

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ «КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені Ігоря Сікорського»

ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ

**Кафедра системного програмування та спеціалізованих комп’ютерних систем**

**Лабораторна робота №3**

з дисципліни **Бази даних і засоби управління**

*на тему: “* *Засоби оптимізації роботи СУБД PostgreSQL”*

Виконав:

студент ІII курсу

групи КВ-91

Русаков М. В.

Перевірив: Павловський В. І.

Київ – 2021

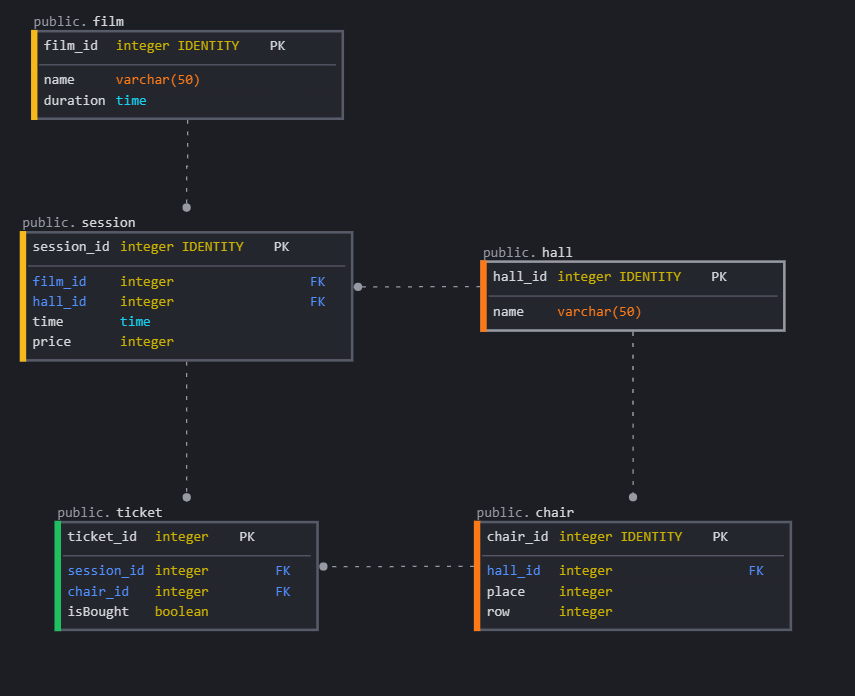
**Варіант 18**

Завдання роботи полягає у наступному:

1. Перетворити модуль “Модель” з шаблону MVC лабораторної роботи №2 у вигляд об’єктно-реляційної проекції (ORM).
2. Створити та проаналізувати різні типи індексів у PostgreSQL.
3. Розробити тригер бази даних PostgreSQL.
4. Навести приклади та проаналізувати рівні ізоляції транзакцій у PostgreSQL.

**Логічна модель бази даних**

Нижче (Рис. 1) наведено логічну модель бази даних:



Логічна модель бази даних

Зміни у порівнянні з першою лабораторною роботою відсутні.

**Класи ORM**

Chair.java

import javax.persistence.Column;  
import javax.persistence.Entity;  
import javax.persistence.Id;  
import javax.persistence.Table;  
import java.io.Serializable;  
  
  
@Entity  
@Table (name = "chair")  
public class Chair implements Serializable{  
  
 @Id  
 @Column  
 private int chair\_id;  
  
 @Column  
 private int hall\_id;  
  
 @Column  
 private int place;  
  
 @Column  
 private int row;  
  
 public int getChair\_id()  
 {  
 return chair\_id;  
 }  
  
 public void setChair\_id(int chair\_id)  
 {  
 this.chair\_id = chair\_id;  
 }  
  
 public int getHall\_id()  
 {  
 return hall\_id;  
 }  
  
 public void setHall\_id(int hall\_id)  
 {  
 this.hall\_id = hall\_id;  
 }  
  
 public int getPlace()  
 {  
 return place;  
 }  
  
 public void setPlace(int place)  
 {  
 this.place = place;  
 }  
  
 public int getRow()  
 {  
 return row;  
 }  
  
 public void setRow(int row)  
 {  
 this.row = row;  
 }  
  
 @Override  
 public String toString(){  
 return "Chair {" +  
 " chair\_id = " + chair\_id +  
 " hall\_id = " + hall\_id +  
 " place = " + place +  
 " row = " + row +  
 " }";  
 }  
  
}

Film.java

import javax.persistence.Column;  
import javax.persistence.Entity;  
import javax.persistence.Id;  
import javax.persistence.Table;  
import java.io.Serializable;  
import java.sql.Time;  
import java.text.SimpleDateFormat;  
import java.util.Date;  
  
  
@Entity  
@Table (name = "film")  
public class Film implements Serializable{  
  
 @Id  
 @Column  
 private int film\_id;  
  
 @Column  
 private String name;  
  
 @Column  
 private Time duration;  
  
 public int getFilm\_id()  
 {  
 return film\_id;  
 }  
  
 public void setFilm\_id(int film\_id)  
 {  
 this.film\_id = film\_id;  
 }  
  
 public String getName()  
 {  
 return name;  
 }  
  
 public void setName(String name)  
 {  
 this.name = name;  
 }  
  
 public Time getDuration()  
 {  
 return duration;  
 }  
  
 public void setDuration(Time duration)  
 {  
 this.duration = duration;  
 }  
  
 @Override  
 public String toString(){  
 return "Film {" +  
 " film\_id = " + film\_id +  
 " name = " + name +  
 " duration = " + duration +  
 " }";  
 }  
  
}

Hall.java

import javax.persistence.Column;  
import javax.persistence.Entity;  
import javax.persistence.Id;  
import javax.persistence.Table;  
import java.io.Serializable;  
  
@Entity  
@Table (name = "hall")  
public class Hall implements Serializable{  
  
 @Id  
 @Column  
 private int hall\_id;  
  
 @Column  
 private String name;  
  
  
 public int getHall\_id()  
 {  
 return hall\_id;  
 }  
  
 public void setHall\_id(int hall\_id)  
 {  
 this.hall\_id = hall\_id;  
 }  
  
 public String getName()  
 {  
 return name;  
 }  
  
 public void setName(String name)  
 {  
 this.name = name;  
 }  
  
  
 @Override  
 public String toString(){  
 return "Hall {" +  
 " hall\_id = " + hall\_id +  
 " name = " + name +  
 " }";  
 }  
  
}

Session1.java (Клас Session вже існує в проекті, тому назва була видозмінена)

import javax.persistence.Column;  
import javax.persistence.Entity;  
import javax.persistence.Id;  
import javax.persistence.Table;  
import java.io.Serializable;  
import java.sql.Time;  
import java.util.Date;  
  
  
@Entity  
@Table (name = "session")  
public class Session1 implements Serializable{  
  
 @Id  
 @Column  
 private int session\_id;  
  
 @Column  
 private int film\_id;  
  
 @Column  
 private int hall\_id;  
  
 @Column  
 private Time time;  
  
 @Column  
 private int price;  
  
 public int getSession\_id()  
 {  
 return session\_id;  
 }  
  
 public void setSession\_id(int session\_id)  
 {  
 this.session\_id = session\_id;  
 }  
  
 public int getFilm\_id()  
 {  
 return film\_id;  
 }  
  
 public void setFilm\_id(int film\_id)  
 {  
 this.film\_id = film\_id;  
 }  
  
 public int getHall\_id()  
 {  
 return hall\_id;  
 }  
  
 public void setHall\_id(int hall\_id)  
 {  
 this.hall\_id = hall\_id;  
 }  
  
 public Time getTime()  
 {  
 return time;  
 }  
  
 public void setTime(Time time)  
 {  
 this.time = time;  
 }  
  
 public int getPrice()  
 {  
 return price;  
 }  
  
 public void setPrice(int price)  
 {  
 this.price = price;  
 }  
  
 @Override  
 public String toString(){  
 return "Session {" +  
 " session\_id = " + session\_id +  
 " film\_id = " + film\_id +  
 " time = " + time +  
 " price = " + price +  
 " }";  
 }  
}

Ticket.java

import javax.persistence.Column;  
import javax.persistence.Entity;  
import javax.persistence.Id;  
import javax.persistence.Table;  
import java.io.Serializable;  
  
@Entity  
@Table (name = "ticket")  
public class Ticket implements Serializable{  
  
 @Id  
 @Column  
 private int ticket\_id;  
  
 @Column  
 private int session\_id;  
  
 @Column  
 private int chair\_id;  
  
 @Column  
 private Boolean isbought;  
  
 public int getTicket\_id()  
 {  
 return ticket\_id;  
 }  
  
 public void setTicket\_id(int ticket\_id)  
 {  
 this.ticket\_id = ticket\_id;  
 }  
  
 public int getSession\_id()  
 {  
 return session\_id;  
 }  
  
 public void setSession\_id(int session\_id)  
 {  
 this.session\_id = session\_id;  
 }  
  
 public int getChair\_id()  
 {  
 return chair\_id;  
 }  
  
 public void setChair\_id(int chair\_id)  
 {  
 this.chair\_id = chair\_id;  
 }  
  
 public Boolean getIsbought()  
 {  
 return isbought;  
 }  
  
 public void setIsbought(Boolean isbought)  
 {  
 this.isbought = isbought;  
 }  
  
 @Override  
 public String toString(){  
 return "Ticket {" +  
 " ticket\_id = " + ticket\_id +  
 " session\_id = " + session\_id +  
 " chair\_id = " + chair\_id +  
 " isbought = " + isbought +  
 " }";  
 }  
}

**Приклад запитів ORM:**

* Film film = session.get(Film.class, Integer.*parseInt*(film\_id));
* session.delete(hall);
* session.save(ticket);
* session.update(film);

**BTree**

Для дослідження індексу була створена таблиця, яка має дві колонки:

числову і текстову. Вони проіндексовані як BTree. У таблицю було занесено

1000000 записів.

Створення таблиці та її заповнення:

DROP TABLE IF EXISTS "test\_btree";

CREATE TABLE "test\_btree"(

"id" bigserial PRIMARY KEY,

"test\_text" varchar(255)

);

INSERT INTO "test\_btree"("test\_text")

SELECT

substr(characters, (random() \* length(characters) + 1)::integer, 10)

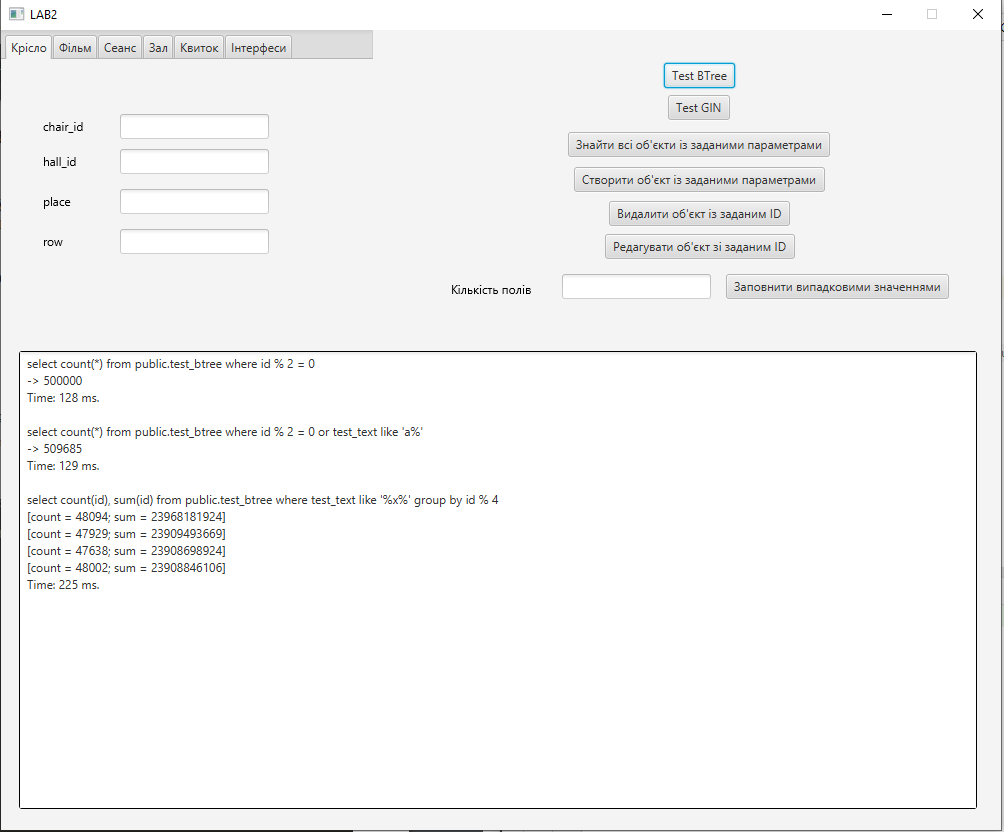
FROM

(VALUES('qwertyuiopasdfghjklzxcvbnmQWERTYUIOPASDFGHJKLZXCVBNM'))

as symbols(characters),

generate\_series(1, 1000000) as q;

**Вибір даних без індексу:**

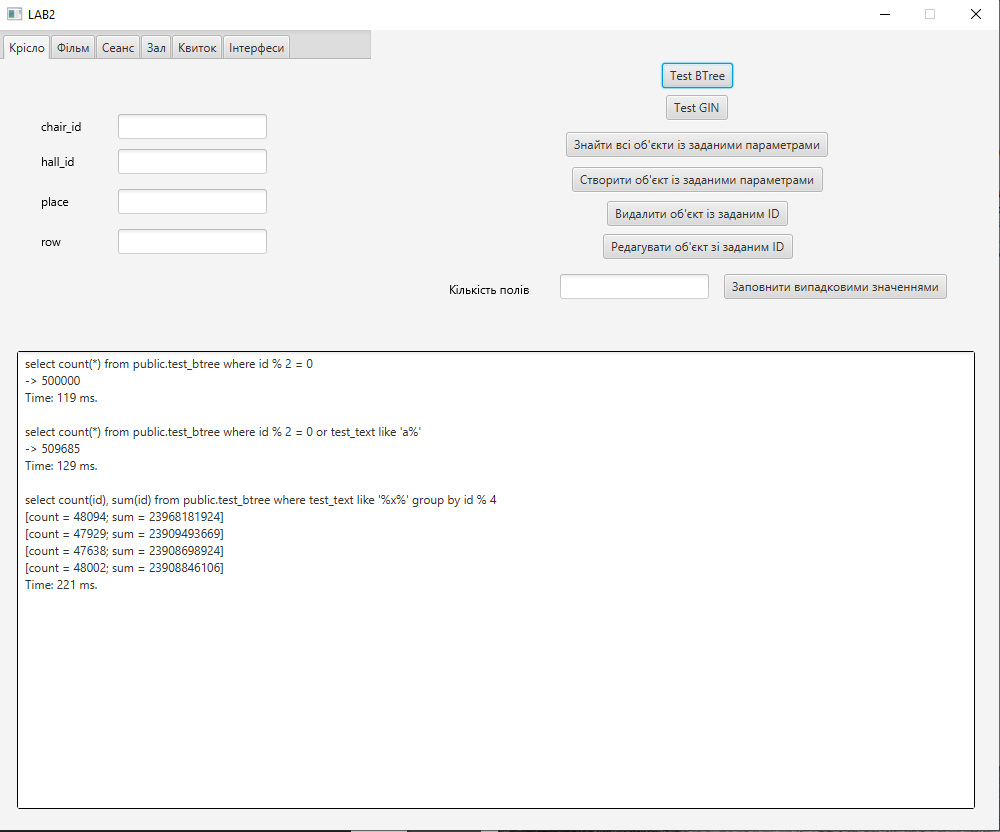


**Сворюємо індекс:**

DROP INDEX IF EXISTS "test\_btree\_test\_text\_index";

CREATE INDEX "test\_btree\_test\_text\_index" ON "test\_btree" USING btree ("test\_text");

**Вибір даних з створеним індексом:**



**GIN**

Для дослідження індексу була створена таблиця, яка має дві колонки:

test\_text типу text[] (масив рядків) і id типу integer. Колонка test\_text проіндексована як GIN. У таблицю занесено 1000000 записів.

Створення таблиці та її заповнення:

DROP TABLE IF EXISTS "test\_gin";

CREATE TABLE "test\_gin"(

"id" bigserial PRIMARY KEY,

"test\_text" text[5]

);

DO $$

begin

for i IN 1..1000000 LOOP

INSERT INTO "test\_gin"("test\_text")

values(array[(SELECT

substr(characters, (random() \* length(characters) + 1)::integer, 10)

FROM

(VALUES('qwertyuiopasdfghjklzxcvbnmQWERTYUIOPASDFGHJKLZXCVBNM')) as symbols(characters))

,(SELECT

substr(characters, (random() \* length(characters) + 1)::integer, 10)

FROM

(VALUES('qwertyuiopasdfghjklzxcvbnmQWERTYUIOPASDFGHJKLZXCVBNM')) as symbols(characters))

, (SELECT

substr(characters, (random() \* length(characters) + 1)::integer, 10)

FROM

(VALUES('qwertyuiopasdfghjklzxcvbnmQWERTYUIOPASDFGHJKLZXCVBNM')) as symbols(characters))

, (SELECT

substr(characters, (random() \* length(characters) + 1)::integer, 10)

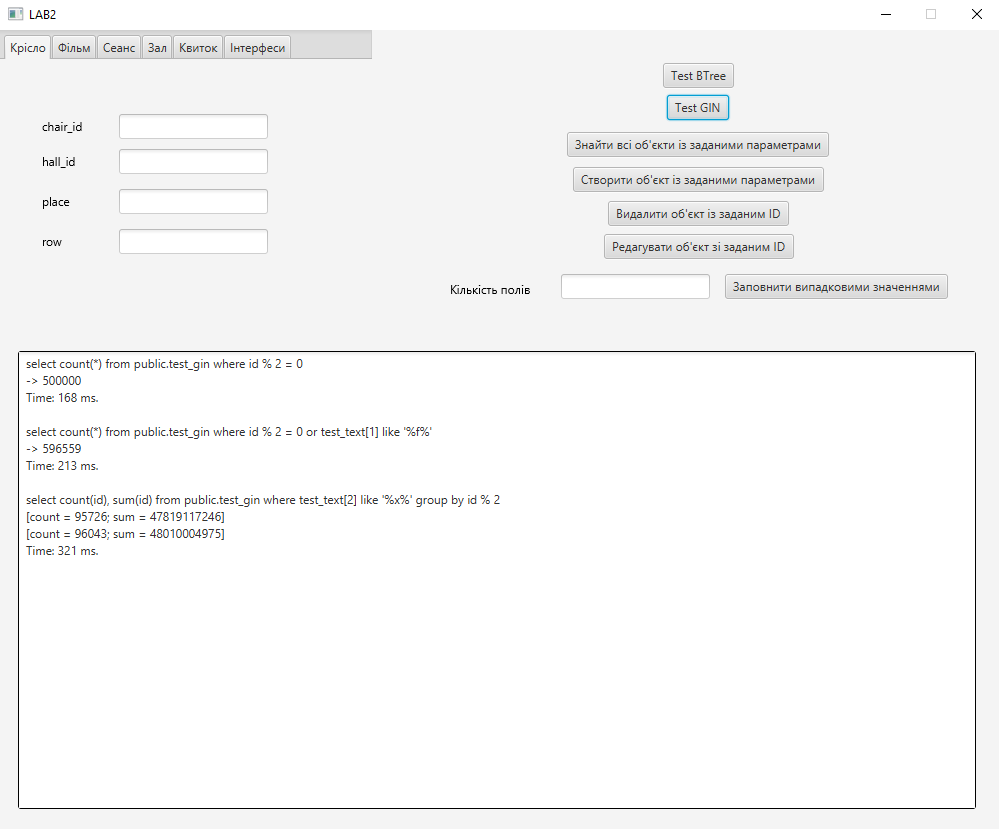
FROM

(VALUES('qwertyuiopasdfghjklzxcvbnmQWERTYUIOPASDFGHJKLZXCVBNM')) as symbols(characters))]);

END LOOP;

end$$;

**Вибір даних без індексу:**

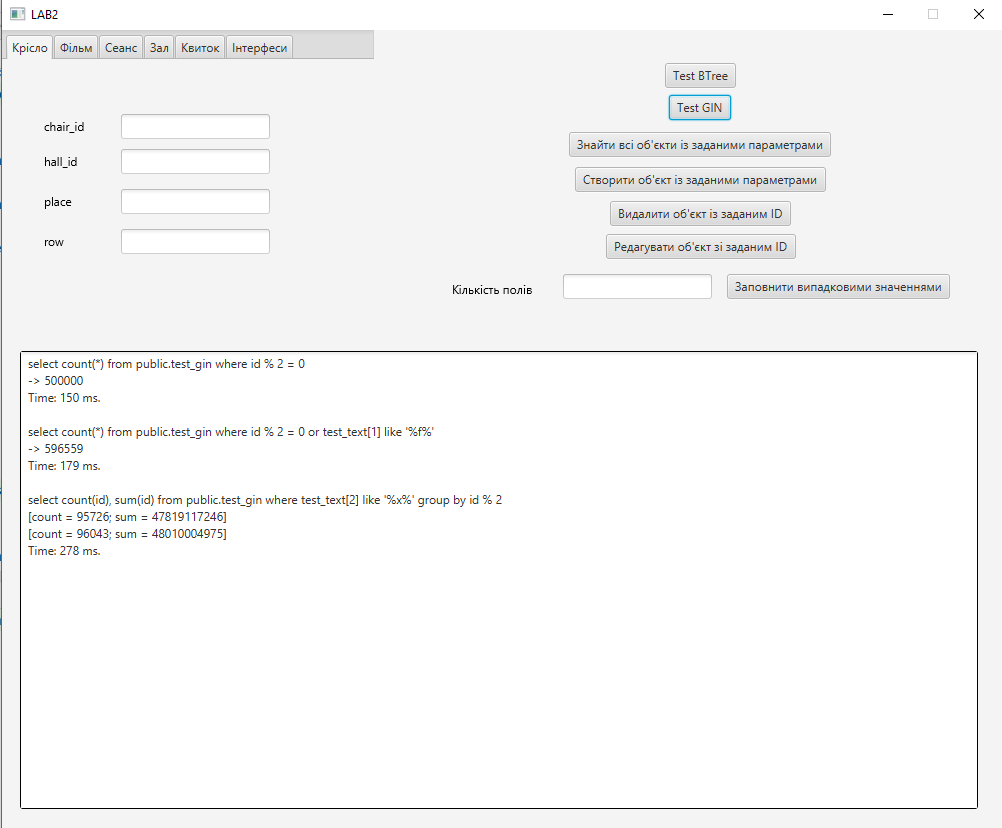


**Сворюємо індекс:**

DROP INDEX IF EXISTS "test\_gin\_test\_text\_index";

CREATE INDEX "test\_gin\_test\_text\_index" ON test\_gin USING gin (test\_text);

**Вибір даних з створеним індексом:**



В даному випадку індекси пришвидшили роботу запиту, але не на багато. Побачити значну відмінність можна на великих базах даних з багатьма елементами. Тому для маленьких баз даних немає необхідності у використанні індексів.

**Тригер бази даних PostgreSQL. Умова для тригера – after update, insert**

**Таблиці:**

CREATE TABLE public.film

(

film\_id integer NOT NULL,

name character varying(50) NOT NULL,

duration time without time zone NOT NULL,

CONSTRAINT film\_pkey PRIMARY KEY (film\_id)

)

CREATE TABLE public.session

(

session\_id integer NOT NULL,

film\_id integer NOT NULL,

hall\_id integer NOT NULL,

"time" time without time zone NOT NULL,

price integer NOT NULL,

CONSTRAINT session\_pkey PRIMARY KEY (session\_id),

CONSTRAINT session\_film\_id\_fkey FOREIGN KEY (film\_id)

REFERENCES public.film (film\_id) MATCH SIMPLE

ON UPDATE NO ACTION ON DELETE NO ACTION,

CONSTRAINT session\_hall\_id\_fkey FOREIGN KEY (hall\_id)

REFERENCES public.hall (hall\_id) MATCH SIMPLE

ON UPDATE NO ACTION ON DELETE NO ACTION

)

**Тригер:**

-- Function: public."updateSession"()

-- DROP FUNCTION public."updateSession"();

CREATE OR REPLACE FUNCTION public."updateSession"()

RETURNS trigger AS

$BODY$declare

h\_i int := (select hall\_id from hall order by random() limit 1);

time\_buf time := '00:00';

begin

IF ((select max(time) from public.session where hall\_id = h\_i) is not null or (select max(time) from public.session where hall\_id = h\_i) < '22:00')

THEN

time\_buf := (select (max(time)) from public.session where hall\_id = h\_i) + interval '2 hours';

END IF;

IF ((select count(film\_id) from public.session where film\_id = new.film\_id) < 1)

THEN

INSERT INTO public.session (session\_id, film\_id, hall\_id, time, price)

VALUES ((select max(session\_id) from session) + 1,

(select film\_id from public.film where film\_id = new.film\_id),

h\_i,

time\_buf,

100);

END IF;

return new;

end

$BODY$

LANGUAGE plpgsql VOLATILE

COST 100;

ALTER FUNCTION public."updateSession"()

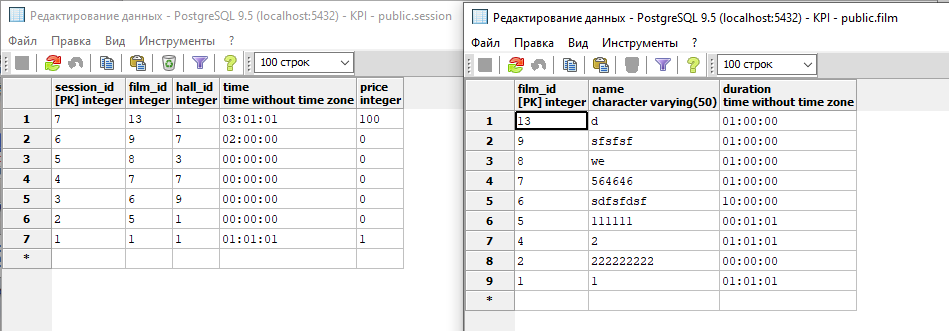
OWNER TO sysdba;

**Принцип роботи**

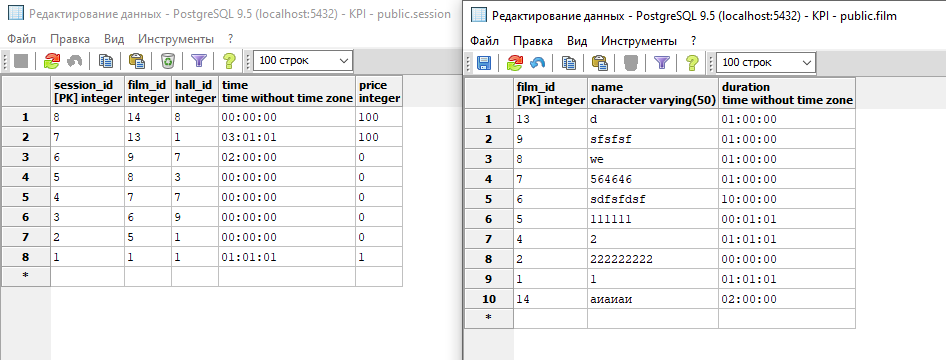
Тригер спрацьовує після вставки або редагування в таблиці «film». При додаванні будь-якого фільму у таблиці «session» буде створений сеанс у випадковий зал на вільний час, тобто час останнього сеансу в залі + 2 години (якщо вільного часу нема, або ще немає жодного сеансу в цьому залі – записуємо «00:00:00») та з ціною у 100 грн. При редагуванні новий сеанс утвориться за умови зміни id, в іншому новий сеанс не створюється.

***Приклад***

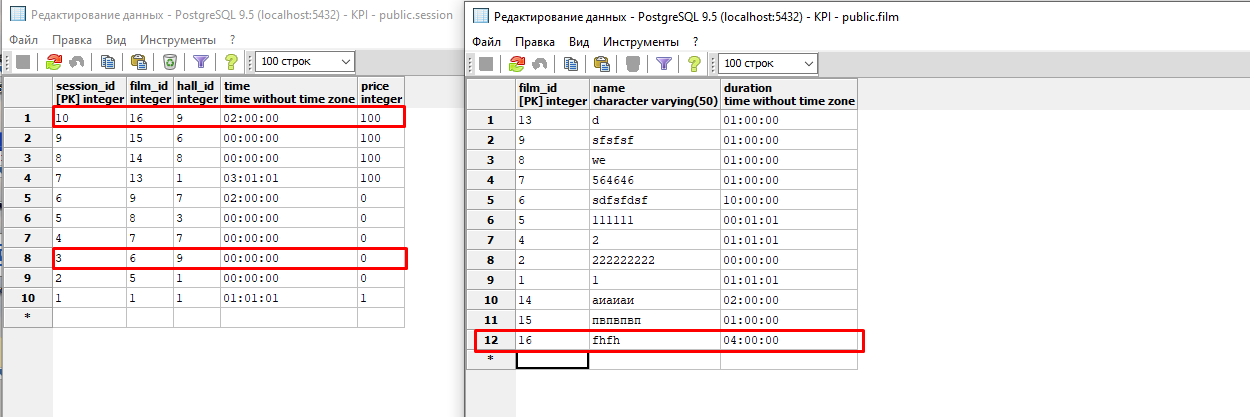
Таблиці до змін:



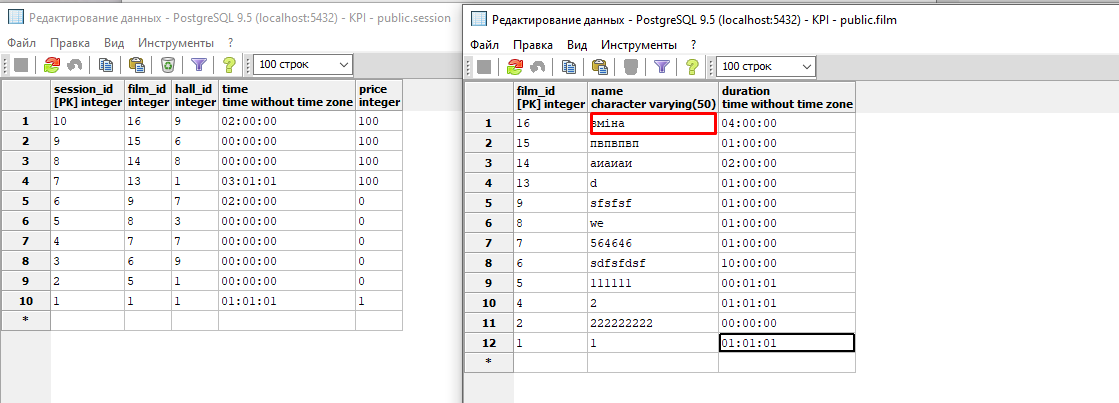
Додали новий фільм:



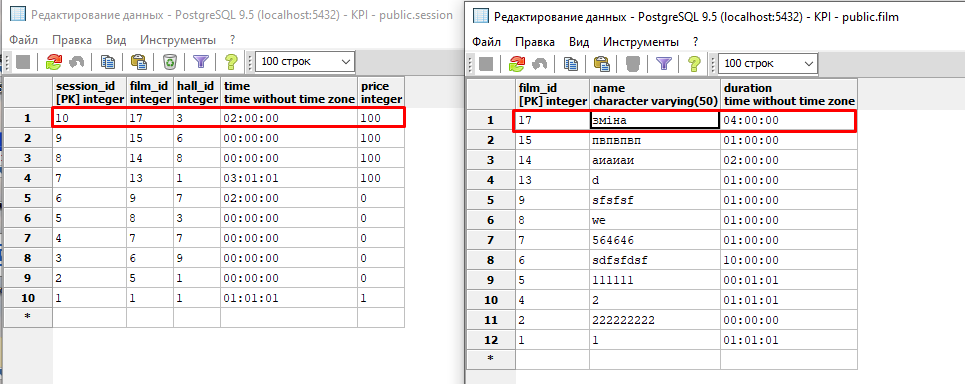
Новий фільм потрапив у зал, в якому вже будуть показувати один фільм:



Зміна назви:



Зміна id:



**Рівні ізоляції транзакцій у PostgreSQL**

Транзакція — це N (N≥1) запитів до БД, які успішно виконуються всі разом або зовсім невиконуються. Ізольованість транзакції показує те, наскільки сильно вони впливають одне на одного паралельно виконуються транзакції.

Вибираючи рівень транзакції, ми намагаємося дійти консенсусу у

виборі між високою узгодженістю даних між транзакціями та швидкістю

виконання цих транзакцій.

Варто зазначити, що найвищу швидкість виконання та найнижчу

узгодженість має рівень read uncommitted. Найнижчу швидкість виконання та

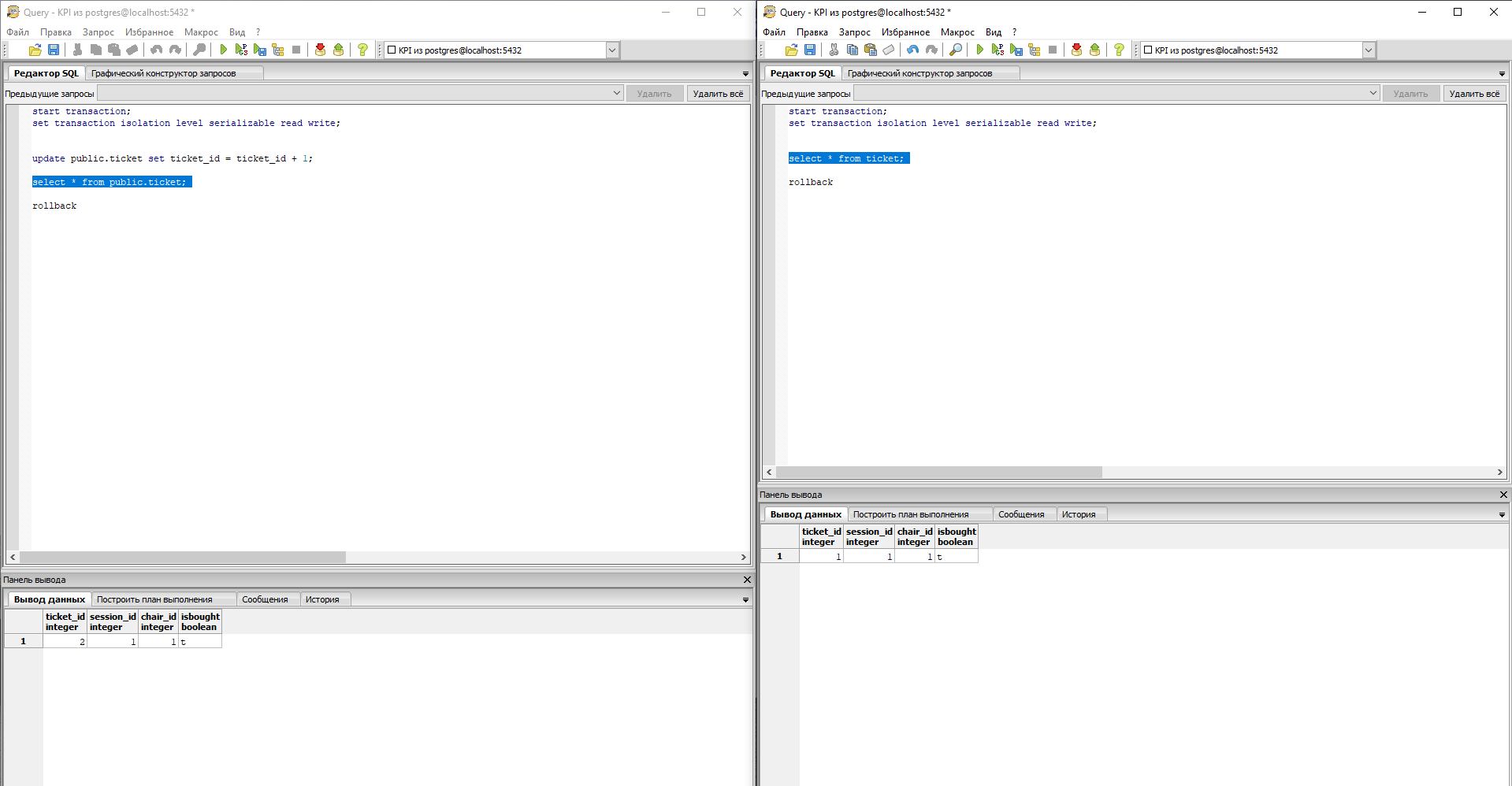
найвищу узгодженість — serializable.

При паралельному виконанні транзакцій можливі виникненя таких проблем:

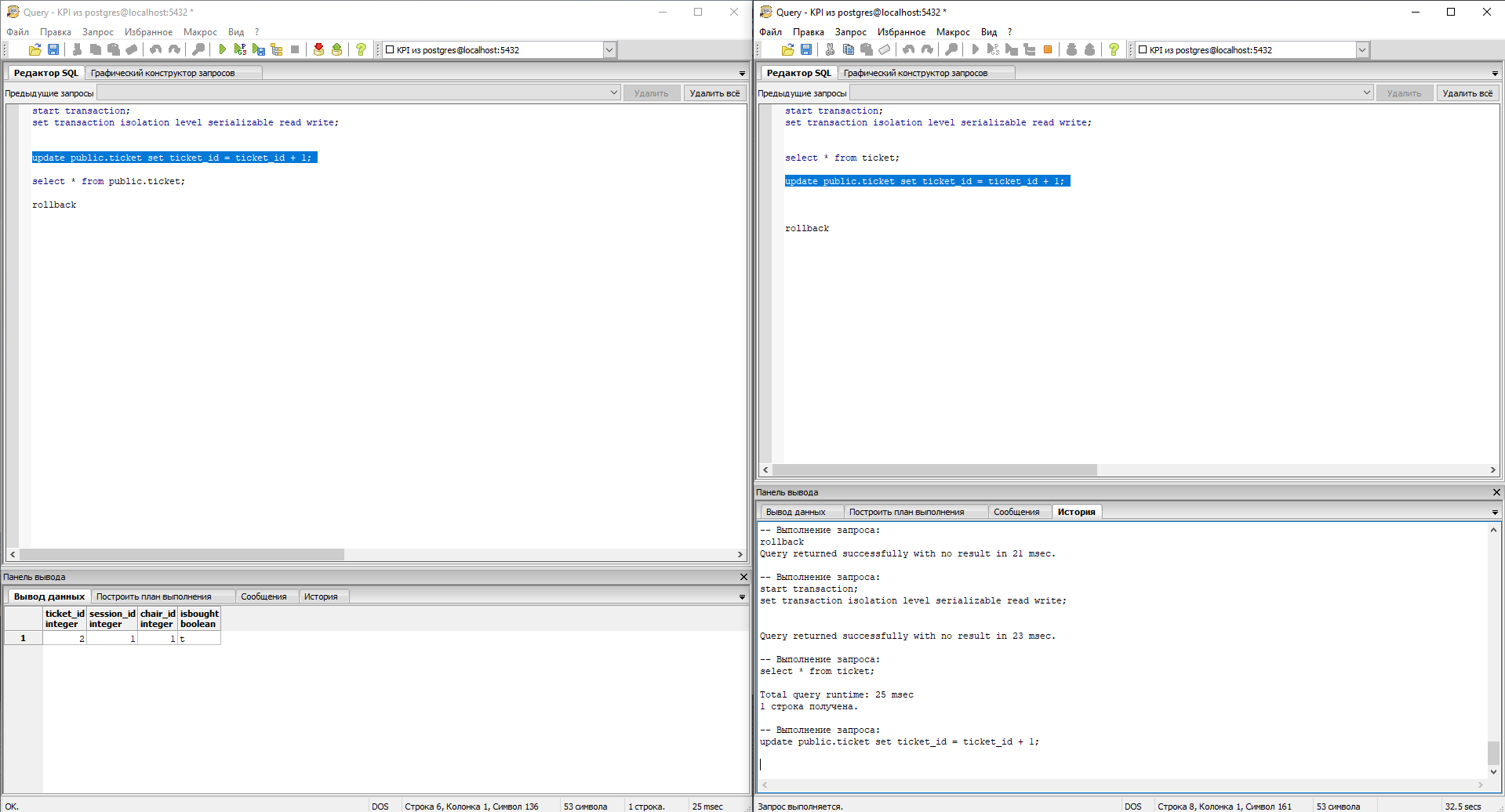
1. Втрачене оновлення - ситуація, коли при одночасній зміні одного блоку даних різними транзакціями, одна зі змін втрачається.
2. «Брудне» читання- читання даних, які додані чи змінені транзакцією, яка згодом не підтвердиться (відкотиться).
3. Неповторюване читання - ситуація, коли при повторному читанні в рамках однієї транзакції, раніше прочитані дані виявляються зміненими.
4. Фантомне читання - ситуація, коли при повторному читанні в рамках однієї транзакції одна і та ж вибірка дає різні множини рядків.

**Стандарт SQL-92**

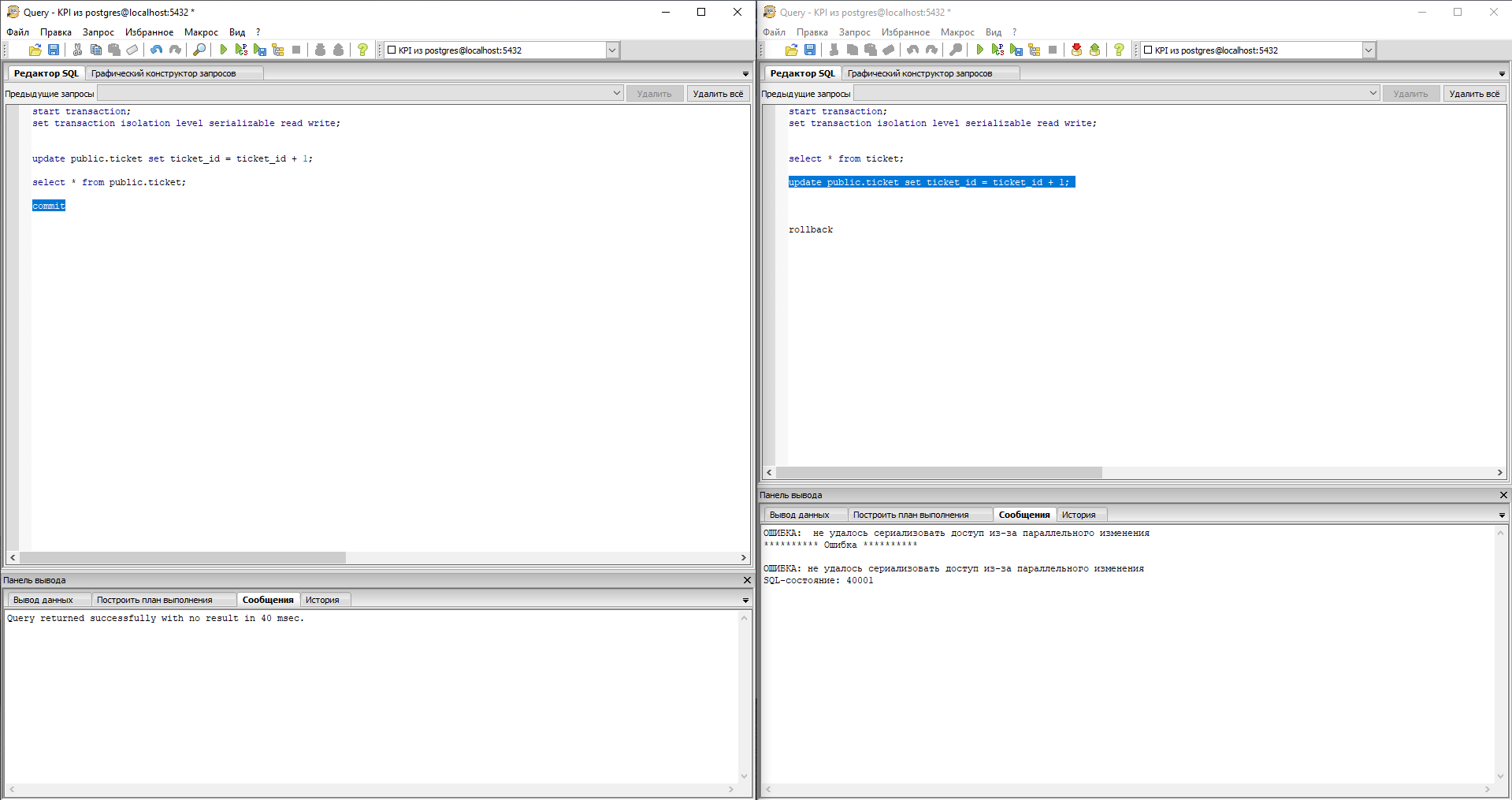
1. **Serializable (впорядкованість)**



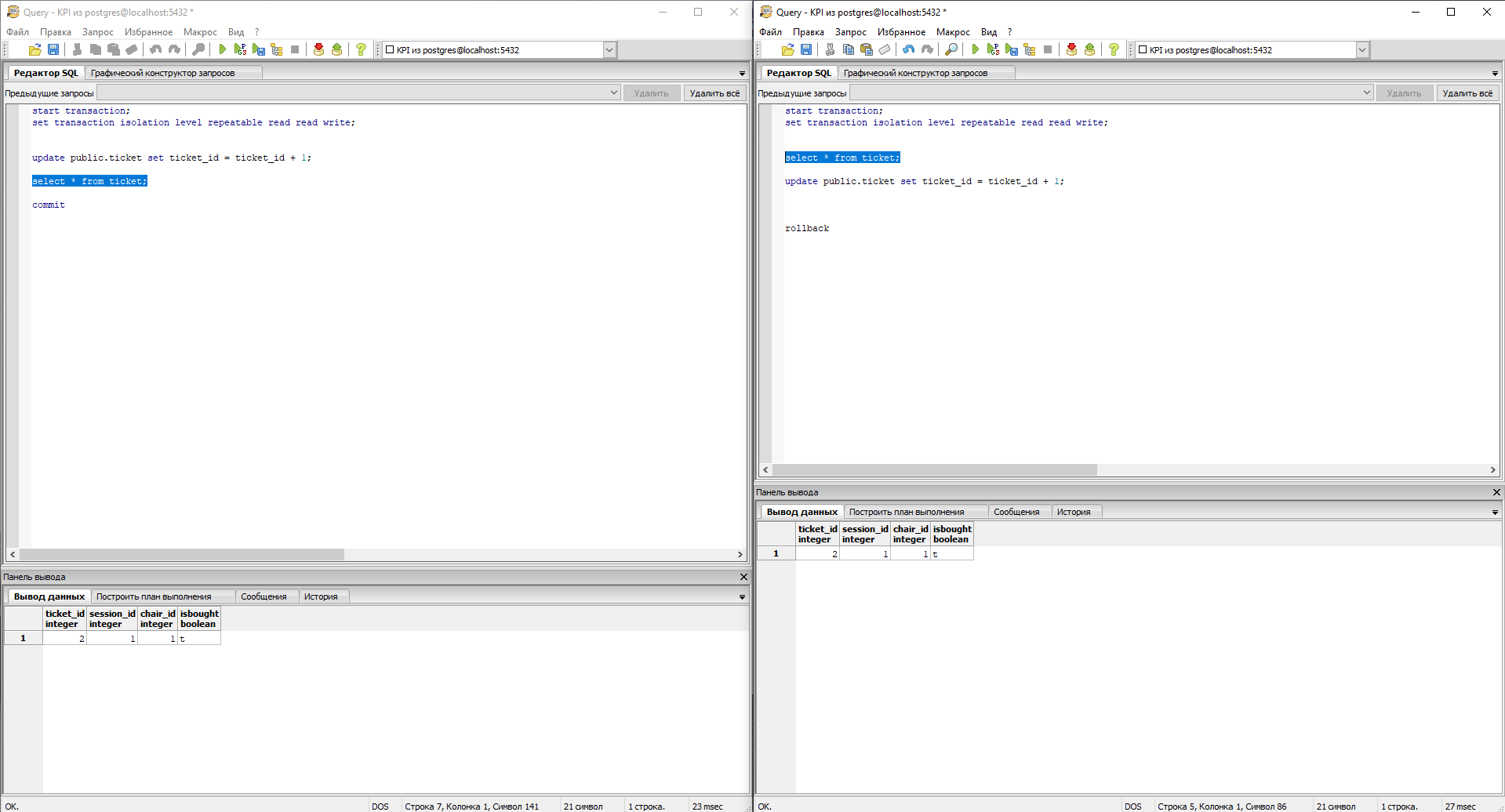
Як бачимо, дані у транзаціях ізольовано.



Тепер при оновлені даних в T2(фото зправа) бачимо, що T2 блокується поки T1 не не зафіксує зміни або не відмінить іх.

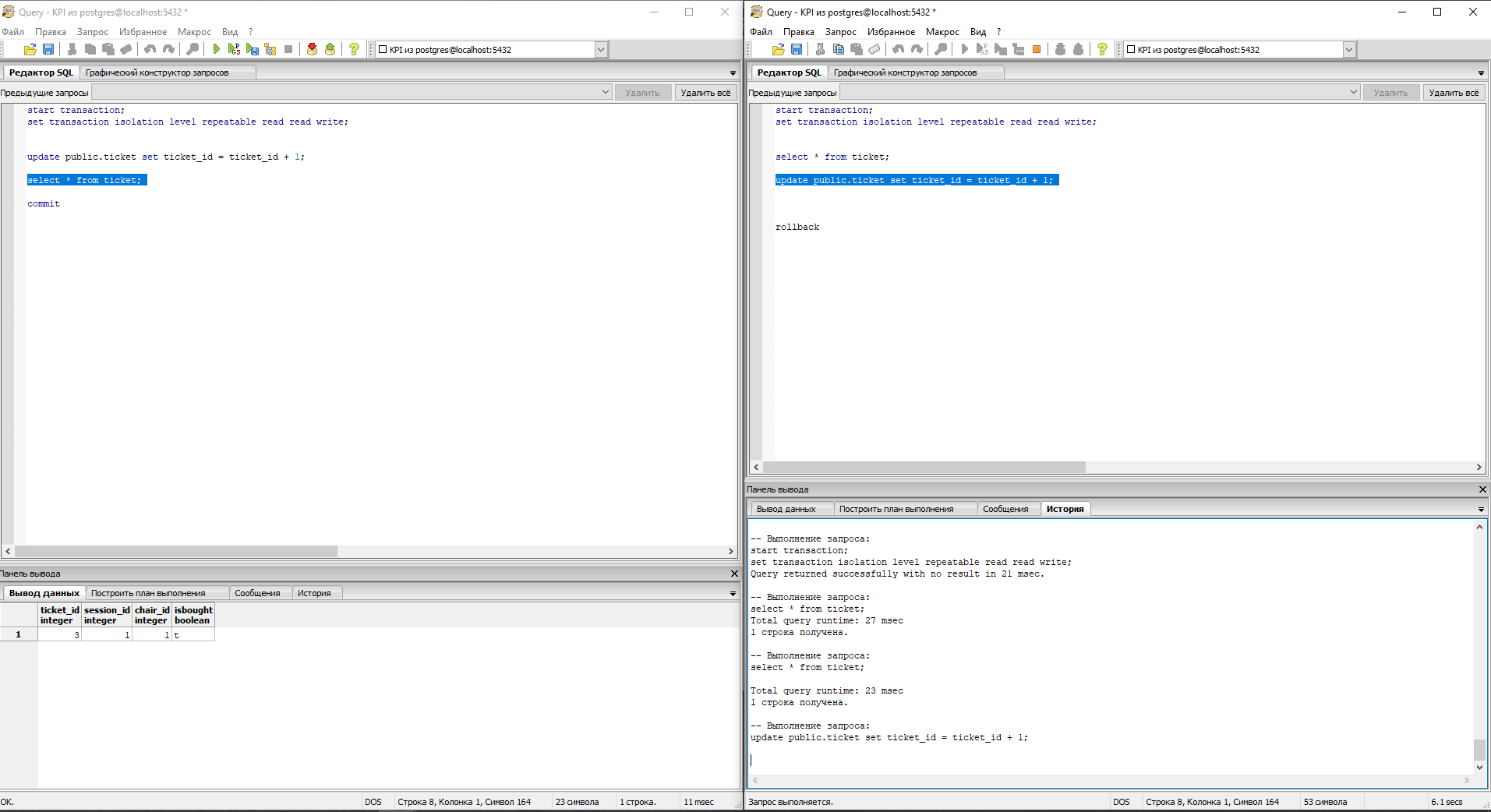


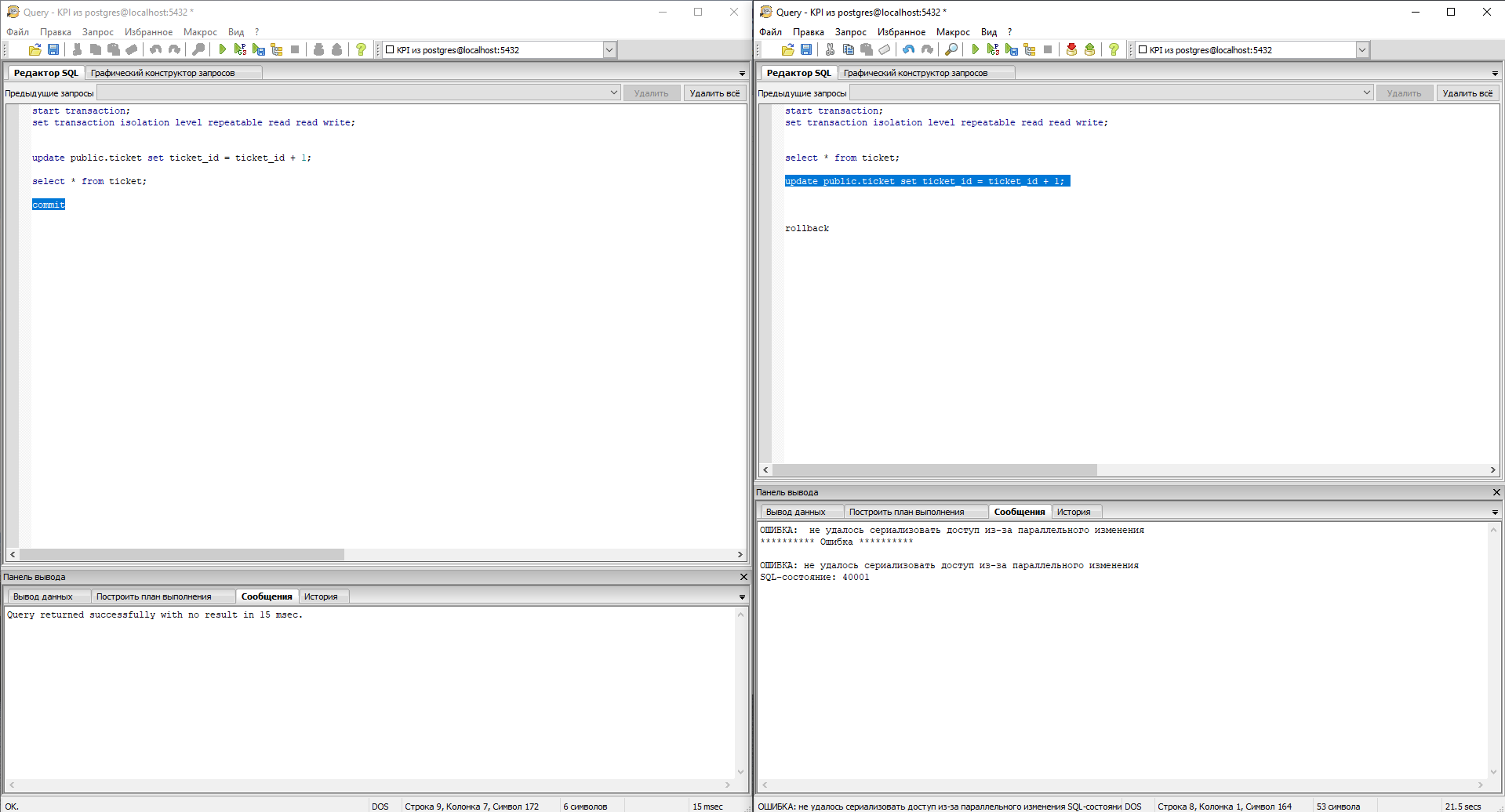
1. **Repeatable read (повторюваність читання)**

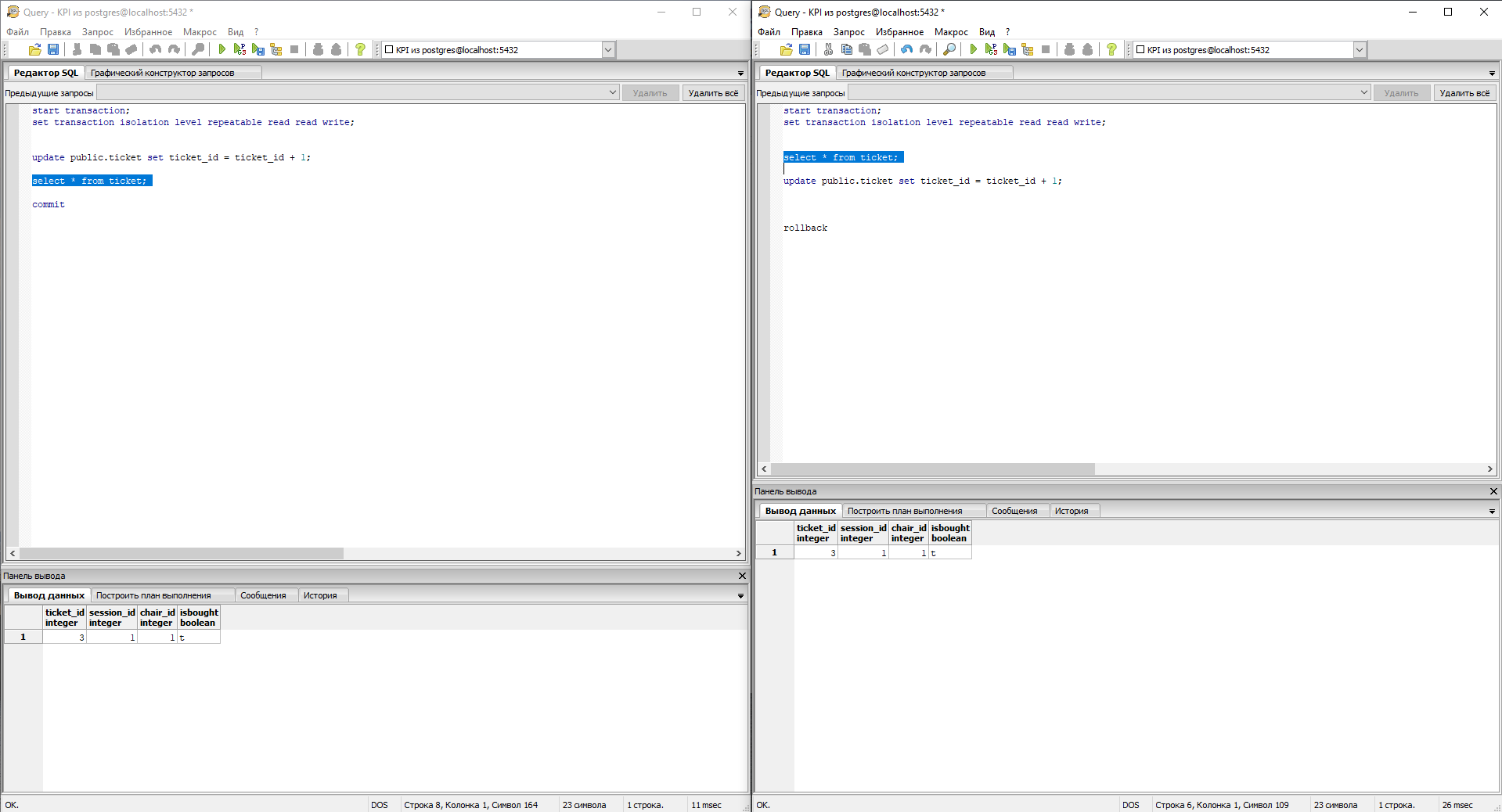


Тепер транзакція T2(зправа) буде чекати поки T1 не не зафіксує зміни

або не відмінить іх



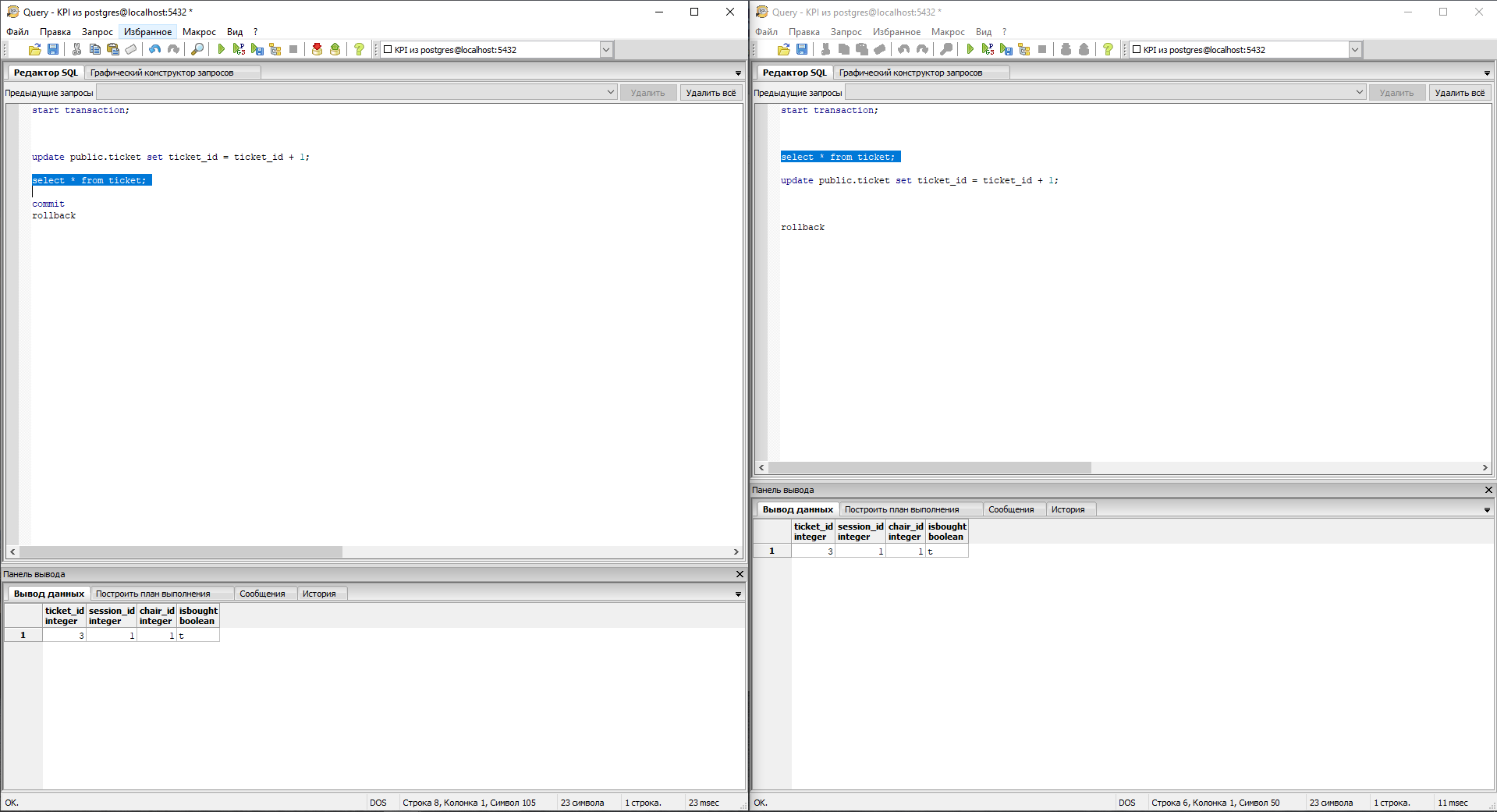


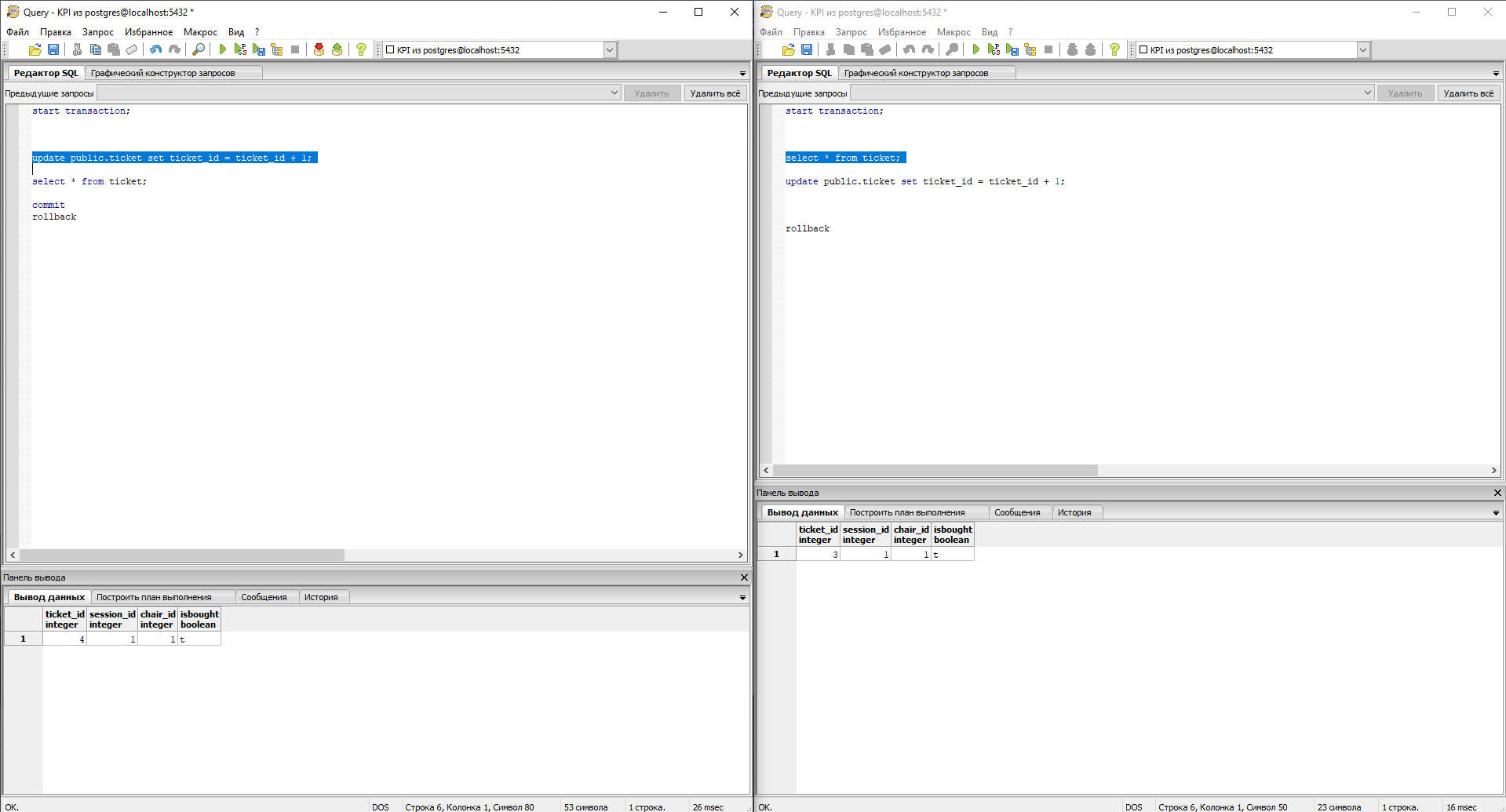


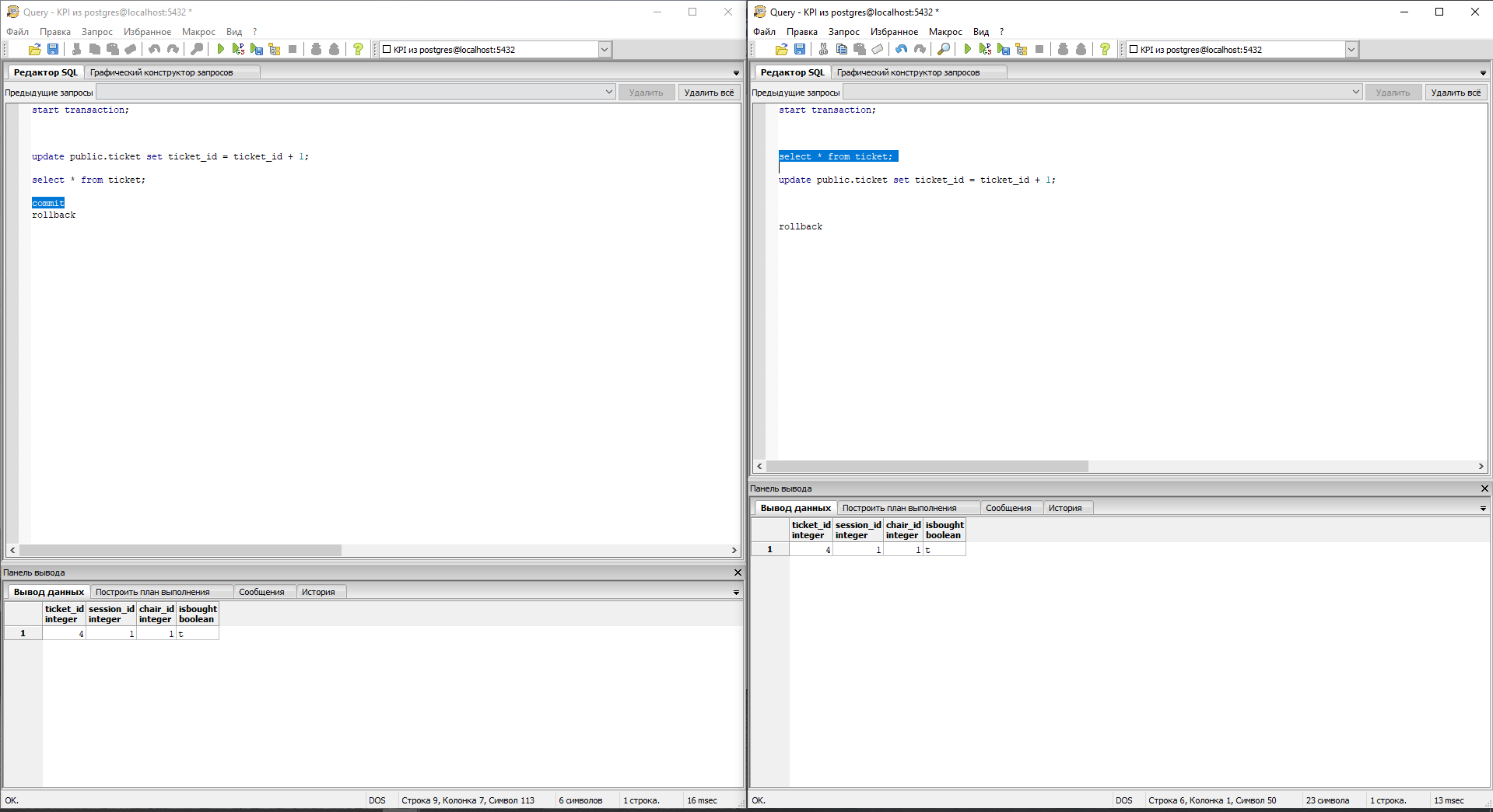
Як бачимо, Repeatable read не дозволяє виконувати операції зміни даних, якщо дані вже було модифіковано у іншій незавершеній транзакції. Тому використання Repeatable read рекомендоване тільки для режиму читаня.

1. **Read committed (читання фіксованих даних)**

Прийнятий за замовчуванням рівень для PostgreSQL. Закінчене читання, при якому відсутнє «брудне» читання (тобто, читання одним користувачем даних, що не були зафіксовані в БД командою COMMIT). Проте, в процесі роботи однієї транзакції інша може бути успішно закінчена, і зроблені нею зміни зафіксовані. В підсумку, перша транзакція буде працювати з іншим набором даних. Це проблема неповторюваного читання







1. **Read uncommitted (читання незафіксованих даних)**

Найнижчий рівень ізоляції, який відповідає рівню 0. Він гарантує тільки відсутність втрачених оновлень. Якщо декілька транзакцій одночасно намагались змінювати один і той же рядок, то в кінцевому варіанті рядок буде мати значення, визначений останньою успішно виконаною транзакцією. У PostgreSQL READ UNCOMMITTED розглядається як READ COMMITTED.