



Міністерство освіти і науки України
Національний технічний університет України
«Київський політехнічний інститут»

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

Лабораторна робота №2
з дисципліни
«Бази даних та засоби управління»

Тема: «Засоби оптимізації роботи СУБД PostgreSQL»

Виконав студент 3 курсу

ФПМ групи КВ-11

Парієнко Віктор

Київ – 2023

Метою роботи є здобуття практичних навичок використання засобів оптимізації СУБД PostgreSQL

Завдання роботи полягає у наступному:

1. Перетворити модуль “Модель” з шаблону MVC РГР у вигляд об’єктно-реляційної проєкції (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.

Вимоги до пункту завдання №1

Для перетворення функцій, що реалізують запити до об’єктної бази даних, необхідно встановити бібліотеку sqlalchemy, налаштувати програму на роботу з ORM, розробити класи-сутності для об’єктів-сутностей, представлених відповідними таблицями БД та пов’язаних зв’язками 1:М, М:М та 1:1 виконати опис схеми бази даних. Особливу увагу приділити контролю зовнішніх зв’язків між таблицями засобами ORM.

Замінити виклики запитів мовою SQL на відповідні запити засобами SQLAlchemy по роботі з об’єктами. Обов’язковим є реалізація вставки, вилучення та редагування екземплярів класів-сутностей. Розробка запитів на генерацію даних та пошук екземплярів класів-сутностей вітається, але не є обов’язковою.

Інтерфейси функцій (вхідні та вихідні аргументи функцій модуля “Модель”) мають залишитись без змін.

Вимоги до пункту завдання №2

Відповідно до варіанту індексування продемонструвати на прикладах запитів SQL SELECT підвищення швидкодії їх виконання з використанням індексів, а також пояснити чому для деяких випадків індексування використовувати недоцільно. При цьому для наочного представлення слід використати функцію генерування рандомізованих даних з лабораторної роботи №2, створивши необхідну кількість тестових даних. Навести 4-5

прикладів запитів SELECT (із виведенням результуючих даних), що містять фільтрацію, агрегатні функції, групування та сортування (у необхідних комбінаціях).

Вимоги до пункту завдання №3

Створити тригер бази даних PostgreSQL відповідно до варіанта. Тригерна функція має включати обробку запису, що модифікується (вставляється або вилучається), умовні оператори, курсорні цикли та обробку виключних ситуацій. Виконати відлагодження тригера при різних вхідних даних, навівши 2-3 приклади його використання.

Вимоги до пункту завдання №4

Проаналізувати на прикладах використання рівнів ізоляції транзакцій READ COMMITTED, REPEATABLE READ та SERIALIZABLE, продемонструвавши феномени, які виникають, і спосіб їх уникнення завдяки встановленню відповідного рівня ізоляції транзакцій. Для виконання завдання необхідно відкрити дві транзакції у різних вікнах pgAdmin4 і виконати послідовність запитів INSERT, UPDATE або DELETE у обох транзакціях, що доводять наявність або відсутність певних феноменів.

Варіант 19

<i>№ варіанта</i>	<i>Види індексів</i>	<i>Умови для тригера</i>
<i>19</i>	<i>BTree, BRIN</i>	<i>before insert, delete</i>

GitHub репозиторій: <https://github.com/SkaLe3/KPI-data-base-labs/tree/orm>

Контакт в Telegram: <https://t.me/SkaLe9> (@SkaLe9)

Модель “сутність-зв’язок” предметної галузі “Музичний каталог для зберігання даних про виконавців та пісні”

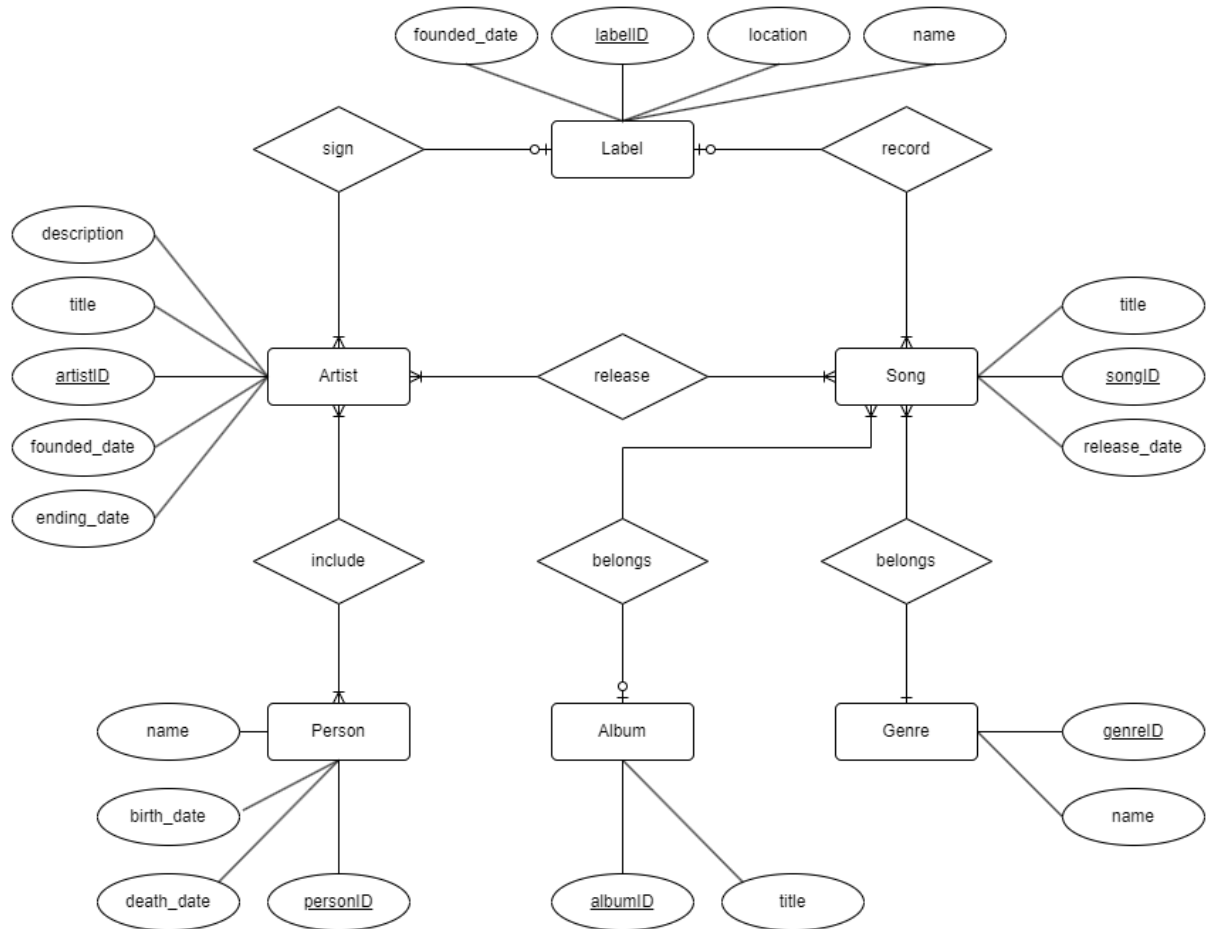


Рис 1. ER-діаграма з використанням нотації “Crow’s foot”

Короткий опис бази даних

Перелік сутностей і їх атрибутів:

бази даних в предметній галузі “Музичний каталог” включає 6 сутностей з наступними атрибутами:

- Artist
 - artist_id, title, description, founded_date, closed_date
- Person
 - person_id, name, birth_date, death_date
- Label
 - label_id, founded_date, location
- Song
 - song_id, title, release_date
- Genre
 - genre_id, name
- Album
 - album_id, title

Опис сутностей:

Artist - сутність для представлення музичного проєкту що може бути групою або однією людиною. Причому одна людина може мати декілька сольних проєктів. Кожен такий проєкт має id, назву, опис, дату створення, і дату закінчення.

Person - сутність для представлення людини. Людина є учасником певного музичного проєкту, або декількох. Атрибути людини це ID, ім'я, дата народження, дата смерті.

Label - сутність для представлення звукозаписуючої компанії, що є брендом а також має право власності на пісні і співпрацює з авторами. Містить id, назву, локацію, дату створення

Song - сутність для представлення пісні яку випускають виконавці, містить id, назву, дату випуску

Genre - сутність для представлення жанру пісні. Складається лише з 2 атрибутів id і назва

Album - сутність для представлення альбому з піснями. Має id і назву.

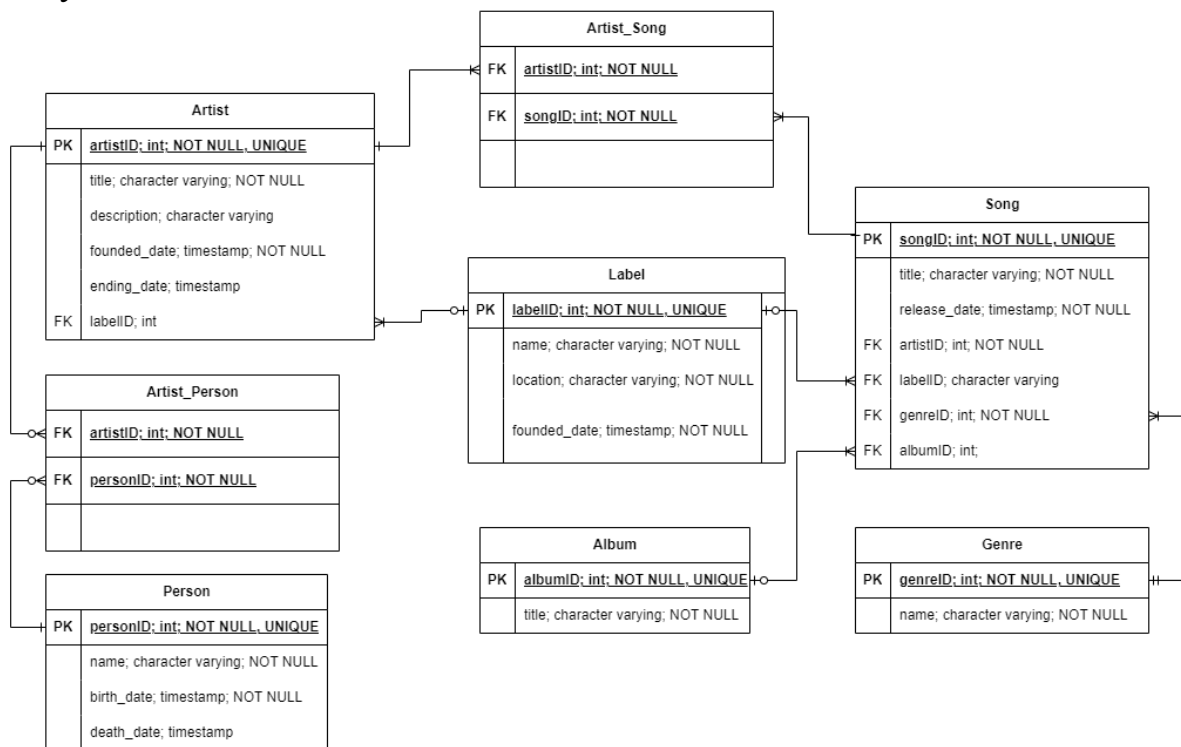


Рис 2. Схема бази даних

Для створення застосунку було використано:

Мова програмування: C++ 20

Середовище розробки: Visual Studio 2022

Бібліотеки:

- GLFW - для створення вікна і прийняття введення
- Glad - для завантаження вказівників на функції OpenGL
- VieM - власна бібліотека, на базі Walnut, для основи програми
- ImGui - для створення графічного інтерфейсу
- libpq - для взаємодії з PostgreSQL
- TinyORM - для використання ORM

Завдання 1

Для використання ORM було створено 6 класів моделей для існуючих 6 таблиць бази даних:

1. Genre
2. Label
3. Album
4. Song
5. Person
6. Artist

Genre

Таблиця genre складається з 2 стовпців genre_id і name, і має зв'язок 1:N з таблицею song. В класі моделі Genre встановлений зв'язок **hasMany<Song>()**

Програмний код реалізації:

```
class Genre final : public Model<Genre, Song>
{
    friend Model;
    using Model::Model;

public:
    std::unique_ptr<HasMany<Genre, Song>>
        songs()
    {
        return hasMany<Song>();
    }

private:
    QString u_table{ "genre" };
    QString u_primaryKey{ "genre_id" };
    bool u_timestamps = false;

private:
    QHash<QString, RelationVisitor> u_relations
    {
        {"songs", [](auto& v) { v(&Genre::songs); }},
    };
};
```

Label

Таблиця label складається з 4 стовпців: label_id, name, location, founded_date, і має два зв'язки, з таблицею artist 1:N і таблицею song 1:N. Для цього в класі моделі Label встановлений зв'язок **hasMany<Artist>()** і **hasMany<Song>()**

Програмний код реалізації:


```

class Label final : public Model<Label, Artist, Song>
{
    friend Model;
    using Model::Model;

public:
    std::unique_ptr<HasMany<Label, Artist>>
        artists()
    {
        return hasMany<Artist>();
    }

    std::unique_ptr<HasMany<Label, Song>>
        songs()
    {
        return hasMany<Song>();
    }

private:
    QString u_table{ "label" };
    QString u_primaryKey{ "label_id" };
    bool u_timestamps = false;

private:
    QHash<QString, RelationVisitor> u_relations
    {
        {
            {"artists", [](auto& v) { v(&Label::artists); }},
            {"songs",   [](auto& v) { v(&Label::songs); }},
        };
    };
};

```

Album

Таблиця Album складається з 2 стовпців: album_id, title, і має 1 зв'язок, з таблицею song 1:N. Для цього в класі моделі Album встановлений зв'язок **hasMany<Song>()**

Програмний код реалізації:

```
class Album final : public Model<Album, Song>
{
    friend Model;
    using Model::Model;

public:
    std::unique_ptr<HasMany<Album, Song>>
        songs()
    {
        return hasMany<Song>();
    }

private:
    QString u_table{ "album" };
    QString u_primaryKey{ "album_id" };
    bool u_timestamps = false;

private:
    QHash<QString, RelationVisitor> u_relations
    {
        { "songs", [](auto& v) { v(&Album::songs); } },
    };
};
```

Song

Таблиця Song складається з 6 стовпців: song_id, title, release_date, label_id, genre_id, album_id і має 3 зв'язки з таблицями album 1:N, genre 1:N, artists N:M. Для цього в класі моделі Song встановлені зв'язки **belongsTo<Album>()**, **belongsTo<Genre>()**, **belongsToMany<Artist>()**

Програмний код реалізації:

```
class Song final : public Model<Song, Album, Genre, Artist, Pivot>
{
    friend Model;
    using Model::Model;

public:
    std::unique_ptr<BelongsTo<Song, Album>>
        album()
    {
        return belongsTo<Album>();
    }

    std::unique_ptr<BelongsTo<Song, Genre>>
        genre()
    {
        return belongsTo<Genre>();
    }

    std::unique_ptr<BelongsToMany<Song, Artist>>
        artists()
    {
        return belongsToMany<Artist>("artist_song");
    }

private:
    QString u_table{ "song" };
    QString u_primaryKey{ "song_id" };
    bool u_timestamps = false;

private:
    QHash<QString, RelationVisitor> u_relations
    {
        {"album", [](auto& v) { v(&Song::album); }},
        {"genre", [](auto& v) { v(&Song::genre); }},
        {"artists", [](auto& v) { v(&Song::artists); }},
    };
};
```

Person

Таблиця Person складається з 4 стовпців: person_id, name, birth_date, death_date і має зв'язок з таблицею artist N:M. Для цього в класі моделі Person встановлений зв'язок **belongsToMany<Artist>("artist_person")**

Програмний код реалізації:

```
class Person final : public Model<Person, Artist, Pivot>
{
    friend Model;
    using Model::Model;

public:
    std::unique_ptr<BelongsToMany<Person, Artist>>
        artists()
    {
        return belongsToMany<Artist>("artist_person");
    }

private:
    QString u_table{ "person" };
    QString u_primaryKey{ "person_id" };
    bool u_timestamps = false;

private:
    QHash<QString, RelationVisitor> u_relations
    {
        {"artists", [](auto& v) { v(&Person::artists); }},
    };
};
```

Artist

Таблиця Artist складається з 6 стовпців: artist_id, title, description, founded_date, closed_date, label_id і має 3 зв'язки з таблицями label 1:N, person N:M, song N:M. Для цього в класі моделі Artist встановлено зв'язки **belongsTo<Label>()**, **belongsToMany<Person>("artist_person", "artist_id", "person_id")**, **belongsToMany<Song>("artist_song", "artist_id", "song_id")**

Програмний код реалізації:

```
class Artist final : public Model<Artist, Label, Person, Song, Pivot>
{
    friend Model;
    using Model::Model;

public:
    std::unique_ptr<BelongsTo<Artist, Label>>
        label()
    {
        return belongsTo<Label>();
    }

    std::unique_ptr<BelongsToMany<Artist, Person>>
        persons()
    {
        return belongsToMany<Person>("artist_person", "artist_id", "person_id");
    }

    std::unique_ptr<BelongsToMany<Artist, Song>>
        songs()
    {
        return belongsToMany<Song>("artist_song", "artist_id", "song_id");
    }

private:
    QString u_table{ "artist" };
    QString u_primaryKey{ "artist_id" };
    bool u_timestamps = false;

private:
    QHash<QString, RelationVisitor> u_relations
    {
        {"label", [](auto& v) { v(&Artist::label); }},
        {"persons", [](auto& v) { v(&Artist::persons); }},
        {"songs", [](auto& v) { v(&Artist::songs); }},
    };
};
```

Реалізація класу моделі шаблону MVC

Для реалізації нової моделі, необхідні функції вибору, створення, оновлення і видалення даних було перетворено на віртуальні в класі моделі, і перевизначено в новому класі ModelORM

```
class ModelORM final : public ModelBase
{
public:
    ModelORM();
    virtual bool Connect(const std::string& dbname, const std::string& username, const std::string& password, std::string& errorMessage) override;
    virtual std::shared_ptr<TableData> FetchTableData(Table table, std::vector<std::string> pkeysTitles, std::string& errorMessage) override;
    virtual bool CreateRecord(Table table, std::vector<std::string> recordData, std::string& errorMessage) override;
    virtual bool UpdateRecord(Table table, std::vector<std::string> recordData, std::vector<Column> pkeyColumns, std::vector<std::string> pkeysData, std::string& errorMessage) override;
    virtual bool DeleteRecord(Table table, std::vector<Column> pkeyColumns, std::vector<std::string> pkeysData, std::string& errorMessage);
};
```

А також створено нові функції для виконання запитів у вигляді ORM

```
private:
    /* CRUD */
    std::shared_ptr<TableData> FetchGenres(std::string& errorMessage);
    std::shared_ptr<TableData> FetchLabels(std::string& errorMessage);
    std::shared_ptr<TableData> FetchAlbums(std::string& errorMessage);
    std::shared_ptr<TableData> FetchSongs(std::string& errorMessage);
    std::shared_ptr<TableData> FetchPersons(std::string& errorMessage);
    std::shared_ptr<TableData> FetchArtists(std::string& errorMessage);
    std::shared_ptr<TableData> FetchArtistPersons(std::string& errorMessage);
    std::shared_ptr<TableData> FetchArtistSongs(std::string& errorMessage);

    void CreateGenreRecord(std::vector<std::string> recordData, std::string& errorMessage);
    void CreateLabelRecord(std::vector<std::string> recordData, std::string& errorMessage);
    void CreateAlbumRecord(std::vector<std::string> recordData, std::string& errorMessage);
    void CreateSongRecord(std::vector<std::string> recordData, std::string& errorMessage);
    void CreatePersonRecord(std::vector<std::string> recordData, std::string& errorMessage);
    void CreateArtistRecord(std::vector<std::string> recordData, std::string& errorMessage);
    void CreateArtistPersonRecord(std::vector<std::string> recordData, std::string& errorMessage);
    void CreateArtistSongRecord(std::vector<std::string> recordData, std::string& errorMessage);

    void UpdateGenreRecord(std::vector<std::string> recordData, std::vector<std::string> pkeysData, std::string& errorMessage);
    void UpdateLabelRecord(std::vector<std::string> recordData, std::vector<std::string> pkeysData, std::string& errorMessage);
    void UpdateAlbumRecord(std::vector<std::string> recordData, std::vector<std::string> pkeysData, std::string& errorMessage);
    void UpdateSongRecord(std::vector<std::string> recordData, std::vector<std::string> pkeysData, std::string& errorMessage);
    void UpdatePersonRecord(std::vector<std::string> recordData, std::vector<std::string> pkeysData, std::string& errorMessage);
    void UpdateArtistRecord(std::vector<std::string> recordData, std::vector<std::string> pkeysData, std::string& errorMessage);
    void UpdateArtistPersonRecord(std::vector<std::string> recordData, std::vector<std::string> pkeysData, std::string& errorMessage);
    void UpdateArtistSongRecord(std::vector<std::string> recordData, std::vector<std::string> pkeysData, std::string& errorMessage);

    void DeleteGenreRecord(std::vector<std::string> pkeysData, std::string& errorMessage);
    void DeleteLabelRecord(std::vector<std::string> pkeysData, std::string& errorMessage);
    void DeleteAlbumRecord(std::vector<std::string> pkeysData, std::string& errorMessage);
    void DeleteSongRecord(std::vector<std::string> pkeysData, std::string& errorMessage);
    void DeletePersonRecord(std::vector<std::string> pkeysData, std::string& errorMessage);
    void DeleteArtistRecord(std::vector<std::string> pkeysData, std::string& errorMessage);
    void DeleteArtistPersonRecord(std::vector<std::string> pkeysData, std::string& errorMessage);
    void DeleteArtistSongRecord(std::vector<std::string> pkeysData, std::string& errorMessage);
};
```

Запити вибору даних

Запити вибору даних з таблиць сутностей реалізовано за допомогою функції `all()`, що повертає колекцію з усіх моделей зазначеного класу, які існують в базі даних.

```
std::shared_ptr<TableData> ModelORM::FetchGenres(std::string& errorMessage)
{
    Orm::Tiny::Types::ModelsCollection<Models::Genre> data1 = Models::Genre::all();
    return CollectionToTableData<Models::Genre>(data1);
}

std::shared_ptr<TableData> ModelORM::FetchLabels(std::string& errorMessage)
{
    Orm::Tiny::Types::ModelsCollection<Models::Label> data1 = Models::Label::all();
    return CollectionToTableData(data1);
}

std::shared_ptr<TableData> ModelORM::FetchAlbums(std::string& errorMessage)
{
    Orm::Tiny::Types::ModelsCollection<Models::Album> data1 = Models::Album::all();
    return CollectionToTableData(data1);
}

std::shared_ptr<TableData> ModelORM::FetchSongs(std::string& errorMessage)
{
    Orm::Tiny::Types::ModelsCollection<Models::Song> data1 = Models::Song::all();
    return CollectionToTableData(data1);
}

std::shared_ptr<TableData> ModelORM::FetchPersons(std::string& errorMessage)
{
    Orm::Tiny::Types::ModelsCollection<Models::Person> data1 = Models::Person::all();
    return CollectionToTableData(data1);
}

std::shared_ptr<TableData> ModelORM::FetchArtists(std::string& errorMessage)
{
    Orm::Tiny::Types::ModelsCollection<Models::Artist> data1 = Models::Artist::all();
    return CollectionToTableData(data1);
}
```

Запити вибору даних з проміжних таблиць зв'язку N:M реалізовано за допомогою функції `getRelation()` щоб отримати репрезентацію таблиці і `getAttribute()`, щоб отримати значення з проміжної таблиці.

```
std::shared_ptr<TableData> ModelORM::FetchArtistPersons(std::string& errorMessage)
{
    std::shared_ptr<TableData> data = std::make_shared<TableData>();

    auto artists1 = Models::Artist::with("persons")->get();
    for (auto& artist : artists1)
    {
        auto artist_persons = artist.getRelation<Models::Person>("persons");
        if (!artist_persons.empty())
        {
            for (auto* person : artist_persons)
            {
                std::vector<std::string> rowData;
                auto artist_id = person->getRelation<Orm::Tiny::Relations::Pivot, Orm::One>("pivot")->getAttribute("artist_id");
                auto person_id = person->getRelation<Orm::Tiny::Relations::Pivot, Orm::One>("pivot")->getAttribute("person_id");
                rowData.push_back(artist_id.toString().toString());
                rowData.push_back(person_id.toString().toString());
                data->push_back(rowData);
            }
        }
    }
    return data;
}

std::shared_ptr<TableData> ModelORM::FetchArtistSongs(std::string& errorMessage)
{
    std::shared_ptr<TableData> data = std::make_shared<TableData>();

    auto artists1 = Models::Artist::with("songs")->get();
    for (auto& artist : artists1)
    {
        auto artist_songs = artist.getRelation<Models::Song>("songs");
        if (!artist_songs.empty())
        {
            for (auto* song : artist_songs)
            {
                std::vector<std::string> rowData;
                auto artist_id = song->getRelation<Orm::Tiny::Relations::Pivot, Orm::One>("pivot")->getAttribute("artist_id");
                auto song_id = song->getRelation<Orm::Tiny::Relations::Pivot, Orm::One>("pivot")->getAttribute("song_id");
                rowData.push_back(artist_id.toString().toString());
                rowData.push_back(song_id.toString().toString());
                data->push_back(rowData);
            }
        }
    }
    return data;
}
```


Запити створення даних

Запити створення даних для таблиць сутностей реалізовано за допомогою створення екземпляру моделі, і встановлення її атрибутів функцією `setAttribute()`, а також збереження методом `save()`

```
void ModelORM::CreateGenreRecord(std::vector<std::string> recordData, std::string& errorMessage)
{
    Models::Genre genre;
    if (!recordData[1].empty()) genre.setAttribute("name", recordData[1].c_str());
    genre.save();
}

void ModelORM::CreateLabelRecord(std::vector<std::string> recordData, std::string& errorMessage)
{
    Models::Label label;
    if (!recordData[1].empty()) label.setAttribute("name", recordData[1].c_str());
    if (!recordData[2].empty()) label.setAttribute("location", recordData[2].c_str());
    if (!recordData[3].empty()) label.setAttribute("founded_date", recordData[3].c_str());
    label.save();
}

void ModelORM::CreateAlbumRecord(std::vector<std::string> recordData, std::string& errorMessage)
{
    Models::Album album;
    if (!recordData[1].empty()) album.setAttribute("title", recordData[1].c_str());
    album.save();
}

void ModelORM::CreateSongRecord(std::vector<std::string> recordData, std::string& errorMessage)
{
    Models::Song song;
    if (!recordData[1].empty()) song.setAttribute("title", recordData[1].c_str());
    if (!recordData[2].empty()) song.setAttribute("release_date", recordData[2].c_str());
    if (!recordData[3].empty()) song.setAttribute("label_id", recordData[3].c_str());
    if (!recordData[4].empty()) song.setAttribute("genre_id", recordData[4].c_str());
    if (!recordData[5].empty()) song.setAttribute("album_id", recordData[5].c_str());
    song.save();
}

void ModelORM::CreatePersonRecord(std::vector<std::string> recordData, std::string& errorMessage)
{
    Models::Person person;
    if (!recordData[1].empty()) person.setAttribute("name", recordData[1].c_str());
    if (!recordData[2].empty()) person.setAttribute("birth_date", recordData[2].c_str());
    if (!recordData[3].empty()) person.setAttribute("death_date", recordData[3].c_str());
    person.save();
}

void ModelORM::CreateArtistRecord(std::vector<std::string> recordData, std::string& errorMessage)
{
    Models::Artist artist;
    if (!recordData[1].empty()) artist.setAttribute("title", recordData[1].c_str());
    if (!recordData[2].empty()) artist.setAttribute("description", recordData[2].c_str());
    if (!recordData[3].empty()) artist.setAttribute("founded_date", recordData[3].c_str());
    if (!recordData[4].empty()) artist.setAttribute("closed_date", recordData[4].c_str());
    if (!recordData[5].empty()) artist.setAttribute("label_id", recordData[5].c_str());
    artist.save();
}
```

Запити створення даних для проміжних таблиць зв'язку N:M реалізовано за допомогою функції `attach(id)`, що створює зв'язок N:M між моделлю що його викликає, і моделлю аргументом

```
void ModelORM::CreateArtistPersonRecord(std::vector<std::string> recordData, std::string& errorMessage)
{
    int32_t artist_id = std::stoi(recordData[0]);
    int32_t person_id = std::stoi(recordData[1]);
    auto artist = Models::Artist::find(artist_id);
    artist->persons()->attach(person_id);
}

void ModelORM::CreateArtistSongRecord(std::vector<std::string> recordData, std::string& errorMessage)
{
    int32_t artist_id = std::stoi(recordData[0]);
    int32_t song_id = std::stoi(recordData[1]);
    auto artist = Models::Artist::find(artist_id);
    artist->songs()->attach(song_id);
}
```

Запити видалення даних

Запити видалення даних для таблиць сутностей реалізовано за допомогою функції `destroy` відповідного класу моделі

```
void ModelORM::DeleteGenreRecord(std::vector<std::string> pkeysData, std::string& errorMessage)
{
    Models::Genre::destroy(std::stoi(pkeysData[0]));
}

void ModelORM::DeleteLabelRecord(std::vector<std::string> pkeysData, std::string& errorMessage)
{
    Models::Label::destroy(std::stoi(pkeysData[0]));
}

void ModelORM::DeleteAlbumRecord(std::vector<std::string> pkeysData, std::string& errorMessage)
{
    Models::Album::destroy(std::stoi(pkeysData[0]));
}

void ModelORM::DeleteSongRecord(std::vector<std::string> pkeysData, std::string& errorMessage)
{
    Models::Song::destroy(std::stoi(pkeysData[0]));
}

void ModelORM::DeletePersonRecord(std::vector<std::string> pkeysData, std::string& errorMessage)
{
    Models::Person::destroy(std::stoi(pkeysData[0]));
}

void ModelORM::DeleteArtistRecord(std::vector<std::string> pkeysData, std::string& errorMessage)
{
    Models::Artist::destroy(std::stoi(pkeysData[0]));
}
```

Запити видалення даних проміжних таблиць зв'язку N:M реалізовано за допомогою функції `detach(id)`, що видаляє зв'язок N:M між моделлю що його викликає, і моделлю аргументом

```
void ModelORM::DeleteArtistPersonRecord(std::vector<std::string> pkeysData, std::string& errorMessage)
{
    int32_t artist_id = std::stoi(pkeysData[0]);
    int32_t person_id = std::stoi(pkeysData[1]);
    auto artist = Models::Artist::find(artist_id);
    artist->songs()->detach(person_id, false);
}

void ModelORM::DeleteArtistSongRecord(std::vector<std::string> pkeysData, std::string& errorMessage)
{
    int32_t artist_id = std::stoi(pkeysData[0]);
    int32_t song_id = std::stoi(pkeysData[1]);
    auto artist = Models::Artist::find(artist_id);
    artist->songs()->detach(song_id, false);
}
```

Запити оновлення даних

Запити оновлення даних таблиць сутностей реалізовано за допомогою встановлення атрибутів функцією `setAttribute` існуючої моделі.

```
void ModelORM::UpdateGenreRecord(std::vector<std::string> recordData, std::vector<std::string> pkeysData, std::string& errorMessage)
{
    auto genre = Models::Genre::find(std::stoi(pkeysData[0]));
    if (!recordData[1].empty()) genre->setAttribute("name", recordData[1].c_str());
    genre->save();
}

void ModelORM::UpdateLabelRecord(std::vector<std::string> recordData, std::vector<std::string> pkeysData, std::string& errorMessage)
{
    auto label = Models::Label::find(std::stoi(pkeysData[0]));
    if (!recordData[1].empty()) label->setAttribute("name", recordData[1].c_str());
    if (!recordData[2].empty()) label->setAttribute("location", recordData[2].c_str());
    if (!recordData[3].empty()) label->setAttribute("founded_date", recordData[3].c_str());
    label->save();
}

void ModelORM::UpdateAlbumRecord(std::vector<std::string> recordData, std::vector<std::string> pkeysData, std::string& errorMessage)
{
    auto album = Models::Album::find(std::stoi(pkeysData[0]));
    if (!recordData[1].empty()) album->setAttribute("title", recordData[1].c_str());
    album->save();
}

void ModelORM::UpdateSongRecord(std::vector<std::string> recordData, std::vector<std::string> pkeysData, std::string& errorMessage)
{
    auto song = Models::Song::find(std::stoi(pkeysData[0]));
    if (!recordData[1].empty()) song->setAttribute("title", recordData[1].c_str());
    if (!recordData[2].empty()) song->setAttribute("release_date", recordData[2].c_str());
    if (!recordData[3].empty()) song->setAttribute("label_id", recordData[3].c_str());
    if (!recordData[4].empty()) song->setAttribute("genre_id", recordData[4].c_str());
    if (!recordData[5].empty()) song->setAttribute("album_id", recordData[5].c_str());
    song->save();
}

void ModelORM::UpdatePersonRecord(std::vector<std::string> recordData, std::vector<std::string> pkeysData, std::string& errorMessage)
{
    auto person = Models::Person::find(std::stoi(pkeysData[0]));
    if (!recordData[1].empty()) person->setAttribute("name", recordData[1].c_str());
    if (!recordData[2].empty()) person->setAttribute("birth_date", recordData[2].c_str());
    if (!recordData[3].empty()) person->setAttribute("death_date", recordData[3].c_str());
    person->save();
}

void ModelORM::UpdateArtistRecord(std::vector<std::string> recordData, std::vector<std::string> pkeysData, std::string& errorMessage)
{
    auto artist = Models::Artist::find(std::stoi(pkeysData[0]));
    if (!recordData[1].empty()) artist->setAttribute("title", recordData[1].c_str());
    if (!recordData[2].empty()) artist->setAttribute("description", recordData[2].c_str());
    if (!recordData[3].empty()) artist->setAttribute("founded_date", recordData[3].c_str());
    if (!recordData[4].empty()) artist->setAttribute("closed_date", recordData[4].c_str());
    if (!recordData[5].empty()) artist->setAttribute("label_id", recordData[5].c_str());
    artist->save();
}
```

Запити оновлення даних проміжних таблиць зв'язку N:M
реалізовано за допомогою видалення існуючого зв'язку, і створення нового
вже з новими даними

```
void ModelORM::UpdateArtistPersonRecord(std::vector<std::string> recordData, std::vector<std::string> pkeysData, std::string& errorMessage)
{
    DeleteArtistPersonRecord(pkeysData, errorMessage);
    CreateArtistPersonRecord(recordData, errorMessage);
}

void ModelORM::UpdateArtistSongRecord(std::vector<std::string> recordData, std::vector<std::string> pkeysData, std::string& errorMessage)
{
    DeleteArtistSongRecord(pkeysData, errorMessage);
    CreateArtistSongRecord(recordData, errorMessage);
}
```

Звдання 2

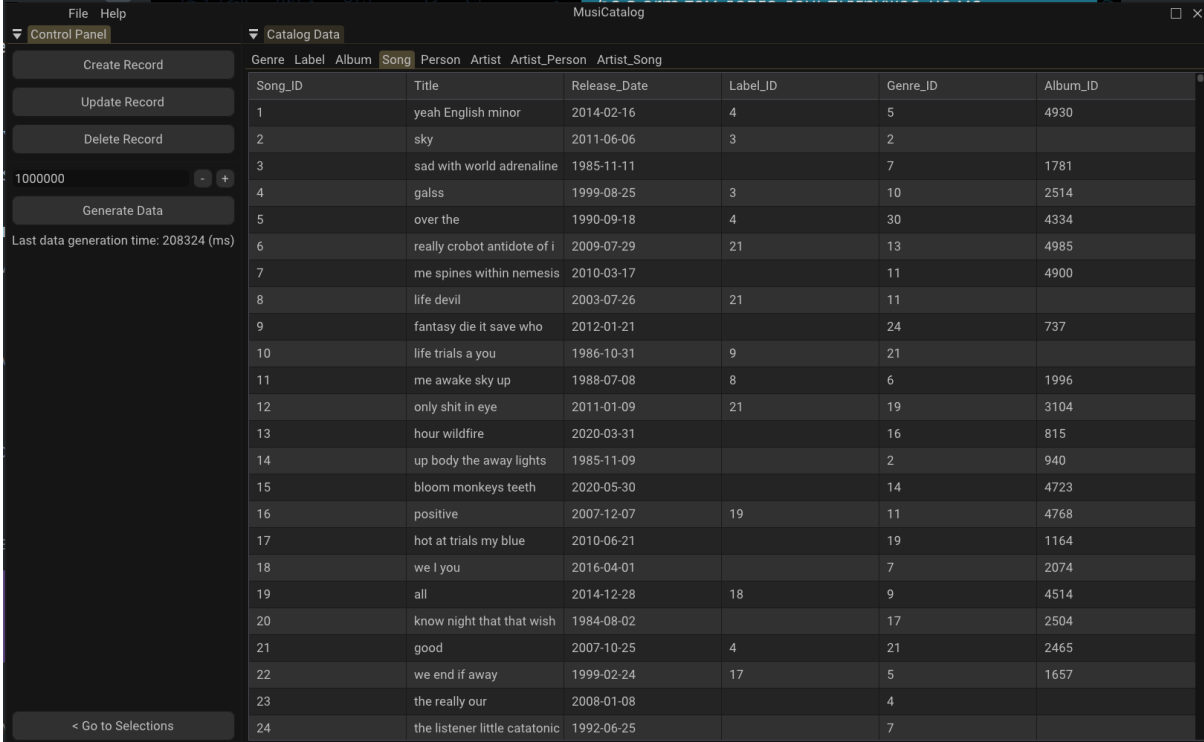
В цій частині роботи досліджено роботу індексів і їх вплив на ефективність виконання запитів.

Індекс - це копія частини таблиці, впорядкована таким чином, що дозволяє PostgreSQL швидко знаходити та отримувати рядки, які відповідають певній умові запиту.

В дослідженні використано 2 типи індексів: BTree і BRIN

Тестові дані:

Для дослідження створено 1 000 000 псевдо-випадкових рядків таблиці song (song_id, title, release_date, label_id, genre_id, album_id) з використанням генерації даних раніше створеною програмою



The screenshot shows the MusiCatalog application interface. On the left is a 'Control Panel' with buttons for 'Create Record', 'Update Record', 'Delete Record', and 'Generate Data'. The 'Generate Data' button is active, and below it, it says 'Last data generation time: 208324 (ms)'. The main area is titled 'Catalog Data' and displays a table with columns: Genre, Label, Album, Song, Person, Artist, Artist_Person, Artist_Song, Song_ID, Title, Release_Date, Label_ID, Genre_ID, and Album_ID. The table contains 24 rows of data.

Genre	Label	Album	Song	Person	Artist	Artist_Person	Artist_Song	Song_ID	Title	Release_Date	Label_ID	Genre_ID	Album_ID
								1	yeah English minor	2014-02-16	4	5	4930
								2	sky	2011-06-06	3	2	
								3	sad with world adrenaline	1985-11-11		7	1781
								4	galss	1999-08-25	3	10	2514
								5	over the	1990-09-18	4	30	4334
								6	really crobot antidote of i	2009-07-29	21	13	4985
								7	me spines within nemesis	2010-03-17		11	4900
								8	life devil	2003-07-26	21	11	
								9	fantasy die it save who	2012-01-21		24	737
								10	life trials a you	1986-10-31	9	21	
								11	me awake sky up	1988-07-08	8	6	1996
								12	only shit in eye	2011-01-09	21	19	3104
								13	hour wildfire	2020-03-31		16	815
								14	up body the away lights	1985-11-09		2	940
								15	bloom monkeys teeth	2020-05-30		14	4723
								16	positive	2007-12-07	19	11	4768
								17	hot at trials my blue	2010-06-21		19	1164
								18	we I you	2016-04-01		7	2074
								19	all	2014-12-28	18	9	4514
								20	know night that that wish	1984-08-02		17	2504
								21	good	2007-10-25	4	21	2465
								22	we end if away	1999-02-24	17	5	1657
								23	the really our	2008-01-08		4	
								24	the listener little catatonic	1992-06-25		7	

BTree

Цей тип індексу організований у вигляді дерева. Індекс починається з кореневого вузла з вказівниками на дочірні вузли. Кожен вузол дерева містить декілька пар ключ-значення, де ключі використовуються для індексування, а значення вказують на відповідні дані в таблиці.

Для тестування швидкодії створено 4 запити:

```
explain analyze SELECT count(*) FROM song WHERE title LIKE 'h%'
explain analyze SELECT count(*) FROM song WHERE title = 'hell nemesis';
explain analyze SELECT * FROM song ORDER BY title;
explain analyze SELECT sum(song_id) FROM song GROUP BY title;
```

Результати виконання запитів:

Оскільки час виконання швидких запитів коливається, для зменшення похибки вимірювання вибрано середній показник 20 виконань запитів, для кожного прикладу поточного вимірювання, і всіх наступних

1. Запит:

1	explain analyze SELECT count(*) FROM song WHERE title LIKE 'h%'
Data Output Messages Notifications	
<div><div><div></div><div></div><div></div><div></div><div></div><div></div><div></div></div></div>	
QUERY PLAN	
1	text
1	Finalize Aggregate (cost=14285.02..14285.03 rows=1 width=8) (actual time=64.121..69.941 rows=1 loops=1)
2	→ Gather (cost=14284.80..14285.01 rows=2 width=8) (actual time=63.949..69.933 rows=3 loops=1)
3	Workers Planned: 2
4	Workers Launched: 2
5	→ Partial Aggregate (cost=13284.80..13284.81 rows=1 width=8) (actual time=38.546..38.547 rows=1 loops=3)
6	→ Parallel Seq Scan on song (cost=0.00..13222.33 rows=24988 width=0) (actual time=0.030..37.651 rows=20587 loop=1)
7	Filter: (title)::text ~~ 'h%':text
8	Rows Removed by Filter: 312747
9	Planning Time: 0.097 ms
10	Execution Time: 69.985 ms

avg = 199.2 ms

2. Запит:

2	explain analyze SELECT count(*) FROM song WHERE title = 'hell nemesis';	
Data Output	Messages	Notifications
<div></div>		
QUERY PLAN		
text		
1	Finalize Aggregate (cost=14222.56..14222.57 rows=1 width=8) (actual time=56.421..61.893 rows=1 loops=1)	
2	-> Gather (cost=14222.34..14222.55 rows=2 width=8) (actual time=56.262..61.885 rows=3 loops=1)	
3	Workers Planned: 2	
4	Workers Launched: 2	
5	-> Partial Aggregate (cost=13222.34..13222.35 rows=1 width=8) (actual time=33.575..33.576 rows=1 loops=3)	
6	-> Parallel Seq Scan on song (cost=0.00..13222.33 rows=4 width=0) (actual time=13.339..33.569 rows=1 loop=3)	
7	Filter: ((title)::text = 'hell nemesis')::text	
8	Rows Removed by Filter: 333332	
9	Planning Time: 0.106 ms	
10	Execution Time: 61.931 ms	








avg = 205.1 ms

3. Запит:

3	explain analyze SELECT * FROM song ORDER BY title;		
Data Output Messages Notifications			
<div></div>			
QUERY PLAN			
text			
1	Gather Merge (cost=63469.50..160698.59 rows=833334 width=36) (actual time=1698.436..2766.008 rows=1000000 loops=1)		
2	Workers Planned: 2		
3	Workers Launched: 2		
4	→ Sort (cost=63469.48..63511.15 rows=416667 width=36) (actual time=1574.750..1968.268 rows=333333 loops=3)		
	Sort Key: title		
	Sort Method: external merge Disk: 1591368		
	Worker 0: Sort Method: external merge Disk: 1421688		
	Worker 1: Sort Method: external merge Disk: 1421688		
	→ Parallel Seq Scan on song (cost=0.00..12180.67 rows=416667 width=36) (actual time=0.007..19.903 rows=333333 loop=1)		
10	Planning Time: 0.079 ms		
11	Execution Time: 2797.807 ms		

avg = 5306 ms

4. Запит:

4	explain analyze SELECT sum(song_id) FROM song GROUP BY title;
Data Output Messages Notifications	
      	
	QUERY PLAN text
1	Gather Merge (cost=63469.50..160698.59 rows=833334 width=36) (actual time=1698.436..2766.008 rows=1000000 loops=1)
2	Workers Planned: 2
3	Workers Launched: 2
4	-> Sort (cost=62469.48..63511.15 rows=416667 width=36) (actual time=1574.750..1968.268 rows=333333 loops=3)
5	Sort Key: title
6	Sort Method: external merge Disk: 15912kB
7	Worker 0: Sort Method: external merge Disk: 14216kB
8	Worker 1: Sort Method: external merge Disk: 14216kB
9	-> Parallel Seq Scan on song (cost=0.00..12180.67 rows=416667 width=36) (actual time=0.007..19.903 rows=333333 loop...
10	Planning Time: 0.079 ms
11	Execution Time: 2797.307 ms








avg = 5081 ms

Створено індекс “song_btree” для таблиці song по стовпцю title:

1	drop index if exists "song_btree";
2	create index "song_btree" on "song" using btree("title")
Data Output Messages Notifications	
CREATE INDEX	
Query returned successfully in 6 secs 565 msec.	







Результати виконання запитів:

1. Запит:

1	explain analyze SELECT count(*) FROM song WHERE title LIKE 'h%';
Data Output Messages Notifications	
      	
	QUERY PLAN text
1	Finalize Aggregate (cost=14285.02..14285.03 rows=1 width=8) (actual time=75.042..81.252 rows=1 loops=1)
2	-> Gather (cost=14284.80..14285.01 rows=2 width=8) (actual time=74.887..81.243 rows=3 loops=1)
3	Workers Planned: 2
4	Workers Launched: 2
5	-> Partial Aggregate (cost=13284.80..13284.81 rows=1 width=8) (actual time=42.139..42.140 rows=1 loops=3)
6	-> Parallel Seq Scan on song (cost=0.00..13222.33 rows=24988 width=0) (actual time=0.022..41.137 rows=20587 loop...
7	Filter: ((title)::text ~~ 'h%')::text
8	Rows Removed by Filter: 312747
9	Planning Time: 0.175 ms
10	Execution Time: 81.309 ms

avg = 185.3

2. Запит:

2	explain analyze SELECT count(*) FROM song WHERE title = 'hell nemesi';
Data Output Messages Notifications	
      	
	QUERY PLAN text
1	Aggregate (cost=4.60..4.61 rows=1 width=8) (actual time=0.657..0.657 rows=1 loops=1)
2	-> Index Only Scan using song_btree on song (cost=0.42..4.58 rows=9 width=0) (actual time=0.648..0.651 rows=4 loop...
3	Index Cond: (title = 'hell nemesi')::text
4	Heap Fetches: 0
5	Planning Time: 0.111 ms
6	Execution Time: 0.683 ms

avg = 92.6

3. Запит:

3	explain analyze SELECT * FROM song ORDER BY title;		
Data Output Messages Notifications			
<div><div><div><div></div><div></div><div></div><div></div><div></div></div><div><div></div><div></div><div></div><div></div><div></div></div><div><div></div><div></div><div></div><div></div><div></div></div><div><div></div><div></div><div></div><div></div><div></div></div><div><div></div><div></div><div></div><div></div><div></div></div></div></div>			
	QUERY PLAN		
	text		
1	Index Scan using song_btree on song (cost=0.42..62255.96 rows=1000000 width=36) (actual time=0.020..616.351 rows=1000000 loop=		
2	Planning Time: 0.099 ms		
3	Execution Time: 641.981 ms		

avg = 2262 ms

4. Запит:

4	explain analyze SELECT sum(song_id) FROM song GROUP BY title;	
Data Output Messages Notifications		
<div><div><div><div></div><div></div><div></div><div></div><div></div><div></div><div></div></div><div><div></div><div></div><div></div><div></div><div></div><div></div><div></div></div></div></div>		
	QUERY PLAN	
	text	
1	GroupAggregate (cost=0.42..68235.55 rows=97959 width=24) (actual time=0.194..921.439 rows=704570 loops=1)	
2	Group Key: title	
3	-> Index Scan using song_btree on song (cost=0.42..62255.96 rows=1000000 width=20) (actual time=0.016..674.029 rows=1000000 loop=1)	
4	Planning Time: 0.097 ms	
5	Execution Time: 943.138 ms	

avg = 1781 ms

Загальні результати

Запит	№1	№2	№3	№4
Без індексу	199.2	205.1	5306	5081
З індексом	185.3	92.6	2262	1781

Також потрібно звернути увагу на показник cost в QUERY PLAN. З використанням індексу значення значно зменшились, що і є причиною зменшення часу виконання.

Як можна побачити з результатів, в більшості випадків за допомогою індексів запити виконуються швидше. А саме в тих випадках, коли в умовах WHERE, ORDER BY, GROUP BY використовується стовпець, по якому створений індекс. Дивлячись на результати запиту з використанням WHERE, перший запит показав мінімальну різницю, в межах похибки, так як в ньому дуже багато рядків відповідали умові, тому індекс не був задіяний.. Натомість в другому запиті лише 4 рядки відповідали умові WHERE, тому використання індексу значно вплинуло на швидкість виконання запиту. Така сама поведінка спостерігається і для інших умов ORDER BY і GROUP BY, коли в них використовується стовпець, по якому створений індекс. У випадках коли доводиться переглядати великий об'єм рядків, від загальної кількості, індекс застосовувати недоцільно, він не дасть ніяких переваг в швидкості, оскільки його сильна сторона це точні співпадиння, сортування, агрегатні функції в зв'язку з WHERE і т.д. Недоцільно використовувати індекс з таблицями малими за розміром, і коли стовпець по якому створений індекс не використовується в умовах, також в складних запитах, запитах що включають велику кількість JOIN,

підзапитів і т.д. ВTree має недоліки перед іншими видами індексів у використанні з стовпцями з довгими значеннями таких як текст.

BRIN

BRIN - Block Range Index. Цей тип індексу призначений для ефективного індексування великих таблиць із відсортованими даними. Він ділить таблицю на логічні блоки та зберігає підсумкову інформацію про кожен блок. Ці блоки містять діапазони значень, а індекс зберігає мінімальні та максимальні значення в блоках. Це робить індекс меншим за розміром, порівняно з іншими типами.

Перед початком виконання дослідження, створено нову таблицю song_copy з сортуванням записів по даті, інакше brin індекс по стовпцю release_date не буде використаний запитом

```
1 CREATE TABLE song_copy AS
2 SELECT *
3 FROM song
4 ORDER BY release_date;
```

Для тестування швидкодії створено 4 запити:

```
explain analyze SELECT * FROM song_copy
WHERE release_date BETWEEN '2013-06-01' AND '2013-06-30';

explain analyze SELECT date_trunc('day', release_date) AS day,
AVG(genre_id) AS avg_genre FROM song_copy GROUP BY day;

explain analyze SELECT * FROM song_copy
ORDER BY release_date desc;

explain analyze SELECT MAX(title) AS max_title
FROM song_copy WHERE release_date BETWEEN '2013-06-01' AND '2013-06-30';
```

Результати виконання запитів:

Таким самим шляхом як і для попереднього індексу проведено вимірювання швидкості виоконання запитів

1. Запит:

1	<code>explain analyze SELECT * FROM song_copy</code>
2	<code>WHERE release_date BETWEEN '2013-06-01' AND '2013-06-30';</code>
Data Output Messages Notifications	
<div><div><div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div></div><div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div></div></div><div>QUERY PLAN text</div><div></div></div>	
1	Gather (cost=1000.00..15468.40 rows=2014 width=36) (actual time=68.671..147.859 rows=2001 loops=1)
2	Workers Planned: 2
3	Workers Launched: 2
4	-> Parallel Seq Scan on song_copy (cost=0.00..14267.00 rows=839 width=36) (actual time=33.633..44.080 rows=667 loop=1)
5	Filter: ((release_date >= '2013-06-01'::date) AND (release_date <= '2013-06-30'::date))
6	Rows Removed by Filter: 332666
7	Planning Time: 0.071 ms
8	Execution Time: 147.949 ms

avg = 122.9 ms

2. Запит:

4	<code>explain analyze SELECT date_trunc('day', release_date) AS day,</code>
5	<code>AVG(genre_id) AS avg_genre FROM song_copy GROUP BY day;</code>
Data Output Messages Notifications	
<div><div><div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div></div><div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div></div></div><div>QUERY PLAN text</div><div></div></div>	
1	Finalize HashAggregate (cost=20627.24..20682.11 rows=14564 width=40) (actual time=252.056..257.293 rows=14600 loops=1)
2	Group Key: (date_trunc('day', release_date)::text, (release_date)::timestamp with time zone)
3	Batches: 1 Memory Usage: 3089kB
4	-> Gather (cost=17350.34..20481.60 rows=29128 width=40) (actual time=241.494..245.343 rows=16541 loops=1)
5	Workers Planned: 2
6	Workers Launched: 2
7	-> Partial HashAggregate (cost=16350.34..16568.80 rows=14564 width=40) (actual time=219.566..221.402 rows=5514 loops=3)
8	Group Key: date_trunc('day', release_date)::text, (release_date)::timestamp with time zone)
9	Batches: 1 Memory Usage: 1425kB
10	Worker 0: Batches: 1 Memory Usage: 1425kB
11	Worker 1: Batches: 1 Memory Usage: 1425kB
12	-> Parallel Seq Scan on song_copy (cost=0.00..14267.00 rows=416667 width=12) (actual time=0.374..152.250 rows=333333 loop=1)
13	Planning Time: 0.150 ms
14	Execution Time: 258.277 ms

avg = 277.15 ms

3. Запит:

7	<code>explain analyze SELECT * FROM song_copy</code>
8	<code>ORDER BY release_date desc;</code>
Data Output Messages Notifications	
<div><div><div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div></div><div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div></div></div><div>QUERY PLAN text</div><div></div></div>	
1	Gather Merge (cost=63472.50..160701.59 rows=833334 width=36) (actual time=187.654..389.233 rows=1000000 loops=1)
2	Workers Planned: 2
3	Workers Launched: 2
4	-> Sort (cost=62472.48..63514.15 rows=416667 width=36) (actual time=126.039..171.930 rows=333333 loops=3)
5	Sort Key: release_date DESC
6	Sort Method: external merge Disk: 21584kB
7	Worker 0: Sort Method: external merge Disk: 17760kB
8	Worker 1: Sort Method: external merge Disk: 5008kB
9	-> Parallel Seq Scan on song_copy (cost=0.00..12183.67 rows=416667 width=36) (actual time=0.098..33.354 rows=333333 loop=1)
10	Planning Time: 0.107 ms
11	Execution Time: 422.944 ms

avg = 410 ms

4. Запит:

10	<code>explain analyze SELECT MAX(title) AS max_title</code>
11	<code>FROM song_copy WHERE release_date BETWEEN '2013-06-01' AND '2013-06-30';</code>
Data Output Messages Notifications	
<div><div><div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div></div><div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div></div></div><div>QUERY PLAN text</div><div></div></div>	
1	Finalize Aggregate (cost=15269.31..15269.32 rows=1 width=32) (actual time=152.297..155.664 rows=1 loops=1)
2	-> Gather (cost=15269.10..15269.31 rows=2 width=32) (actual time=76.093..155.636 rows=3 loops=1)
3	Workers Planned: 2
4	Workers Launched: 2
5	-> Partial Aggregate (cost=14269.10..14269.11 rows=1 width=32) (actual time=39.452..39.453 rows=1 loops=3)
6	-> Parallel Seq Scan on song_copy (cost=0.00..14267.00 rows=839 width=16) (actual time=30.437..39.204 rows=667 loop=1)
7	Filter: ((release_date >= '2013-06-01'::date) AND (release_date <= '2013-06-30'::date))
8	Rows Removed by Filter: 332666
9	Planning Time: 0.125 ms
10	Execution Time: 155.697 ms

avg = 127 ms

Створено індекс “song_brin” для таблиці song по стовпцю title:

1	drop index if exists "song_brin";
2	create index "song_brin" on "song_copy" using brin("release_date")

Data Output	Messages	Notifications
-------------	----------	---------------

NOTICE: index "song_brin" does not exist, skipping
CREATE INDEX

Query returned successfully in 179 msec.

Результати виконання запитів:

1. Запит:

1	explain analyze SELECT * FROM song_copy
2	WHERE release_date BETWEEN '2013-06-01' AND '2013-06-30';

Data Output	Messages	Notifications
-------------	----------	---------------

QUERY PLAN	text
1	Bitmap Heap Scan on song_copy (cost=12.54..8287.41 rows=2014 width=36) (actual time=0.268..1.760 rows=2001 loop=1)
2	Recheck Cond: ((release_date >= '2013-06-01':date) AND (release_date <= '2013-06-30':date))
3	Rows Removed by Index Recheck: 13972
4	Heap Blocks: lossy=128
5	-> Bitmap Index Scan on song_brin (cost=0.00..12.03 rows=15873 width=0) (actual time=0.024..0.024 rows=1280 loop=1)
6	Index Cond: ((release_date >= '2013-06-01':date) AND (release_date <= '2013-06-30':date))
7	Planning Time: 4.057 ms
8	Execution Time: 1.846 ms

avg = 57 ms

2. Запит:

4	explain analyze SELECT date_trunc('day', release_date) AS day,
5	AVG(genre_id) AS avg_genre FROM song_copy GROUP BY day;

Data Output	Messages	Notifications
-------------	----------	---------------

QUERY PLAN	text
1	Finalize HashAggregate (cost=20627.24..20882.11 rows=14564 width=40) (actual time=214.591..219.810 rows=14600 loops=1)
2	Group Key: (date_trunc('day', release_date):timestamp with time zone)
3	Batches: 1 Memory Usage: 3089kB
4	-> Gather (cost=17350.34..20481.60 rows=29128 width=40) (actual time=206.359..209.253 rows=16517 loops=1)
5	Workers Planned: 2
6	Workers Launched: 2
7	-> Partial HashAggregate (cost=16350.34..16568.80 rows=14564 width=40) (actual time=178.885..180.335 rows=5506 loops=3)
8	Group Key: date_trunc('day', release_date):timestamp with time zone
9	Batches: 1 Memory Usage: 1425kB
10	Worker 0: Batches: 1 Memory Usage: 1425kB
11	Worker 1: Batches: 1 Memory Usage: 1425kB
12	-> Parallel Seq Scan on song_copy (cost=0.00..14267.00 rows=416667 width=12) (actual time=0.022..122.038 rows=333333 loop=1)
13	Planning Time: 0.137 ms
14	Execution Time: 220.864 ms

avg = 264.7 ms

3. Запит:

7	explain analyze SELECT * FROM song_copy
8	ORDER BY release_date desc;

Data Output	Messages	Notifications
-------------	----------	---------------

QUERY PLAN	text
1	Gather Merge (cost=63472.50..160701.59 rows=833334 width=36) (actual time=145.823..354.195 rows=1000000 loops=1)
2	Workers Planned: 2
3	Workers Launched: 2
4	-> Sort (cost=62472.48..63514.15 rows=416667 width=36) (actual time=115.928..162.781 rows=333333 loops=3)
5	Sort Key: release_date DESC
6	Sort Method: external merge Disk: 17752kB
7	Worker 0: Sort Method: external merge Disk: 12488kB
8	Worker 1: Sort Method: external merge Disk: 14112kB
9	-> Parallel Seq Scan on song_copy (cost=0.00..12183.67 rows=416667 width=36) (actual time=0.008..20.737 rows=333333 loop=1)
10	Planning Time: 0.949 ms
11	Execution Time: 388.879 ms

avg = 415 ms

4. Запит:

10	explain analyze SELECT MAX(title) AS max_title
11	FROM song_copy WHERE release_date BETWEEN '2013-06-01' AND '2013-06-30';
Data Output Messages Notifications	
<div> <div></div> <div></div> <div></div> <div></div> <div></div> <div></div> </div>	
	QUERY PLAN
	text
1	Aggregate (cost=8292.44..8292.45 rows=1 width=32) (actual time=2.311..2.312 rows=1 loops=1)
2	-> Bitmap Heap Scan on song_copy (cost=12.54..8287.41 rows=2014 width=16) (actual time=0.278..1.748 rows=2001 loops=1)
3	Recheck Cond: ((release_date >= '2013-06-01'::date) AND (release_date <= '2013-06-30'::date))
4	Rows Removed by Index Recheck: 13972
5	Heap Blocks: lossy=128
6	-> Bitmap Index Scan on song_brin (cost=0.00..12.03 rows=15873 width=0) (actual time=0.025..0.025 rows=1280 loops=1)
7	Index Cond: ((release_date >= '2013-06-01'::date) AND (release_date <= '2013-06-30'::date))
8	Planning Time: 0.164 ms
9	Execution Time: 2.341 ms

avg = 47 ms

Загальні результати

Запит	№1	№2	№3	№4
Без індексу	122.9	277	410	127
З індексом	57	264	415	47

brin індекс виявився ефективним лише в 2 запитах, тому лише в них і був застосований. Значення cost в цих запитах значно знизилось.

З результатів можна побачити, що BRIN індекс значно прискорює швидкість виконання запитів, коли використовується в умові WHERE, якщо дані мають відсортовані блоки, а індекс був створений по стовпцю, по якому дані відсортовані. Особливо ефективний для стовпців з даними типу date або timestamp. Найбільш оптимальне використання BRIN індексу для таких даних як лог файли, або стрімінгові дані, де вони неперервно додаються в таблицю в певному порядку, і природньо впорядковані. BRIN індекс доцільно використовувати в запитах які потребують послідовного доступу, таких як сканування по певному проміжку значень, а також для великих об'ємів даних. Цей тип індексу недоцільно використовувати з даними, що мають мало унікальних значень і рандомно впорядковані. Також недоцільне використання з малими таблицями, з складними умовами WHERE, з GROUP BY і агрегатними функціями, і ORDER BY. Для останніх потреб кращим вибором буде BTree індекс.

Завдання 3

Умови для тригера: before insert, before delete

Для реалізації тригерів, спочатку створено таблицю song_logs, куди тригер буде записувати логи, про insert і delete в таблиці song.

```
CREATE TABLE song_logs (  
    id serial PRIMARY KEY,  
    log text,  
    title character varying(64),  
    action_time timestamp  
);
```

Тригер спрацьовує перед операцією вставки і операцією видалення. Тригер записує в таблицю логів повідомлення про успішну вставку або видалення запису, а також назву пісні що була вставлена/видалена і мітку часу коли це було зроблено. У випадку insert тригера виконується перевірка, щоб у вставленого запису дата випуску пісні була не пізніше ніж сьогодні. Якщо дата випуску пізніше, виникає помилка з відповідним повідомленням. Якщо вставлена нова пісня має назву таку саму як в пісні що вже додавалась в логи, то до старої пісні в логах додається примітка '_older'

Нижче представлений запит реалізації тригера:

```
create or replace function song_trigger_func() returns trigger as $$  
declare  
    cursor_log cursor FOR SELECT * from "song_logs";  
    row_ song_logs % rowtype;  
    log_message text;  
begin  
    if (tg_op = 'INSERT') then  
        if (new.release_date > now()::date) then  
            insert into song_logs(log, title, action_time)  
                values('incorrect release date during insertion a record in table song: ', new.title, now()::timestamp);  
            raise exception 'release date cant be in future';  
            return null;  
        else  
            FOR row_ IN cursor_log LOOP  
                UPDATE song_logs SET title = row_.title || '_older' WHERE row_.title = new.title;  
            END LOOP;  
        end if;  
        log_message := 'inserted a record in table song: ' || new.title;  
        insert into song_logs(log, title, action_time) values(log_message, new.title, now()::timestamp);  
        raise notice 'Successfull insertion';  
        return new;  
    elsif (tg_op = 'DELETE') then  
        log_message := 'deleted from table song: ' || old.title;  
        insert into song_logs(log, title, action_time) values(log_message, old.title, now()::timestamp);  
        raise notice 'Successfull deletion';  
        return null;  
    else return null;  
    end if;  
end;  
$$language plpgsql  
  
create trigger song_trigger before insert or delete on public.song  
for each row execute procedure song_trigger_func();
```

Тестування роботи тригера:

Для тестування викнано кілька запитів

1. Запит: створення звичайного запису в таблицю song

```
INSERT INTO song(title, release_date, genre_id) VALUES('newsong1', '2023-01-01', '1')
```

В таблиці song_logs з'явився відповідний запис

	id [PK] integer	log text	title character varying (64)	action_time timestamp without time zone
1	1	inserted a record in table song: newsong1	newsong1	2023-12-21 19:19:57.176388

2. Запит: створення запису з датою більшою за сьогодні

```
INSERT INTO song(title, release_date, genre_id) VALUES('newsong1', '2024-01-01', '1')
```

виведено відповідне повідомлення про помилку

```
ERROR:  release date cant be in future
```

```
CONTEXT:  PL/pgSQL function song_trigger_func() line 11 at RAISE
```

```
SQL state: P0001
```

Створено ще один звичайний запис для збільшення кількості даних для тестування

```
INSERT INTO song(title, release_date, genre_id) VALUES('newsong2', '2023-01-02', '1')
```

	id [PK] integer	log text	title character varying (64)	action_time timestamp without time zone
1	1	inserted a record in table song: newsong1	newsong1	2023-12-21 19:19:57.176388
2	3	inserted a record in table song: newsong2	newsong2	2023-12-21 19:20:46.946275

3. Запит: створення запису із вже існуючою назвою пісні

```
INSERT INTO song(title, release_date, genre_id) VALUES('newsong1', '2023-01-02', '1')
```

В таблиці song_logs з'явився новий запис, а в старому записі з такою ж назвою з'явилась примітка _older

	id [PK] integer	log text	title character varying (64)	action_time timestamp without time zone
1	3	inserted a record in table song: newsong2	newsong2	2023-12-21 19:20:46.946275
2	1	inserted a record in table song: newsong1	newsong1_older	2023-12-21 19:19:57.176388
3	4	inserted a record in table song: newsong1	newsong1	2023-12-21 19:21:22.421796

4. Запит: видалення запису

```
DELETE FROM song WHERE title = 'newsong2';
```

З'явилися відповідні записи в таблиці song_logs. (їх з'явилось 4 а не 1, бо під час налагодження в таблиці song було створено зайві записи з таким полем title)

	id [PK] integer	log text	title character varying (64)	action_time timestamp without time zone
1	3	inserted a record in table song: newsong2	newsong2	2023-12-21 19:20:46.946275
2	1	inserted a record in table song: newsong1	newsong1_older	2023-12-21 19:19:57.176388
3	4	inserted a record in table song: newsong1	newsong1	2023-12-21 19:21:22.421796
4	5	deleted from table song: newsong2	newsong2	2023-12-21 19:24:11.648371
5	6	deleted from table song: newsong2	newsong2	2023-12-21 19:24:11.648371
6	7	deleted from table song: newsong2	newsong2	2023-12-21 19:24:11.648371
7	8	deleted from table song: newsong2	newsong2	2023-12-21 19:24:11.648371

Завдання 4

Вибір рівня ізоляції транзакцій включає компроміс між узгодженістю даних між транзакціями і швидкістю виконання транзакцій

PostgreSQL реалізує 3 рівні ізоляції транзакцій:

1. Read committed
2. Repeatable read
3. Serializable

За замовчуванням у postgres read committed рівень ізоляції

При паралельному виконанні транзакцій можливі виникнення проблем:

1. *Втрачене оновлення*: втрата даних однієї зміни, внаслідок одночасної зміни одного блоку даних різними транзакціями.
2. *Брудне читання*: читання даних, які додані чи змінені транзакцією яка буде відкочена.
3. *Неповторюване читання*: повторно прочитані дані в рамках однієї транзакції, можуть не співпадати з попередньо прочитаними.
4. *Фантомне читання*: ситуація, коли при повторному читанні в рамках однієї транзакції одна і та ж вибірка дає різні множини рядків.

Перед початком тестування створено таблицю transaction_test

```
CREATE TABLE transaction_test(  
    id serial PRIMARY KEY,  
    value int,  
    log text  
)
```

і додано декілька записів

```
INSERT INTO transaction_test (value, log)  
VALUES (3, 'data3'), (7, 'data7'), (13, 'data13');
```

Read Committed

Він гарантує що читання даних бачить лише зафіксовані дані перед початком запиту, і не бачить незафіксованих даних або змін, зафіксованих під час виконання запиту.

Запит SELECT бачить наслідки попередніх оновлень, виконаних у власній транзакції, навіть якщо вони ще не зафіксовані. Два послідовних запити SELECT можуть бачити різні дані, навіть в межах однієї транзакції, якщо інші транзакції фіксують зміни після запуску першого SELECT і до початку другого SELECT.

При цьому рівні ізоляції відсутнє брудне читання, проте наявна проблема неповторюваного читання, коли в процесі роботи однієї транзакції інша може бути успішно виконана, і зроблені нею зміни зафіксовані. Тоді перша транзакція буде працювати з новим набором даних

Рівень ізоляції Read Committed встановлено

SET TRANSACTION ISOLATION LEVEL READ COMMITTED;

Для наглядості послідовності дій, запити з першого вікна будуть відмічені червоним, а з другого синім кольором.

Початковий стан даних:

	id [PK] integer	value integer	log text
1	2	7	data7
2	3	13	data13
3	1	3	data3

Виконання транзакцій:

BEGIN;

SELECT * FROM transaction_test;

	id [PK] integer	value integer	log text
1	2	7	data7
2	3	13	data13
3	1	3	data3

START TRANSACTION;

UPDATE transaction_test SET log = 'NEW DATA' WHERE id = 1;

COMMIT;

SELECT * FROM transaction_test;

COMMIT;

	id [PK] integer	value integer	log text
1	2	7	data7
2	3	13	data13
3	1	3	NEW DATA

Зміни зроблені синьою транзакцією були зафіксовані, перша транзакція працює з ыншим набором даних. Присутня проблема неповторюваного читання і фантомного читання

Repeatable Read

При цьому рівні ізоляції читання одного і того ж рядку чи рядків в транзакції дає однаковий результат. Поки транзакція не закінчена, дані не змінюються іншими транзакціями.

Транзакція бачить лише дані, які були наявні до початку транзакції, і не бачить зміни, внесені під час виконання інших транзакцій. Проте бачить наслідки попередніх оновлень, виконаних у власній транзакції, навіть якщо вони не зафіксовані. Такий рівень ізоляції дає сильнішу гарантію, і запобігає появі феноменів крім аномалій серіалізації.

Початковий стан даних:

	id [PK] integer	value integer	log text
1	2	7	data7
2	3	13	data13
3	1	3	NEW DATA

Виконання транзакцій:

```
BEGIN;
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ READ WRITE;

SELECT * FROM transaction_test;
```

	id [PK] integer	value integer	log text
1	3	13	data13
2	1	3	NEW DATA
3	2	7	data7

```
BEGIN;
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ READ WRITE;

SELECT * FROM transaction_test;
```

	id [PK] integer	value integer	log text
1	3	13	data13
2	1	3	NEW DATA
3	2	7	data7

```
UPDATE transaction_test SET value = 999 WHERE id=2;
```

```
SELECT * FROM transaction_test;
```

	id [PK] integer	value integer	log text
1	3	13	data13
2	1	3	NEW DATA
3	2	999	data7

```
SELECT * FROM transaction_test;
```

	id [PK] integer	value integer	log text
1	3	13	data13
2	1	3	NEW DATA
3	2	7	data7

```
COMMIT;
```

```
SELECT * FROM transaction_test;
```

	id [PK] integer	value integer	log text
1	3	13	data13
2	1	3	NEW DATA
3	2	7	data7

Дані в червоній транзакції не були змінені

```
UPDATE transaction_test SET value = 0 WHERE id=2;
```

ERROR: could not serialize access due to concurrent update

SQL state: 40001

```
SELECT * FROM transaction_test;
```

	id [PK] integer	value integer	log text
1	3	13	data13
2	1	3	NEW DATA
3	2	999	data7

```
ROLLBACK;
```

```
SELECT * FROM transaction_test;
```

	id [PK] integer	value integer	log text
1	3	13	data13
2	1	3	NEW DATA
3	2	999	data7

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 читання.

Serializable

Рівень ізоляції транзакцій Serializable забезпечує найсуворішу ізоляцію. Транзакції повністю ізолюються одна від одної. Цей рівень емулює послідовне виконання транзакцій для всіх здійснених, ніби транзакції виконувались одна за одною, а не одночасно.

```
BEGIN;  
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

```
SELECT * FROM transaction_test;
```

	id [PK] integer	value integer	log text
1	3	13	data13
2	1	3	NEW DATA
3	2	999	data7

```
BEGIN;  
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

```
SELECT * FROM transaction_test;
```

	id [PK] integer	value integer	log text
1	3	13	data13
2	1	3	NEW DATA
3	2	999	data7

```
UPDATE transaction_test SET value = 555 WHERE id=3;
```

	id [PK] integer	value integer	log text
1	1	3	NEW DATA
2	2	999	data7
3	3	555	data13

```
UPDATE transaction_test SET value = 1337 WHERE id=3;
```

Синя транзакція блокується, поки червона транзакція не зафіксує зміни або не відмінить їх.

Data Output				Messages	Notifications
	id	value	log		
	[PK] integer	integer	text		
1	1	3	NEW DATA		
2	2	999	data7		
3	3	13	data13		

Waiting for the query to complete...

COMMIT;

ERROR: could not serialize access due to concurrent update

SQL state: 40001

ROLLBACK;

SELECT * FROM transaction_test;

	id	value	log
	[PK] integer	integer	text
1	1	3	NEW DATA
2	2	999	data7
3	3	555	data13