

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені Ігоря Сікорського»

ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ

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

**Лабораторна робота № 2**

з дисципліни

**«Бази даних та засоби управління»**

**Тема: «Засоби оптимізації роботи СУБД PostgreSQL»**

Виконав:

студент ІII курсу

групи КВ-12

Дмитрієвцев М.В.

Перевірив:

Павловский В. І.

Київ – 2023

**Проектування бази даних та ознайомлення з базовими операціями СУБД PostgreSQL**

*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 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.

Посилання на Github:

*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*

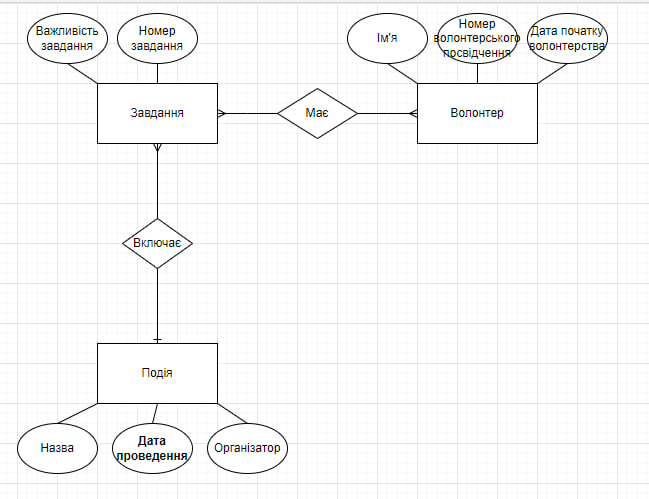
Проаналізувати на прикладах використання рівнів ізоляції транзакцій READ COMMITTED, REPEATABLE READ та SERIALIZABLE, продемонструвавши феномени, які виникають, і спосіб їх уникнення завдяки встановленню відповідного рівня ізоляції транзакцій. Для виконання завдання необхідно відкрити дві транзакції у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

Варіант №8

|  |  |  |
| --- | --- | --- |
| *№ варіанта* | *Види індексів* | *Умови для тригера* |
| *8* | *BTree, GIN* | *after insert, update* |

Посилання на репозиторій y github: <https://github.com/Dmitriievtsev-Mykhail/BD/tree/main/LAB2>

**ER-діаграма, побудована за нотацією «Crow`s foot»**

****

**Сутності з описом призначення**

Предметна галузь «Система управління волонтерами на подіях» включає в себе 3 сутності, кожна сутність містить декілька атрибутів:

1. Event (event\_id, title, organizer, date\_of\_event).

2. Task (task\_id, importance, event\_id).

3. Volunteer (volunteer\_id, name, start\_date\_of\_volunteering).

Сутність Event описує події, які мають бути проведені волонтерами. Кожна подія має свій ідентифікатор, а також містить інформацію про свою назву, дату проведення та організатора.

Сутність Task описує завдання, яке потрібно виконати на певній події. Також кожне завдання містить свій ідентифікатор і певну задачу, яку повинні виконати волонтери.

Сутність Volunteer описує волонтера, який виконує певне завдання під час якоїсь волонтерської події. Кожен волонтер має свій ідентифікатор, а також ім’я та дату початку волонтерства.

**Опис зв’язків між сутностями предметної області**

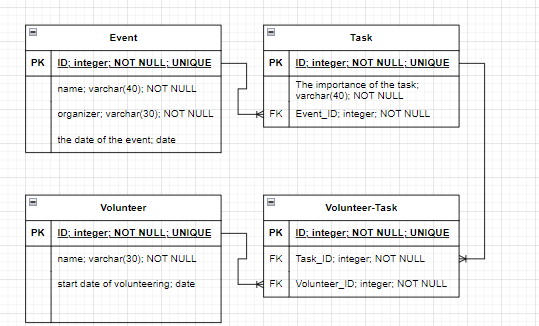
*Зв’язок між Task та Event*:

Кожна подія має завдання, яке потрібно виконати. Зв'язок 1:N – до однієї події може належати багато різних завдань, а певне завдання може відноситися лише до однієї події.

*Зв'язок між Task та Volunteer*:

Кожне завдання має волонтерів, які виконують їх. Зв'язок M:N – кожен волонтер може виконувати декілька завдань на певній події, також одне завдання можуть виконувати кілька волонтерів.

**Схема бази даних у графічному вигляді**



**Завдання №1**

У даній лабораторній роботі було реалізовано 4 класи відповідно до 4 існуючих таблиць у розробленій базі даних, а саме:

1. Event
2. Task
3. Volunteer
4. VolunteerTask

**Event**

Таблиця Events має такі стовпці: event\_id (ідентифікатор заходу), title (назва заходу), organizer (організатор заходу), date\_of\_event (дата проведення заходу). Також наявний зв’язок із таблицею Tasks, тому в цьому класі встановлений зв’язок relationship("Task").

Програмна реалізація класу Event:

**class** Event(Base):

\_\_tablename\_\_ = 'events'

event\_id = Column(Integer, primary\_key=True)

title = Column(String)

organizer = Column(String)

date\_of\_event = Column(Date)

task = relationship("Task")

**def** \_\_init\_\_(self, event\_id, title, organizer, date\_of\_event):

self.event\_id = event\_id

self.title = title

self.organizer = organizer

self.date\_of\_event = date\_of\_event

**def** \_\_repr\_\_(self):

**return** f"<Events(event\_id={self.event\_id}, title={self.title}, organizer={self.organizer}, date\_of\_event={self.date\_of\_event})>"

**Task**

Таблиця Tasks має такі стовпці: task\_id (ідентифікатор завдання), importance (важливість завдання), event\_id (зовнішній ключ, який посилається на захід). Також наявний зв’язок із таблицею Volunteer\_Tasks, тому в цьому класі встановлений зв’язок relationship("VolunteerTask").

Програмна реалізація класу Task:

**class** Task(Base):

\_\_tablename\_\_ = 'tasks'

task\_id = Column(Integer, primary\_key=True)

importance = Column(String)

event\_id = Column(Integer, ForeignKey('events.event\_id'), primary\_key=True)

volunteer\_task = relationship("VolunteerTask")

**def** \_\_init\_\_(self, task\_id, importance, event\_id):

self.task\_id = task\_id

self.importance = importance

self.event\_id = event\_id

**def** \_\_repr\_\_(self):

**return** f"<Tasks(task\_id={self.task\_id}, importance={self.importance}, event\_id={self.event\_id})>"

**Volunteer**

Таблиця Volunteers має такі стовпці: volunteer\_id (ідентифікатор волонтера), name (ім’я волонтера), state\_date\_of\_volunteering (дата початку волонтерства). Також наявний зв’язок із таблицею Volunteer\_Tasks, тому в цьому класі встановлений зв’язок relationship("VolunteerTask").

Програмна реалізація класу Volunteer:

**class** Volunteer(Base):

\_\_tablename\_\_ = 'volunteers'

volunteer\_id = Column(Integer, primary\_key=True)

name = Column(String)

state\_date\_of\_volunteering = Column(Date)

volunteer\_task = relationship("VolunteerTask")

**def** \_\_init\_\_(self, volunteer\_id, name, state\_date\_of\_volunteering):

self.volunteer\_id = volunteer\_id

self.name = name

self.state\_date\_of\_volunteering = state\_date\_of\_volunteering

**def** \_\_repr\_\_(self):

**return** f"<Volunteers(volunteer\_id={self.volunteer\_id}, name={self.name}, state\_date\_of\_volunteering={self.state\_date\_of\_volunteering})>"

**VolunteerTask**

Таблиця Volunteer\_Tasks має такі стовпці: volunteer\_task\_id (ідентифікатор волонтера-завдання), volunteer\_id (зовнішній ключ, який посилається на волонтера), task\_id (зовнішній ключ, який посилається на завдання).

Програмна реалізація класу VolunteerTask:

**class** VolunteerTask(Base):

\_\_tablename\_\_ = 'volunteer\_tasks'

volunteer\_task\_id = Column(Integer, primary\_key=True)

volunteer\_id = Column(Integer, ForeignKey('volunteers.volunteer\_id'), primary\_key=True)

task\_id = Column(Integer, ForeignKey('tasks.task\_id'), primary\_key=True)

**def** \_\_init\_\_(self, volunteer\_task\_id, volunteer\_id, task\_id):

self.volunteer\_task\_id = volunteer\_task\_id

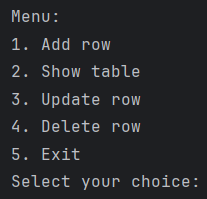
self.volunteer\_id = volunteer\_id

self.task\_id = task\_id

**def** \_\_repr\_\_(self):

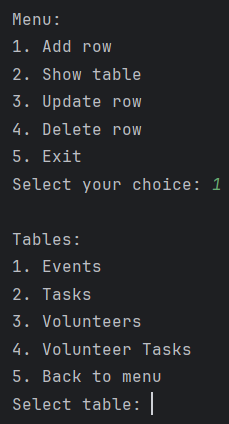
**return** f"<VolunteerTasks(volunteer\_task\_id={self.volunteer\_task\_id}, volunteer\_id={self.volunteer\_id}, task\_id={self.task\_id})>"

Меню складається із 5 пунктів, кожен з яких буде розглянуто далі.



1. **Add row (Додати рядок)**

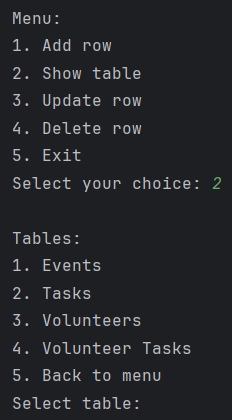
Цей пункт створений для додавання рядка у таблицю. Після його вибору, відкривається список всіх таблиць БД, де потрібно обрати таблицю, до якої хочемо додати рядок:



Після вибору таблиці, користувачу потрібно ввести всі необхідні дані для нового рядка.

1. **Show table (Показ таблиці)**

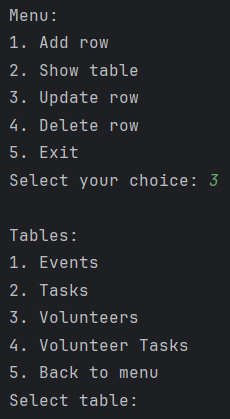
Цей пункт створений для показу таблиць. Після його вибору, відкривається список доступних таблиць БД, де потрібно вибрати таблицю, яку бажаємо побачити.



Після вибору таблиці, мають вивестися всі рядки і стовпці з обраної таблиці БД.

1. **Update row (Редагувати рядок)**

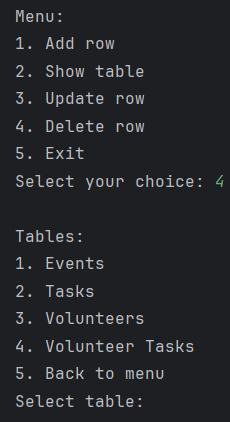
Цей пункт створений для редагування рядків у таблицях. Після вибору цього пункту, відкривається список доступних таблиць, де потрібно вибрати таблицю, в якій бажаємо зробити зміну.



Після вибору таблиці, користувачу потрібно ввести ідентифікатор існуючого рядка в таблиці. Потім записати нові дані для обраного рядка.

1. **Delete row (Видалити рядок)**

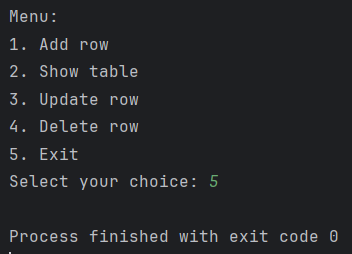
Цей пункт створений для видалення рядків у таблицях. Після вибору цього пункту, відкривається список доступних таблиць БД, де потрібно вибрати таблицю, в якій бажаємо видалити рядок.



Після вибору таблиці, користувачу потрібно ввести ідентифікатор існуючого рядка в таблиці для видалення.

1. **Exit (Вихід)**

Пункт виходу з програми: закривається з’єднання і програма завершується.

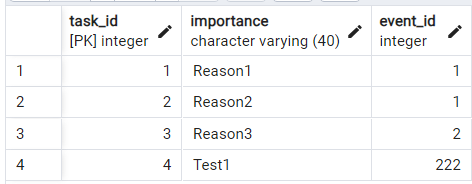


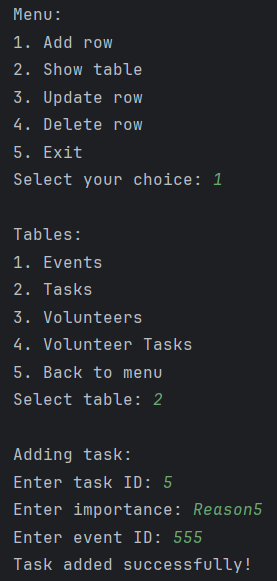
**Приклади запитів у вигляді ORM**

Для демонстрації запитів виберемо по 1-2 таблиці до кожного.

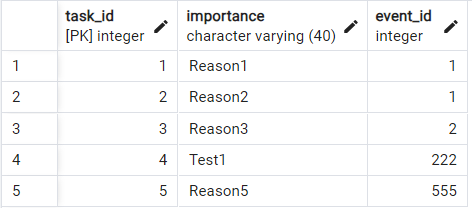
***Запити вставки*** реалізовані за допомогою функцій insert. Спочатку в меню користувач обирає опцію додавання, далі обирає таблицю, до якої хоче додати рядок і вводить необхідні дані.

Таблиця “tasks” до вставки:





Таблиця “tasks” після вставки:



Лістинг функцій insert для кожної таблиці:

**def** insert\_event(self, event\_id: int, title: str, organizer: str, date\_of\_event: Date) -> None:

event = Event(event\_id=event\_id, title=title, organizer=organizer, date\_of\_event=date\_of\_event)

s.add(event)

s.commit()

**def** insert\_task(self, task\_id: int, importance: str, event\_id: int) -> None:

task = Task(task\_id=task\_id, importance=importance, event\_id=event\_id)

s.add(task)

s.commit()

**def** insert\_volunteer(self, volunteer\_id: int, name: str, state\_date\_of\_volunteering: Date) -> None:

volunteer = Volunteer(volunteer\_id=volunteer\_id, name=name, state\_date\_of\_volunteering=state\_date\_of\_volunteering)

s.add(volunteer)

s.commit()

**def** insert\_volunteer\_task(self, volunteer\_task\_id: int, volunteer\_id: int, task\_id: int) -> None:

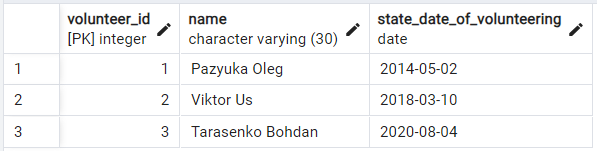
volunteer\_task = VolunteerTask(volunteer\_task\_id=volunteer\_task\_id, volunteer\_id=volunteer\_id, task\_id=task\_id)

s.add(volunteer\_task)

s.commit()

***Запити показу*** реалізовані за допомогою функцій show. Спочатку в меню користувач обирає опцію показу, далі обирає таблицю, яку хоче побачити.

Таблиця “volunteers”:





Лістинг функцій show для кожної таблиці:

**def** show\_events(self):

**return** s.query(Event.event\_id, Event.title, Event.organizer, Event.date\_of\_event).all()

**def** show\_tasks(self):

**return** s.query(Task.task\_id, Task.importance, Task.event\_id).all()

**def** show\_volunteers(self):

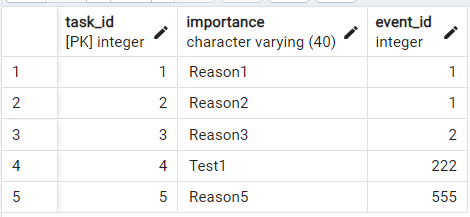
**return** s.query(Volunteer.volunteer\_id, Volunteer.name, Volunteer.state\_date\_of\_volunteering).all()

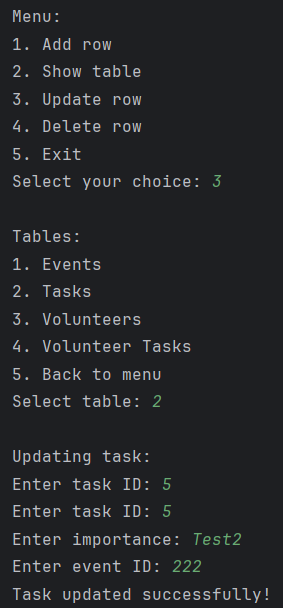
**def** show\_volunteer\_tasks(self):

**return** s.query(VolunteerTask.volunteer\_task\_id, VolunteerTask.volunteer\_id, VolunteerTask.task\_id).all()

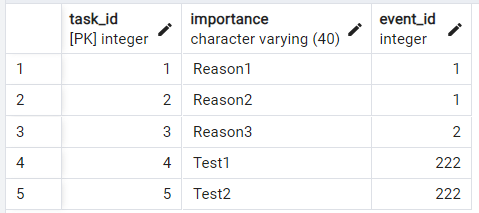
***Запит редагування*** реалізовано за допомогою функції update. Спочатку користувач обирає, у якій таблиці потрібно змінити запис і за яким ідентифікатором. Потім треба ввести всі необхідні дані для реадгування рядка.

Таблиця “tasks” до редагування:





Таблиця “tasks” після редагування:



Лістинг функцій update для кожної таблиці:

@staticmethod

**def** update\_event(event\_id: int, title: str, organizer: str, date\_of\_event: Date, id: int) -> None:

s.query(Event).filter\_by(event\_id=id).update({Event.event\_id: event\_id, Event.title: title, Event.organizer: organizer, Event.date\_of\_event: date\_of\_event})

s.commit()

@staticmethod

**def** update\_task(task\_id: int, importance: str, event\_id: int, id: int) -> None:

s.query(Task).filter\_by(task\_id=id).update({Task.task\_id: task\_id, Task.importance: importance, Task.event\_id: event\_id})

s.commit()

@staticmethod

**def** update\_volunteer(self, volunteer\_id: int, name: str, state\_date\_of\_volunteering: Date, id: int) -> None:

s.query(Volunteer).filter\_by(volunteer\_id=id).update({Volunteer.volunteer\_id: volunteer\_id, Volunteer.name: name, Volunteer.state\_date\_of\_volunteering: state\_date\_of\_volunteering})

s.commit()

@staticmethod

**def** update\_volunteer\_task(self, volunteer\_task\_id: int, volunteer\_id: int, task\_id: int, id: int) -> None:

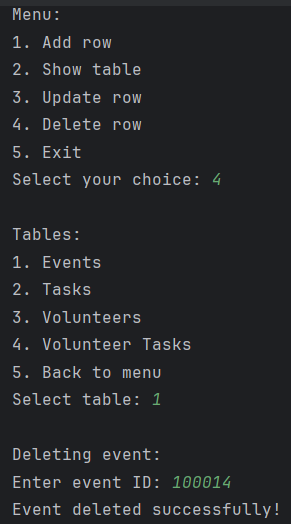
s.query(VolunteerTask).filter\_by(volunteer\_task\_id=id).update({VolunteerTask.volunteer\_task\_id: volunteer\_task\_id, VolunteerTask.volunteer\_id: volunteer\_id, VolunteerTask.task\_id: task\_id})

s.commit()

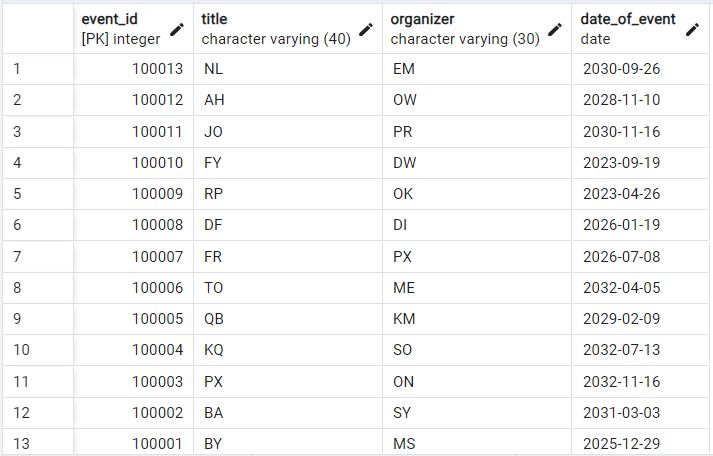
***Запити видалення*** реалізовані за допомогою функцій delete. Спочатку користувач обирає таблицю, з якої потрібно видалити дані. Потім потрібно ввести номер ідентифікатора рядка для видалення.

Таблиця “events” до видалення:





Таблиця “events” після видалення:



Лістинг функцій delete для кожної таблиці:

**def** delete\_event(self, event\_id) -> None:

event = s.query(Event).filter\_by(event\_id=event\_id).one()

s.delete(event)

s.commit()

**def** delete\_task(self, task\_id) -> None:

task = s.query(Task).filter\_by(task\_id=task\_id).one()

s.delete(task)

s.commit()

**def** delete\_volunteer(self, volunteer\_id) -> None:

volunteer = s.query(Volunteer).filter\_by(volunteer\_id=volunteer\_id).one()

s.delete(volunteer)

s.commit()

**def** delete\_volunteer\_task(self, volunteer\_task\_id) -> None:

volunteer\_task = s.query(VolunteerTask).filter\_by(volunteer\_task\_id=volunteer\_task\_id).one()

s.delete(volunteer\_task)

s.commit()

**Завдання №2**

Індекс – це спеціальна структура даних, яка зберігає групу ключових значень та покажчиків. Індекс використовується для управління даними. Для тестування індексів було створено окремі таблиці у базі даних test з 1000000 записами.

***BTree***

Індекс BTree, відоме також як B-дерево, придатне для даних, які можна відсортувати. Іншими словами, для типу даних повинні бути визначені оператори «більше», «більше або дорівнює», «менше», «менше або дорівнює» та «дорівнює». Звернемо увагу, що одні й ті ж дані іноді можна сортувати різними способами, що повертає нас до концепції сімейства операторів.

Як завжди, індексні записи B-дерева упаковані в сторінки. У листкових сторінках ці записи містять індексовані дані (ключі) та посилання на рядки таблиці (TID-и); у внутрішніх сторінках кожен запис посилається на дочірню сторінку індексу та містить мінімальне значення ключа в цій сторінці.

B-дерева мають кілька важливих властивостей:

1. Вони збалансовані, тобто будь-яку листкову сторінку відділяє від кореня однакова кількість внутрішніх сторінок. Таким чином, пошук будь-якого значення займає однаковий час.

2. Вони сильно розгалужені, тобто кожна сторінка (зазвичай, 8 КБ) містить відразу багато (сотні) TID-ів. Завдяки цьому глибина B-дерев стає невеликою; на практиці вона може сягати до 4–5 для дуже великих таблиць.

3. Дані в індексі впорядковані за неспаданням (як між сторінками, так і всередині кожної сторінки), а сторінки одного рівня пов'язані між собою двостороннім списком. Таким чином, отримати впорядкований набір даних ми можемо, просто проходячи по списку в одну або іншу сторону, не повертаючись кожного разу до кореня.

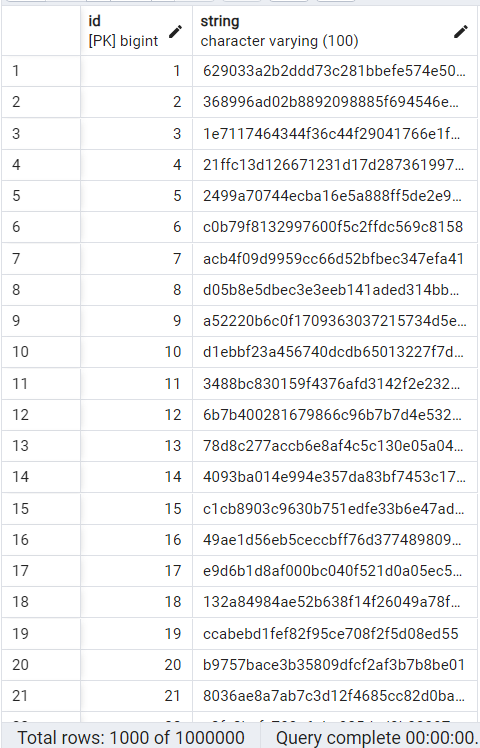
Для дослідження індексу була створена таблиця btree\_test, яка має дві колонки: “id” та “string”:

**CREATE** **TABLE** "btree\_test"("id" bigserial **PRIMARY** **KEY**, "string" **varchar**(100));

**INSERT** **INTO** "btree\_test"("id", "string")

**SELECT** generate\_series **as** "id", md5(**random**()::text)

**FROM** generate\_series(1, 1000000)



Для тестування візьмемо 4 запити:

**SELECT** \* **FROM** btree\_test **WHERE** "string" **LIKE** 'abc%';

**SELECT** **COUNT**(\*), **MAX**("id"), **MIN**("id") **FROM** btree\_test;

**SELECT** "string", **COUNT**(\*) **FROM** btree\_test **GROUP** **BY** "string" **ORDER** **BY** "string" **DESC**;

**SELECT** **LEFT**("string", 1) **AS** first\_letter, **COUNT**(\*), **MAX**("id") **FROM** btree\_test **WHERE** "id" < 500000 **GROUP** **BY** first\_letter **ORDER** **BY** first\_letter;

Створення індексу:

**CREATE** **INDEX** btree\_index **ON** btree\_test (string);

Результати виконання запитів

Без індекса BTree

Запит №1



Запит №2



Запит №3



Запит №4



З індексом BTree

Запит №1



Запит №2



Запит №3



Запит №4



Отже, з отриманих результатів можна побачити, що виконання обраних запитів з використанням індексу BTree не завжди є швидшим за виконання без індексу.

Індекси можуть не завжди призводити до покращення продуктивності запитів, і є кілька причин, чому це може трапитися в конкретних випадках:

1. Якщо таблиця має невеликий обсяг даних або низьку кардинальність для колонки "string", то оптимізатор може вирішити, що сканування всієї таблиці без використання індексу є ефективнішим, оскільки читання з індексу може виявитися дорожчим.
2. Запит 1 фільтрує рядки, які починаються з "abc%". В індексі B-tree пошук за патерном, якщо префікс заданої строки не фіксований, може бути менш ефективним порівняно зі скануванням всієї таблиці для знаходження відповідних записів.
3. Запит 4 використовує функцію LEFT("string", 1), і застосування функцій до колонок може ускладнити використання індексів.
4. Якщо обсяг результатів з використанням індексу виявляється значною частиною таблиці або виявляється дуже великим, то оптимізатор може вирішити, що простіше прочитати всю таблицю.

***GIN***

GIN розшифровується як Generalized Inverted Index — це так званий зворотний індекс. Він працює з типами даних, значення яких не є атомарними, а складаються з елементів. При цьому індексуються не самі значення, а окремі елементи; кожен елемент посилається на ті значення, де він зустрічається.

Хороша аналогія для цього методу — алфавітний покажчик в кінці книги, де для кожного терміна наведений список сторінок, де цей термін згадується. Як і покажчик у книзі, індексний метод повинен забезпечувати швидкий пошук проіндексованих елементів. Для цього вони зберігаються у вигляді вже знайомого нам B-дерева (для нього використовується інша, більш проста, реалізація, але у даному випадку це несуттєво). Кожному елементу прив'язаний впорядкований набір посилань на рядки таблиці, що містять значення з цим елементом. Впорядкованість не принципова для вибірки даних (порядок сортування TID-ів не несе в собі особливого сенсу), але важлива з точки зору внутрішньої структури індексу.

Елементи ніколи не видаляються з GIN-індексу. Вважається, що значення, що містять елементи, можуть зникати, з'являтися, змінюватися, але набір елементів, з яких вони складаються, досить статичний. Таке рішення суттєво спрощує алгоритми, що забезпечують паралельну роботу з індексом кількох процесів.

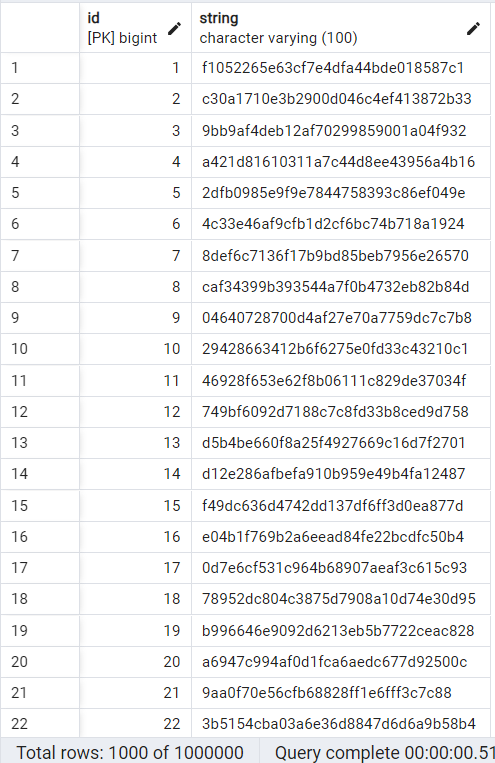
Для дослідження індексу була створена таблиця gin\_test, яка має дві колонки: “id” та “string”:

**CREATE** **TABLE** "gin\_test"("id" bigserial **PRIMARY** **KEY**, "string" **varchar**(100));

**INSERT** **INTO** "gin\_test"("id", "string")

**SELECT** generate\_series **as** "id", md5(**random**()::text)

**FROM** generate\_series(1, 1000000)



Для тестування візьмемо 4 запити:

**SELECT** \* **FROM** gin\_test **WHERE** "string" **LIKE** 'abc%';

**SELECT** **COUNT**(\*), **MAX**("id"), **MIN**("id") **FROM** gin\_test;

**SELECT** "string", **COUNT**(\*) **FROM** gin\_test **GROUP** **BY** "string" **ORDER** **BY** "string" **DESC**;

**SELECT** **LEFT**("string", 1) **AS** first\_letter, **COUNT**(\*), **MAX**("id") **FROM** gin\_test **WHERE** "id" < 500000 **GROUP** **BY** first\_letter **ORDER** **BY** first\_letter;

Створення індексу:

**CREATE** **INDEX** gin\_index **ON** gin\_test **USING** gin("string" gin\_trgm\_ops);

Результати виконання запитів

Без індекса GIN

Запит №1



Запит №2



Запит №3



Запит №4



З індексом GIN

Запит №1



Запит №2



Запит №3



Запит №4



Отже, з отриманих результатів можна побачити, що виконання обраних запитів з використанням індексу GIN не завжди є швидшим за виконання без індексу, а саме в запитах під номерами 2, 3 і 4. Розглянемо їх:

1. Запит №2 (SELECT COUNT(\*), MAX("id"), MIN("id") FROM gin\_test)

Цей запит обчислює агрегатні функції (COUNT, MAX, MIN) для всіх рядків у таблиці. Тут індекс GIN може не бути корисним для поліпшення продуктивності, оскільки операції агрегації вимагають прочитання всіх рядків у будь-якому випадку.

1. Запит №3 (SELECT "string", COUNT(\*) FROM gin\_test GROUP BY "string" ORDER BY "string" DESC;)

Цей запит також містить операції групування та сортування. Індекс GIN може бути менш ефективним в цьому контексті, оскільки групування та сортування може вимагати доступу до багатьох рядків, а не просто швидкого пошуку конкретних значень.

1. Запит №4 (SELECT LEFT("string", 1) AS first\_letter, COUNT(\*), MAX("id") FROM gin\_test WHERE "id" < 500000 GROUP BY first\_letter ORDER BY first\_letter;)

Цей запит використовує функцію LEFT та оператор GROUP BY. Подібно до попередніх запитів, тут також важливим фактором є те, що операції групування та сортування можуть вимагати доступу до багатьох рядків, що може зробити індекс GIN менш ефективним.