

**Національний технічний університет України
“Київський політехнічний інститут імені Ігоря
Сікорського”**

**Факультет прикладної математики
Кафедра системного програмування і спеціалізованих
комп'ютерних систем**

ЛАБОРАТОРНА РОБОТА № 3

з дисципліни
“Бази даних та засоби управління”

ТЕМА: “Засоби оптимізації роботи СУБД PostgreSQL”

Група: КВ-03
Виконав: Семенков М.С.
[GitHub](#)

[Telegram](#)

Оцінка:

Київ – 2022

Завдання на лабораторну роботу і вимоги до виконання

Метою роботи є здобуття практичних навичок використання засобів оптимізації СУБД PostgreSQL.

Загальне 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.

Варіант:

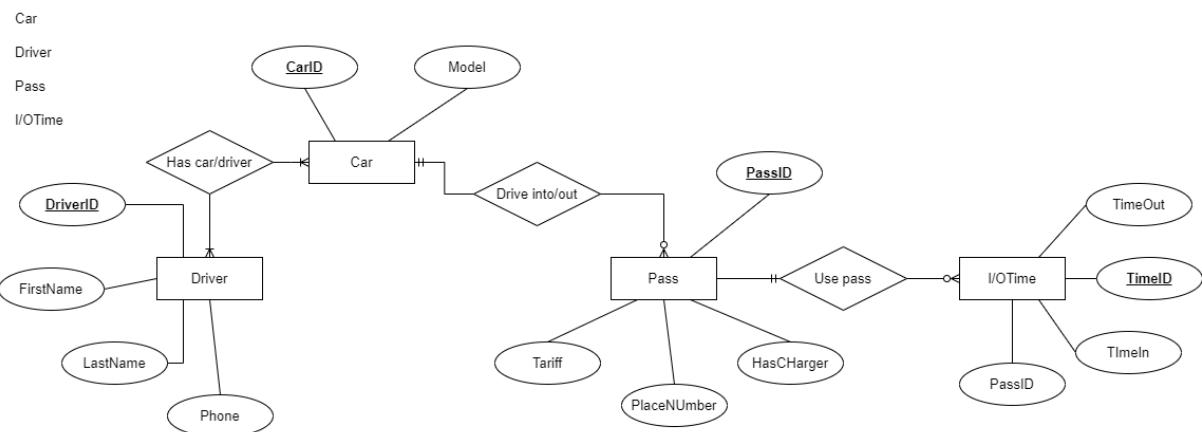
<i>№ варіанта</i>	<i>Види індексів</i>	<i>Умови для тригера</i>
20	<i>GIN, BRIN</i>	<i>after insert, update</i>

Опис та структура бази даних

Галузь розбита на 4 сутності: водій, машина, пропуск і час.

- Driver – сутність з описом водія, містить його ім’я, фамілію, унікальний ідентифікатор (номер прав) і контактний телефон. Ця сутність дозволяє ідентифікувати власників авто.
- Car – сутність з описом авто, містить назву моделі і унікальний ідентифікатор авто.
- Pass – сутність пропуска, містить його унікальний номер, ідентифікатор авто, до якого прив’язаний пропуск, номер місця на якому стане машина, інформацію про саме місце (наявність зарядки для електрокарів і тариф для цього місця). Ця сутність зв’язує авто з місцем на якому стоїть це авто.
- I/OTime – досить абстрактна сутність, призначена для нормалізації бази даних, це дозволяє не створювати нові записи при в’їзді/виїзді авто до/з паркінгу до таблиці сутності Pass. Ця сутність зберігає інформацію про час і ідентифікатор пропуску, до якого відносяться записи про час.

Parking:



Діаграма моделі «сущність-зв’язок» предметної галузі “Паркінг” у нотації Чена

Перетворити модуль “Модель” з шаблону MVC лабораторної роботи №2 у вигляд об’єктно-реляційної проекції (ORM)

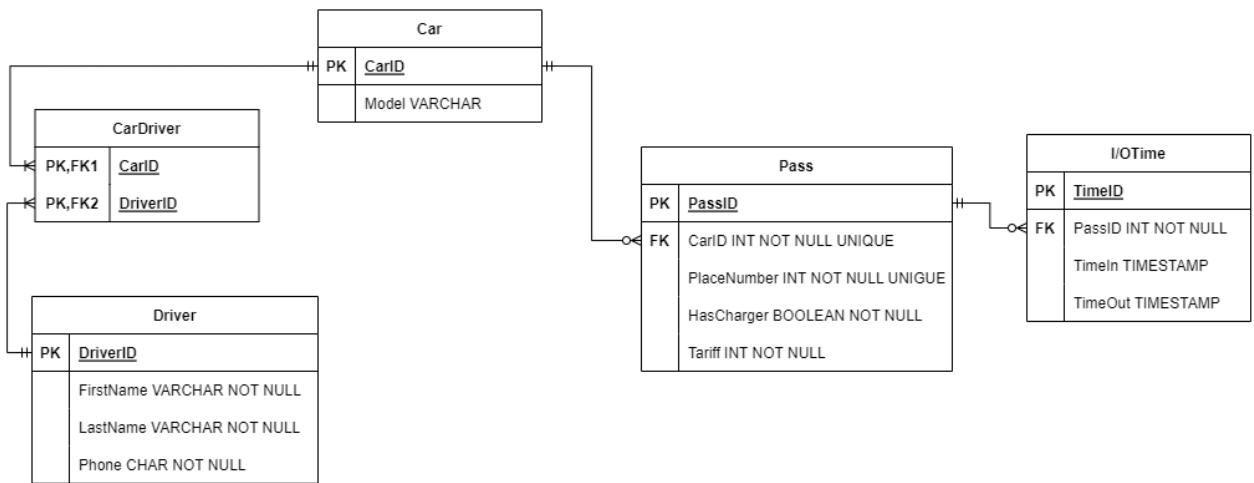


Схема бази даних у вигляді таблиць і зв’язків між ними

Для створення об’єктно-реляційної проекції, було використано Java-фреймворк Hibernate. Також, для зменшення кількості шаблонного коду було використано Lombok.

Класи, що описують сутності і їх зв'язки

Car.java

```
package lab3.hibernate.entities;

import jakarta.persistence.*;
import lombok.AllArgsConstructor;
import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;

import java.util.Collection;

@Entity
@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
public class Car {
    @Id
    @Column(name = "car_id")
    private int carId;

    private String model;

    @ManyToMany
    @JoinTable(name = "cardriver",
               joinColumns = {@JoinColumn(name = "car_id")},
               inverseJoinColumns = {@JoinColumn(name =
"driver_id")})
    private Collection<Driver> drivers = new LinkedList<>();

    @OneToMany(mappedBy = "car")
    private Collection<Pass> passes = new LinkedList<>();
}
```

Driver.java

```
package lab3.hibernate.entities;

import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.Id;
import jakarta.persistence.ManyToOne;
import lombok.AllArgsConstructor;
import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;

import java.util.Collection;

@Entity
@Getter
```

```

@Setter
@NoArgsConstructor
@AllArgsConstructor
public class Driver {
    @Id
    @Column(name = "driver_id")
    private int driverId;

    @Column(name = "first_name")
    private String firstName;

    @Column(name = "last_name")
    private String lastName;

    private String phone;

    @ManyToMany(mappedBy = "drivers")
    private Collection<Car> cars = new LinkedList<>();
}

```

IOTime.java

```

package lab3.hibernate.entities;

import jakarta.persistence.*;
import lombok.AllArgsConstructor;
import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;

import java.sql.Timestamp;

@Entity
@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
@Table(name = "io_time")
public class IOTime {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "time_id", nullable = false, insertable =
false, updatable = false)
    private int timeId;

    @Column(name = "pass_id", insertable = false, updatable =
false)
    private int passId;

    @Column(name = "time_in")
    private Timestamp timeIn;

    @Column(name = "time_out")
    private Timestamp timeOut;
}

```

```

    @ManyToOne
    @JoinColumn(name = "pass_id", referencedColumnName =
"pass_id")
    private Pass pass;
}

```

Pass.java

```

package lab3.hibernate.entities;

import jakarta.persistence.*;
import lombok.AllArgsConstructor;
import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;

import java.util.Collection;

@Entity
@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
public class Pass {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "pass_id")
    private int passId;

    @Column(name = "car_id", insertable = false, updatable =
false)
    private int carId;

    @Column(name = "place_number")
    private int placeNumber;

    @Column(name = "has_charger")
    private boolean hasCharger;

    private int tariff;

    @ManyToOne
    @JoinColumn(name = "car_id", referencedColumnName =
"car_id")
    private Car car;

    @OneToMany(mappedBy = "pass")
    private Collection<IOTime> times = new LinkedList<>();
}

```

Перетворені методи CRUD з використанням ORM

Read

```
List<List<String>> read(List<String> params, String tableName) {
    List<List<String>> tablesList = new LinkedList<>();
    CriteriaBuilder cb = session.getCriteriaBuilder();
    switch (tableName) {
        case "car" -> {
            CriteriaQuery<Car> cq = cb.createQuery(Car.class);
            Root<Car> rootEntry = cq.from(Car.class);
            CriteriaQuery<Car> all = cq.select(rootEntry);

            TypedQuery<Car> allQuery = session.createQuery(all);
            List<Car> resultList = allQuery.getResultList();

            int i = 0;
            for (Car car : resultList) {
                tablesList.add(new LinkedList<>());
                for (String param : params) {
                    switch (param) {
                        case "car_id" ->
                            tablesList.get(i).add(String.valueOf(car.getId()));
                        case "model" ->
                            tablesList.get(i).add(car.getModel());
                    }
                }
                i++;
            }
        }
        case "cardriver" -> {
            CriteriaQuery<Car> cq = cb.createQuery(Car.class);
            Root<Car> rootEntry = cq.from(Car.class);
            CriteriaQuery<Car> all = cq.select(rootEntry);

            TypedQuery<Car> allQuery = session.createQuery(all);
            List<Car> resultList = allQuery.getResultList();

            int i = 0;
            for (Car car : resultList) {
                for (Driver driver : car.getDrivers()) {
                    tablesList.add(new LinkedList<>());
                    for (String param : params) {
                        switch (param) {
                            case "car_id" ->
                                tablesList.get(i).add(String.valueOf(car.getId()));
                            case "driver_id" ->
                                tablesList.get(i).add(String.valueOf(driver.getId()));
                        }
                    }
                    i++;
                }
            }
        }
        case "driver" -> {
```

```

        CriteriaQuery<Driver> cq =
cb.createQuery(Driver.class);
        Root<Driver> rootEntry = cq.from(Driver.class);
        CriteriaQuery<Driver> all = cq.select(rootEntry);

        TypedQuery<Driver> allQuery =
session.createQuery(all);
        List<Driver> resultList = allQuery.getResultList();

        int i = 0;
        for (Driver driver : resultList) {
            tablesList.add(new LinkedList<>());
            for (String param : params) {
                switch (param) {
                    case "driver_id" ->
tablesList.get(i).add(String.valueOf(driver.getDriverId()));
                    case "first_name" ->
tablesList.get(i).add(driver.getFirstName());
                    case "last_name" ->
tablesList.get(i).add(driver.getLastName());
                    case "phone" ->
tablesList.get(i).add(driver.getPhone());
                }
            }
            i++;
        }
    }
    case "pass" -> {
        CriteriaQuery<Pass> cq = cb.createQuery(Pass.class);
        Root<Pass> rootEntry = cq.from(Pass.class);
        CriteriaQuery<Pass> all = cq.select(rootEntry);

        TypedQuery<Pass> allQuery =
session.createQuery(all);
        List<Pass> resultList = allQuery.getResultList();

        int i = 0;
        for (Pass pass : resultList) {
            tablesList.add(new LinkedList<>());
            for (String param : params) {
                switch (param) {
                    case "car_id" ->
tablesList.get(i).add(String.valueOf(pass.getCarId()));
                    case "place_number" ->
tablesList.get(i).add(String.valueOf(pass.getPlaceNumber()));
                    case "has_charger" ->
tablesList.get(i).add(String.valueOf(pass.isHasCharger()));
                    case "tariff" ->
tablesList.get(i).add(String.valueOf(pass.getTariff()));
                    case "pass_id" ->
tablesList.get(i).add(String.valueOf(pass.getId()));
                }
            }
            i++;
        }
    }
}

```

```

        case "i_otime" -> {
            CriteriaQuery<IOTime> cq =
cb.createQuery(IOTime.class);
            Root<IOTime> rootEntry = cq.from(IOTime.class);
            CriteriaQuery<IOTime> all = cq.select(rootEntry);

            TypedQuery<IOTime> allQuery =
session.createQuery(all);
            List<IOTime> resultList = allQuery.getResultList();

            int i = 0;
            for (IOTime ioTime : resultList) {
                tablesList.add(new LinkedList<>());
                for (String param : params) {
                    switch (param) {
                        case "pass_id" ->
tablesList.get(i).add(String.valueOf(ioTime.getPassId()));
                        case "time_in" ->
tablesList.get(i).add(String.valueOf(ioTime.getTimeIn()));
                        case "time_out" ->
tablesList.get(i).add(String.valueOf(ioTime.getTimeOut()));
                        case "time_id" ->
tablesList.get(i).add(String.valueOf(ioTime.getTimeId()));
                    }
                }
                i++;
            }
        }
        default -> System.out.println("Wrong table name");
    }

    return tablesList;
}

```

Create

```

int create(String tableName, List<String> values) {
    int result = -1;
    try {
        switch (tableName) {
            case "car" -> {
                Car car = new Car();
                car.setCarId(Integer.parseInt(values.get(0)));
                car.setModel(values.get(1));
                session.persist(car);
                result = 1;
            }
            case "cardriver" -> {
                Car car = session.find(Car.class,
Integer.parseInt(values.get(0)));
                Driver driver = session.find(Driver.class,
Integer.parseInt(values.get(1)));
                if (car != null && driver != null) {
                    car.getDrivers().add(driver);
                }
            }
        }
    }
    catch (Exception e) {
        e.printStackTrace();
    }
}

```

```

                result = 1;
            } else {
                System.out.println("Wrong car_id or
driver_id value");
            }
        }
    case "driver" -> {
        Driver driver = new Driver();

        driver.setDriverId(Integer.parseInt(values.get(0)));
        driver.setFirstName(values.get(1));
        driver.setLastName(values.get(2));
        driver.setPhone(values.get(3));
        session.persist(driver);
        result = 1;
    }
    case "pass" -> {
        Car car = session.find(Car.class,
Integer.parseInt(values.get(0)));
        if (car != null) {
            Pass pass = new Pass();

            pass.setCarId(Integer.parseInt(values.get(0)));
            pass.setPlaceNumber(Integer.parseInt(values.get(1)));
            pass.setHasCharger(Boolean.parseBoolean(values.get(2)));
            pass.setTariff(Integer.parseInt(values.get(3)));
            pass.setCar(car);
            session.persist(pass);
            result = 1;
        } else {
            System.out.println("Wrong car_id value");
        }
    }
    case "i_otime" -> {
        Pass pass = session.find(Pass.class,
Integer.parseInt(values.get(0)));
        if (pass != null) {
            IOTime ioTime = new IOTime();

            ioTime.setPassId(Integer.parseInt(values.get(0)));
            ioTime.setTimeIn(Timestamp.valueOf(values.get(1)));
            ioTime.setTimeOut(Timestamp.valueOf(values.get(2)));
            ioTime.setPass(pass);
            session.persist(ioTime);
            result = 1;
        } else {
            System.out.println("Wrong pass_id value");
        }
    }
    default -> System.out.println("Wrong table name");
}

```

```

        } catch (NumberFormatException ex) {
            System.out.println("Wrong value");
        }
    return result;
}

```

Update

```

int update(String tableName, String changeValue, String newValue,
String[] filterValues) {
    int result = -1;
    switch (tableName) {
        case "car" -> {
            Car car = session.find(Car.class, filterValues[0]);
            if (car != null) {
                switch (changeValue) {
                    case "car_id" ->
car.setCarId(Integer.parseInt(newValue));
                    case "model" -> car.setModel(newValue);
                }
                result = 1;
            }
        }
        case "cardriver" -> {
            Car car = session.find(Car.class, filterValues[0]);
            Driver driver = session.find(Driver.class,
filterValues[1]);
            if (car != null && driver != null) {
                car.getDrivers().remove(driver);
                driver.getCars().remove(car);
                switch (changeValue) {
                    case "car_id" -> car =
session.find(Car.class, Integer.parseInt(newValue));
                    case "driver_id" -> driver =
session.find(Driver.class, Integer.parseInt(newValue));
                }
                driver.getCars().add(car);
                car.getDrivers().add(driver);
                result = 1;
            }
        }
        case "driver" -> {
            Driver driver = session.find(Driver.class,
filterValues[0]);
            if (driver != null) {
                switch (changeValue) {
                    case "driver_id" ->
driver.setDriverId(Integer.parseInt(newValue));
                    case "first_name" ->
driver.setFirstName(newValue);
                    case "last_name" ->
driver.setLastName(newValue);
                    case "phone" -> driver.setPhone(newValue);
                }
            }
        }
    }
}

```

```

        }
        result = 1;
    }
}
case "pass" -> {
    Pass pass = session.find(Pass.class,
filterValues[0]);
    if (pass != null) {
        switch (changeValue) {
            case "car_id" ->
pass.setCarId(Integer.parseInt(newValue));
            case "place_number" ->
pass.setPlaceNumber(Integer.parseInt(newValue));
            case "has_charger" ->
pass.setHasCharger(Boolean.parseBoolean(newValue));
            case "tariff" ->
pass.setTariff(Integer.parseInt(newValue));
            case "pass_id" ->
pass.setPassId(Integer.parseInt(newValue));
        }
        result = 1;
    }
}
case "i_otime" -> {
    IOTime ioTime = session.find(IOTime.class,
filterValues[0]);
    if (ioTime != null) {
        switch (changeValue) {
            case "pass_id" ->
ioTime.setPassId(Integer.parseInt(newValue));
            case "time_in" ->
ioTime.setTimeIn(Timestamp.valueOf(newValue));
            case "time_out" ->
ioTime.setTimeOut(Timestamp.valueOf(newValue));
            case "time_id" ->
ioTime.setTimeId(Integer.parseInt(newValue));
        }
        result = 1;
    }
}
default -> System.out.println("Wrong table name");
}
return result;
}

```

Delete

```

int delete(String tableName, String[] primaryKeyValues) {
    int result = -1;
    switch (tableName) {
        case "car" -> {
            Car car = session.find(Car.class,
primaryKeyValues[0]);

```

```

        if (car != null) {
            session.remove(car);
            result = 1;
        }
    }
    case "cardriver" -> {
        Car car = session.find(Car.class,
primaryKeyValue[0]);
        Driver driver = session.find(Driver.class,
primaryKeyValue[1]);
        if (car != null && driver != null) {
            car.getDrivers().remove(driver);
            driver.getCars().remove(car);
            result = 1;
        }
    }
    case "driver" -> {
        Driver driver = session.find(Driver.class,
primaryKeyValue[0]);
        if (driver != null) {
            session.remove(driver);
            result = 1;
        }
    }
    case "pass" -> {
        Pass pass = session.find(Pass.class,
primaryKeyValue[0]);
        if (pass != null) {
            session.remove(pass);
            result = 1;
        }
    }
    case "i_otime" -> {
        IOTime ioTime = session.find(IOTime.class,
primaryKeyValue[0]);
        if (ioTime != null) {
            session.remove(ioTime);
            result = 1;
        }
    }
    default -> System.out.println("Wrong table name");
}
return result;
}

```

Результати виконання запитів

Read

```

Choose table
1 car
2 cardriver
3 driver
4 i_otime
5 pass
1
Choose columns (write numbers with comma without spaces)
1 All
2 car_id
3 model
1
[Hibernate]
select
    c1_0.car_id,
    c1_0.model
from
    Car c1_0
car_id          | model
1               | Audi
2               | Volkswagen
4               | BMW
5               | Mercedes Benz
3               | Tesla
Choose table
1 car
2 cardriver
3 driver
4 i_otime
5 pass
5
Choose columns (write numbers with comma without spaces)
1 All
2 driver_id
3 first_name
4 last_name
5 phone
5,4,2
[Hibernate]
select
    d1_0.driver_id,
    d1_0.first_name,
    d1_0.last_name,
    d1_0.phone
from
    Driver d1_0
First_name      | last_name      | driver_id
Alex            | Brown          | 22
Jhon            | Smith          | 11
Joanna          | Smith          | 55

```

Create

```

2
Choose table
1 car
2 cardriver
3 driver
4 i_otime
5 pass
1
Set new values
Enter car_id
11
Enter model
Toyota
Successfully created 1 rows

```

car_id	model
1	Audi
2	Volkswagen
4	BMW
5	Mercedes Benz
3	Tesla
11	Toyota

```

2
Choose table
1 car
2 cardriver
3 driver
4 i_otime
5 pass
5
Set new values
Enter car_id
11
Enter place_number
10
Enter has_charger
false
Enter tariff
9
[Hibernate]
insert
into
    Car
    (model, car_id)
values
    (?, ?)
[Hibernate]
insert
into
    Pass
    (car_id, has_charger, place_number, tariff)
values
    (?, ?, ?, ?)
Successfully created 1 rows

```

car_id	place_number	has_charger	tariff	pass_id
3	1	true	10	1
1	9	false	8	2
2	5	false	8	3
11	10	false	9	22
5	7	false	9	4

Update

```

3
Choose table
1 car
2 cardriver
3 driver
4 i_otime
5 pass
1
Choose value that you want change
1 car_id
2 model
2
Enter new value
Honda
Filter value (PK) should be equals to
11
Successfully updated 1 rows

```

car_id	model
1	Audi
2	Volkswagen
4	BMW
5	Mercedes Benz
3	Tesla
11	Honda

car_id	driver_id
1	55
1	11
2	55
4	22
5	22
3	11

car_id	driver_id
1	55
2	55
4	22
5	22
3	11
11	11

```

3
Choose table
1 car
2 cardriver
3 driver
4 i_otime
5 pass
2
Choose value that you want change
1 car_id
2 driver_id
1
Enter new value
11
Filter value (PK) should be equals to
1
Second part of filter should be equals to
11
[Hibernate]
  select
    c1_0.driver_id,
    c1_1.car_id,
    c1_1.model
  from
    cardriver c1_0
  join
    Car c1_1
    on c1_1.car_id=c1_0.car_id
  where
    c1_0.driver_id=?
Successfully updated 1 rows

```

Delete

```
4
Choose table
1 car
2 cardriver
3 driver
4 i_otime
5 pass
1
Filter value (PK) should be equals to
11
Successfully deleted 1 rows
```

car_id	model
1	Audi
2	Volkswagen
4	BMW
5	Mercedes Benz
3	Tesla

```
4
Choose table
1 car
2 cardriver
3 driver
4 i_otime
5 pass
5
Filter value (PK) should be equals to
22
[Hibernate]
select
    p1_0.pass_id,
    c1_0.car_id,
    c1_0.model,
    p1_0.car_id,
    p1_0.has_charger,
    p1_0.place_number,
    p1_0.tariff
from
    Pass p1_0
left join
    Car c1_0
        on c1_0.car_id=p1_0.car_id
where
    p1_0.pass_id=?
Successfully deleted 1 rows
```

car_id	place_number	has_charger	tariff	pass_id
3	1	true	10	1
1	9	false	8	2
2	5	false	8	3
5	7	false	9	4

Створити та проаналізувати різні типи індексів у PostgreSQL

GIN

Створення таблиці для тестування:

```
CREATE TABLE users (
    first_name text,
    last_name text
);
```

Заповнення даними:

```
INSERT into users(first_name, last_name) SELECT md5(random()::text),
md5(random()::text) FROM
(SELECT * FROM generate_series(1,1000000) AS id) AS x;
```

Створення індексів:

```
CREATE EXTENSION IF NOT EXISTS pg_trgm;
CREATE INDEX users_search_idx ON users USING gin (first_name
gin_trgm_ops, last_name gin_trgm_ops);
```

Порівняння швидкодії:

```
SELECT * FROM users;
```

Statistics per Node Type			
Node type	Count	Time spent	% of query
Seq Scan	1	288.262 ms	100%
Statistics per Relation			
Relation name	Scan count	Total time	% of query
Node type	Count	Sum of times	% of relation
public.users	1	288.262 ms	100%
Seq Scan	1	288.262 ms	100%

Без індексів

Statistics per Node Type			
Node type	Count	Time spent	% of query
Seq Scan	1	437.101 ms	100%
Statistics per Relation			
Relation name	Scan count	Total time	% of query
Node type	Count	Sum of times	% of relation
public.users	1	437.101 ms	100%
Seq Scan	1	437.101 ms	100%

З індексами

```
SELECT * FROM users WHERE first_name LIKE '%aeb%';
```

Statistics per Node Type

Node type	Count	Time spent	% of query
Bitmap Heap Scan	1	56.712 ms	96.64%
Bitmap Index Scan	1	1.974 ms	3.37%

Statistics per Relation

Relation name	Scan count	Total time	% of query
Node type	Count	Sum of times	% of relation
public.users	1	56.712 ms	96.64%
Bitmap Heap Scan	1	56.712 ms	100%

Без індексів

Statistics per Node Type

Node type	Count	Time spent	% of query
Gather	1	161.61 ms	48.22%
Seq Scan	1	173.593 ms	51.79%

Statistics per Relation

Relation name	Scan count	Total time	% of query
Node type	Count	Sum of times	% of relation
public.users	1	173.593 ms	51.79%
Seq Scan	1	173.593 ms	100%

З індексами

```
SELECT * FROM users ORDER BY last_name;
```

Statistics per Node Type

Node type	Count	Time spent	% of query
Gather Merge	1	1292.067 ms	30.07%
Seq Scan	1	15.98 ms	0.38%
Sort	1	2989.627 ms	69.57%

Statistics per Relation

Relation name	Scan count	Total time	% of query
Node type	Count	Sum of times	% of relation
public.users	1	15.98 ms	0.38%
Seq Scan	1	15.98 ms	100%

Без індексів

Statistics per Node Type

Node type	Count	Time spent	% of query
Gather Merge	1	1516.235 ms	32.02%
Seq Scan	1	17.298 ms	0.37%
Sort	1	3203.059 ms	67.63%

Statistics per Relation

Relation name	Scan count	Total time	% of query
Node type	Count	Sum of times	% of relation
public.users	1	17.298 ms	0.37%
Seq Scan	1	17.298 ms	100%

З індексами

```
SELECT * FROM users WHERE first_name LIKE '%aeb%' OR last_name LIKE '%k%' ORDER BY last_name;
```

Statistics per Node Type			
Node type	Count	Time spent	% of query
Bitmap Heap Scan	1	114.701 ms	75.14%
Bitmap Index Scan	2	1.469 ms	0.97%
Bitmap OR	1	0.009 ms	0.01%
Sort	1	36.482 ms	23.9%

Statistics per Relation			
Relation name	Scan count	Total time	% of query
Node type	Count	Sum of times	% of relation
public.users	1	114.701 ms	75.14%
Bitmap Heap Scan	1	114.701 ms	100%

Без індексів

Statistics per Node Type			
Node type	Count	Time spent	% of query
Gather Merge	1	99.03 ms	15.59%
Seq Scan	1	173.938 ms	27.38%
Sort	1	362.336 ms	57.04%

Statistics per Relation			
Relation name	Scan count	Total time	% of query
Node type	Count	Sum of times	% of relation
public.users	1	173.938 ms	27.38%
Seq Scan	1	173.938 ms	100%

З індексами

GIN-індекси зберігають набори пар – ключ і список появ, тобто даний тип індексів зберігає всі місця у які входить ключ. Це дозволяє швидко знаходити входження у якомусь наборі даних певних значень. Саме тому, цей індекс часто використовується для повнотекстового пошуку. Як видно з тестів, даний тип індекса дозволяє суттєво пришвидшити фільтрацію, а ось при сортуванні приросту майже немає. Це виходить з того, як влаштований індекс в середині.

BRIN

Створення таблиці для тестування:

```
CREATE TABLE temperature_log (log_id serial, log_timestamp timestamp
without time zone, temperature int);
```

Заповнення даними:

```
INSERT INTO temperature_log(log_timestamp,temperature)
VALUES
(generate_series('2022-01-01'::timestamp,'2022-12-31'::timestamp,
'1 minute'), round(random()*100)::int);
```

Створення індексів:

```
CREATE INDEX idx_temperature_log_log_timestamp ON temperature_log
USING BRIN (log_timestamp) WITH (pages_per_range = 128);
```

Порівняння швидкодії:

```
SELECT * FROM temperature_log WHERE log_timestamp BETWEEN
'2022-04-01' AND '2022-04-07';
```

Statistics per Node Type			
Node type	Count	Time spent	% of query
Gather	1	87.473 ms	79.54%
Seq Scan	1	22.505 ms	20.47%
Statistics per Relation			
Relation name	Scan count	Total time	% of query
Node type	Count	Sum of times	% of relation
public.temperature_log	1	22.505 ms	20.47%
Seq Scan	1	22.505 ms	100%

Без індексів

Statistics per Node Type			
Node type	Count	Time spent	% of query
Bitmap Heap Scan	1	2.173 ms	57.11%
Bitmap Index Scan	1	1.632 ms	42.9%
Statistics per Relation			
Relation name	Scan count	Total time	% of query
Node type	Count	Sum of times	% of relation
public.temperature_log	1	2.173 ms	57.11%
Bitmap Heap Scan	1	2.173 ms	100%

З індексами

```
SELECT AVG(temperature) FROM temperature_log WHERE log_timestamp
BETWEEN '2022-09-01' AND '2022-10-01';
```

Statistics per Node Type			
Node type	Count	Time spent	% of query
Aggregate	2	19.622 ms	7.21%
Gather	1	244.162 ms	89.69%
Seq Scan	1	8.447 ms	3.11%
Statistics per Relation			
Relation name	Scan count	Total time	% of query
Node type	Count	Sum of times	% of relation
public.temperature_log	1	8.447 ms	3.11%
Seq Scan	1	8.447 ms	100%

Без індексів

Statistics per Node Type			
Node type	Count	Time spent	% of query
Aggregate	1	4.101 ms	21.46%
Bitmap Heap Scan	1	9.526 ms	49.85%
Bitmap Index Scan	1	5.483 ms	28.7%
Statistics per Relation			
Relation name	Scan count	Total time	% of query
Node type	Count	Sum of times	% of relation
public.temperature_log	1	9.526 ms	49.85%
Bitmap Heap Scan	1	9.526 ms	100%

З індексами

```
SELECT MIN(log_timestamp) FROM temperature_log WHERE temperature BETWEEN 10 AND 20;
```

Statistics per Node Type

Node type	Count	Time spent	% of query
Aggregate	2	14.387 ms	6.62%
Gather	1	196.759 ms	90.41%
Seq Scan	1	6.491 ms	2.99%

Statistics per Relation

Relation name	Scan count	Total time	% of query
Node type	Count	Sum of times	% of relation
public.temperature_log	1	6.491 ms	2.99%
Seq Scan	1	6.491 ms	100%

Без індексів

Statistics per Node Type

Node type	Count	Time spent	% of query
Aggregate	2	36.779 ms	22.36%
Gather	1	111.443 ms	67.74%
Seq Scan	1	16.302 ms	9.91%

Statistics per Relation

Relation name	Scan count	Total time	% of query
Node type	Count	Sum of times	% of relation
public.temperature_log	1	16.302 ms	9.91%
Seq Scan	1	16.302 ms	100%

З індексами

```
SELECT * FROM temperature_log WHERE log_timestamp BETWEEN '2022-04-01' AND '2022-04-07' ORDER BY temperature;
```

Statistics per Node Type

Node type	Count	Time spent	% of query
Gather Merge	1	83.964 ms	69.37%
Seq Scan	1	11.625 ms	9.61%
Sort	1	25.464 ms	21.04%

Statistics per Relation

Relation name	Scan count	Total time	% of query
Node type	Count	Sum of times	% of relation
public.temperature_log	1	11.625 ms	9.61%
Seq Scan	1	11.625 ms	100%

Без індексів

Statistics per Node Type

Node type	Count	Time spent	% of query
Bitmap Heap Scan	1	1.775 ms	51.56%
Bitmap Index Scan	1	0.027 ms	0.79%
Sort	1	1.641 ms	47.67%

Statistics per Relation

Relation name	Scan count	Total time	% of query
Node type	Count	Sum of times	% of relation
public.temperature_log	1	1.775 ms	51.56%
Bitmap Heap Scan	1	1.775 ms	100%

З індексами

BRIN-індекси дозволяють пришвидшити пошук потрібної інформації, за рахунок зменшення області пошуку, бо ці індекси дозволяють дізнатися де потрібних значень точно не може бути. Для коректної роботи потрібно, щоб проіндексовані поля були відсортовані, або близькі до цього. Як видно з тестів, даний тип індексів суттєво пришвидшує пошук по проіндексованим параметрам.

Розробити тригер бази даних PostgreSQL

After insert, update

Тестові таблиці:

```
CREATE TABLE student(id SERIAL PRIMARY KEY, name TEXT, mark INTEGER, date TIMESTAMP);
```

```
CREATE TABLE exam(id SERIAL PRIMARY KEY, student_id INTEGER, pass BOOLEAN, in_time BOOLEAN);
```

Створення триггера:

```
CREATE TRIGGER after_insert_update_trigger
AFTER INSERT OR UPDATE ON student
FOR EACH ROW
EXECUTE PROCEDURE after_insert_update_proc();

CREATE OR REPLACE FUNCTION after_insert_update_proc() RETURNS
TRIGGER AS $trigger$
DECLARE
cursor_log CURSOR FOR SELECT * FROM student;
row_student%ROWTYPE;
exist BOOLEAN = false;
BEGIN
    FOR row IN cursor_log LOOP
        exist = EXISTS(SELECT * FROM exam WHERE student_id =
row_.id);

        IF row_.mark > 60 THEN
            IF row_.date <= '2022-12-20' THEN
                IF exist = false THEN
                    INSERT INTO exam(student_id, pass,
in_time) VALUES (row_.id, true, true);
                ELSE
                    UPDATE exam SET pass = true,
in_time = true WHERE student_id = row_.id;
                END IF;
            ELSE
                IF row_.mark >= 70 THEN
```

```

        IF exist = false THEN
            INSERT INTO exam(student_id,
pass, in_time) VALUES (row_.id, true, false);
        ELSE
            UPDATE exam SET pass = true,
in_time = false WHERE student_id = row_.id;
        END IF;
        ELSE
            IF exist = false THEN
                INSERT INTO exam(student_id,
pass, in_time) VALUES (row_.id, false, false);
            ELSE
                UPDATE exam SET pass = false,
in_time = false WHERE student_id = row_.id;
            END IF;
        END IF;
        ELSE
            IF row_.date <= '2022-12-20' THEN
                IF exist = false THEN
                    INSERT INTO exam(student_id, pass,
in_time) VALUES (row_.id, false, true);
                ELSE
                    UPDATE exam SET pass = false,
in_time = true WHERE student_id = row_.id;
                END IF;
            ELSE
                IF exist = false THEN
                    INSERT INTO exam(student_id, pass,
in_time) VALUES (row_.id, false, false);
                ELSE
                    UPDATE exam SET pass = false,
in_time = false WHERE student_id = row_.id;
                END IF;
            END IF;
        END IF;
    END LOOP;
    RETURN new;
EXCEPTION
    WHEN OTHERS THEN
        RAISE NOTICE 'Something went wrong';
    END;
$trigger$ LANGUAGE plpgsql;

```

Триггер, при занесенні нових студентів, або при оновленні даних студента, автоматично оновлює таблицю exam відповідно до кількості балів і своєчасності виконання. Триггер кожен раз знову читує всі записи і перевіряє чи є такий запис у таблиці чи ні. Якщо він є, то цей запис буде оновлено, якщо ні, то буде занесено новий. Також програма має можливість перехоплювати помилки, але вона занадто проста, щоб можна було створити помилкову ситуацію.

Тести:

Початковий стан таблиць

student

id [PK] integer	name text	mark integer	date timestamp without time zone
---------------------------	---------------------	------------------------	--------------------------------------------

exam

id [PK] integer	student_id integer	pass boolean	in_time boolean
---------------------------	------------------------------	------------------------	---------------------------

Вставка нових значень

```
INSERT INTO student(name, mark, date) VALUES('Mikhail', 99, '2022-12-01');
INSERT INTO student(name, mark, date) VALUES('Dmytro', null, '2022-12-01');
INSERT INTO student(name, mark, date) VALUES('Pedro', 70, '2022-12-30');
INSERT INTO student(name, mark, date) VALUES('Ivan', 50, '2022-12-10');
```

student

	id [PK] integer	name text	mark integer	date timestamp without time zone
1	1	Mikhail	99	2022-12-01 00:00:00
2	2	Dmytro	[null]	2022-12-01 00:00:00
3	3	Pedro	70	2022-12-30 00:00:00
4	4	Ivan	50	2022-12-10 00:00:00

exam

	id [PK] integer	student_id integer	pass boolean	in_time boolean
1	1	1	true	true
2	2	2	false	true
3	3	3	true	false
4	4	4	false	true

Оновлення занесених даних

```
UPDATE student SET mark = 90 WHERE id = 2;  
UPDATE student SET date = '2022-12-25' WHERE id = 4;
```

student

	id [PK] integer	name text	mark integer	date timestamp without time zone
1	1	Mikhail	99	2022-12-01 00:00:00
2	3	Pedro	70	2022-12-30 00:00:00
3	2	Dmytro	90	2022-12-01 00:00:00
4	4	Ivan	50	2022-12-25 00:00:00

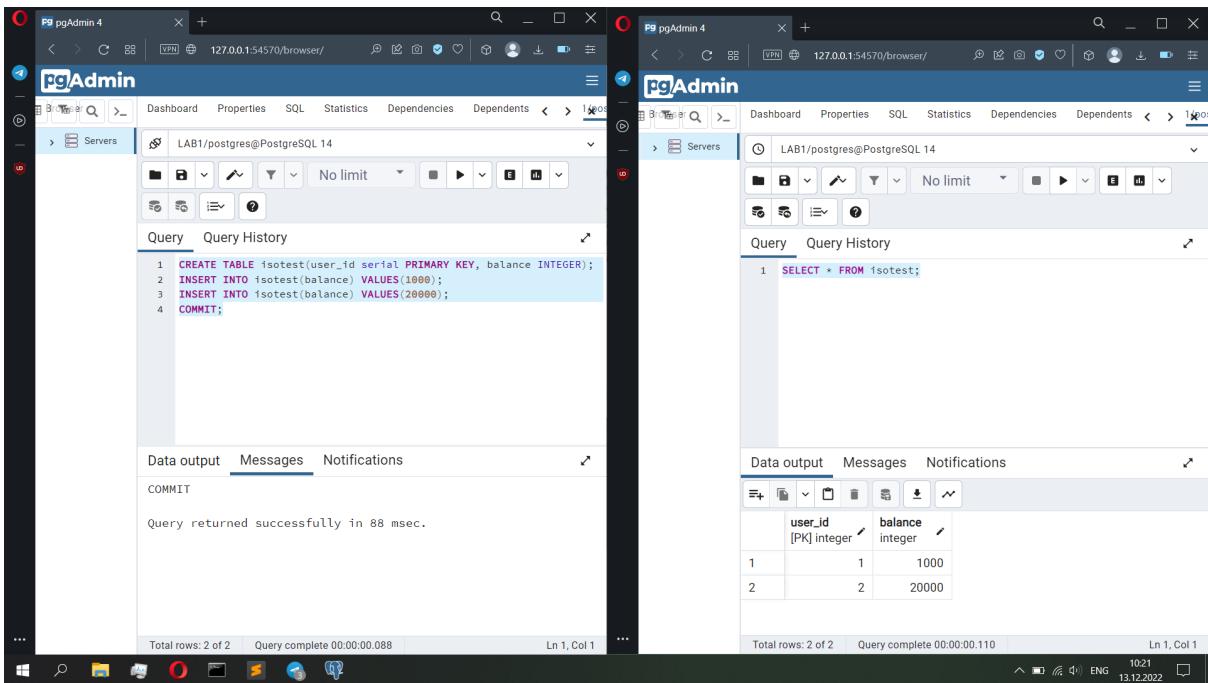
exam

	id [PK] integer	student_id integer	pass boolean	in_time boolean
1	1	1	true	true
2	3	3	true	false
3	2	2	true	true
4	4	4	false	false

Триггер коректно реагує на всі передбачені зміни: нові дані заносяться, старі оновлюються.

Навести приклади та проаналізувати рівні ізоляції транзакцій у PostgreSQL

Створюємо тестову таблицю

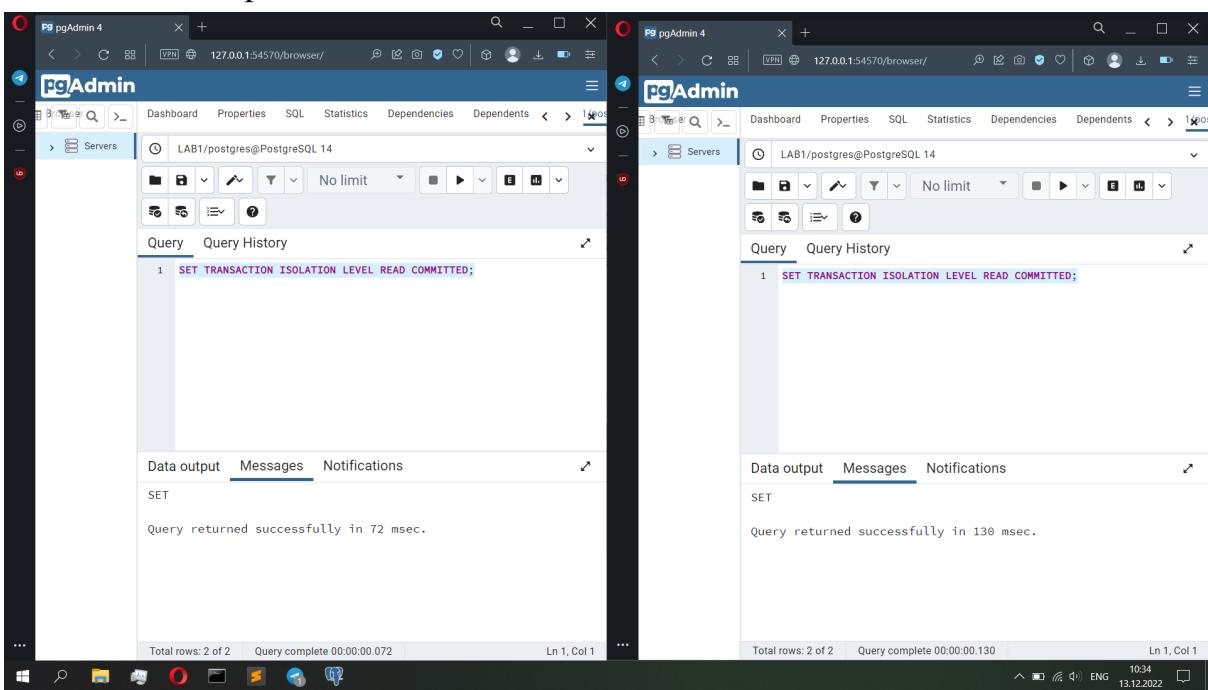


```
1 CREATE TABLE isotest(user_id serial PRIMARY KEY, balance INTEGER);
2 INSERT INTO isotest(balance) VALUES(1000);
3 INSERT INTO isotest(balance) VALUES(20000);
4 COMMIT;
```

user_id	balance
1	1000
2	20000

Read committed

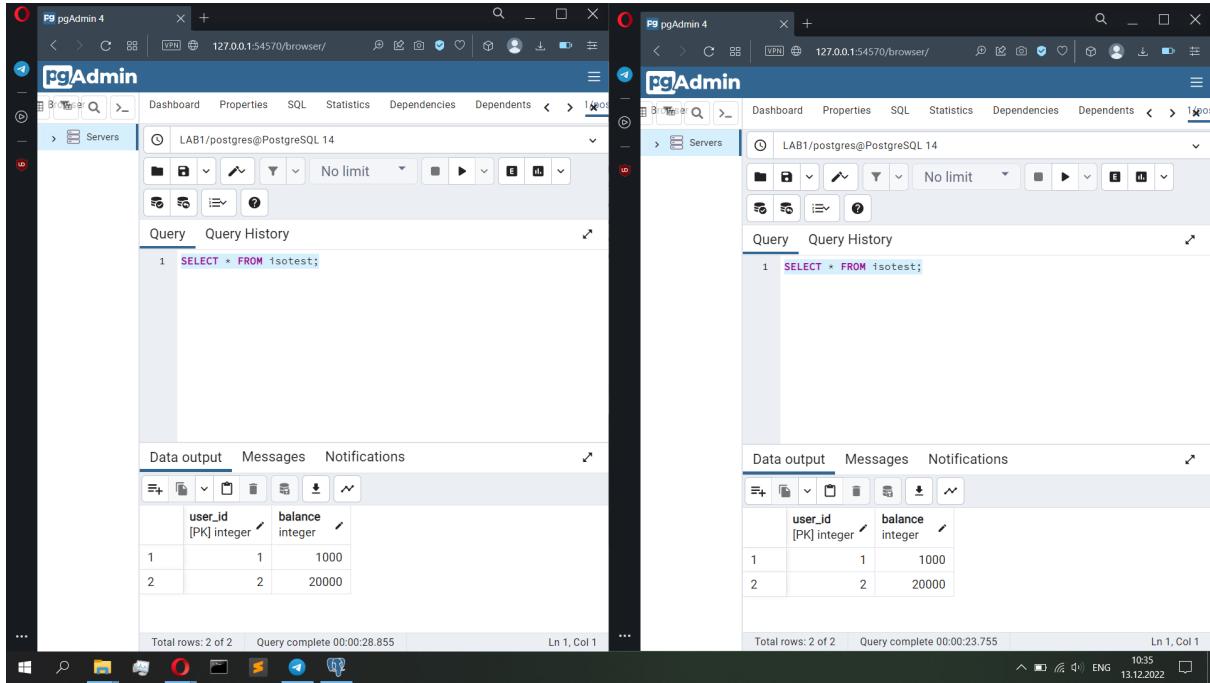
Встановлюємо рівень ізоляції



```
1 SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
```

user_id	balance
1	1000
2	20000

У початковий момент часу всім транзакціям доступна початкова версія даних



The screenshot shows two separate pgAdmin 4 sessions running on the same server (LAB1/postgres@PostgreSQL 14). Both sessions have the same initial data in the 'isotest' table:

user_id	balance
1	1000
2	20000

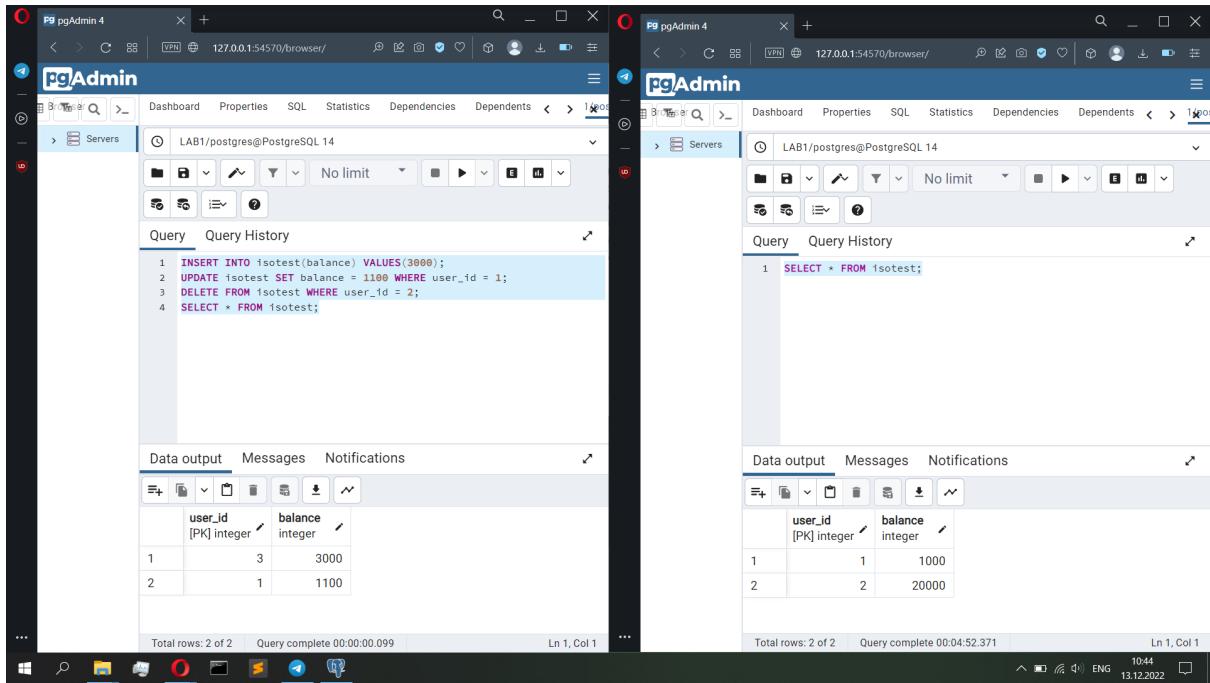
Session 1 (left):
Query: `SELECT * FROM isotest;`
Data output:

user_id	balance
1	1000
2	20000

Session 2 (right):
Query: `SELECT * FROM isotest;`
Data output:

user_id	balance
1	1000
2	20000

Зміни виконані у першій транзакції до її закриття будуть доступні лише всередині неї (заборонений Dirty Read)



The screenshot shows two pgAdmin 4 sessions. The left session (Session 1) contains the following SQL queries:

```
1 INSERT INTO isotest(balance) VALUES(3000);
2 UPDATE isotest SET balance = 1100 WHERE user_id = 1;
3 DELETE FROM isotest WHERE user_id = 2;
4 SELECT * FROM isotest;
```

After executing these queries, the data in the 'isotest' table is updated:

user_id	balance
1	3000
2	1100

The right session (Session 2) shows the original data from before the updates:

user_id	balance
1	1000
2	20000

Після закриття першої транзакції, зміни зберігаються і доступні з інших транзакцій (дозволені Nonrepeatable Read і Phantom Read)

pgAdmin 4

LAB1/postgres@PostgreSQL 14

Query History

1 COMMIT;

Data output Messages Notifications

1 COMMIT

Query returned successfully in 82 msec.

Total rows: 2 of 2 Query complete 00:00:00.082 Ln 1, Col 1

pgAdmin 4

LAB1/postgres@PostgreSQL 14

Query History

1 SELECT * FROM isotest;

Data output Messages Notifications

user_id [PK] integer ↗ balance integer ↗

user_id	balance
1	3
2	1100

Total rows: 2 of 2 Query complete 00:00:00.107 Ln 1, Col 1

Спроба паралельної роботи з даними без завершення транзакції

pgAdmin 4

LAB1/postgres@PostgreSQL 14

Query History

1 UPDATE isotest SET balance = 1200 WHERE user_id = 1;

Data output Messages Notifications

1 UPDATE 1

Query returned successfully in 10 secs 792 msec.

Total rows: 2 of 2 Query complete 00:00:10.792 Ln 1, Col 1

pgAdmin 4

LAB1/postgres@PostgreSQL 14

Query History

1 DELETE FROM isotest;

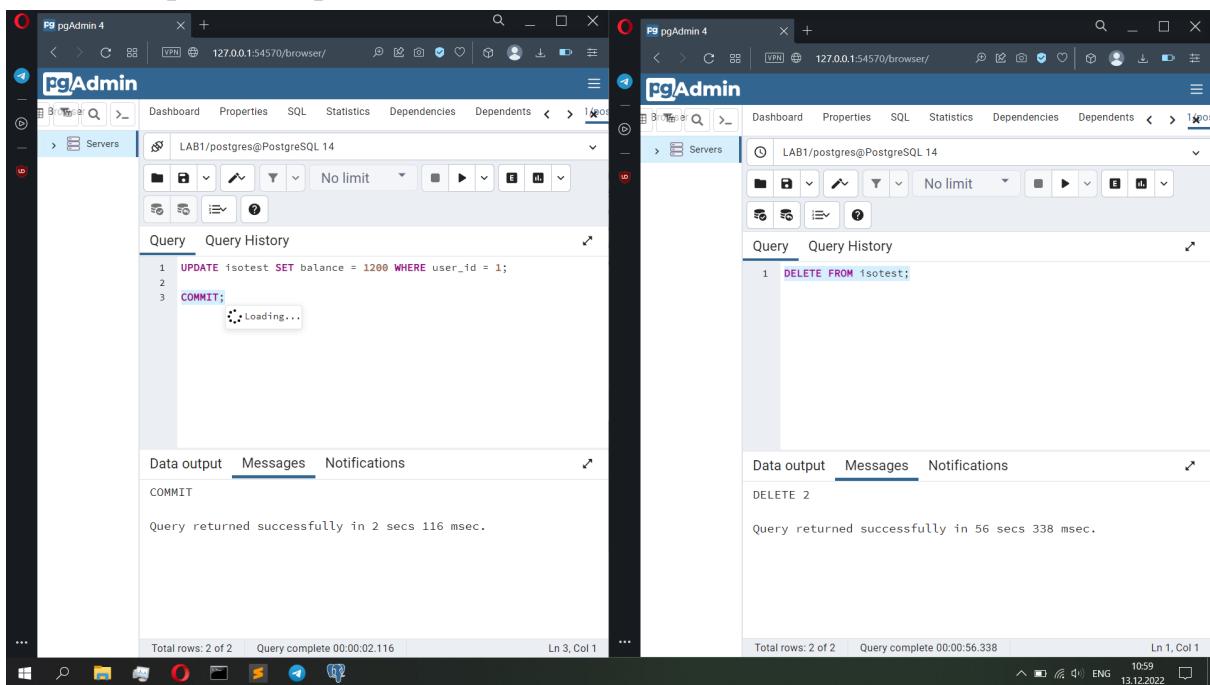
Data output Messages Notifications

user_id [PK] integer ↗ balance integer ↗

user_id	balance
1	3
2	1100

Waiting for the query to complete... Total rows: 2 of 2 Waiting for the query to complete... 00:00:32.029 Ln 1, Col 1

Завершення транзакції



The image shows two side-by-side pgAdmin 4 interface windows. Both windows have the title bar "pgAdmin 4" and the URL "127.0.0.1:54570/browser/". The left window is connected to the server "LAB1/postgres@PostgreSQL 14" and displays the following SQL session:

```
1 UPDATE isotest SET balance = 1200 WHERE user_id = 1;
2
3 COMMIT;
```

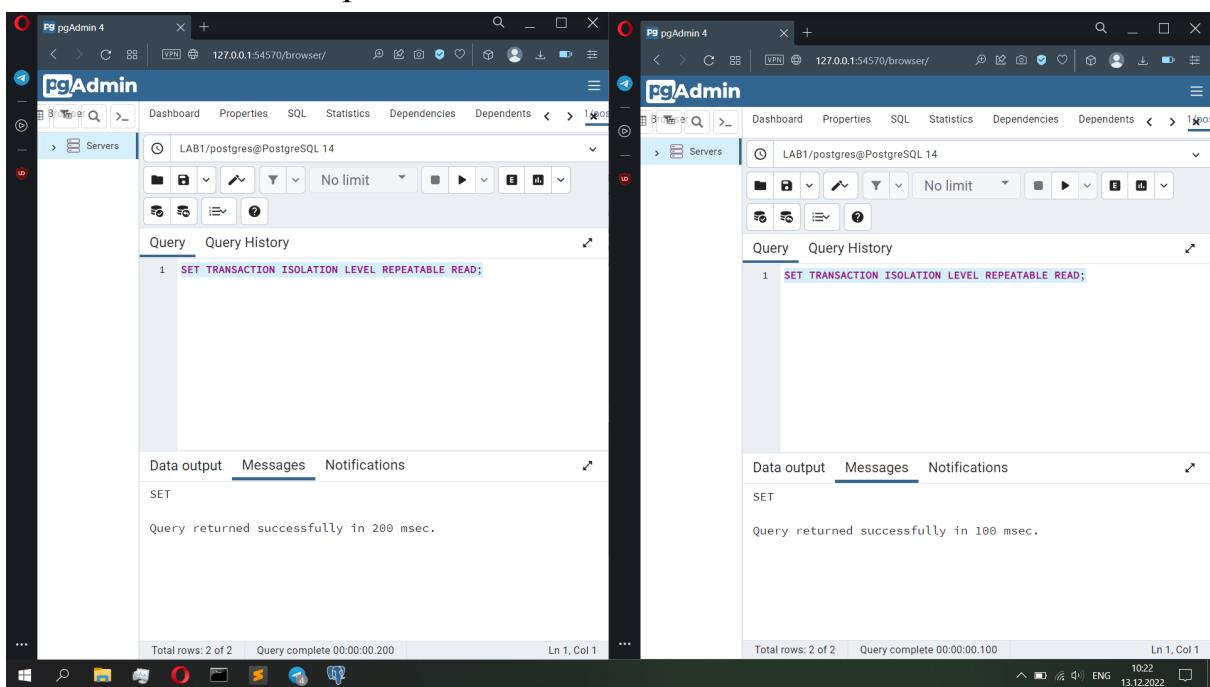
The right window is also connected to the same server and displays the following SQL session:

```
1 DELETE FROM isotest;
```

Отже, рівень Read Committed забезпечує захист лише від брудного зчитування, за рахунок закриття доступу до бачення змін в не закритих транзакціях.

Repeatable read

Встановлюємо рівень ізоляції



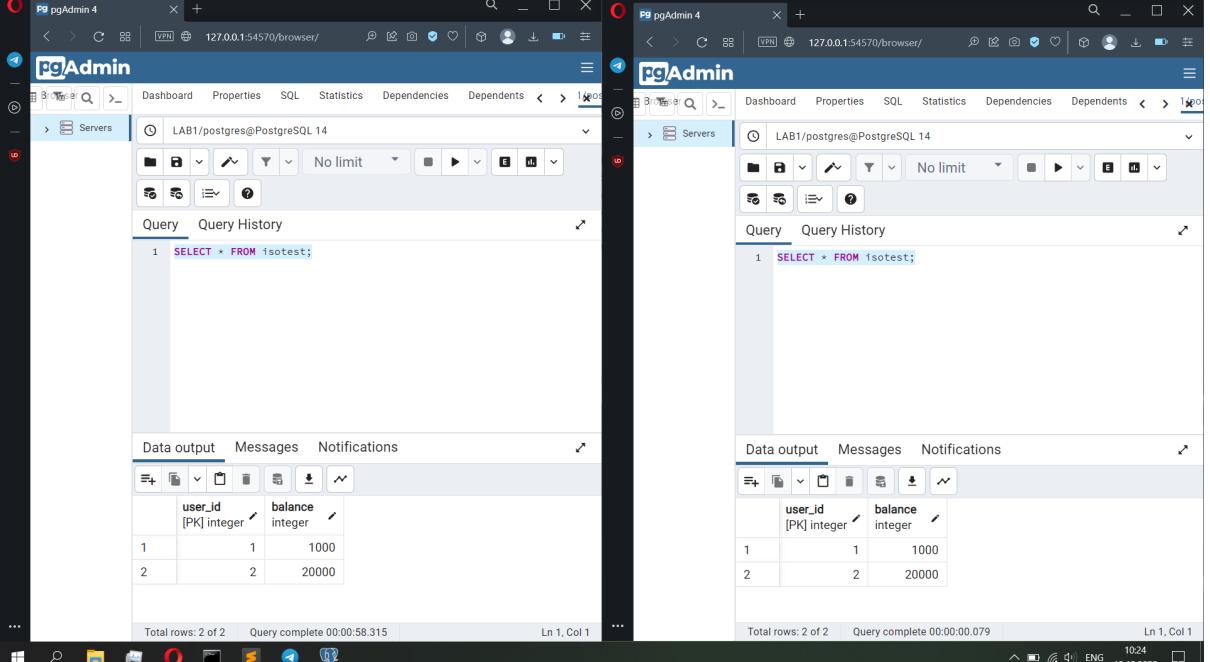
The image shows two side-by-side pgAdmin 4 interface windows. Both windows have the title bar "pgAdmin 4" and the URL "127.0.0.1:54570/browser/". The left window is connected to the server "LAB1/postgres@PostgreSQL 14" and displays the following SQL session:

```
1 SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
```

The right window is also connected to the same server and displays the following SQL session:

```
1 SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
```

У початковий момент часу всім транзакціям доступна початкова версія даних

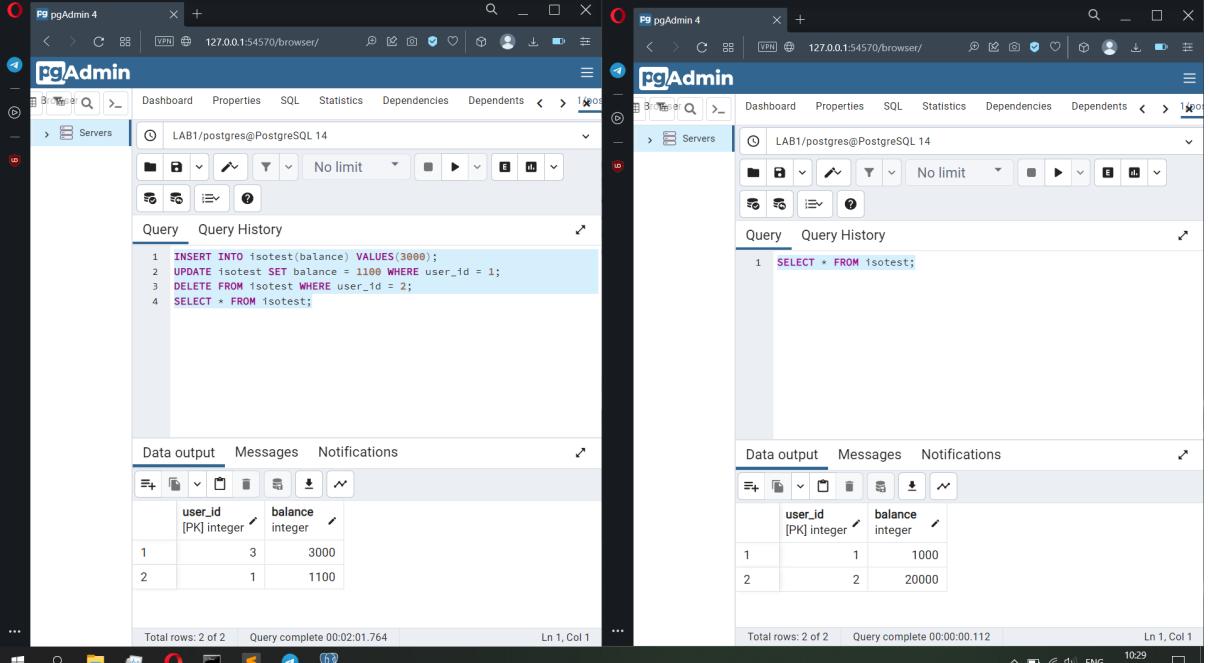


The screenshot shows two pgAdmin windows side-by-side. Both windows have the title 'pgAdmin 4' and are connected to the server 'LAB1/postgres@PostgreSQL 14'. The left window's query editor contains the SQL command 'SELECT * FROM isatest;'. The right window's query editor also contains the same SQL command. Both windows display the same data output table:

	user_id	balance
1	1	1000
2	2	20000

Both windows show 'Total rows: 2 of 2' and 'Query complete 00:00:58.315' or '00:00:00.079' at the bottom.

Зміни виконані у першій транзакції до її закриття будуть доступні лише всередині неї (заборонений Dirty Read)



The screenshot shows two pgAdmin windows side-by-side. The left window's query editor contains the following four SQL statements:

```
1 INSERT INTO isatest(balance) VALUES(3000);
2 UPDATE isatest SET balance = 1100 WHERE user_id = 1;
3 DELETE FROM isatest WHERE user_id = 2;
4 SELECT * FROM isatest;
```

The right window's query editor contains the SQL command 'SELECT * FROM isatest;'. Both windows display the same data output table:

	user_id	balance
1	3	3000
2	1	1100

Both windows show 'Total rows: 2 of 2' and 'Query complete' times at the bottom. The transaction on the left has completed, while the transaction on the right is still active.

Паралельна робота з даними блокується

The screenshot shows two instances of pgAdmin 4 running side-by-side. Both instances are connected to the same PostgreSQL 14 database, specifically the 'LAB1/postgres' schema.

Left Instance:

- Query window:

```
1 INSERT INTO isotest(balance) VALUES(3000);
2 UPDATE isotest SET balance = 1100 WHERE user_id = 1;
3 DELETE FROM isotest WHERE user_id = 2;
4 SELECT * FROM isotest;
```
- Data output window:

user_id	balance
1	3000
2	1100

Right Instance:

- Query window:

```
1 UPDATE isotest SET balance = 1200 WHERE user_id = 1;
```
- Data output window:

user_id	balance
1	1100
2	2000

In the right instance's data output window, there is a message: "Waiting for the query to complete..." indicating a deadlock or conflict between the two transactions.

Після закриття першої транзакції, зміни зберігаються але не доступні з інших транзакцій (заборонені Nonrepeatable Read і Phantom Read)

The screenshot shows two instances of pgAdmin 4 running side-by-side. Both instances are connected to the same PostgreSQL 14 database, specifically the 'LAB1/postgres' schema.

Left Instance:

- Query window:

```
1 INSERT INTO isotest(balance) VALUES(3000);
2 UPDATE isotest SET balance = 1100 WHERE user_id = 1;
3 DELETE FROM isotest WHERE user_id = 2;
4 SELECT * FROM isotest;
5
6 COMMIT;
```
- Data output window:

COMMIT

Query returned successfully in 121 msec.

Right Instance:

- Query window:

```
1 UPDATE isotest SET balance = 1200 WHERE user_id = 1;
```
- Data output window:

UPDATE 1

Query returned successfully in 1 min 2 secs.

This demonstrates that after committing the first transaction, the changes are visible to other transactions, but they are not repeatable (Nonrepeatable Read).

```

1 INSERT INTO isotest(balance) VALUES(3000);
2 UPDATE isotest SET balance = 1100 WHERE user_id = 1;
3 DELETE FROM isotest WHERE user_id = 2;
4 SELECT * FROM isotest;
5
6 COMMIT;

1 UPDATE isotest SET balance = 1200 WHERE user_id = 1;
2
3 SELECT * FROM isotest;

Data output Messages Notifications
user_id [PK] integer ↴
balance integer ↴
1 3 3000
2 1 1100
Total rows: 2 of 2 Query complete 00:00:32.022 Ln 3, Col 1

```

Отже, рівень Repeatable Read забезпечує захист від брудного зчитування, неповторюваного зчитування і фантомного зчитування. Взагалі, в класичному представленні цього рівня читання фантомів має бути дозволеним, але в PostgreSQL це заборонено.

Serializable

Встановлюємо рівень ізоляції

```

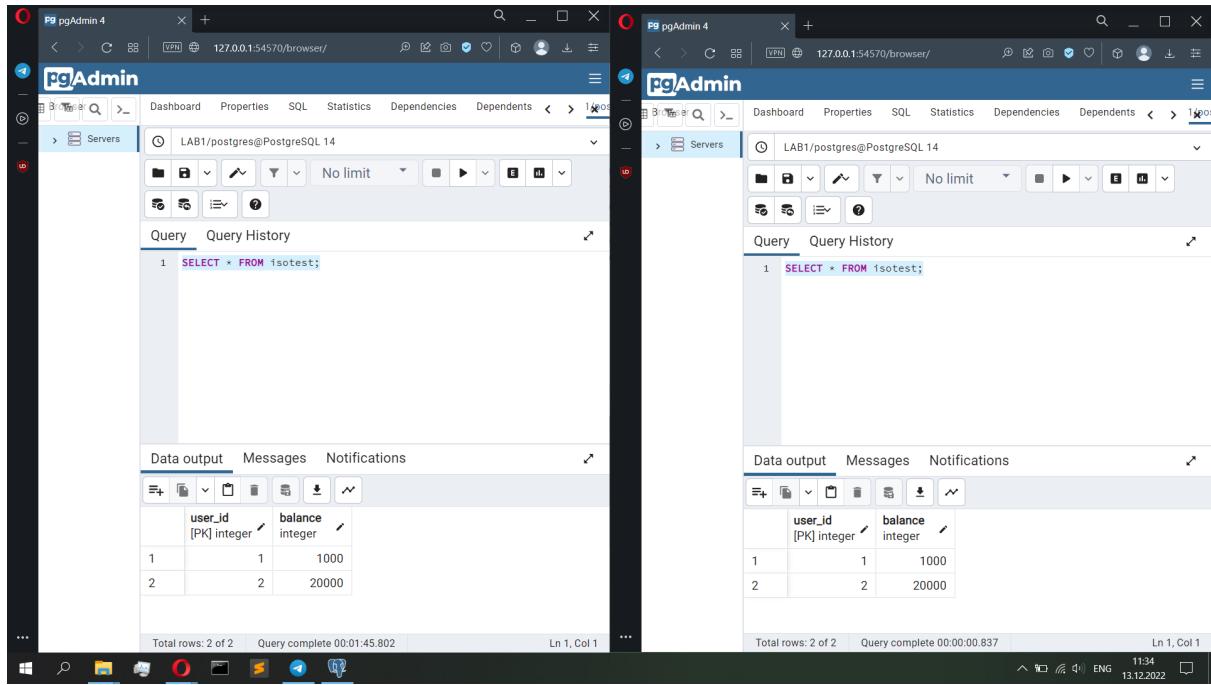
1 SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

1 SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

Data output Messages Notifications
SET
SET
Total rows: 2 of 2 Query complete 00:00:00.110 Ln 1, Col 1
Total rows: 2 of 2 Query complete 00:00:00.130 Ln 1, Col 1

```

У початковий момент часу всім транзакціям доступна початкова версія даних



The screenshot shows two instances of pgAdmin 4 running side-by-side. Both instances are connected to the same PostgreSQL 14 database, specifically the 'LAB1/postgres' server. In the left instance, a query is run:

```
1 SELECT * FROM isotest;
```

The results show the following data:

user_id	balance
1	1000
2	20000

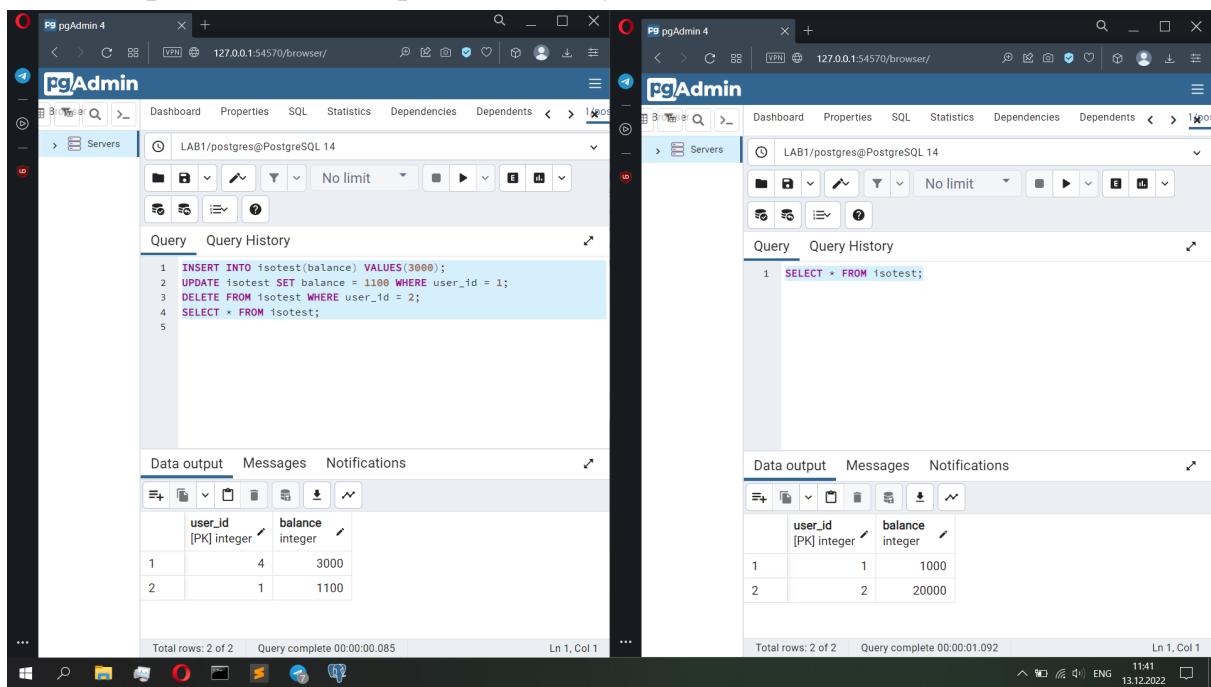
In the right instance, another query is run:

```
1 SELECT * FROM isotest;
```

The results are identical:

user_id	balance
1	1000
2	20000

Зміни виконані у першій транзакції до її закриття будуть доступні лише всередині неї (заборонений Dirty Read)



The screenshot shows two instances of pgAdmin 4. The left instance contains the following SQL code in its query editor:

```
1 INSERT INTO isotest(balance) VALUES(3000);
2 UPDATE isotest SET balance = 1100 WHERE user_id = 1;
3 DELETE FROM isotest WHERE user_id = 2;
4 SELECT * FROM isotest;
```

The right instance shows the results of the final SELECT statement from the left session:

user_id	balance
1	1100
2	3000

This demonstrates that the changes made in the first transaction were visible only within that transaction, preventing dirty reads.

Паралельна робота з даними блокується

The screenshot shows two instances of pgAdmin 4 running side-by-side. Both instances are connected to the same PostgreSQL 14 database, specifically the 'LAB1/postgres' server.

Left Instance:

- Query window:

```
1 INSERT INTO isotest(balance) VALUES(3000);
2 UPDATE isotest SET balance = 1100 WHERE user_id = 1;
3 DELETE FROM isotest WHERE user_id = 2;
4 SELECT * FROM isotest;
```
- Data output window:

user_id	balance
1	4
2	1

Right Instance:

- Query window:

```
1 UPDATE isotest SET balance = 1200 WHERE user_id = 1;
```
- Data output window:

user_id	balance
1	1
2	2000

A circular arrow icon is overlaid on the value '1' in the 'balance' column for user_id 1, indicating a self-modifying row.

Після закриття першої транзакції, зміни зберігаються але не доступні з інших транзакцій (заборонені Nonrepeatable Read і Phantom Read). Також заборонені всі зміни даних з інших транзакцій до наступного коміту в ній.

The screenshot shows two instances of pgAdmin 4 running side-by-side. Both instances are connected to the same PostgreSQL 14 database, specifically the 'LAB1/postgres' server.

Left Instance:

- Query window:

```
1 INSERT INTO isotest(balance) VALUES(3000);
2 UPDATE isotest SET balance = 1100 WHERE user_id = 1;
3 DELETE FROM isotest WHERE user_id = 2;
4 SELECT * FROM isotest;
5 COMMIT;
```
- Data output window:

Commit

Query returned successfully in 60 msec.

Right Instance:

- Query window:

```
1 SELECT * FROM isotest;
2
3 UPDATE isotest SET balance = 1200 WHERE user_id = 1;
4
5 COMMIT;
```
- Data output window:

ERROR: ОШИБКА: не удалось сериализовать доступ из-за параллельного изменения

SQL state: 40001

The screenshot displays four pgAdmin 4 windows side-by-side, each showing a different transaction isolation level:

- Serializable (Left Top):** Shows a successful transaction execution. The SQL code is:

```

1 INSERT INTO isotest(balance) VALUES(3000);
2 UPDATE isotest SET balance = 1100 WHERE user_id = 1;
3 DELETE FROM isotest WHERE user_id = 2;
4 SELECT * FROM isotest;
5 COMMIT;

```

- Read Committed (Right Top):** Shows a failed transaction due to a concurrent update. The SQL code is:

```

1 SELECT * FROM isotest;
2 UPDATE isotest SET balance = 1200 WHERE user_id = 1;
3
4
5 COMMIT;

```

Messages tab output:

ERROR: ОШИБКА: текущая транзакция прервана, команды до конца блока транзакции игнорируются

SQL state: 25P02

- Repeatable Read (Left Bottom):** Shows a successful transaction execution. The SQL code is identical to the Serializable window.
- Read Uncommitted (Right Bottom):** Shows a successful transaction execution. The SQL code is:

```

1 COMMIT;
2 SELECT * FROM isotest;
3
4 UPDATE isotest SET balance = 1200 WHERE user_id = 1;

```

Data output tab output:

user_id	balance
1	1100
2	3000

Отже, рівень Serializable забезпечує повний захист даних, транзакції ніяк не можуть повпливати одна на одну.