Student: Jovan Mladenović 1277

Profesor: Aleksandar Stanimirović

PostgreSQL - Obrada transakcija, planovi izvršavanja transakcija, izolacija i zaključavanje

Sadržaj

[1. Uvod 3](#_Toc105104936)

[2. Obrada upita 3](#_Toc105104937)

[2.2 Parser 3](#_Toc105104938)

[2.3 Analyzer/Analyser 4](#_Toc105104939)

[2.4 Rewriter 5](#_Toc105104940)

[2.5 Planner i Executor 6](#_Toc105104941)

[2.5.1 Procena cene u single-table upitima 8](#_Toc105104942)

[2.5.2 Kreiranje plan stabla za single-table upite 8](#_Toc105104943)

[2.6 Izvršenje višetablarnih upita 12](#_Toc105104944)

[2.6.1 Preprocesiranje 12](#_Toc105104945)

[2.6.2 Pronalaženje najjeftinijeg pristupa 12](#_Toc105104946)

[2. Kontrola konkurentnosti 13](#_Toc105104947)

[2.1 ID transakcije 14](#_Toc105104948)

[2.2 Struktura reda tabele baze podataka 14](#_Toc105104949)

[2.2.1 Upis 15](#_Toc105104950)

[2.2.2 Brisanje 16](#_Toc105104951)

[2.2.3 Ažuriranje 16](#_Toc105104952)

[2.3 Commit log 16](#_Toc105104953)

[2.4 Transaction snapshot 17](#_Toc105104954)

[2.5 Pravila provere vidljivosti 18](#_Toc105104955)

[2.5.1 Status t\_xmin je ABORTED 18](#_Toc105104956)

[2.5.2 Status t\_xmin je IN\_PROGRESS 19](#_Toc105104957)

[2.5.3 Status t\_xmin je COMMITED 19](#_Toc105104958)

[2.6 Provera Vidljivosti 20](#_Toc105104959)

[2.7 Sprečavanje izgubljenih ažuriranja 21](#_Toc105104960)

[2.8 Freeze proces 22](#_Toc105104961)

# Uvod

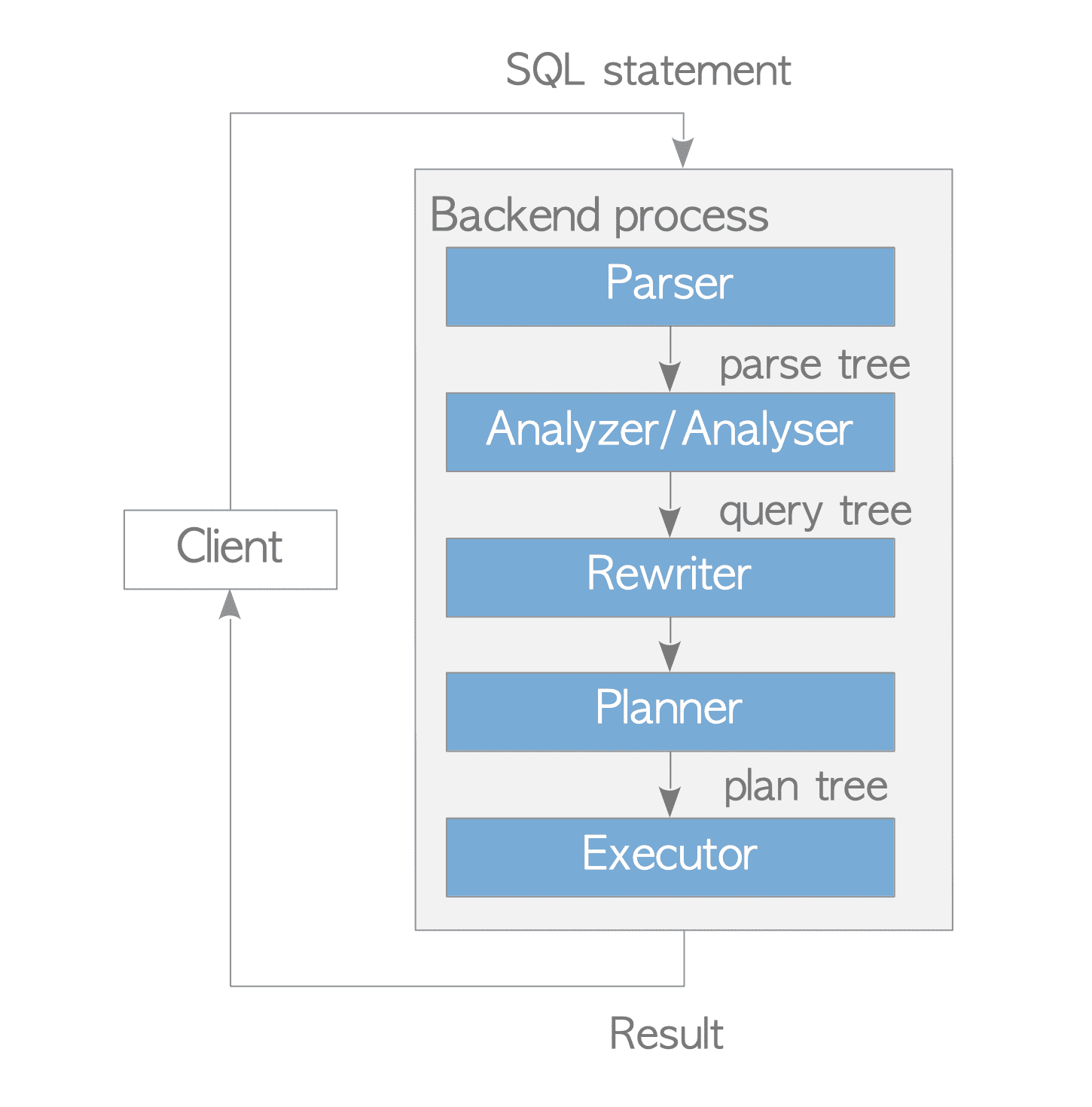
Transakcije su fundamentalni koncept svih DBMS sistema. Srž transakcija je da poveže više koraka u jednu sve ili ništa operaciju, ako dođe do grške koja sprečava transakciju od izvršenja onda nijedan od izvršenih koraka neće uticati na bazu. Takođe ako se uspešno izvrši transakcija, potrebno je da sistem permanentno upamti novo stanje baze i ne izgubi ga čak i ako dođe do prekida aplikacije ubrzo nakon izvršenja. Još jedna bitna osobina transakcija je da budu atomične, kada se više transakcija izvrčavaju istovremeno one međusobno ne bi trebale da vide nepotpune izmene.

# Obrada upita

Obrada upita je najkomplikovaniji podsistem PostrgresSQL i veoma je efikasno u procesiranju podržanog SQL-a.

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

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

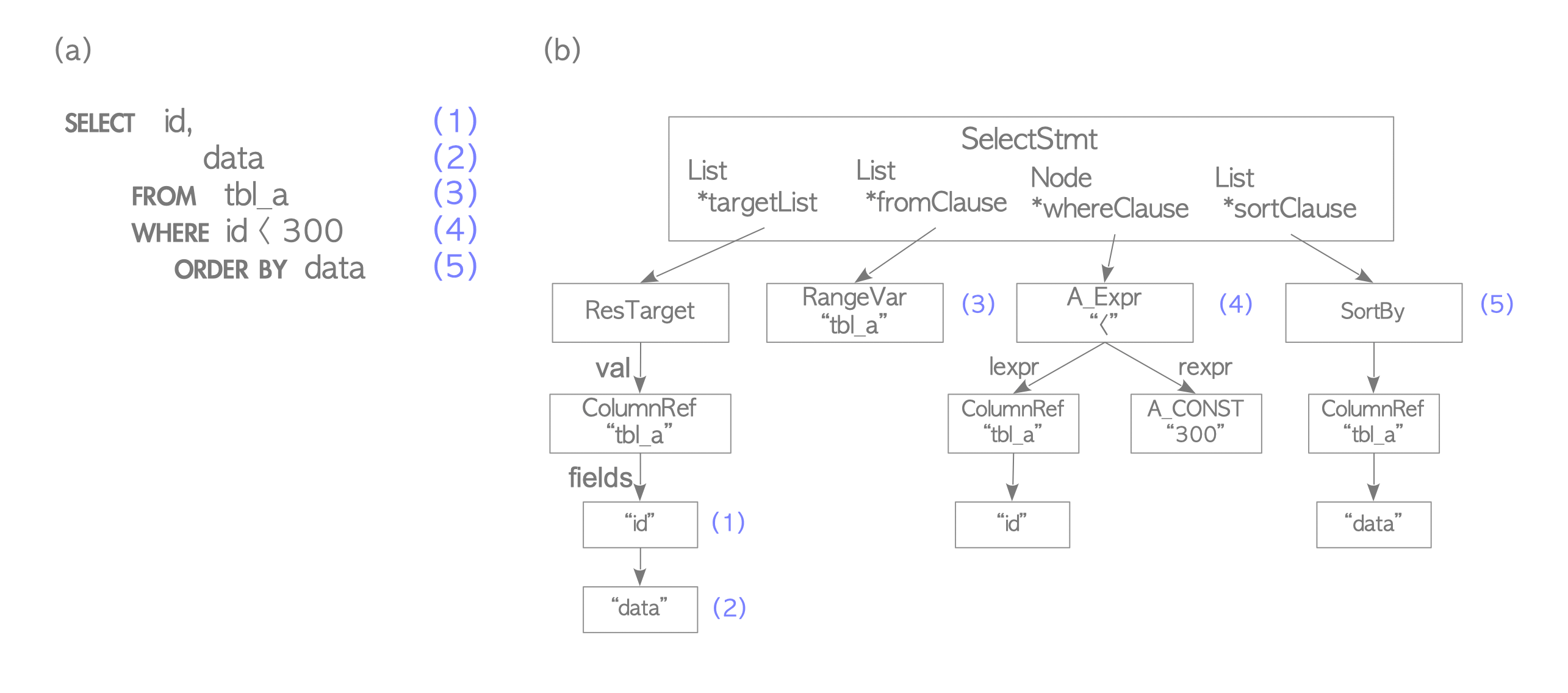


Slika 2.1 – procesiranje Quary-ja

## Parser

Parser generiše parse stablo koje mogu pročitati sledeći podsistemi na osnovu SQL izraza.

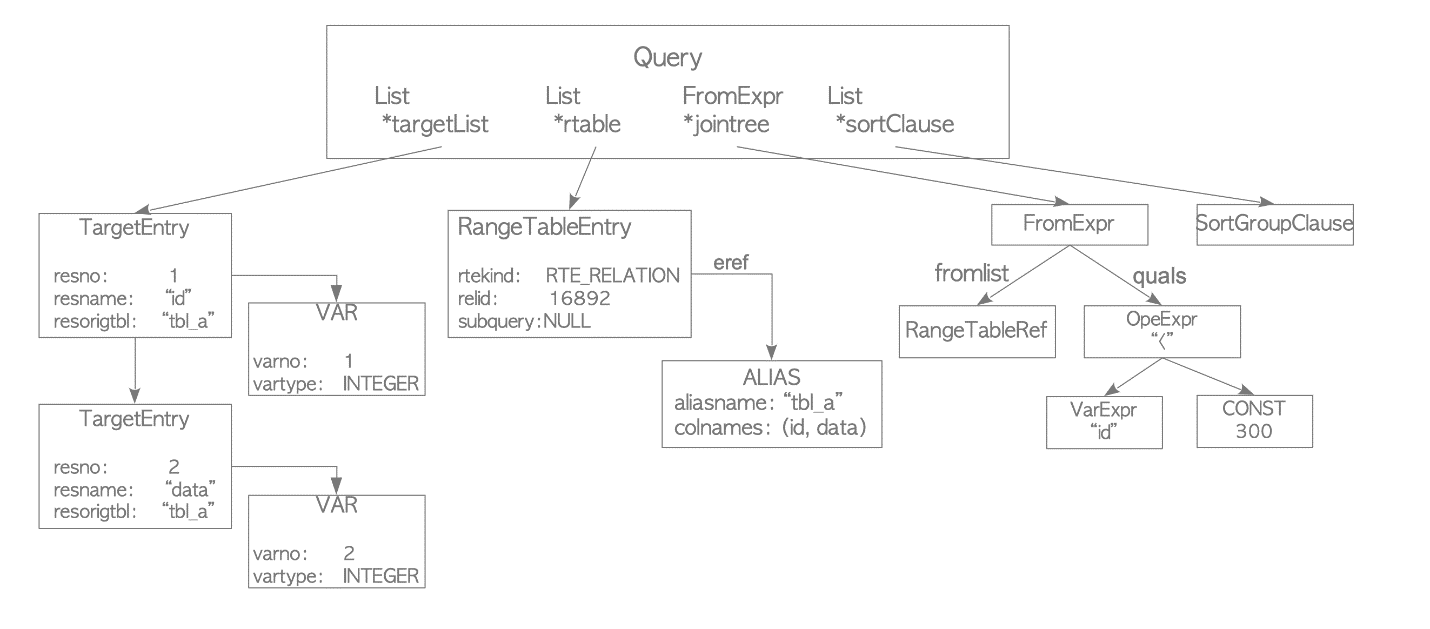
Parser prilikom generisanja parse stabla gleda samo sintaksu, ne uzima u obzir validnost upita kao na primer da li tabela postoji u bazi (ove provere vrši analyzer). On vraća grešku samo u slučaju ako sintaksa upita ne valja.



Slika 2.2 – primer parse stabla

## Analyzer/Analyser

Analyzer vrši semantičku analizu nad parse stablom i generiše query stablo.



Slika 2.3 – primer query stabla

Struktura querry stabla 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 stablo pokazuje strukturu JOIN-a. Restrikcije povezane sa join-ovanjem se pamte kao kvalifikacione ekspresije vezane za join ekspresije, tako da join stablo 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.

## Rewriter

Revriter je sistem koji primenjuje *rule system*, i transformiše query stablo 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

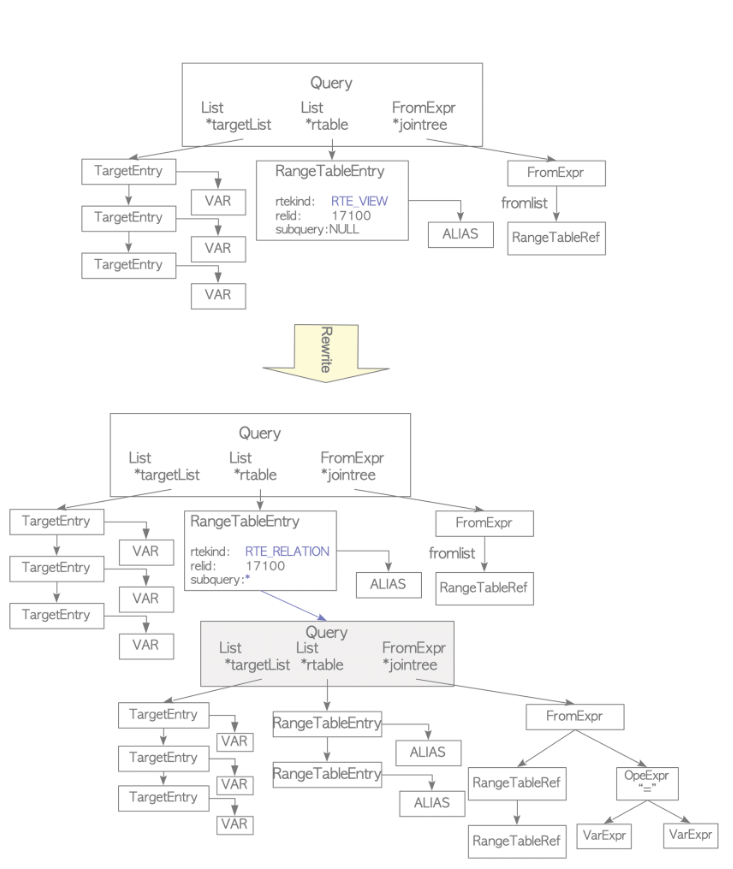
    AS SELECT e.id, e.name, d.name AS department

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

Kada izdamo komandu

SELECT \* FROM employees\_list;

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



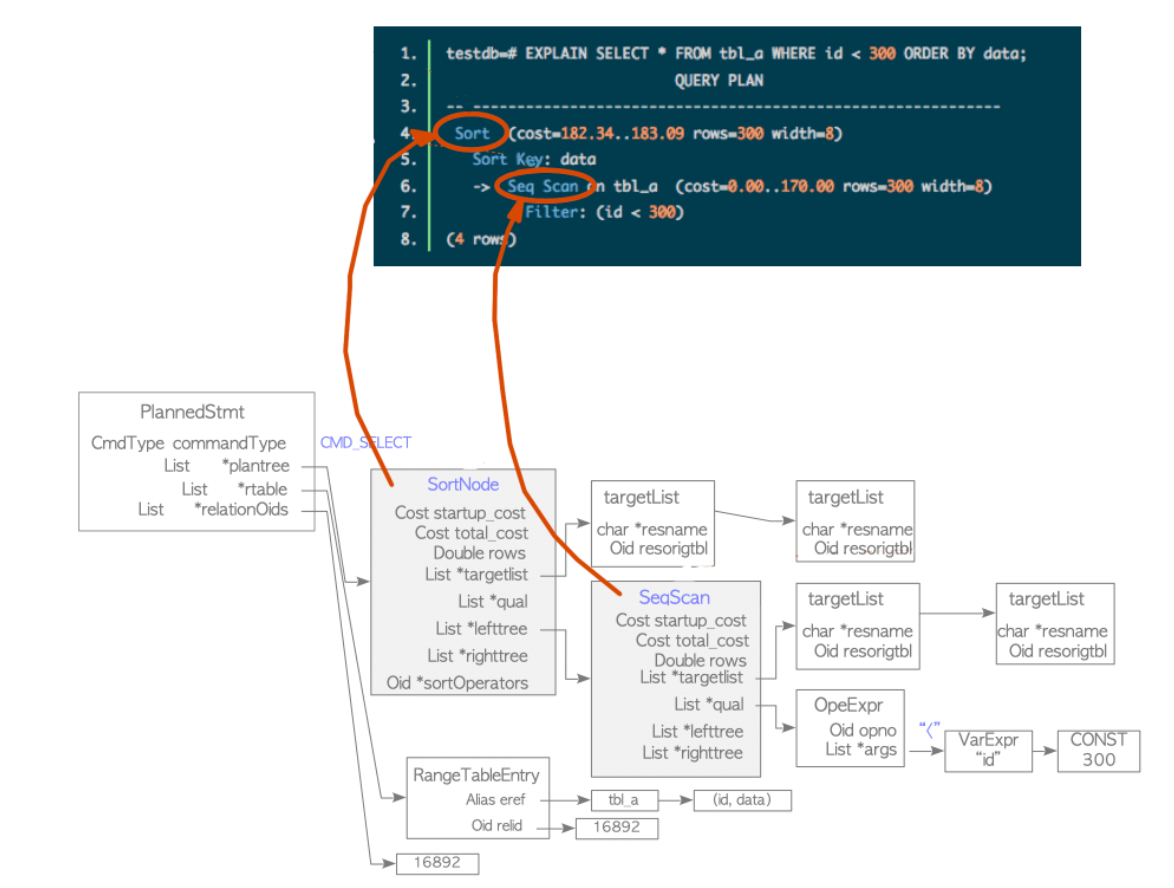
Slika 2.4 – primena pravila nad query stablom

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

## Planner i Executor

Planer dobija querry stablo od rewriter-a i generiše plan stablo koje je najoptimalniji executor-u za izvršenje. 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. Planner u PostgreSQL je baziran isključivo na ceni izvršenja, ne podržava se optimizacija na osnovu pravila ili nagoveštaja. Planner je najkompleksniji podsistem RDBMS-a.

EXPLAIN komanda u PostgreSQL prikazuje plan stablo.



Slika 2.5 – primer planstabla i veza sa EXPLAIN komandom

Plan stablo se sastoji od *plan nodes* (čvorova) 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 stabla do korena u slučaju za upite vezane za samo jednu tabelu.

U single-table upitima executor uzima čvorove plan stabla 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. U primeru sa slike 2.5 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 izdršenju executor koristi *work\_mem* i *temp\_buffers* koji su unapred alocirani u memoriji za izvršenje upita, međutim ako to nije dovoljno kreira se privremeni fajl koji se smešta unutar *base/pg\_tmp* poddirektorijuma. Korišćenjem ANALYZE opcije uz EXPLAIN komandu PostgreSQL izvršava upit i prikazuje pravi broj redova, vreme izvršenja i iskorišćenu memoriju. Sledi primer ove komande, na liniji 6 možemo zapaziti da je je executor kreirao privremeni fajl veličine 10.000kB.

EXPLAIN ANALYZE SELECT id, data FROM tbl\_25m ORDER BY id;

QUERY PLAN

--------------------------------------------------------------------------------------------------------------------------

 Sort  (cost=3944070.01..3945895.01 rows=730000 width=4104) (actual time=885.648..1033.746 rows=730000 loops=1)

   Sort Key: id

   Sort Method: external sort  Disk: 10000kB

   ->  Seq Scan on tbl\_25m  (cost=0.00..10531.00 rows=730000 width=4104) (actual time=0.024..102.548 rows=730000 loops=1)

 Planning time: 1.548 ms

 Execution time: 1109.571 ms

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

### 2.5.1 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.

### 2.5.2 Kreiranje plan stabla za single-table upite

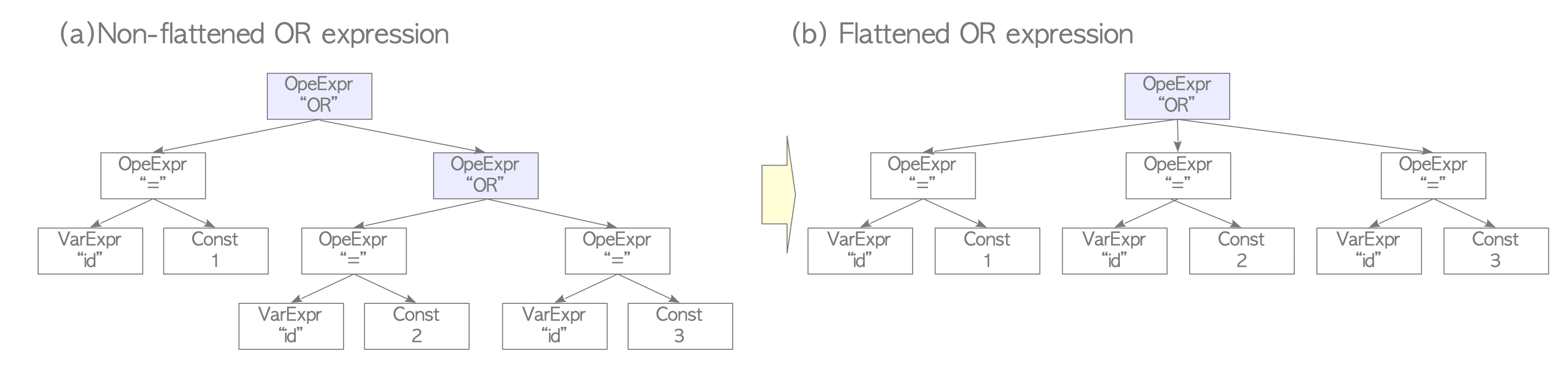
Planner u PostgreSQL prolazi kroz tri faze:

1. Izvrši preprocesiranje.
2. Pronađi najjeftiniji pristup (*access path*) estimacijom cena svih mogućih pristupa
3. Kreirati plan stablo bazirano na pronađenim najjeftinijim pristupom

#### 2.5.2.1 Preprocesiranje

Pre kreiranje plan stabla, planner izvršava preprocesiranje query stabla, koraci u preprocesiranju 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 2.6 – poravnjanje and/or izraza

#### 2.5.2.2 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

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

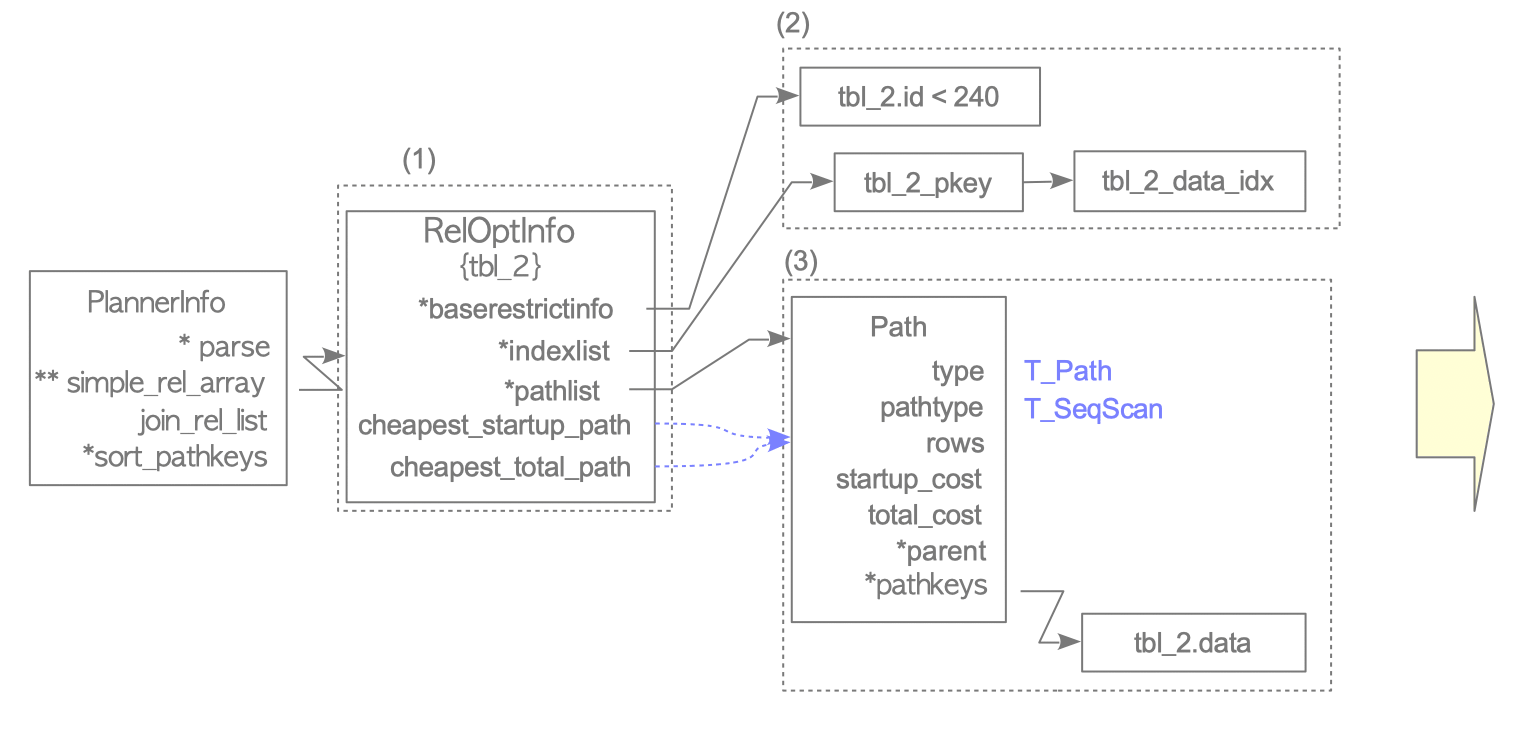
id | integer | not null

data | integer |

Indexes:

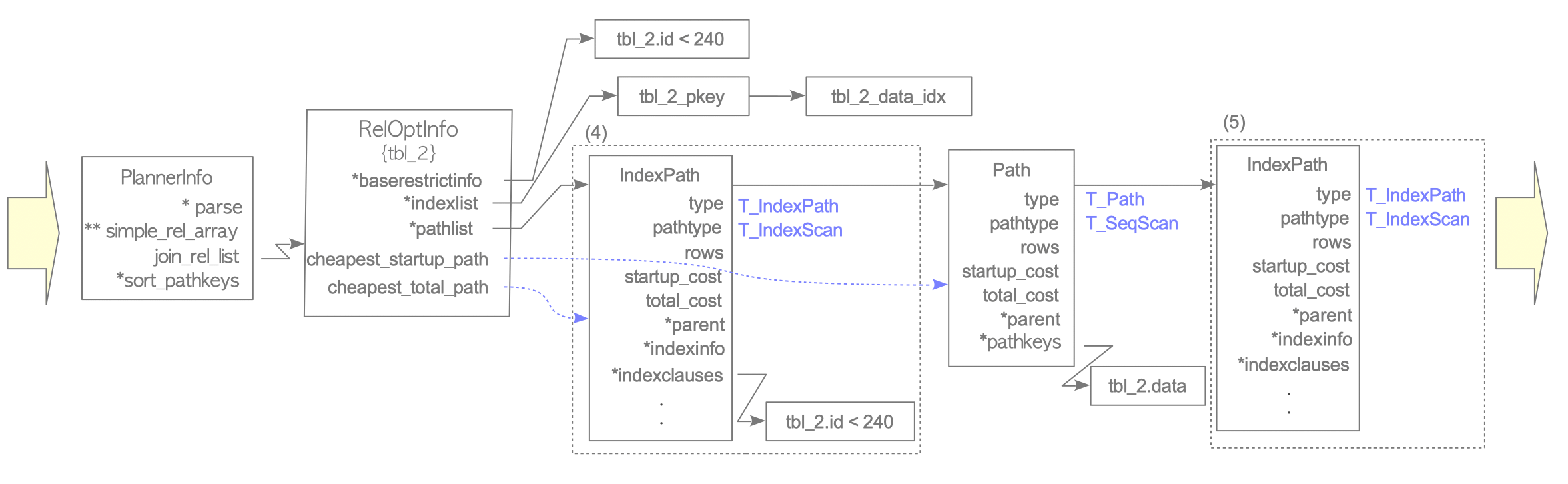
"tbl\_2\_pkey" PRIMARY KEY, btree (id)

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



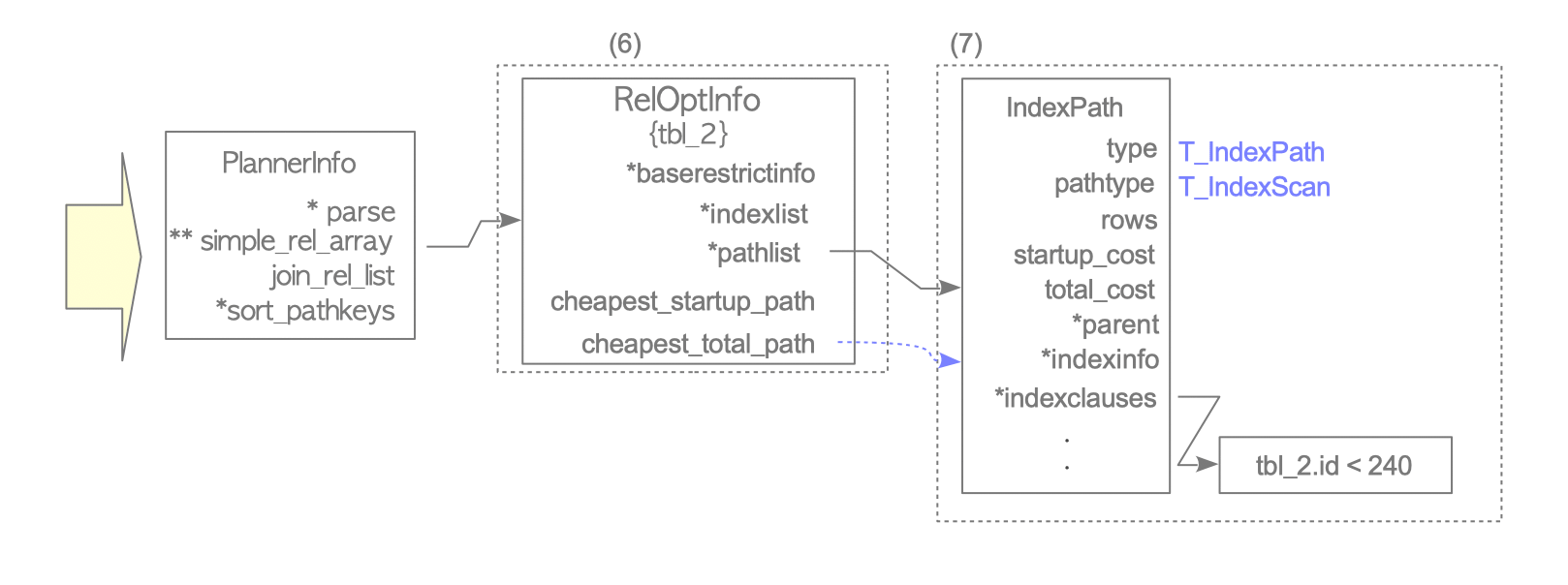
Slika 2.7 – primer pronalaženja najjefitnijeg pristupa

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 2.8 – 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 2.9 – 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.

#### 2.5.2.3 Kreiranje plan stabla

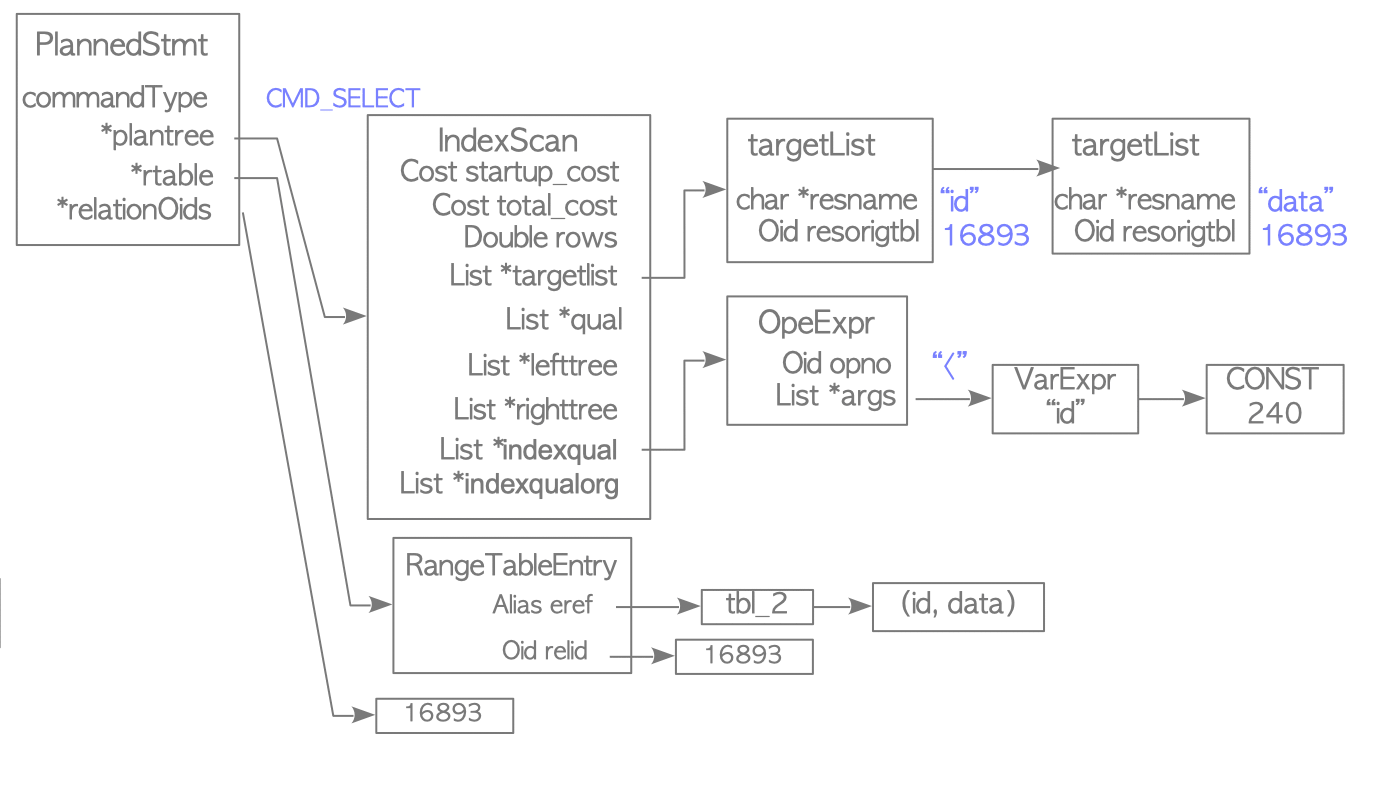
Koren plan stabla [PlannedStmt](https://github.com/postgres/postgres/blob/dec8ad367e46180f826d5b6dc820fbecba1b71d2/src/include/nodes/plannodes.h#L43) struktura, i ako sadrži devetnaest polja, četiri najbitnija su:

* **commandType** – pamti vrstu upita (SELECT, UPDATE, INSERT ...)
* **rtable** – pamti rangeTable-ove za upit
* **relationalOids** – pamti OID-e potrebne za upit
* **plantree** – pamti plan stablo, 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 stabla
* **qual –** lista koja pamti kvalifikacione uslove.
* **lefttree** i **righttree** – čvorovi za dodavanje čvorove potomke.

Sledi slika plan stabla baziranog na primeru iz prethodnog poglavlja.



Slika 2.10 – primer plan stabla

## Izvršenje višetablarnih upita

### 2.6.1 Preprocesiranje

Preprocesiranje jednotablarnih upita je objašnjeno u prethodnim poglavljima ovog dokumenta. U nastavku sledi preprocesiranje višetabelarnih upita, od mnogih postupaka biće objašnjena samo nekoliko.

* Izvlačenje podupita – ako FROM klauzula ima podupit i nema *GROUP BY, HAVING, ORDER BY, LIMIT* ili *DISTINCT* klauzule I ne koristi *INTERSECT* ili *EXCEPT* planner konvertuje podupit u join. U primeru koji sledi upit ima podupit unutar FROM klauzule koji može biti preveden u join upit.

SELECT \* FROM tbl\_a AS a, (SELECT \* FROM tbl\_b) as b WHERE a.id = b.id;

                     ↓

SELECT \* FROM tbl\_a AS a, tbl\_b as b WHERE a.id = b.id;

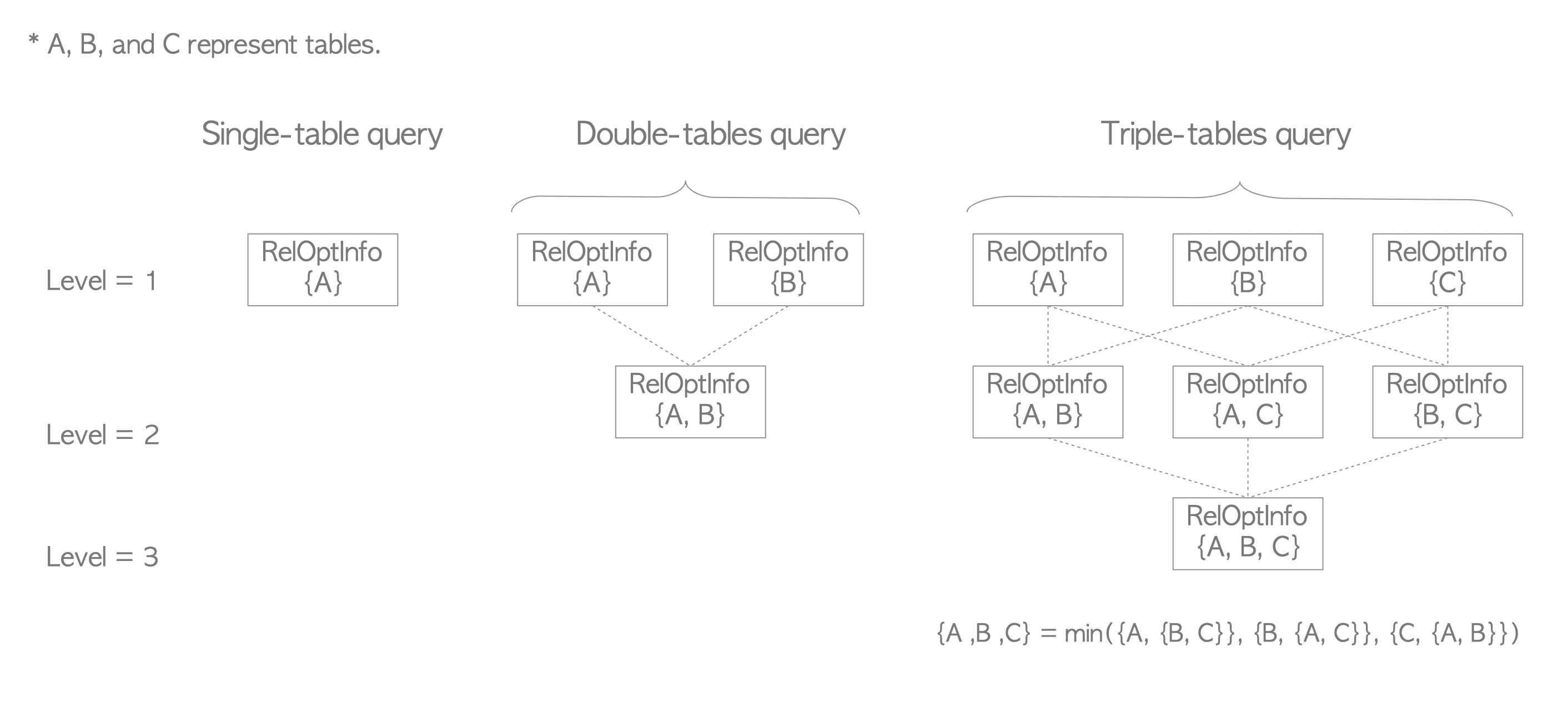
* Transformacija *Outer Join*-a u *Inner Join* ako je to moguće.

### 2.6.2 Pronalaženje najjeftinijeg pristupa

Da bi pronašao optimalno stablo plana, planner mora da uzme u obzir kombinacije svih indeksa i join metoda. Ovo je veoma skup proces i eksponencijalno raste sa povećanjem broja tabela. Ako je broj tabela manji od 12, planner može da u razumnom vremenu dođe do rezultata korišćenjem dinamičkog programiranja. U suprotnom planner koristi *generički algoritam* koji prestavlja algoritam koji aproksimacijom dolazi do razumnog plana u razumnom vremenu, on se poziva ako je broj join-ova veći od promenljive definisanoj u *geqo\_threshold* promenljivoj (difoltna vrednost je 12).

Određivanje najjeftinijeg pristupa dinamičkim programiranjem se može objasniti sledećim koracima:

* Korak 1 – pronaći najjeftiniji pristup svakoj od tabela, on se pamti u zasebnim *RelOptInfo* strukturama.
* Korak 2 – Pronaći najjeftiniji pristup kombinacijom svake tabele sa svakom iz skupa, npr. ako u upitu postoje tri tabele, pronaći najjeftiniji pristup za {A, B}, {A, C} i {B, C}.
* Korak 3 – nastaviti istu obradu dok se ne dostigne do nivoa (level) koji je jednak broju tabela u upitu.



Slika 2.11 – pronalaženje najjeftinijeg pristupa

# 2. 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)**.

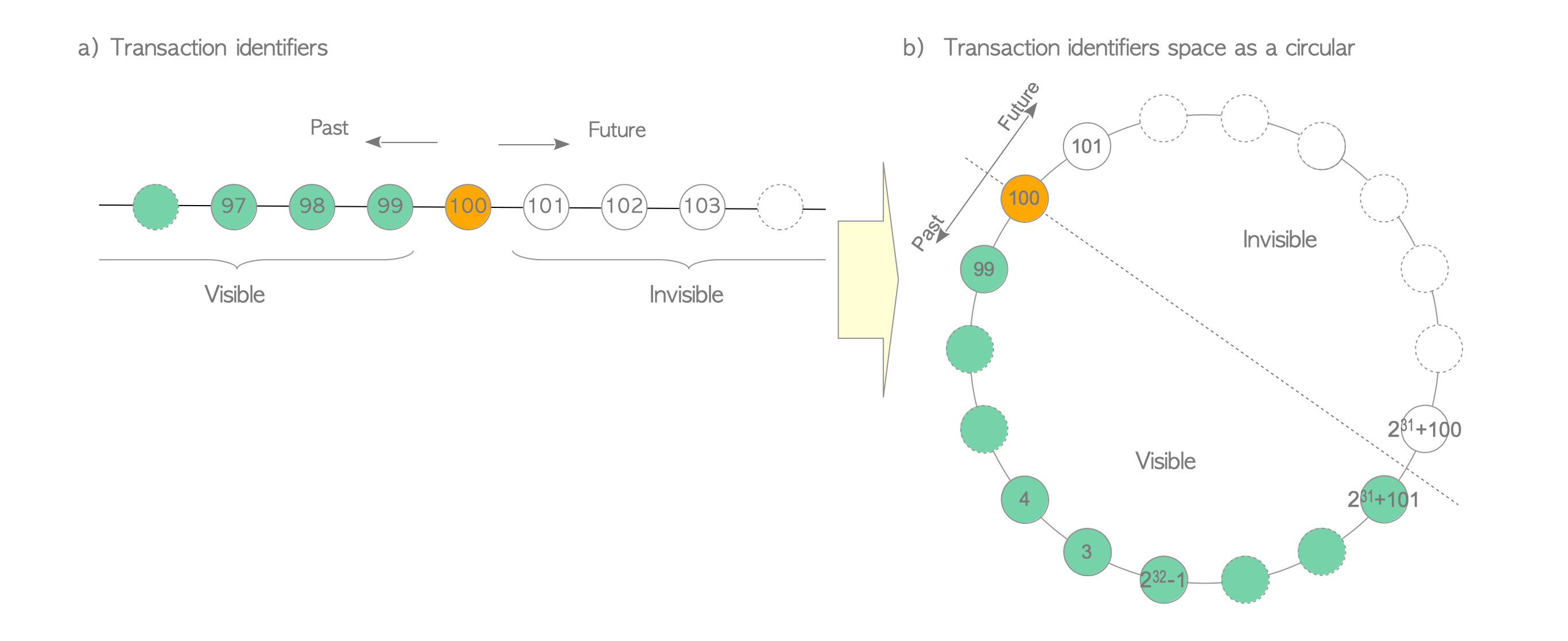
## 2.1 ID transakcije

Kad god se zapošne transakcija, njoj se dodeli ID transakcije **(transaction ID),** nadalje u tekstu predstavljen kao **txid**,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

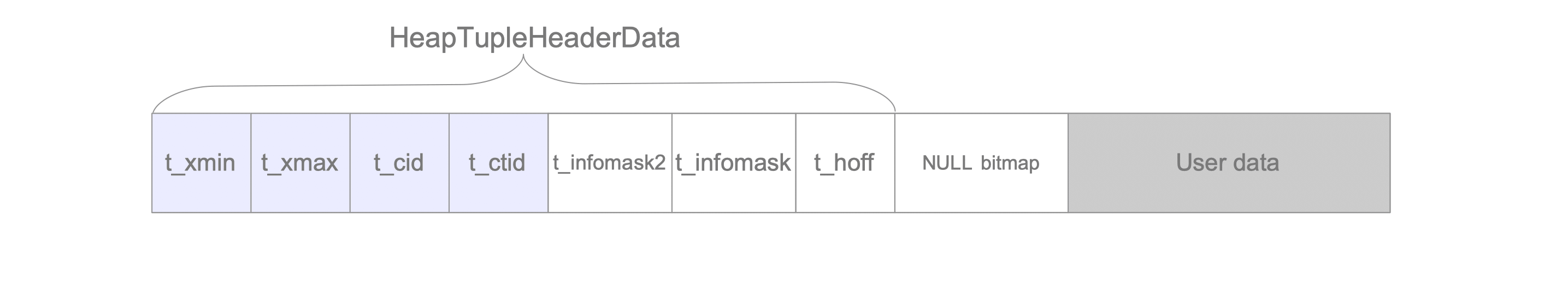
Txid-jevi su uporedivi - značući da za txid od vrednosti N, txid-jevi sa vrednošću manjom od N su u prošlosti a sa većom su u budučnosti



Slika 2.1 – organizacija identifikatora transakcija

## 2.2 Struktura reda tabele baze podataka

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

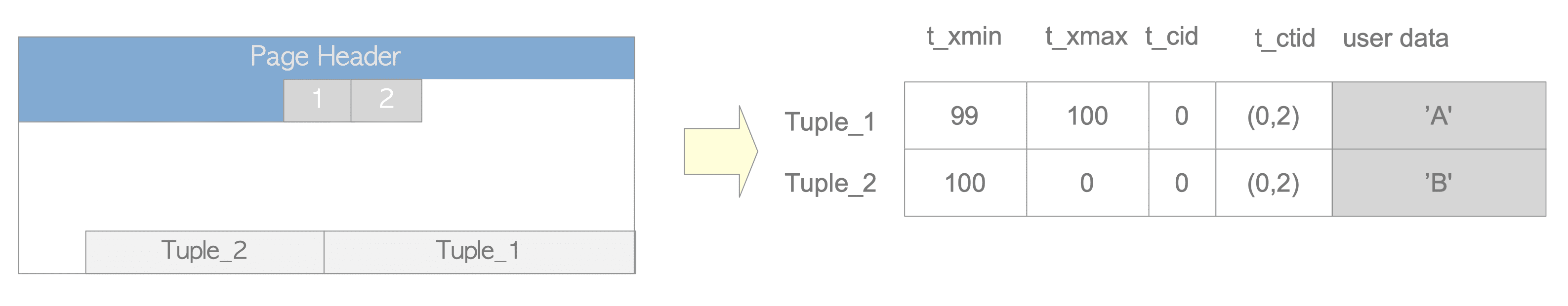


Slika 2.2 – struktura reda-a

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

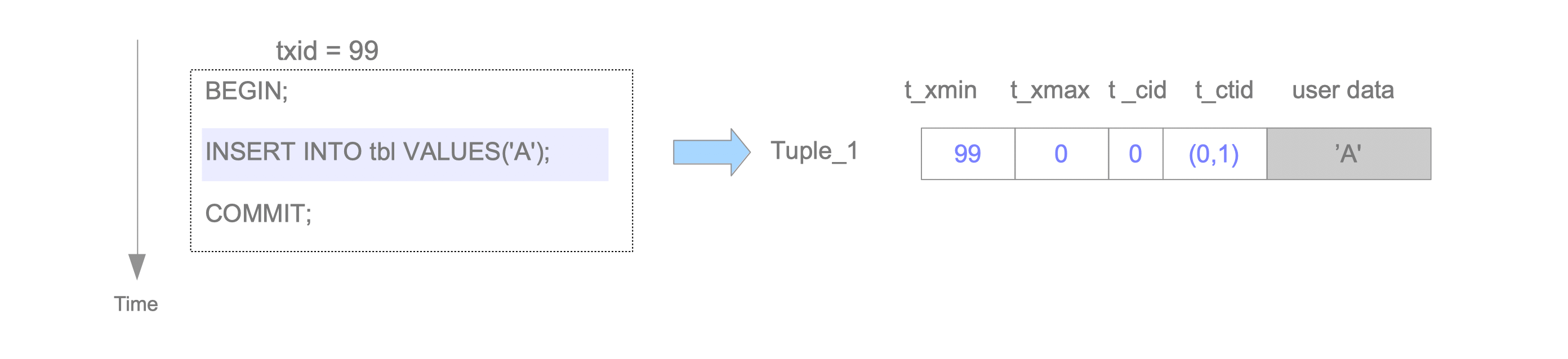
* **t\_xmin** – txid transakcije koja je umetnula red
* **t\_xmax** – txid transakcije koja je obrisala ili ažurirala red. Ako je neuspelo brisanje ili ažuriranje vrednost je 0, što je stanje INVALID.
* **t\_cid** – predstavlja command id, koji označava koliko SQL komandi je izvršeno pre nego što se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

U nastavku slede ilustracije umetanja, brisanja i ažuriranja redova.



Slika 2.3 – reprezentacija reda

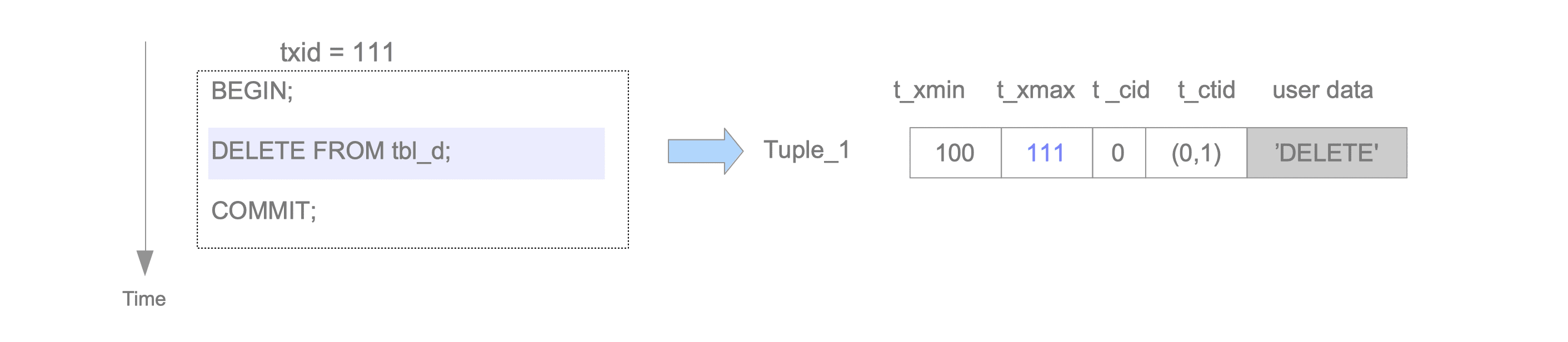
### 2.2.1 Upis



Slika 2.4 – upis reda

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

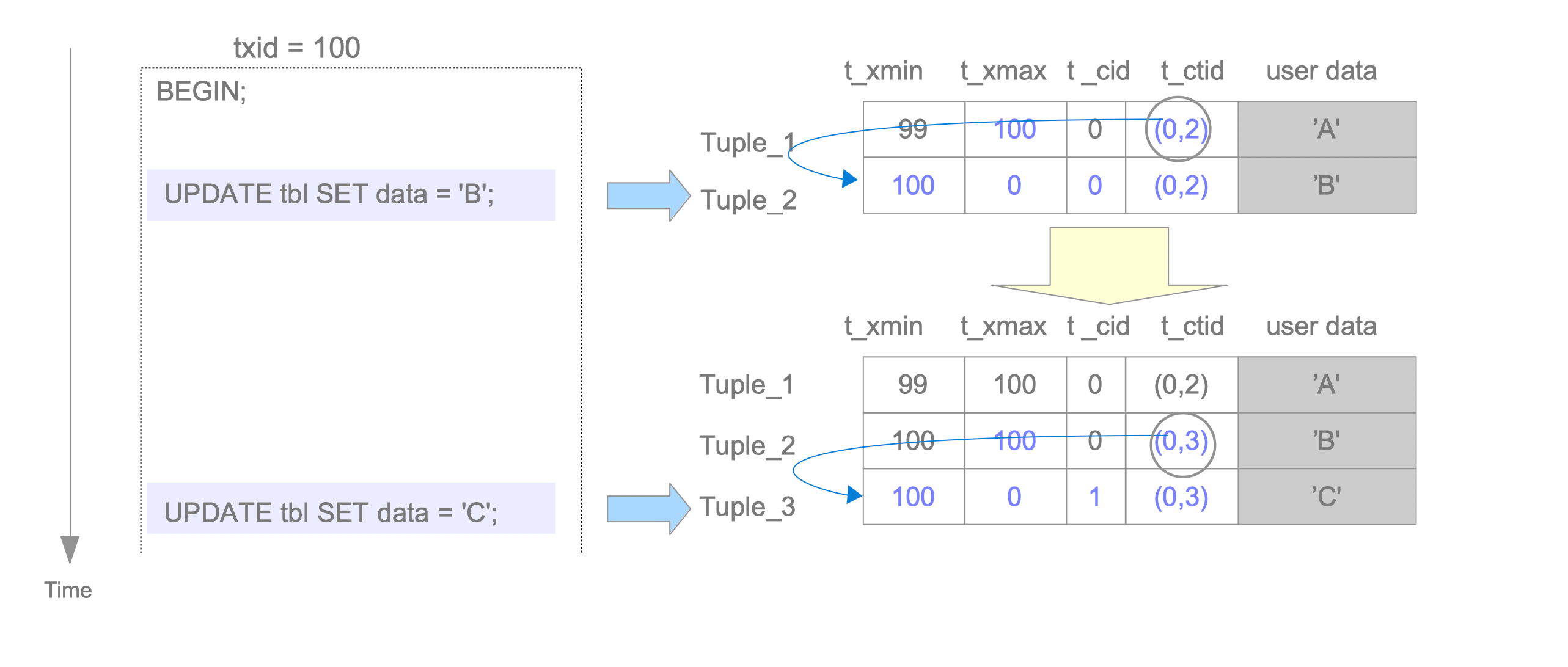
### 2.2.2 Brisanje



Slika 2.5 – brisanje reda

**t\_xmax** je 111 jer je to txid transakcije koja je izvršila brisanje. Red se sada smatra mrtvim I eventualno biće fizički obrisan od strane vakuum procesa.

### 2.2.3 Ažuriranje



Slika 2.6 –ažuriranje reda

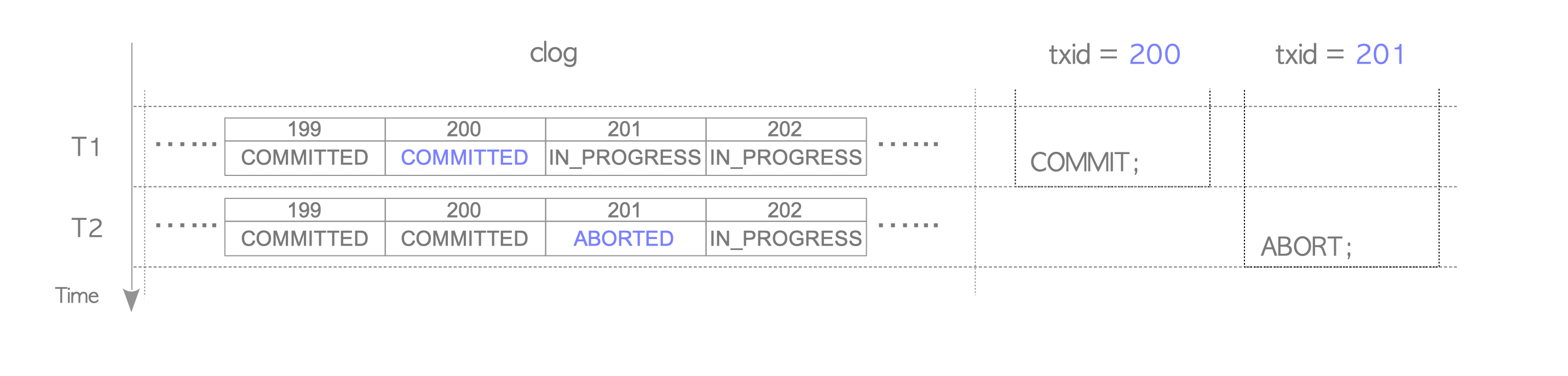
Izvršenjem prve updejt komande tuple\_1 je logiški izbrisanpostavljanje t\_xmax na 100, tuple\_2 je upisan i t\_cid polje unutar tuple\_1 je ažurirano da ukazuje na tuple\_2.

## 2.3 Commit log

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

Postoje tri stanja transakcija: IN\_PROGRESS, COMMITTED, ABORTED.

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



Slika 2.6 –struktura clog-a

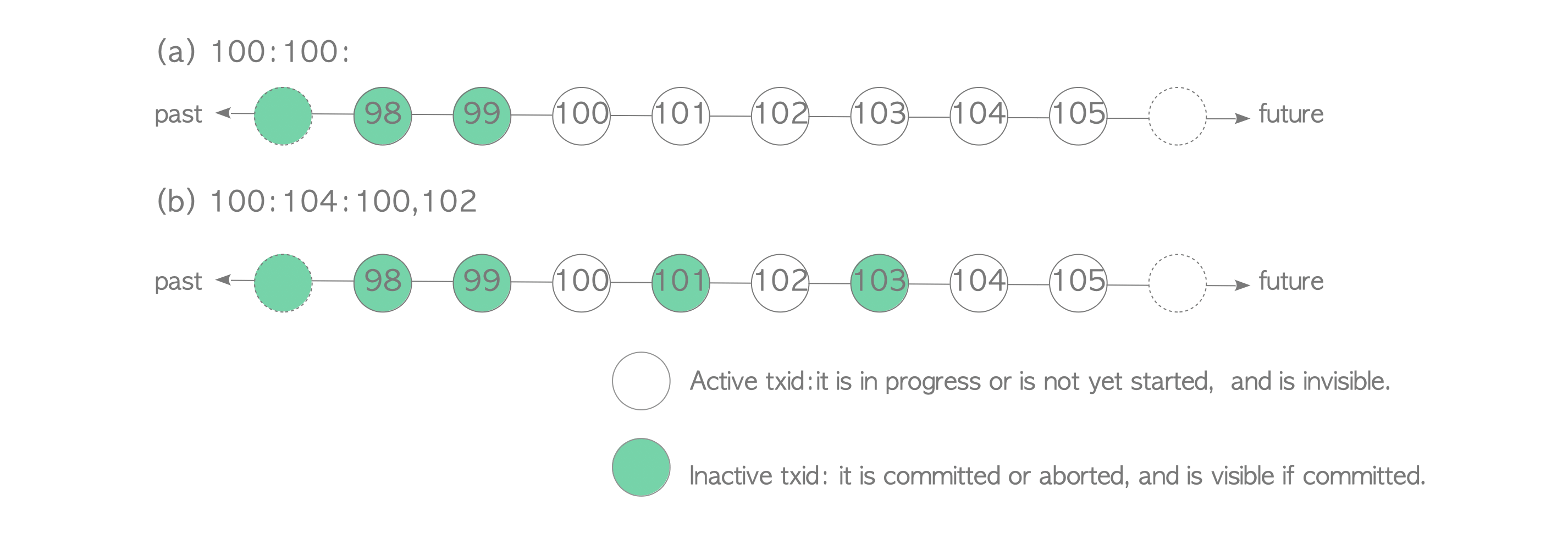
Kada se PostgreSQL gasi ili se pokrene checkpoint process, sadržaj clog-a se upisuje u datoteku unutar **pg\_xact** poddirektorijuma. Ovi fajlovi su imenovani 0000, 0001, 0002 itd. Maksimalna veličina datoteke je 255KB. Pri pokretanju PostgreSQL učitava se sadržaj datoteka, sa obzirom da se clog kontinualno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

## 2.4 Transaction snapshot

Transaction snapshot opisuje koje transakcije su date u datom trenutku iz perspektive neke transakcije. Aktivna transakcija je ona koja je u toku ili još nije počela.

Reprezentacija transaction snapshota je **xmin : xmax : xip\_list** gde je:

* xmin – najmanji txid koji je još aktivan
* xmax – txid koji je sledeći za izdavanje
* xip\_list – txid-jevi transakcija koje su još u toku za vreme izdavanja transaction snapshot-a



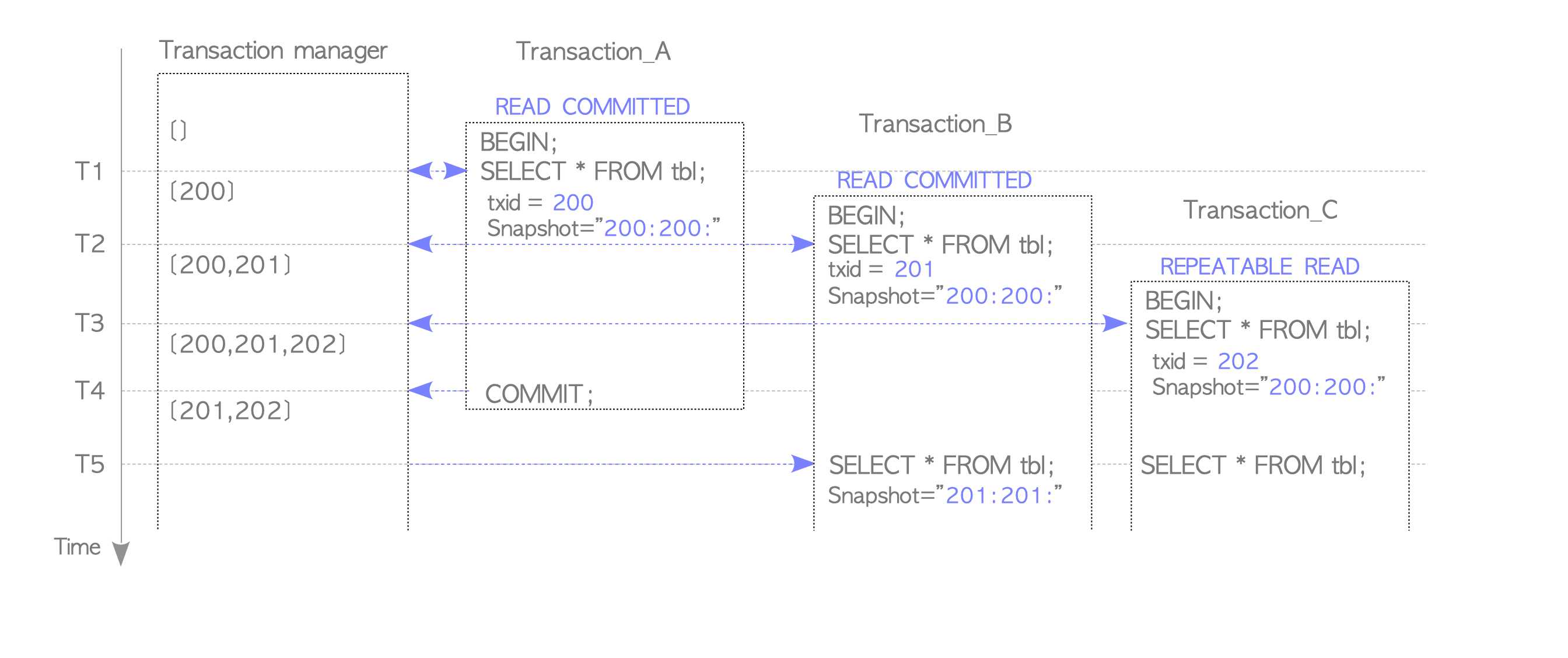
Slika 2.7 –primer reprezentacije transakcionog snapshot-a

Prvi primer je 100:100, značeći da txid-jevi koji su 99 ili manje nisu aktivni jer je xmin 100, txid-jevi jednaki ili veći od 100 su aktivni jer je xmax 100. Drugi primer je 100:104:100,102, značeći da txid-jevi koji su manje ili jednaki sa 99 nisu aktivni, veći ili jednaki sa 104 su aktivni i da txid-jevi sa vrednošću od 100 i 102 su aktivni jer se nalaze u xip\_list-i.

Transaction snapshots su generisani od strane transakcionog menadžera i koriste se za proveru vidljivosti koja je kasnije opisana u dokumentu. Postoje nekoliko nivoa izolacije:

* **READ COMMITTED** – transakcija dobija snapshot pri izvršenju svake SQL komande.
* **REPEATABLE READ i SERIALIZABLE** – transakcija dobija snapshot samo pri prvoj SQL komandi.

Nivo izolacije se menja komandom **SET TRANSACTION transaction\_mode**. U nastavku sledi primer kako bi različite transakcije videle snapshot. Transakcije A i B se izvršavaju pod *READ COMMITTED* izolacionom nivou, dok transakcija C pod *REPEATABLE READ.*



Slika 2.8 –primer transakcionog snapshot-a za različite transakcije

* T1 - Transakcija A izdraje prvu SELECT komadu i zahteva trenutni transaction snapshot. U ovom primeru menadžer transakcija dodeljuje ovoj transakcije txid od 200 i vraća snapshot 200:200.
* T2 - Transakcija B izdraje prvu SELECT komadu i zahteva trenutni transaction snapshot. Menadžer transakcija dodeljuje ovoj transakcije txid od 201 i vraća snapshot 200:200. Jer transakcija A (txid 200) je još u toku i samim tim nije vidljiva transakciji B.
* T3 - Transakcija C izdraje prvu SELECT komadu i zahteva trenutni transaction snapshot. Menadžer transakcija dodeljuje ovoj transakcije txid od 202 i vraća snapshot 200:200, samim tim transakcije A i B joj nisu vidljive.
* T4 - Transakcija A je komitovana.
* T5 – Transakcije B i C izvršavaju svoje naknadne select komande. Transakcija B zahteva snapshot i pošto je u primeru pokrenuta pod *READ COMMITTED* nivou za snapshot dobija 201:201 jer je transakcija A završena i nije više aktivna i samim tim vidljiva transakciji B. Transakcija ce je pokrenuta pod *REPEATABLE READ* nivoom i zbog toga koristi već dodeljeni snapshot 200:200, transakcija A joj i dalje nije vidljiva.

## 2.5 Pravila provere vidljivosti

Pravila provere vidljivosti je skup pravila koji odlučuje da li je neku red vidljiv ili ne korišćenjem t\_xmin i t\_xmax datog reda, clog i dobijeni transaction snapshot. Pravila koja slede su uprošćena zbog velike kompleksnsti. Navešćemo 10 pravila koja mogu biti kategorizovana u tri grupe:

### 2.5.1 Status t\_xmin je ABORTED

Redvoi kojima je status t\_xmin-a *aborted*  su uvek nevidljiva (pravilo 1) jer je transakcija koja je unela ovaj red odbačena.

Rule 1: IF t\_xmin status is 'ABORTED' THEN

                  RETURN 'Invisible'

            END IF

### 2.5.2 Status t\_xmin je IN\_PROGRESS

Red kome je t\_xmin *in\_progress* je uvek nevidljiv (pravila 3 i 4) osim u jednom slučaju:

              IF t\_xmin status is 'IN\_PROGRESS' THEN

                   IF t\_xmin = current\_txid THEN

Rule 2:              IF t\_xmax = INVALID THEN

                  RETURN 'Visible'

Rule 3:              ELSE  /\* ovaj red je izbrisan ili ažuriran u trenutnoj transakciji. \*/

                  RETURN 'Invisible'

                         END IF

Rule 4:        ELSE   /\* t\_xmin ≠ current\_txid \*/

                  RETURN 'Invisible'

                   END IF

             END IF

Ako je ovaj red ubačen od strane druge transakcije i status t\_xmin-a je *in\_progress* on je nevidljiv (pravilo 4). Ako je t\_xmin jednak trenutnom txid-ju (umetnut od strane iste transakcije) i t\_xmax je *invalid,* red je nevidljiv jer je ažuriran ili obrisan unutar iste transakcije (pravilo 3). Izuzetak je ako je t\_xmin jednak trenutnom txid-ju i t\_xmax je *invalid*,tada je red vidljiv jer je umetnut unutar iste transakcije.

### 2.5.3 Status t\_xmin je COMMITED

Red kome je status t\_xmin-a *commited* je vidljiv (pravila 6, 8 i 9) osim u tri slučaja.

            IF t\_xmin status is 'COMMITTED' THEN

Rule 5:      IF t\_xmin is active in the obtained transaction snapshot THEN

                      RETURN 'Invisible'

Rule 6:      ELSE IF t\_xmax = INVALID OR status of t\_xmax is 'ABORTED' THEN

                      RETURN 'Visible'

                 ELSE IF t\_xmax status is 'IN\_PROGRESS' THEN

Rule 7:           IF t\_xmax =  current\_txid THEN

                            RETURN 'Invisible'

Rule 8:           ELSE  /\* t\_xmax ≠ current\_txid \*/

                            RETURN 'Visible'

                      END IF

                 ELSE IF t\_xmax status is 'COMMITTED' THEN

Rule 9:           IF t\_xmax is active in the obtained transaction snapshot THEN

                            RETURN 'Visible'

Rule 10:         ELSE

                            RETURN 'Invisible'

                      END IF

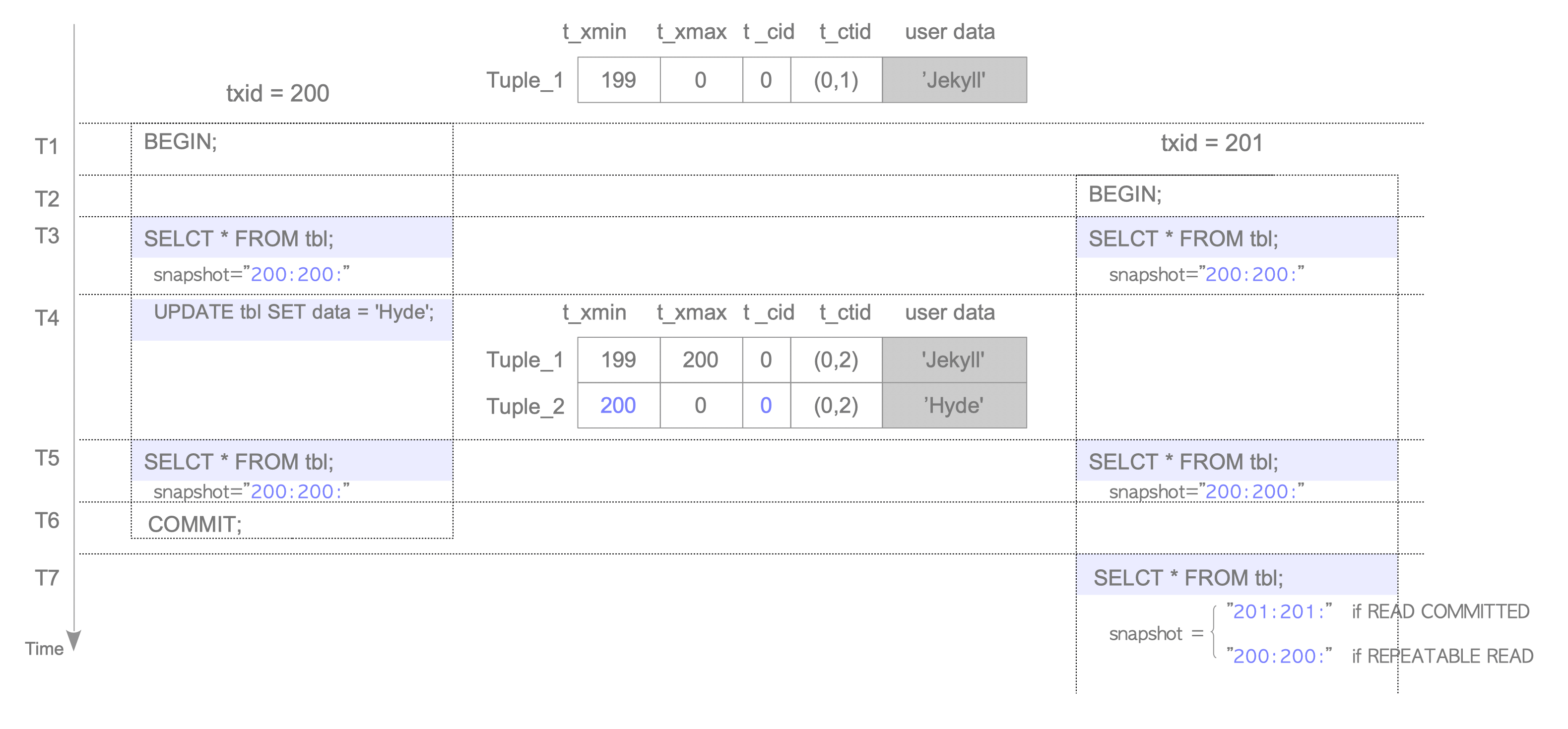
                 END IF

            END IF

Prvi izuzetak je da t\_xmin ima vrednost *active* u dobijenom transaction snapshotu, red je nevidljiv jer status t\_xmin se treba smatrati kao *in\_progress* (pravilo 5)*.* Drugi izuzetak je kada je t\_xmin jednak trenutnom txid-ju i t\_xmax nije *invalid* jer je red izbrisan ili ažuriran u toku iste transakcije (pravilo 7). Treći izuzetak je ako je status t\_xmax -a *commited* i t\_xmax nije aktivan u dobijenom transaction snapshot-u, značeći da je red obrisan ili ažuriran od strane druge transakcije (pravilo 10).

## 2.6 Provera Vidljivosti

U ovom delu biće objašnjeno kako PostgreSQL bira redove adekvatne verzije za datu transakciju i kako sprečava anomalije poput *Dirty Reads, Repeatable Reads* i *Phantom Reads.*



Slika 2.9 –primer provere vidljivosti

Da bi uprostili objašnjenje, smatrajmo da u primeru na slici 2.7 postoje samo sve transakcije: 200 i 201.

* T3 - postoji samo red *Tuple\_1* i vidljiv je po pravilu broj 6, samim tim obe transakcije dobijaju istu vrednost *Jekyll.*
* T5 – prvo analiziramo select komadu txid-ja 200, red Tuple\_1 je nevidljiv na osnovu pravila 7 i red Tuple\_2 je vidljiv na osnovu pravila 2, premda rezultat upita je *Hyde.* Nasuprot njemu, select upit txid-ja 201 vraća ponovo vrednost *Jekyll* jer je red Tuple\_1 je vidljiv na osnovu pravila 8 i red Tuple\_2 je nevidljiv na osnovu pravila 4.
* T7 – ovde će biti analizirana ponašanje select komandi u dva izolaciona nivoa. Prvo ispitujemo kada txid 201 sa nivoom izolacije *read commited.* U ovom slučaju txid 200 se smatra kao *committed* jer je transaction snaphot 201:201, red Tuple\_1 je nevidljiv na osnovu pravila 10 i red Tuple\_2 je vidljiv na osnovu pravila 6, rezultat select upita je *Hyde;* Međutim kada je txid 201 u *repeatable read* nivou izolacije txid 200 se smatra *in\_progress* jer je transaction snaphot 200:200, red Tuple\_1 je vidljiv na osnovu pravila 9 i red Tuple\_2 je nevidljiv na osnovu pravila 5 i kao rezultat select upita dobija se *Jekyll.*

## Sprečavanje izgubljenih ažuriranja

Izgubljeno ažuriranje, takođe poznato kao i *ww-conflict*, je anomalija koja se dešava kada uzastopne transakcije ažuriraju isti red i mora biti sprečeno u *REPEATABLE READ* i *SERIALIZABLE* izolacionim nivoima (kod *READ COMMITTED* nivoa nije potrena zaštita). Sledi pseudokod update komande.

(1)  FOR each row that will be updated by this UPDATE command

(2)    WHILE true

         /\* The First Block \*/

(3)      IF the target row is being updated THEN

(4)           WAIT for the termination of the transaction that updated the target row

(5)           IF (the status of the terminated transaction is COMMITTED)

                AND (the isolation level of this transaction is REPEATABLE READ or SERIALIZABLE) THEN

(6)                 ABORT this transaction  /\* First-Updater-Win \*/

              ELSE

(7)               GOTO step (2)

              END IF

         /\* The Second Block \*/

(8)      ELSE IF the target row has been updated by another concurrent transaction THEN

(9)           IF (the isolation level of this transaction is READ COMMITTED THEN

(10)                UPDATE the target row

              ELSE

(11)                ABORT this transaction  /\* First-Updater-Win \*/

              END IF

         /\* The Third Block \*/

          ELSE  /\* The target row is not yet modified or has been updated by a terminated transaction. \*/

(12)          UPDATE the target row

          END IF

        END WHILE

   END FOR

1. Nabaviti sve redove koji će biti ažurirani ovom update komandom
2. Ponavljati sledeće korake sve dok dati red nije ažuriran
3. Ako dati red u toku ažuriranja nastaviti dalje, u suprotnom ići na korak 8
4. Sačekati da se okonča transakcija koja je ažurirala dati red
5. Ako je status transakcije koja je azvršila ažuriranje COMMITED i nivo izolacije je REPEATABLE READ ili SERIALIZABLE ići na korak 6, u suprotnom na korak 7
6. Ukinuti ovu transakciju kako bi se sprečilo gubljenje ažuriranja
7. Preći na korak 2 i pokušati ažuriranje datog reda u sledećem krugu
8. Ako je red ažuriran od strane druge transakcije nastaviti na korak 9, u suprotnom preći na korak 10
9. Ako je izolacioni nivo READ COMMITTED, nastaviti na korak 10, u suprotnom preći na korak

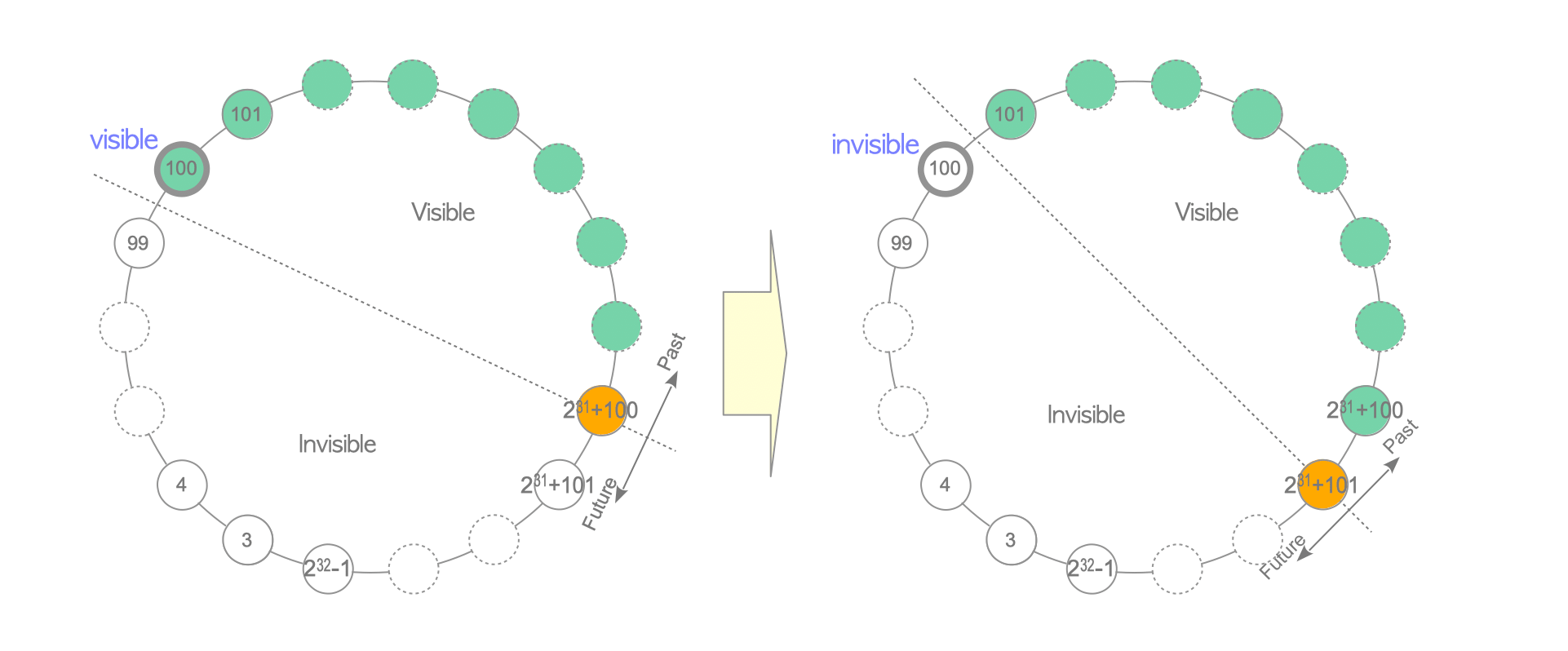
(10) Ažurirati red i preći na korak 1

(11) Ukinuti ovu transakciju kako bi se sprečilo gubljenje ažuriranja

(12) Ažurirati dati red i preći na korak 1 jer dati red nije ili je bio ažuriran od strane terminirane transakcije

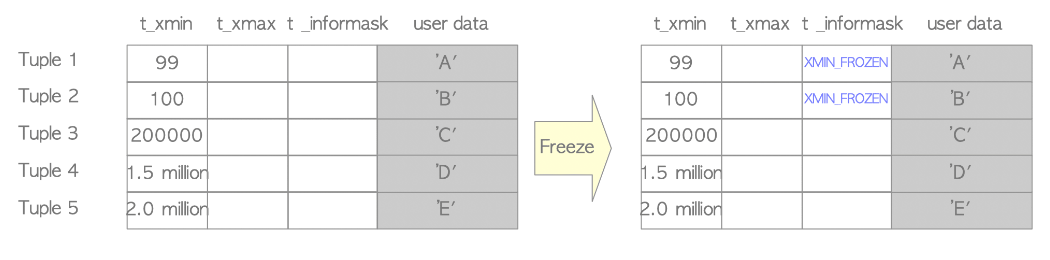
## Freeze proces

U nastavku ovog paragrafa biće primer txid wraparound problema. Uzmimo u obzir da je red Tuple\_1 umetnut od strane txid 100 (njegov t\_xmin je 100). Server radi duže vreme i Tuple\_1 nije bio menjan, trenutni txid je 2.1 milijarda + 100 i izvršava se SELECT komanda. Tuple\_1 je vidljiv jer je txid 100 u prošlosti. SELECT komanda se izvršava i trenutni txid postaje 2.1 milijarde + 101, Tuple\_1 više nije vidljiv jer je u budućnosti, ovo se zove wraparound problem.



Slika 2.10 – wraparound problem

Da bi se ovo sprečilo PostgreSQL je implementirao FREEZE proces. U PostgreSQL-u zamrznuti txid, koje je specijalan rezervisani txid2, je definisan da je uvek stariji nego svi drugi txid-jevi, drugim rečima zamrznuti txid je uvek inaktivan i vidljiv. Freeze proces skenira sve fajlove tabele i postavlja t\_infomask fleg ako je t\_xmin vrednost starija od trenutnog txid-ja minus *vacuum\_freeze\_min\_age* (50 milijona difoltno).



Slika 2.11 – freeze proces