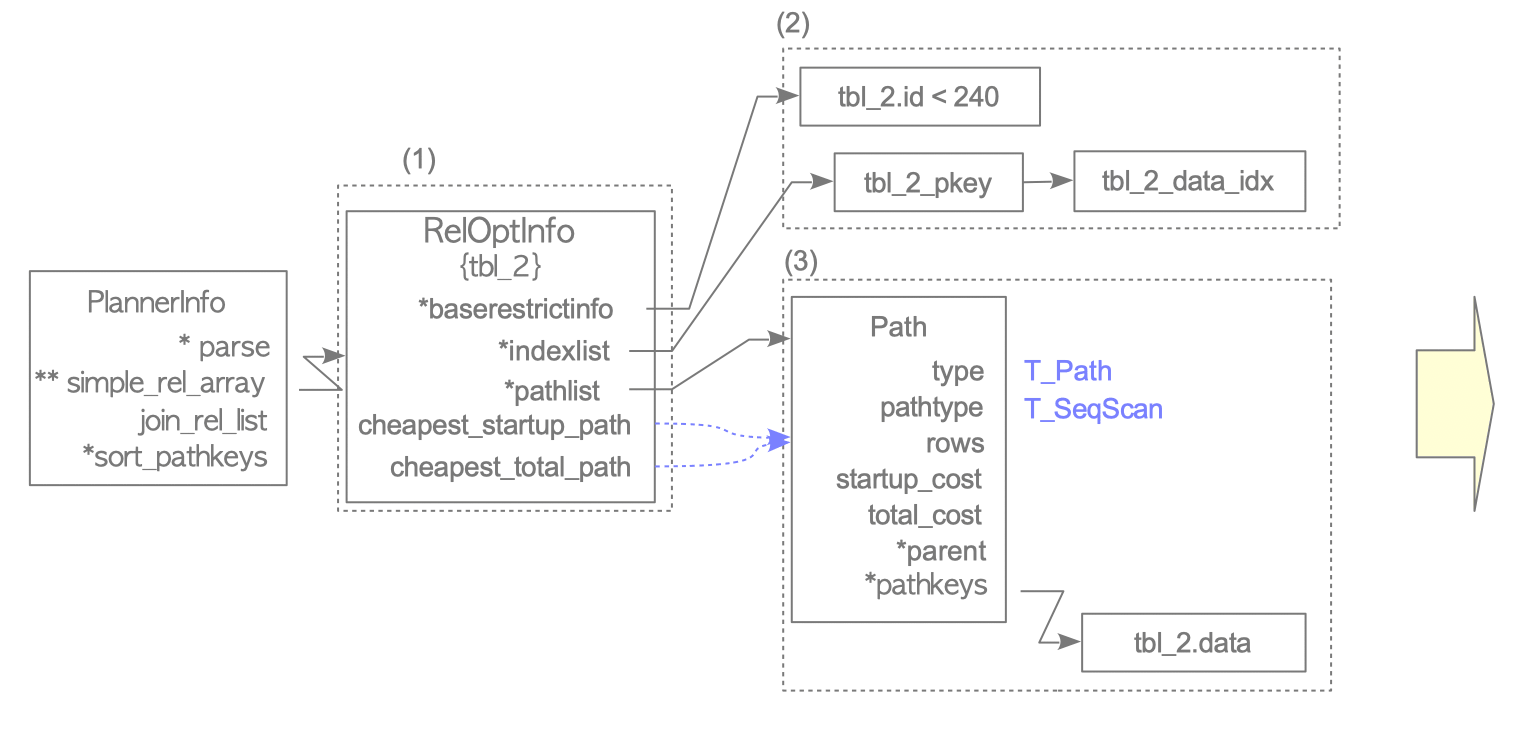
id | integer | not null

data | integer |

Indexes:

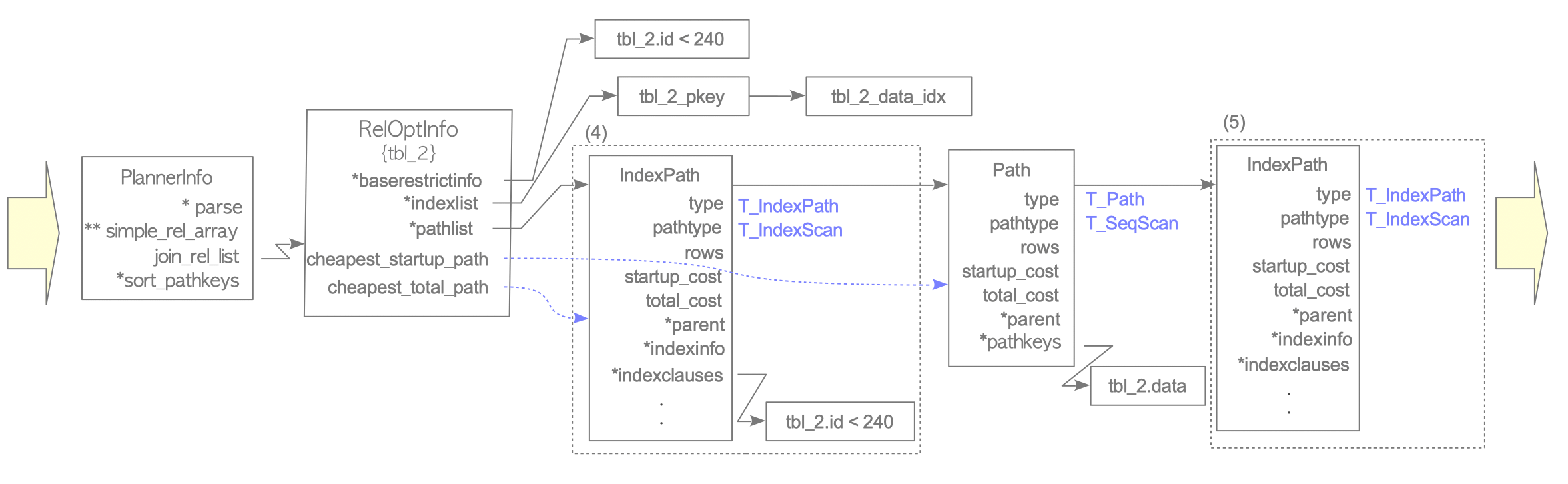
"tbl\_2\_pkey" PRIMARY KEY, btree (id)

"tbl\_2\_data\_idx" btree (data)



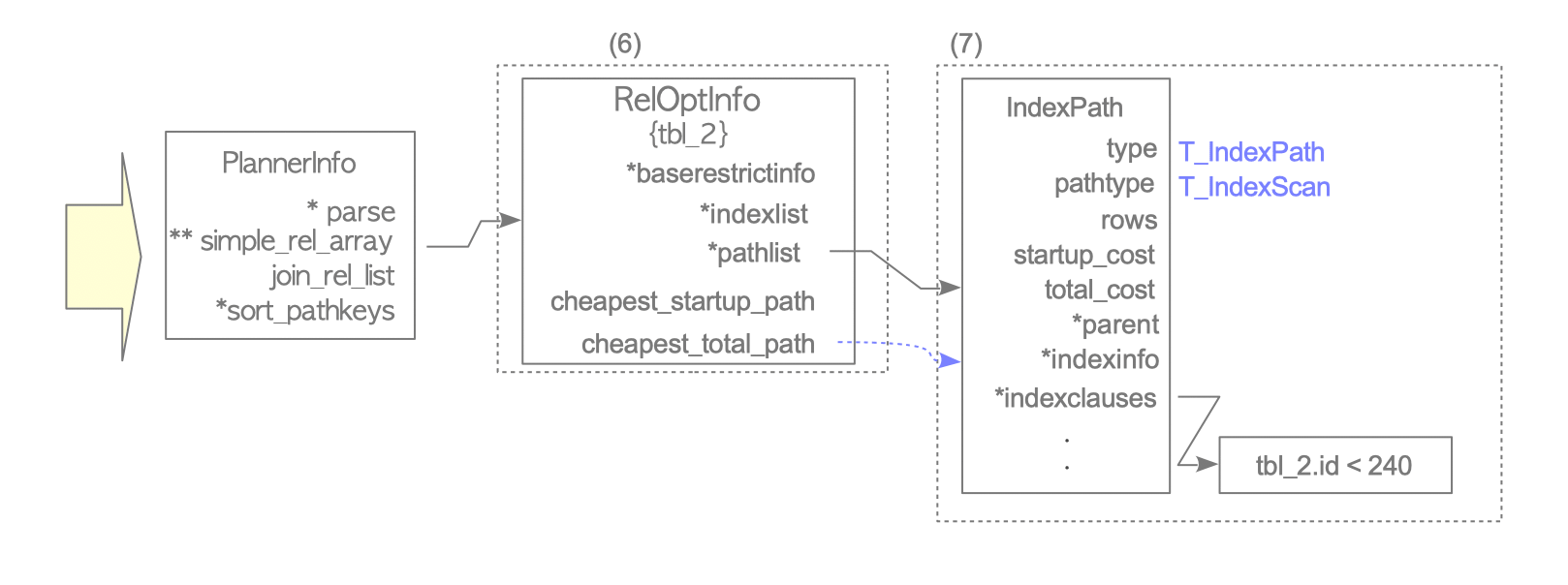
Slika 3.6 – primer pronalaženja najjefitnijeg pristupa

1. Kreirati RelOptInfo strukturu
2. Dodati WHERE klauzulu u *baserestrictinfo* i indeksi ciljne tabele u *indexlist.* U ovom slučaju WHERE clause ‘id < 240’ je dodat u baserestrictinfo, a dva indeksa, tbl\_2\_pkey i tbl\_2\_data\_idx,su dodati u indexlist.
3. Kreirati putanjui estimirati cenu sekvencijalnog skeniranja i dodati putanju u *pathlist.*



Slika 3.7 – nastavak primera pronalaženja najjefitnijeg pristupa

1. Kreirati *IndexPath,* estimirati cenu indeksnog skeniranja i dodati novodobijeni indexpath u pathlist. U ovom primeru postoje dva indeksa, tbl\_2\_pkey i tbl\_2\_data\_idx, koji će biti procesirani tim redosledom. Kreira se IndexPath za index tbl\_2\_pkey, estimiraju se njegova start-up i total cena, pošto se WHERE odnosi na kolonu id za koju je on indeks Where klauzula se dodaje u indexclauses datog IndexPath-a. Primetiti da kada se dodaje nova putanja u pathlist, lista se sortira na osnovu totalne cene u po rastućem redosledu.
2. Kreira se još jedan Index path za drugi indeks i dodaje se u pathlist. U ovom primeru nema ni jedna WHERE klauzula vezana za indeks tbl\_2\_data\_idx, samim tim su indeksne klauzule NULL.



Slika 3.8 – nastavak primera pronalaženja najjefitnijeg pristupa

1. Kreira se nova RelOptInfo struktura.
2. Dodaje se najjefitiniji pristup u novi RelOptInfo. U ovom primeru to je IndexPath vezan za tbl\_2\_pkey indeks.

#### 3.4.2.3 Kreiranje plan tree-ja

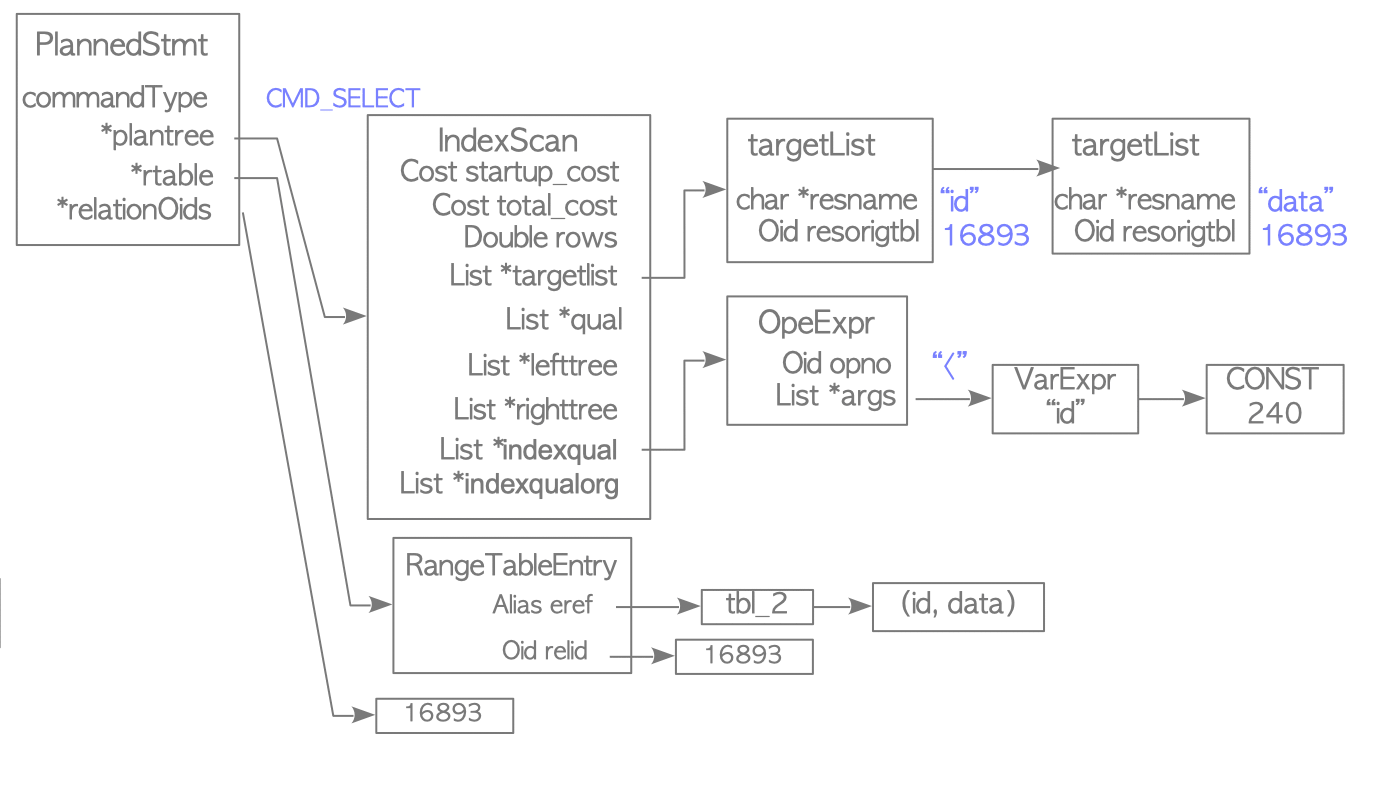
Koren plan tree-ja [PlannedStmt](https://github.com/postgres/postgres/blob/dec8ad367e46180f826d5b6dc820fbecba1b71d2/src/include/nodes/plannodes.h#L43) struktura, i ako sadrži devetnaest polja, četiri najbitnija su:

* **commandType** – pamti vrstu upita (SELECT, UPDATE, INSERT ...)
* **rtable** – pamti rangeTable-ove za upit
* **relationalOids** – pamti OID-e potrebne za upit
* **plantree** – pamti plan tree, gde su svaki čvor predstavlje neku operaciju (sekvencijalno skeniranje, indeksno skeniranje...)

PlanNode je osnovni čvor i drugi ga uvek sadrže, struktura sadrži četrnaest polja, sedam najvažnija su:

* **start-up cost** and **total\_cost** – estimirane cene za dati čvor.
* **rows –** proj redova za skeniranje estimiranih od strane planera.
* **targetlist –** sadrži targetList iz query tree-ja
* **qual –** lista koja pamti kvalifikacione uslove.
* **lefttree** i **righttree** – čvorovi za dodavanje čvorove potomke.

Sledi slika plan tree-ja baziranog na primeru iz prethodnog poglavlja.



Slika 3.9 – primer plan tree-ja

### 3.4.3 Kako radi Executor

U single-table upitima executor uzima čvorove plan tree-ja od krajeva prema korenu i poziva funkcije koje izvršavaju procesiranje datog čvora. Svaki *plan node* ima funkciju za izvršenje date operacije, nalaze se u [src/backend/executor/](https://github.com/postgres/postgres/blob/master/src/backend/executor/) direktorijumu. Pročitaćemo izlaz EXPLAIN komade i izvršiti analizu.

EXPLAIN SELECT \* FROM tbl\_1 WHERE id < 300 ORDER BY data;

                          QUERY PLAN

*---------------------------------------------------------------*

 Sort  (cost=182.34..183.09 rows=300 width=8)

   Sort Key: data

   ->  Seq Scan on tbl\_1  (cost=0.00..170.00 rows=300 width=8)

         Filter: (id < 300)

* Executor prvo izvršava operaciju sekvencijalnog skeniranja korišćenjem funkcija definisanim unutar [nodeSeqscan.c](https://github.com/postgres/postgres/blob/master/src/backend/executor/nodeSeqscan.c).
* Zatim, izvršava sortiranje rezultata sekvencijalnog skeniranja korišćenjem funkcija unutar [nodeSort.c](https://github.com/postgres/postgres/blob/master/src/backend/executor/nodeSort.c).

# 4. Kontrola konkurentnosti

Kontrola konkurentnosti je mehanizam za održavanje atomičnosti i izolacije, koji su dve stavke ACID-a, kada se izvršavaju više upita istovremeno nad bazom.

Postoje tri glavne vrste tehnika za kontrolu konkurentnosti: *Multi-version Concurrency Control* (MVCC), *Strict Two-Phase Locking* (S2PL) i *Optimistic Concurrency Control* (OCC). Svaki od njih ima svoje varijacije.

Kod MVCC -a svaka operacija upisa kreira novu verziju podatka i pamti staru. Kada transakcija zahteva šitanje tog podatka, sistem bira jednu od tih verzija kako bi osigurao izolaciju, sto je glavna prednost MVCC-a, čitanje ne blokira upis i obrnuto. PostgreSQL koristi varijaciju MVCC-a po imenu **Snapshot Isolation (SI).**

Za implementaciju SI-a, RDBMS sistemi poput Oracle-a korsite rollback segmente – pri upisu novog podatka, stari se upisuje u rollback segment, dok se novi upisuje u data delu. PostgreSQL koristi sličan pristup, novi podatak se upisuje u relevantnu table stranicu, pri šitanju podatka PostgreSQL bira prigladnu verziju podatka korišćenjem **pravila provere vidljivosti** (**visibility check rules).** SI ne dozvoljava tri anomalije defenisane ANSI SQL-92 standardu, to su: *Dirty Reads*, *Non-Repeatable Reads*, i *Phantom Reads*. Kasnije je dodata i zaštita od anomalija serijalizacije - **Serializable Snapshot Isolation (SSI)**.

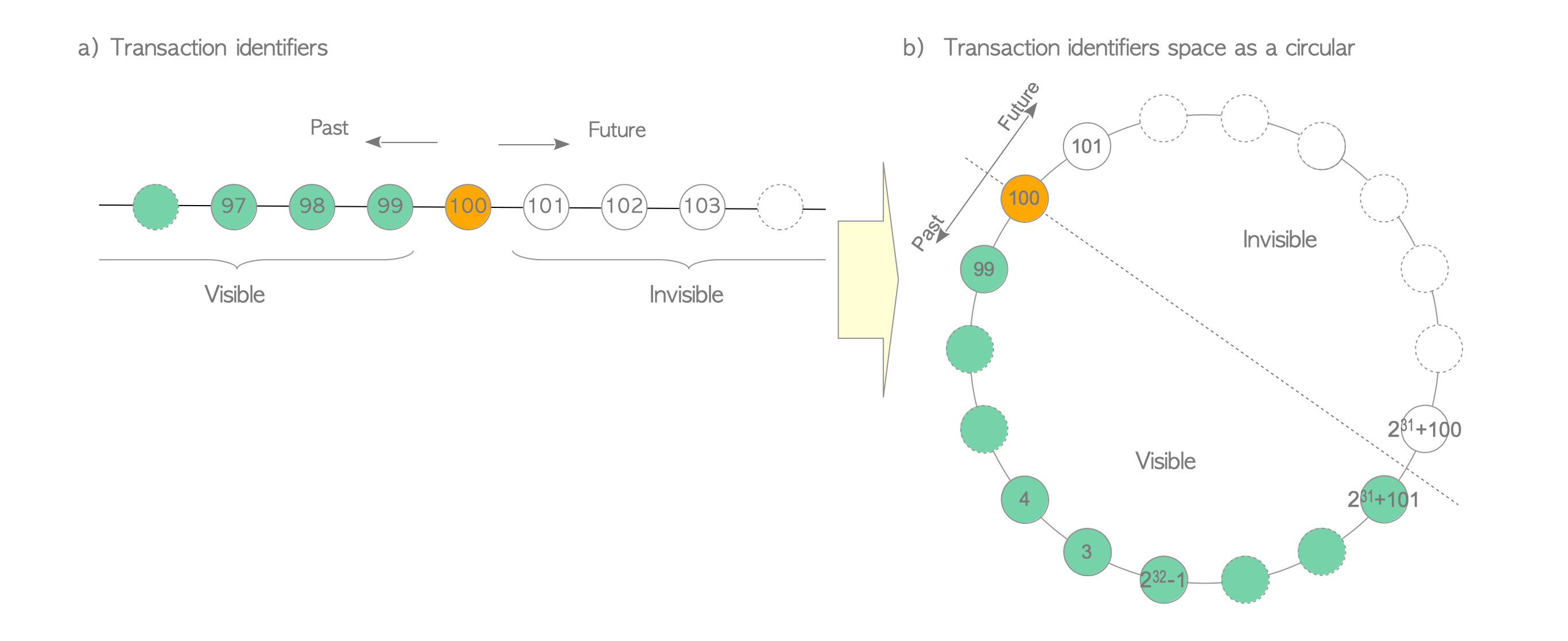
## 4.1 ID transakcije

Kad god se zapošne transakcija, njoj se dodeli ID transakcije **(transaction ID)** od strane menadžera transakcija. PostgreSQL-ov txid je 32-bitni neoznačeni integer, nakon početka transakcije txid možemo pročitati korišćenjem txid\_current() metode.

Postoje tri specijalna txid-ja:

* 0 – znači invalidni txid
* 1 – Bootstrap txid, koristi se samo kod inicijalizacije klastera
* 2 – Zamrznut txid

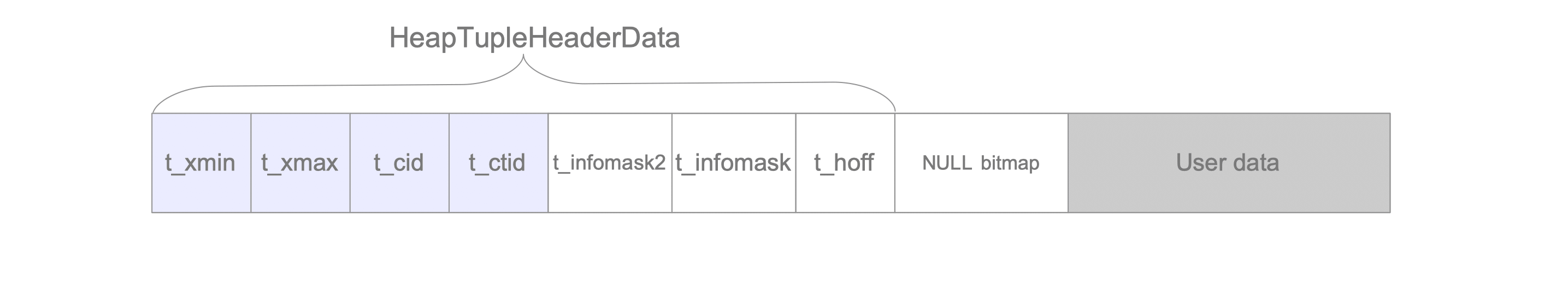
Txid-jevi su uporedivi - značući da za txid od vrednosti N, txid-jevi sa vrednošću manjom od N su u prošlosti a sa većom su u budučnosti



Slika 4.1 – organizacija identifikatora transakcija

## 4.2 Struktura tuple-a

Heap tuple se sastoji od tri dela: HeapTupleHeaderData struktura, NULL bitmapa i korisnički podaci.

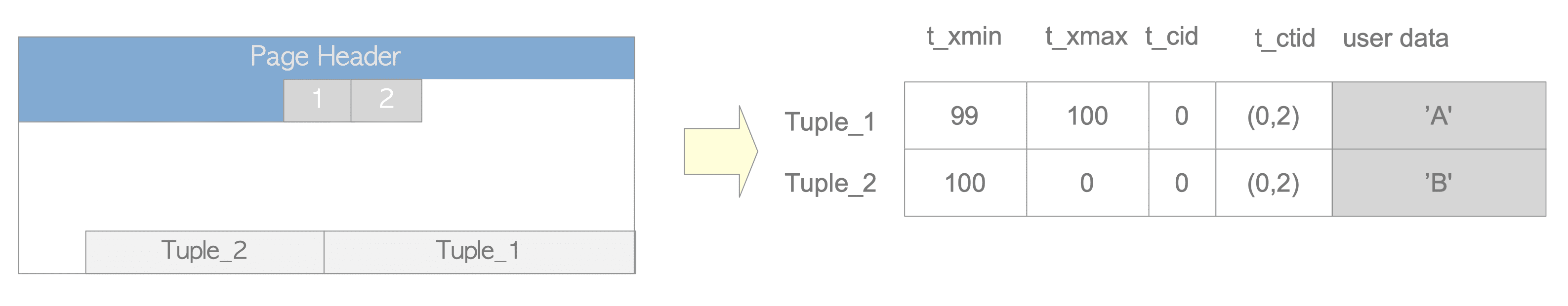


Slika 4.2 – struktura tuple-a

HeapTupleHeaderData ima sedam polja, ali su nam četiri od interesa:

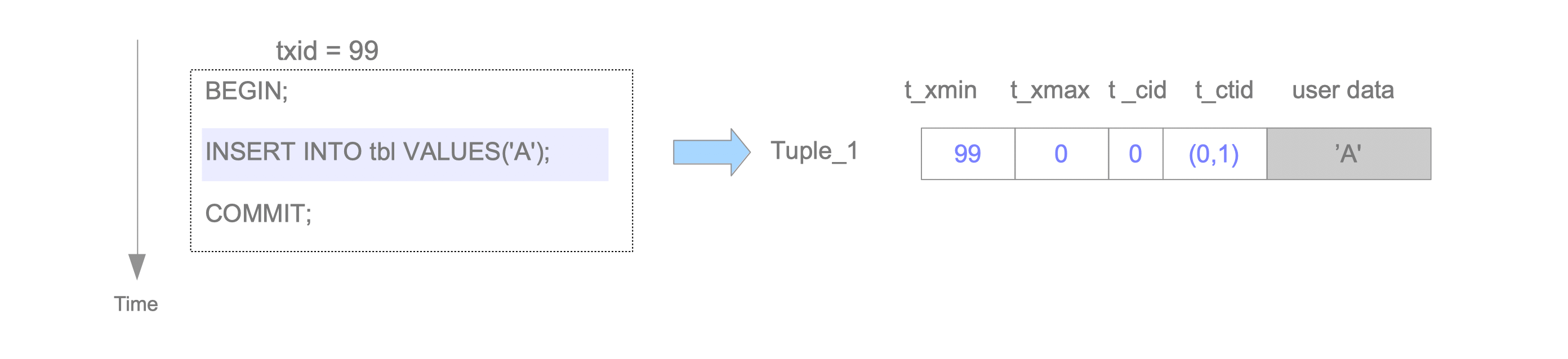
* **t\_xmin** – txid transakcije koja je umetnula tuple
* **t\_xmax** – txid transakcije koja je obrisala ili ažurirala tuple. Ako je neuspelo brisanje ili ažuriranje vrednost je 0, što je stanje INVALID.
* **t\_cid** – predstavlja command id, koji označava koliko SQL komandi je izvršeno pre nego što se izvršila data komanda u okviru jedne transakcije.
* **t\_ctid** –identifikor tuple-a, opisuje tuple unutar tabele. Kada se tuple ažurira, t\_ctid ukazuje na novi tuple, u suprotnom ukazje na sebe.

U nastavku slede ilustracije umetanja, brisanja i ažuriranja tuple-ova.



Slika 4.3 – reprezentacija tuple-a

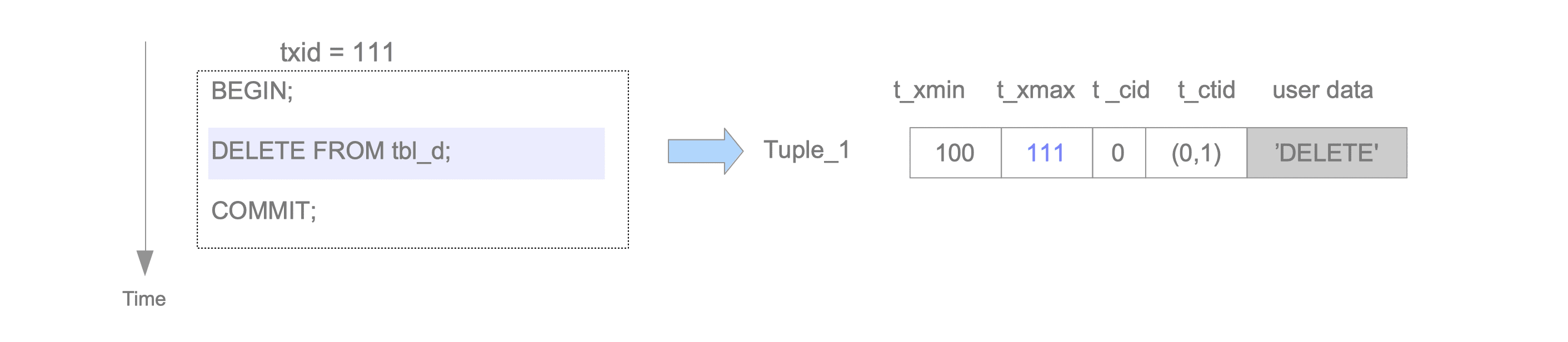
### 4.2.1 Upis



Slika 4.4 – upis tuple-a

**t\_xmin** je 99 jer je to txid koji je izvršio upis, **t\_xmax** je 0 jer nije došlo do brisanja ili ažuriranja tuple-a, **t\_cid** je 0 jer je prva komanda unutar transakcije, **t\_ctid** je (0, 1) jer ukazuje na sebe.

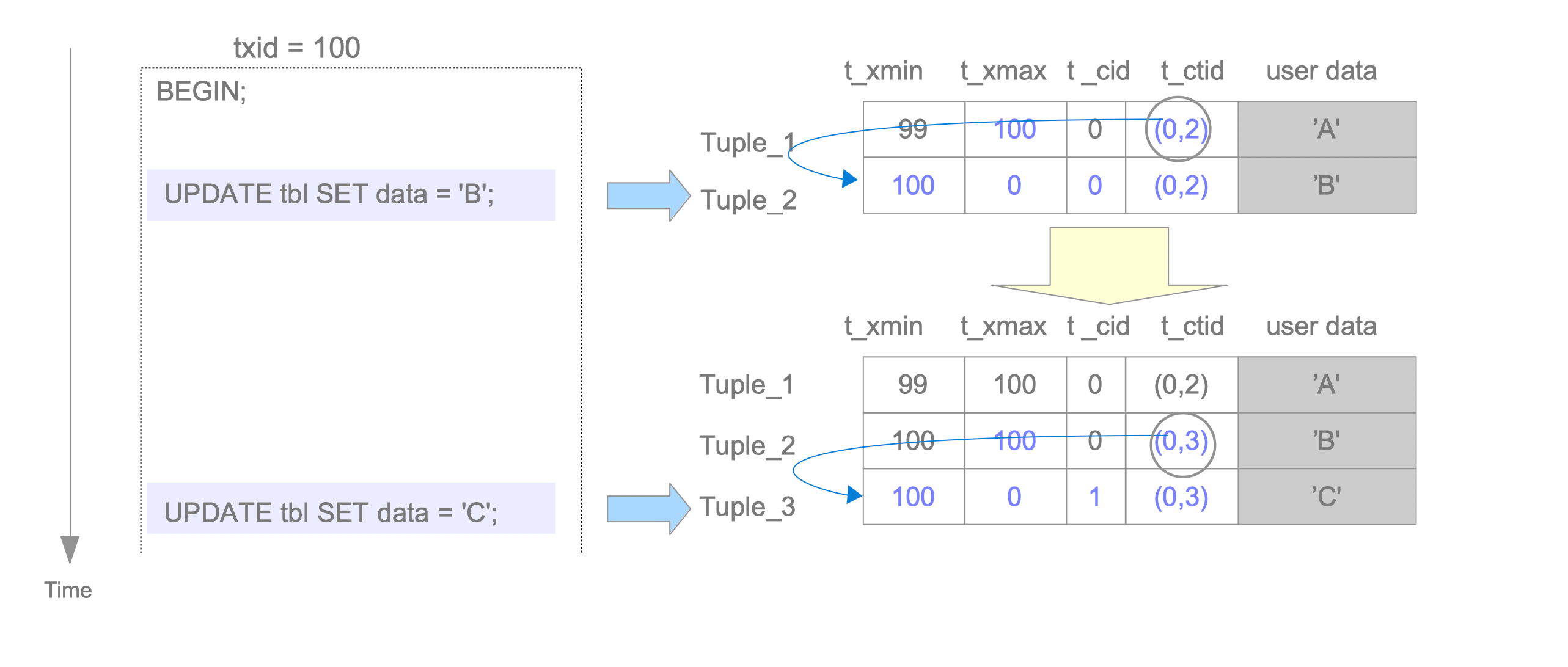
### 4.2.2 Brisanje



Slika 4.5 – brisanje tuple-a

**t\_xmax** je 111 jer je to txid transakcije koja je izvršila brisanje.

### 4.2.3 Ažuriranje



Slika 4.6 –ažuriranje tuple-a

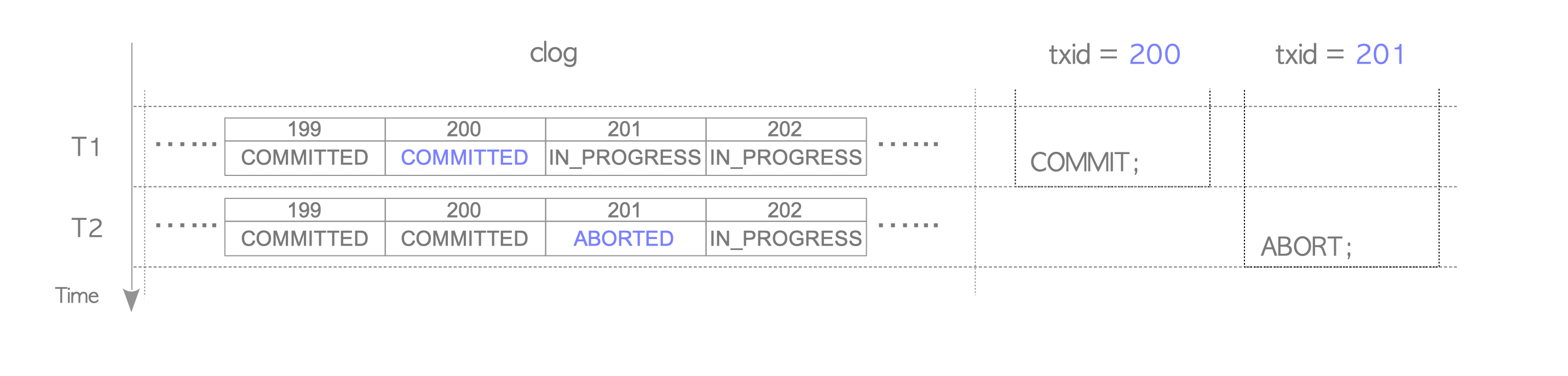
Izvršenjem prve updejt komande tuple\_1 je logiški izbrisanpostavljanje t\_xmax na 100, tuple\_2 je upisan i t\_cid polje unutar tuple\_1 je ažurirano da ukazuje na tuple\_2.

## 4.3 Commit log

PostgreSQL pamti strukturu transakcija unutar commit log-a, poznat kao i **clog.** Nalazi se unutar deljene memorije i koristi se za obradu svih transakcija.

Postoje četiri stanja transakcija: IN\_PROGRESS, COMMITTED, ABORTED, i SUB\_COMMITTED.

Clog se sastoji od jedne ili više stranica od 8 KB unutar deljene memorije i formira niz transakcionih identifikatora i statusa date transakcije.



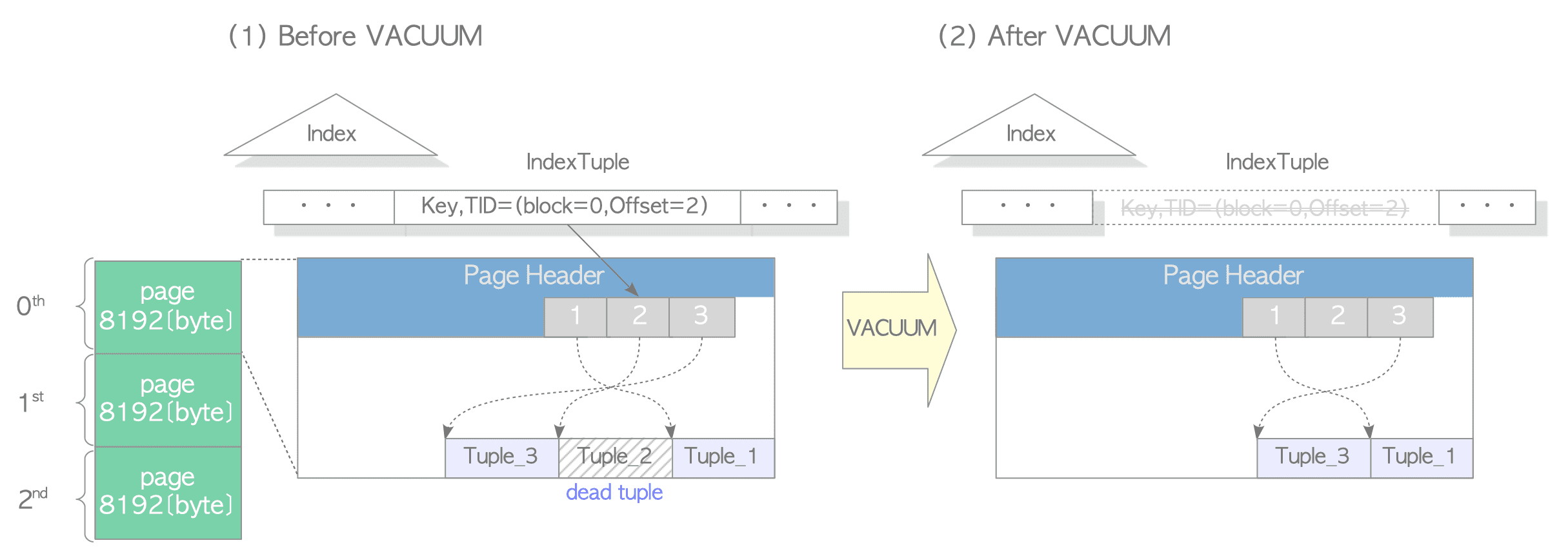
Slika 4.6 –struktura clog-a

Kada se PostgreSQL gasi ili se pokrene checkpoint process, sadržaj clog-a se upisuje u datoteku unutar **pg\_xact** poddirektorijuma. Ovi fajlovi su imenovani 0000, 0001, 0002 itd. Maksimalna veličina datoteke je 255KB. Pri pokretanju PostgreSQL učitava se sadržaj datoteka, sa obzirom da se clog kontinualno povećava, vakuum procesiranje regularno priše zastarele podatke (stranice i datoteke).

# 5. Vakuum procesiranje

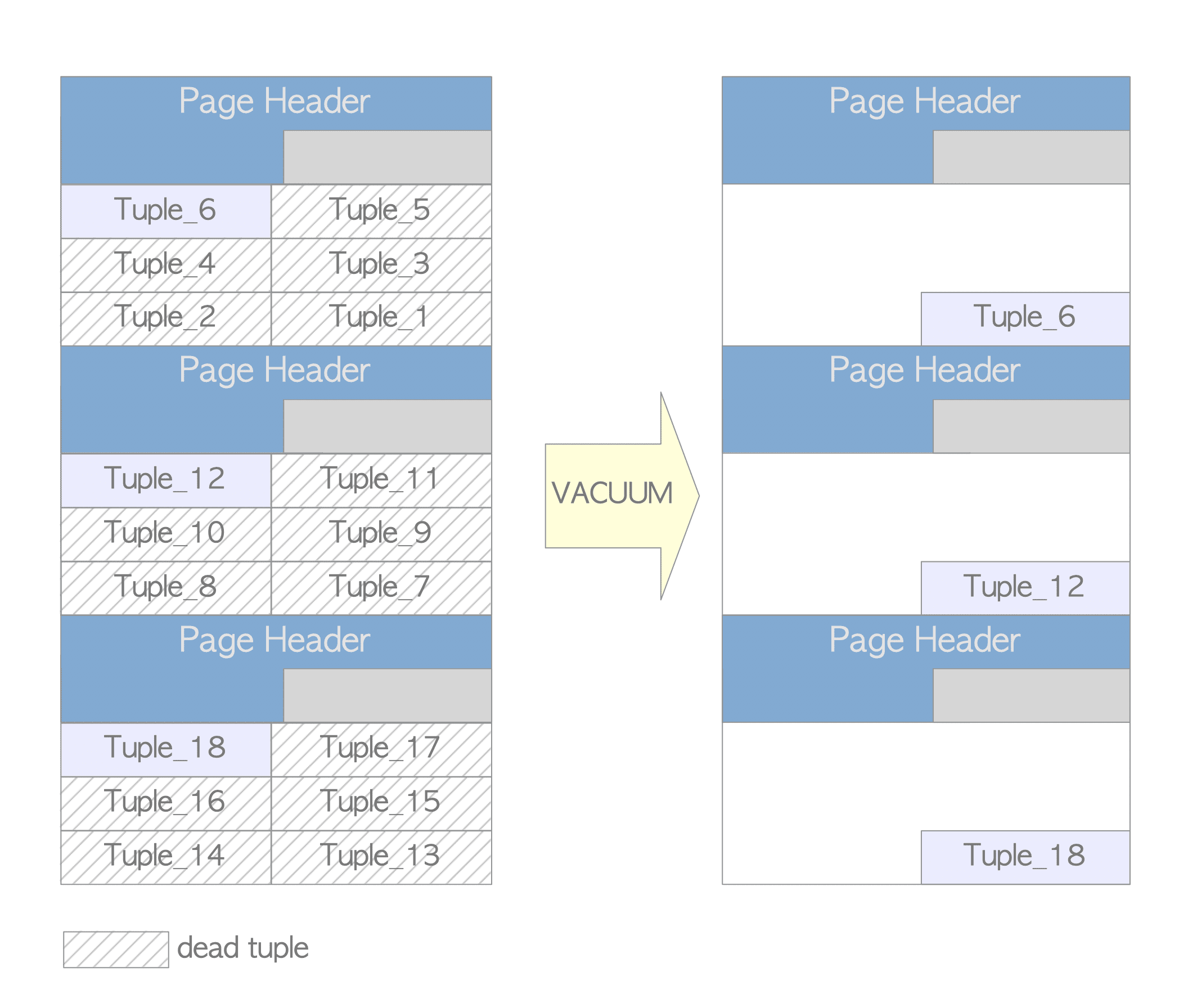
Vakuum procesiranje je proces održavanja unutar PostgreSQL-a, glavne obaveze su brisanje mrtvih tuple-ova i zamrzavanje transakcionih identifikatora.

Za brisanje tuple-ova postoje dva režima rada: **konkurentni** **(Concurrent)** i **potpuni** (**Full) VAKUUM.** Konkuretni vakuum (poznat kao i samo vakuum) briše mertve tuple-ove na svim stranicama tabele dok transakcije mogu čitati sadržaj tabele dok je proces u toku. Potpuni vakuum briše mrtve tuple-ove i vrši defragmentaciju tabele, transakcije ne mogu da čitaju tabele dok je proces u toku.



Slika 5.1 – izgled tabele pre i posle vakuum procesiranja

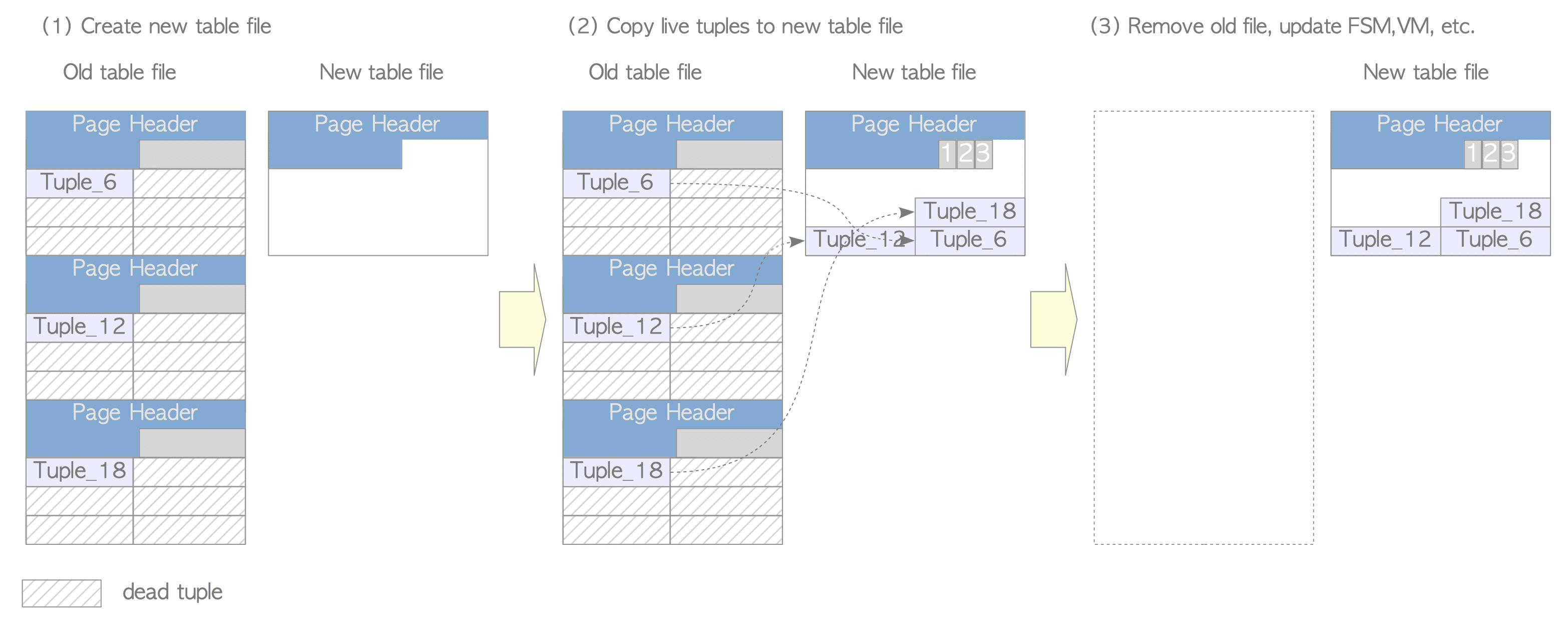
I ako je konkurentni vakuum potreban za funkcionisanje, nije sposoban da smanji veličinu tabele ako je u njoj postoje mnogo mrtvih tuple-ova. Slika 5.2 prikazuje primer toga. Mrtvi tuple-ovi su uklonjeni, veličina tabele nije promenjena što usporava rad baze i nepotrebno troši disk.



Slika 5.2 – loše strane konkurentnog vakuuma

Kako bi se ovaj problem rešio postoji potpuni vakuum:

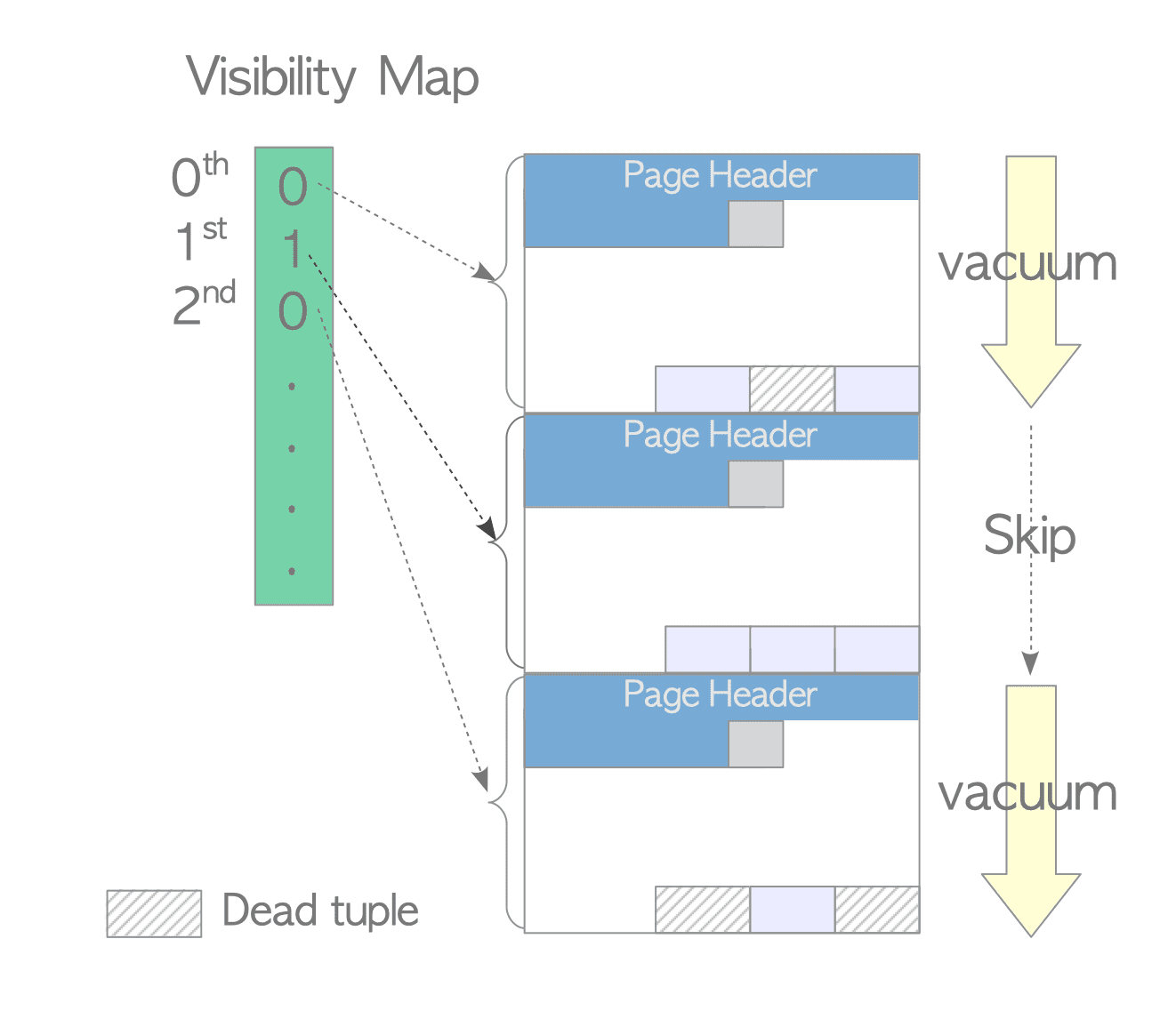
1. kreira se nova table datoteka.
2. Kopiraju se živi tuple-ovi u novu datoteku
3. Brišu se stari fajlovi, ažuriraju se indeksi, statistike, FSM i VM.



Slika 5.3 – ptpuni vakuum

## 5.1 Visbility map

Vakuum procesiranje je veoma skupo, kako bi se to popravilo postoje visibility mape (VM). Svaka tabela ima VM koja drži vidljivost svake stranice unutar table datoteke. Vidljivost stranice ustanovljava da li stranica ima mrtve tuple-ove, ako nema vakuum procesiranje može da preskoči datu stranicu. Svaka VM datoteka se sastoji od jedne ili više stranica od 8KB i pamti se sa sufiksom “vm”.



Slika 5.4 – kako se koristi visibility map

# 6. Linkovi

[PostgreSql 14 dokumentacija](https://www.postgresql.org/files/documentation/pdf/14/postgresql-14-A4.pdf)

[Hironobu SUZUKI - The Internals of PostgreSQL](https://www.interdb.jp/pg/)