A black and white drawing of a building

Description automatically generated

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені Ігоря Сікорського» ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ

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

**Розрахунково-графічна робота**

з дисципліни **Бази даних і засоби управління**

*на тему: “Створення додатку бази даних, орієнтованого на взаємодію з СУБД PostgreSQL”*

Виконала:

студентка ІІI курсу

групи КВ-34   
Понамарьова Сніжана

Перевірив:

Павловский В. І.

Київ – 2025

**Мета:** є здобуття вмінь програмування прикладних додатків баз даних PostgreSQL.

Завдання:

1. Реалізувати функції перегляду, внесення, редагування та вилучення даних у таблицях бази даних, створених у лабораторній роботі №1, засобами консольного інтерфейсу.Перетворити розроблену модель у схему бази даних (таблиці) PostgreSQL.
2. Передбачити автоматичне пакетне генерування «рандомізованих» даних у базі.
3. Забезпечити реалізацію пошуку за декількома атрибутами з двох та більше сутностей одночасно: для числових атрибутів – у рамках діапазону, для рядкових – як шаблон функції LIKE оператора SELECT SQL, для логічного типу – значення True/False, для дат – у рамках діапазону дат.
4. Програмний код виконати згідно шаблону MVC (модель-подання-контролер).

Деталізоване завдання:

1. Забезпечити можливість уведення/редагування/вилучення даних у таблицях бази даних з можливістю контролю відповідності типів даних атрибутів таблиць (рядків, чисел, дати/часу). Для контролю пропонується два варіанти: контроль при введенні (валідація даних) та перехоплення помилок (try..except) від сервера PostgreSQL при виконанні відповідної команди SQL. Особливу увагу варто звернути на дані таблиць, що мають зв’язок 1:N. При цьому з боку батьківської таблиці необхідно контролювати вилученнярядків за умови наявності даних у підлеглій таблиці. З точки зору підлеглої таблиці варто контролювати наявність відповідного рядка у батьківській таблиці при виконанні внесеннянових даних. Унеможливити виведення програмою системних помилок на екрані шляхом їх перехоплення і адекватної обробки. Внесення даних виконується користувачем у консольному вікні програми.
2. Забезпечити можливість автоматичної генерації великої кількості даних у таблицях за допомогою вбудованих у PostgreSQL функцій роботи з псевдовипадковими числами. Дані мають бути згенерованими не мовою програмування, а відповідним SQL-запитом! Кількість даних для генерування має вводити користувач з клавіатури. Для тесту взяти 100 000 записів для однієї-двох таблиць. Особливу увагу слід звернути на відповідність даних вимогам зовнішніх ключів з метою уникнення помилок порушення обмежень цілісності (foreign key).
3. Для реалізації пошуку необхідно підготувати 3 запити, що включають дані з декількох таблиць і фільтрують (WHERE) та групують (GROUP BY) рядки за 3-4 атрибутами цих таблиць. Забезпечити можливість уведення конкретних значень констант для фільтрації з клавіатури користувачем. Крім того, після виведення даних необхідно вивести час виконання запиту у мілісекундах. Перевірити швидкодію роботи запитів на попередньо згенерованих даних.
4. Програмний код організувати згідно шаблону Model-View-Controller(MVC). При цьому модель, подання та контролер мають бути реалізовані у окремих файлах. Для доступу до бази даних використовувати лише мову SQL (без ORM).

Сутності предметної області

1. Учень (Student)

Атрибути: ім’я, прізвище, дата народження, клас, електронна пошта.

**Призначення:** зберігає основну інформацію про учня — персональні дані та належність до класу.

1. Батьки (Parents)

Атрибути: ім’я, прізвище, мобільний телефон, електронна пошта.

**Призначення:** зберігає контактні дані батьків і як доступ до інформації про успішність та відвідуваність їхніх дітей.

1. Вчитель (Teacher)

Атрибути: ім’я, прізвище, електронна пошта.

**Призначення:** зберігає дані викладача.

1. Предмет (Subject)

Атрибути: назва.

**Призначення:** зберігає дані предмета.

1. Журнал (Journal)

Атрибути: дата запису, оцінка, статус відвідування.

**Призначення:** зберігає дані про навчальні події: оцінки та відвідуваність і зв’язують їх із конкретними учнями.

Опис зв’язків між сутностями

**Зв’язок-сутність: «Батьки-Учень». Зв’язок 1:N.** Батьки можуть бути відповідальним за кількох учнів (наприклад, мати/тато декількох дітей), а один учень може мати тільки одних батьків.

Зв’язок-сутність: «Учень-Журнал». Зв’язок 1:N. Для кожного учня є багато записів у журналі (різні дати, предмети, оцінки, відвідування). Кожен запис журналу прив’язаний до одного учня.

Зв’язок-сутність: «Вчитель-Журнал». Зв’язок 1:N. Кожен запис прив’язаний до одного вчителя, але вчитель має багато записів.

Зв’язок-сутність: «Предмет-Журнал». Зв’язок 1:N. Один предмет може мати багато записів у журналі (різні дати, різні учні, різні вчителі), а кожен запис журналу відноситься до одного предмета.

У зв’язку «Журнал» реалізовано 3-хмістне відношення: Вчитель–Учень–Предмет. Тобто журнал не просто зв’язує учня з вчителем, а також фіксує, з яким предметом пов’язана ця взаємодія.

Концептуальна модель даних

Изображение выглядит как зарисовка, рисунок, диаграмма, Штриховая графика

Содержимое, созданное искусственным интеллектом, может быть неверным.

*Рисунок 1 – Концептуальна модель даних*

Логічна модель даних

Графічне подання логічної моделі зображено на рисунку 2.

Изображение выглядит как текст, снимок экрана, Шрифт, число

Содержимое, созданное искусственным интеллектом, может быть неверным.

*Рисунок 2 – Логічна модель бази даних*

Середовище розробки та налаштування підключення до бази даних

Програмне забезпечення розроблено мовою Python. Розробка виконувалася у середовищі PyCharm Community Edition.

Для підключення до СУБД PostgreSQL застосовано бібліотеку psycopg3, яка надає API для виконання SQL-запитів із програми.

Шаблон проєктування програмного додатку

У проєкті застосовано шаблон **MVC** (Model – View – Controller).

**Model** — відповідає за опис структури й логіки зберігання даних. У програмі всі класи моделі розміщені в папці Models і відображають бізнес-об’єкти та правила роботи з даними.

**View** — відповідає за консольний інтерфейс, через який відбувається взаємодія з користувачем. Компоненти представлення готують і виводять необхідну інформацію у вигляді, зручному для користувача (текстові меню, таблиці, повідомлення).

**Controller** — клас, що забезпечує зв’язок між користувачем і системою зберігання даних. Контролер приймає введені дані, передає їх моделям для обробки та, залежно від результату, формує відповідь або оновлює відображення (наприклад, показує повідомлення про успіх/помилку чи перелік записів).

Структура програми та її опис



Изображение выглядит как текст, снимок экрана, Шрифт, программное обеспечение

Содержимое, созданное искусственным интеллектом, может быть неверным.

*Рисунок 3 – Структура програми*

Програма умовно поділена на три модулі, що відповідають шаблону MVC:

1. controller.py
2. model.py
3. view.py

Як видно з назв файлів, кожен модуль виконує свою роль у відповідності до патерну MVC.

* У файлі model.py реалізовано класи та функції, які відповідають за роботу з даними: встановлення з’єднання з базою даних та виконання низькорівневих SQL-запитів.
* У файлі controller.py описано логіку взаємодії з користувачем — обробку команд, виклик операцій над моделлю та управління потоком виконання програми.
* У файлі view.py знаходяться компоненти, що відповідають за виведення інформації: формування текстових меню та відображення результатів у консольному інтерфейсі.

Таке розділення спрощує підтримку коду: модель ізольована від представлення, а контролер зв’язує обидві частини й координує виконання операцій.

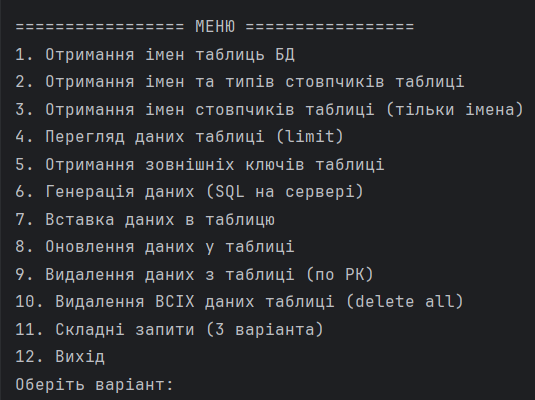
Файл main.py — запуск програми.

Структура меню програми

Меню програми — простий нумерований текстовий інтерфейс для роботи з базою даних: перегляд метаданих, генерація тестових даних і виконання готових складних запитів.

Короткий опис пунктів:

* **Отримання імен таблиць БД** — виводить список таблиць у схемі.
* **Отримання імен та типів стовпчиків** — показує назви полів і їх типи для обраної таблиці.
* **Отримання імен стовпчиків (тільки імена)** — повертає тільки назви стовпців.
* **Перегляд даних (limit)** — показує перші N рядків таблиці.
* **Отримання зовнішніх ключів** — виводить FK-звʼязки для таблиці.
* **Генерація даних** — заповнення таблиці випадковими записами.
* **Вставка даних** — додати один рядок у таблицю.
* **Оновлення даних** — змінити рядок(и) за PK або умовою.
* **Видалення за PK** — видалити конкретний рядок за первинним ключем.
* **Видалення ВСІХ даних** — очистити таблицю.
* **Складні запити (3 варіанта)** — виконати один з трьох наперед підготовлених запитів.
* **Вихід** — завершення роботи та закриття зʼєднання.



*Рисунок 4 – Структура меню програми*

Лістинги та скріншоти результатів виконання операції вилучення запису батьківської таблиці та виведення вмісту дочірньої таблиці після цього вилучення

Опис: наведений код (контролер handle\_delete + модель delete\_by\_pk, preview\_child\_counts, select\_by\_pk) реалізує безпечне видалення рядка батьківської таблиці по первинному ключу. Перед видаленням виконується підрахунок дочірніх записів — якщо такі існують, операція не виконується: модель піднімає ChildRowsExistError, контролер перехоплює помилку і виводить кількість залежних записів та приклади дочірніх рядків (до 10) для наочності. Це запобігає утворенню «висячих» посилань і пояснює користувачу причини неможливості видалення.

controller.py — частина методу handle\_delete (фрагмент)

**try**:

preview = self.model.preview\_child\_counts(table, pk\_col, pkv)

**if** preview.get(table, 0) == 0:

self.view.show\_message("Рядок з таким PK не знайдено.")

**return**

self.view.show\_message("Попередній підрахунок (рядок батька та дочірні записи):")

**for** t, c **in** preview.items():

self.view.show\_message(f" {t}: {c}")

**try**:

row, deleted\_counts = self.model.delete\_by\_pk(table, pk\_col, pkv)

**if** row:

self.view.show\_message(f"Видалено батьківський рядок: {row}")

**if** deleted\_counts:

self.view.show\_message("Фактично видалено:")

**for** tt, cc **in** deleted\_counts.items():

self.view.show\_message(f" {tt}: {cc}")

**else**:

self.view.show\_message("Нічого не видалено.")

**except** ChildRowsExistError **as** e:

self.view.show\_message("Неможливо видалити — знайдені залежні (дочірні) записи:")

**for** t, c **in** e.counts.items():

self.view.show\_message(f" {t}: {c}")

self.view.show\_message("Приклади дочірніх рядків (перші 10) по кожній дочірній таблиці:")

fks = self.model.get\_referencing\_fks(table)

**for** child\_table, \_ **in** e.counts.items():

fk\_cols = [fk['child\_column'] **for** fk **in** fks **if** fk['child\_table'] == child\_table]

**if** **not** fk\_cols:

**continue**

fk\_col = fk\_cols[0]

rows = self.model.select\_by\_pk(child\_table, fk\_col, pkv)

self.view.show\_message(f"--- {child\_table} where {fk\_col} = {pkv} ---")

self.view.show\_rows(rows[:10])

self.view.show\_message("Причина: існують рядки в дочірніх таблицях, які посилаються на цей батьківський PK (референційна цілісність).")

**except** Exception **as** e:

self.view.show\_message(f"Помилка при видаленні: {e}")

model.py — потрібні методи (фрагменти)

# Попередній підрахунок дочірніх записів (для видалення по PK)

**def** preview\_child\_counts(self, table, pk\_col, pk\_val):

self.\_validate\_table(table)

counts = {}

**with** self.conn.cursor() **as** cur:

cur.execute(f'SELECT COUNT(\*) AS cnt FROM "{table}" WHERE "{pk\_col}" = %s;', (pk\_val,))

r = cur.fetchone()

parent\_exists = (r["cnt"] **if** isinstance(r, dict) **else** r[0]) > 0

counts[table] = 1 **if** parent\_exists **else** 0

fks = self.get\_referencing\_fks(table)

**for** fk **in** fks:

child = fk['child\_table']; child\_col = fk['child\_column']

cur.execute(f'SELECT COUNT(\*) AS cnt FROM "{child}" WHERE "{child\_col}" = %s;', (pk\_val,))

rr = cur.fetchone()

cnt = rr["cnt"] **if** isinstance(rr, dict) **else** rr[0]

counts[child] = cnt

**return** counts

# Видалення по PK з забороною при наявності дочірніх записів

**def** delete\_by\_pk(self, table, pk\_col, pk\_val):

self.\_validate\_table(table)

counts = self.preview\_child\_counts(table, pk\_col, pk\_val)

**if** counts.get(table, 0) == 0:

**return** None, {}

child\_totals = {t: c **for** t, c **in** counts.items() **if** t != table **and** c > 0}

**if** child\_totals:

**raise** ChildRowsExistError(child\_totals)

**with** self.conn.cursor() **as** cur:

cur.execute(f'DELETE FROM "{table}" WHERE "{pk\_col}" = %s RETURNING \*;', (pk\_val,))

row = cur.fetchone()

self.conn.commit()

deleted\_counts = {table: (1 **if** row **else** 0)}

**return** row, deleted\_counts

# Повернути всі рядки за PK (використовується контролером для прикладів)

**def** select\_by\_pk(self, table, pk\_col, pk\_val):

self.\_validate\_table(table)

**if** pk\_val **is** None:

q = f'SELECT \* FROM "{table}" ORDER BY 1 LIMIT 10;'

**with** self.conn.cursor() **as** cur:

cur.execute(q)

**return** cur.fetchall()

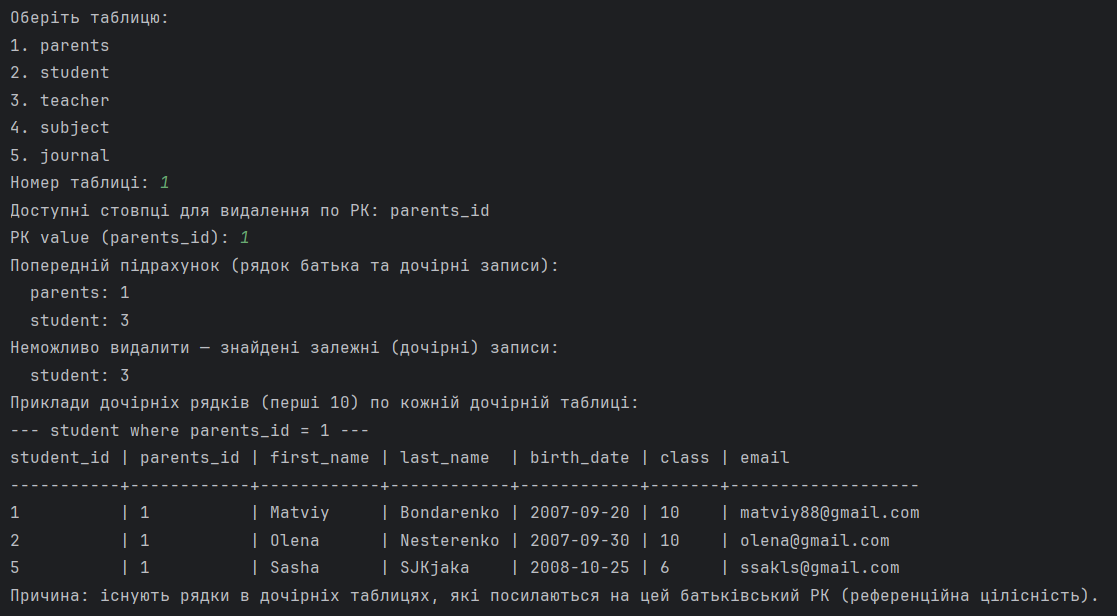
q = f'SELECT \* FROM "{table}" WHERE "{pk\_col}" = %s;'

**with** self.conn.cursor() **as** cur:

cur.execute(q, (pk\_val,))

**return** cur.fetchall()

На Рис. 5 наведено результат спроби видалення parents\_id = 1. У терміналі видно попередній підрахунок, повідомлення про наявність дочірніх записів та виведені приклади рядків з дочірньої таблиці student.



*Рисунок 5 – Результат виконання видалення батьківського рядка (термінал): блокування видалення через наявність дочірніх записів*

Рис. 6 показує таблицю parents в PgAdmin. Тут видно, що батьківські записи з parents\_id = 1 і parents\_id = 2 присутні.

Изображение выглядит как текст, снимок экрана, Шрифт, программное обеспечение

Содержимое, созданное искусственным интеллектом, может быть неверным.

*Рисунок 6 – Знімок таблиці parents в PgAdmin*

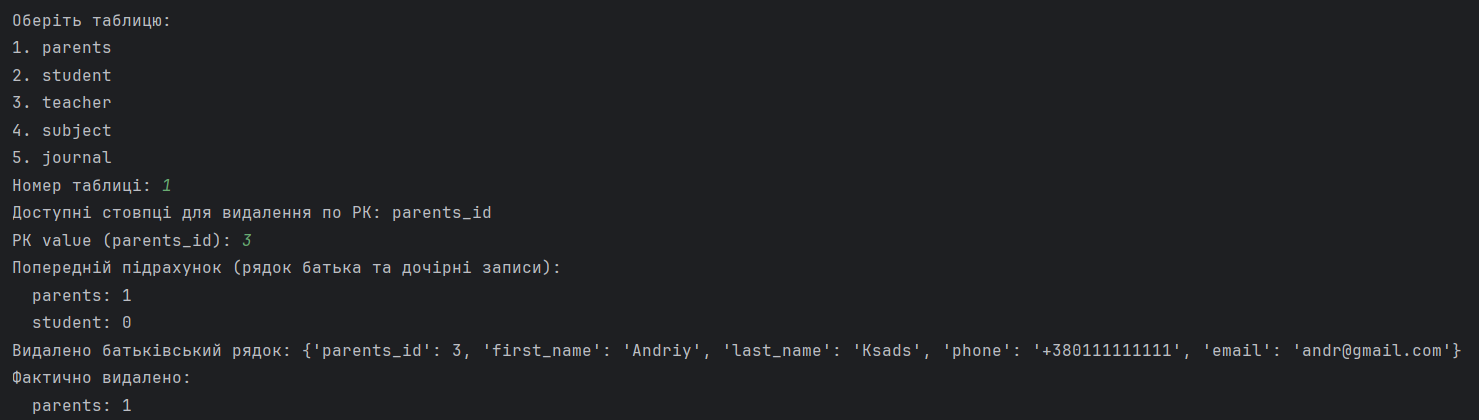
Причини помилки:

* дочірні записи містять FK, що посилаються на видаляємий батьківський запис — референційна цілісність БД не дозволяє залишити дітей без батька;
* рішення: видалити дочірні записи вручну або оновити їхні FK, або свідомо дозволити каскадне видалення, змінивши обмеження (ON DELETE CASCADE) — але це змінює поведінку даних і може призвести до небажаної втрати даних.

У прикладі було обрано parents\_id = 3 див. Рис. 7. Метод preview\_child\_counts повернув parents: 1 (батьківський рядок існує) та student: 0 (дочірні записи відсутні). Оскільки для жодної дочірньої таблиці не знайдено залежних рядків, delete\_by\_pk виконав DELETE ... RETURNING \* і повернув видалений рядок. Контролер вивів інформацію про видалений рядок та кількість фактично видалених записів. Це означає, що референційна цілісність не була порушена — жодні дочірні записи не залишилися «висячими» див. Рис. 8.



*Рисунок 7 – Знімок таблиці parents в PgAdmin* *(перед видаленням)*



*Рисунок 8 – Термінал: успішне видалення parents\_id = 3 (немає дочірніх записів)*

На Рис. 9 показано результат виконання операції видалення по PK, коли введено значення, якого немає в таблиці parents (вміст таблиці такий як на Рис.7) Контролер викликає preview\_child\_counts, перевіряє наявність батьківського рядка і, побачивши, що рядок відсутній, виводить повідомлення «Рядок з таким PK не знайдено.» — видалення не виконується. Таким чином код коректно обробляє випадок відсутнього ключа і не намагається виконати операцію DELETE.

Изображение выглядит как текст, снимок экрана, Шрифт

Содержимое, созданное искусственным интеллектом, может быть неверным.

*Рисунок 9 – Термінал: спроба видалення батьківського рядка з parents\_id = 6 — запис не знайдено*

Лістинги та скріншоти результатів виконання операції вставки запису в дочірню таблицю та виведення повідомлення про її неможливість, якщо в батьківські таблиці немає відповідного запису

Опис: наведений фрагмент реалізує вставку запису в дочірню таблицю student з подвійною перевіркою наявності відповідного батьківського запису (parents). Спочатку контролер робить швидку перевірку через Model.row\_exists(...) і просить користувача ввести існуючий parents\_id, а потім модель повторно перевіряє наявність батька і в разі його відсутності кидає ValidationError. Така комбінація контролера й моделі гарантує кращий UX (швидке повідомлення перед запитом до БД) та надійність на рівні бізнес-логіки (перевірка в моделі), додатковно захищену обмеженнями FK на боці БД.

controller.py — фрагмент з Controller.handle\_insert (для таблиці "student")

**elif** table == "student":

**while** True:

parents\_raw = input("parents\_id: ").strip()

**if** **not** parents\_raw.isdigit():

**print**("parents\_id має бути цілим числом.")

**continue**

parents\_val = int(parents\_raw)

**if** **not** self.model.row\_exists("parents", "parents\_id", parents\_val):

**print**("Parent з таким ID не знайдений. Введіть існуючий parents\_id.")

**continue**

**break**

fn = self.\_read\_nonempty("First name: ")

ln = self.\_read\_nonempty("Last name: ")

birth = self.\_read\_date("Birth date (YYYY-MM-DD): ")

class\_ = self.\_read\_class("Class (наприклад 10A): ")

email = self.\_read\_email("Email: ")

**try**:

sid = self.model.insert\_student(pk\_val, parents\_val, fn, ln, birth, class\_, email)

self.view.show\_message(f"Inserted student\_id={sid}")

**except** ValidationError **as** e:

self.view.show\_message(f"Помилка валідації: {e}")

**except** psycopg.errors.ForeignKeyViolation:

self.view.show\_message("Помилка: батько з таким ID не знайдений (FK).")

**except** Exception **as** e:

self.view.show\_message(f"Помилка при вставці: {e}")

model.py — метод insert\_student

**def** insert\_student(self, student\_id, parents\_id, first\_name, last\_name, birth\_date, class\_, email):

**if** student\_id **is** None:

**raise** ValidationError("student\_id обов'язковий для вставки (не можна автогенерувати).")

**if** parents\_id **is** **not** None **and** **not** self.row\_exists("parents", "parents\_id", parents\_id):

**raise** ValidationError("Parent with given parents\_id not found.")

**with** self.conn.cursor() **as** cur:

q = ('INSERT INTO "student"(student\_id, parents\_id, first\_name, last\_name, birth\_date, class, email) '

'VALUES (%s,%s,%s,%s,%s,%s,%s) RETURNING student\_id;')

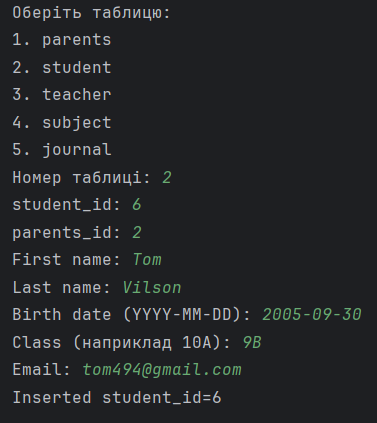
cur.execute(q, (student\_id, parents\_id, first\_name, last\_name, birth\_date, class\_, email))

sid = cur.fetchone()["student\_id"]

self.conn.commit()

**return** sid

У терміналі видно послідовність вводів і повідомлення Inserted student\_id=6 див. Рис. 10, що підтверджує успішне додавання рядка в student див. Рис.11 . Оскільки parents\_id = 2 існував у таблиці parents, валідація FK пройшла і INSERT виконано.



*Рисунок 10 – Термінал: успішна вставка запису в таблицю student (виведено "Inserted student\_id=6")*

Изображение выглядит как текст, снимок экрана, Шрифт, число

Содержимое, созданное искусственным интеллектом, может быть неверным.

*Рисунок 11 – Таблиця student в PgAdmin* *(після додавання рядка)*

На Рис.12 термінала показано сценарій, коли користувач спробував додати запис у дочірню таблицю student, вказавши parents\_id = 999, якого немає в таблиці parents. Контролер виявляє відсутність батька під час попередньої перевірки Model.row\_exists(...) та негайно інформує користувача: «Parent з таким ID не знайдений. Введіть існуючий parents\_id.» — й не дозволяє продовжити вставку.

Изображение выглядит как текст, снимок экрана, программное обеспечение, Мультимедийное программное обеспечение

Содержимое, созданное искусственным интеллектом, может быть неверным.

*Рисунок 12 – Термінал: спроба вставки студента з невірним parents\_id (999) — контролер відхиляє введення та просить ввести існуючий parents\_id*

Причини такої поведінки (чому вставка відхилена):

* Відсутній батьківський запис з таким parents\_id — порушення референційної цілісності могло б виникнути, якби дочірній запис створився без існуючого батька.
* Контролер запобігає зайвим запитам і повідомляє користувача, а модель/БД забезпечують додатковий захист.

SQL-запити, що ілюструють генерацію

1. **Генерація parents**

(копія з model.py)

Опис: додає n нових рядків у таблицю parents, генерує випадкові імена, телефон та email.

WITH maxv AS (

SELECT COALESCE(MAX(parents\_id), 0) AS m FROM "parents"

), gens AS (

SELECT m + row\_number() OVER () AS new\_id,

left(md5(random()::text),8) AS fn,

left(md5(random()::text),8) AS ln,

('+380' || (100000000 + floor(random()\*900000000)::bigint)::text) AS phone,

lower(left(md5(random()::text),8) || '@example.com') AS em

FROM maxv, generate\_series(1, %s)

)

INSERT INTO "parents"(parents\_id, first\_name, last\_name, phone, email)

SELECT new\_id, fn, ln, phone, em FROM gens;

До операції в таблиці вже були три ручні записи з parents\_id = 1, 2, 3 (Olena, Mark, Andriy). Команда генерації додала n = 10 рядків з автоматично згенерованими значеннями — у таблиці це відповідає parents\_id = 4 … parents\_id = 13. Тобто записи з parents\_id від 1 до 3 — до генерації; записи з parents\_id від 4 до 13 — згенеровані під час запуску (значення імен/прізвищ/телефонів/emails випадкові, згенеровані функціями md5(random()...) та random()).

Изображение выглядит как текст, снимок экрана, Шрифт

Содержимое, созданное искусственным интеллектом, может быть неверным.

*Рисунок 12 – Термінал:введення параметрів для генерації*

Изображение выглядит как текст, снимок экрана, число, Шрифт

Содержимое, созданное искусственным интеллектом, может быть неверным.

*Рисунок 13 – Вміст таблиці parents в PgAdmin* *після генерації*

1. **Генерація teacher**

(копія з model.py)

Опис: генерує n вчителів з випадковими іменами та email.

WITH maxv AS (

SELECT COALESCE(MAX(teacher\_id), 0) AS m FROM "teacher"

), gens AS (

SELECT m + row\_number() OVER () AS new\_id,

left(md5(random()::text),8) AS fn,

left(md5(random()::text),8) AS ln,

lower(left(md5(random()::text),8) || '@example.com') AS em

FROM maxv, generate\_series(1, %s)

)

INSERT INTO "teacher"(teacher\_id, first\_name, last\_name, email)

SELECT new\_id, fn, ln, em FROM gens;

Команда генерації додала n = 5 нових записів з teacher\_id = 4 … teacher\_id = 8 див. Рис. 15. Записи з teacher\_id 1–3 — до генерації (ручні), а записи з teacher\_id 4–8 — згенеровані під час виконання (значення first\_name, last\_name, email отримані випадково функціями md5(random()...)).

Изображение выглядит как текст, снимок экрана, Шрифт

Содержимое, созданное искусственным интеллектом, может быть неверным.

*Рисунок 14 – Термінал: генерація teacher — введено n = 5*

Изображение выглядит как текст, снимок экрана, Шрифт, число

Содержимое, созданное искусственным интеллектом, может быть неверным.

*Рисунок 15 –Вміст таблиці teacher в PgAdmin* *після генерації*

1. **Генерація subject**

(копія з model.py)

Опис: генерує n предметів із випадковими назвами.

WITH maxv AS (

SELECT COALESCE(MAX(subject\_id), 0) AS m FROM "subject"

), gens AS (

SELECT m + row\_number() OVER () AS new\_id,

left(md5(random()::text),10) AS nm

FROM maxv, generate\_series(1, %s)

)

INSERT INTO "subject"(subject\_id, name)

SELECT new\_id, nm FROM gens;

Команда генерації з параметром n = 6 додала шість нових рядків з subject\_id = 4 … subject\_id = 9 див. Рис. 17. Тобто записи з subject\_id 1–3 — до генерації; записи з subject\_id 4–9 — згенеровані під час виконання.

Изображение выглядит как текст, снимок экрана, Шрифт

Содержимое, созданное искусственным интеллектом, может быть неверным.

*Рисунок 16 –* *Термінал: генерація subject — введено n = 6*

Изображение выглядит как текст, снимок экрана, Шрифт, число

Содержимое, созданное искусственным интеллектом, может быть неверным.

*Рисунок 16 –* *Вміст таблиці subject в PgAdmin* *після генерації*

1. **Генерація student**

(копія з model.py)

Опис: генерує n студентів; для parents\_id обирається випадковий існуючий parents\_id (через join з pids), генерує ім'я, прізвище, дату народження, клас і email.

WITH maxv AS (

SELECT COALESCE(MAX(student\_id), 0) AS m FROM "student"

), pids AS (

SELECT row\_number() OVER (ORDER BY parents\_id) AS idx, parents\_id

FROM "parents"

), gens AS (

SELECT m + row\_number() OVER () AS new\_id,

(floor(random() \* (SELECT count(\*) FROM "parents"))::int + 1) AS pidx,

left(md5(random()::text),6) AS fn,

left(md5(random()::text),6) AS ln,

(date '2005-01-01' + (trunc(random()\*4000)::int))::date AS bd,

(floor(1 + random()\*11)::int)::text

|| (CASE WHEN random() < 0.25 THEN chr((65 + floor(random()\*2))::int) ELSE '' END) AS cls,

lower(left(md5(random()::text),6) || '@example.com') AS em

FROM maxv, generate\_series(1, %s)

)

INSERT INTO "student"(student\_id, parents\_id, first\_name, last\_name, birth\_date, **class**, email)

SELECT g.new\_id, p.parents\_id, g.fn, g.ln, g.bd, g.cls, g.em

FROM gens g

JOIN pids p ON p.idx = g.pidx;

На Рис. 18 видно результати генерації для таблиці student. До виконання операції в таблиці вже були наявні ручні записи з student\_id від **1** до **6**. Команда генерації з параметром n = 12 додала дванадцять нових рядків, що відповідає student\_id = 7 … student\_id = 18. Значення полів first\_name, last\_name, email для згенерованих рядків — випадкові (формуються як фрагменти хешів / md5(random()) і @example.com).

Изображение выглядит как текст, снимок экрана, Шрифт

Содержимое, созданное искусственным интеллектом, может быть неверным.

*Рисунок 17 –* *Термінал: генерація student — введено n = 12*

Изображение выглядит как текст, снимок экрана, число, меню

Содержимое, созданное искусственным интеллектом, может быть неверным.

*Рисунок 18 –* *Вміст таблиці student в PgAdmin* *після генерації*

1. **Генерація journal**

(копія з model.py)

Опис: генерує n записів журналу, вибираючи випадкові існуючі student, teacher, subject.

WITH maxv AS (

SELECT COALESCE(MAX(journal\_id), 0) AS m FROM "journal"

), counts AS (

SELECT (SELECT count(\*) FROM "student") AS students\_count,

(SELECT count(\*) FROM "teacher") AS teachers\_count,

(SELECT count(\*) FROM "subject") AS subjects\_count

), gens AS (

SELECT

m + row\_number() OVER () AS new\_id,

(floor(random() \* (SELECT students\_count FROM counts))::int + 1) AS s\_idx,

(floor(random() \* (SELECT teachers\_count FROM counts))::int + 1) AS t\_idx,

(floor(random() \* (SELECT subjects\_count FROM counts))::int + 1) AS sb\_idx,

(date '2020-01-01' + (trunc(random()\*2000)::int))::date AS ed,

(floor(random()\*12)::int + 1) AS gr\_rand

FROM maxv, counts, generate\_series(1, %s)

)

INSERT INTO "journal"(journal\_id, student\_id, teacher\_id, subject\_id, entry\_date, grade, attendance\_status)

SELECT

g.new\_id,

s.student\_id,

t.teacher\_id,

sb.subject\_id,

g.ed,

g.gr\_rand,

'present'

FROM gens g

JOIN (SELECT row\_number() OVER (ORDER BY student\_id) AS idx, student\_id FROM "student") s ON s.idx = g.s\_idx

JOIN (SELECT row\_number() OVER (ORDER BY teacher\_id) AS idx, teacher\_id FROM "teacher") t ON t.idx = g.t\_idx

JOIN (SELECT row\_number() OVER (ORDER BY subject\_id) AS idx, subject\_id FROM "subject") sb ON sb.idx = g.sb\_idx;

На Рис. 19 показано результат генерації даних у таблиці journal. До виконання операції в таблиці вже містилися попередні записи. Команда генерації з параметром n = 8 додала в таблицю 8 нових рядків. У згенерованих рядках поля student\_id, teacher\_id, subject\_id вибрані випадково зі списків наявних записів у відповідних таблицях, entry\_date — випадкова дата в діапазоні, grade — випадкове число 1..12 (або NULL для випадків attendance\_status = 'absent'), а attendance\_status розподіляється випадково між 'present', 'absent', 'late'

Изображение выглядит как текст, снимок экрана, Шрифт

Содержимое, созданное искусственным интеллектом, может быть неверным.

*Рисунок 18 –* *Термінал: генерація journal — введено n = 8*

Изображение выглядит как текст, снимок экрана, число, Шрифт

Содержимое, созданное искусственным интеллектом, может быть неверным.

*Рисунок 19 –* *Вміст таблиці journal в PgAdmin* *після генерації*

За умовою методичних вказівок необхідно також провести тестову генерацію 100 000 записів для однієї таблиці (вибраноjournal*)* з метою оцінки продуктивності та поведінки БД при масовому вставленні.

Перед запуском тесту всі рядки в цільовій таблиці було видалено щоб гарантувати, що після операції в таблиці буде саме n = 100000 нових записів.

На Рис. 20 видно, що в journal зустрічаються NULL у полі grade — це очікувана поведінка для записів з attendance\_status = 'absent' (у генераторі після вставки є UPDATE, що робить grade = NULL для absent). Також видно, що student\_id, teacher\_id, subject\_id лежать в розумних межах — отже логіка вибору існуючих FK спрацювала коректно.

Висока швидкість генерації спричинена тим, що генерація виконувалась в одному set-based SQL (CTE + INSERT ... SELECT) без численних клієнтських round-trips. Також на результат впливають фактори середовища (швидкий диск, відсутність зайвих індексів/тригерів, конфігурація PostgreSQL).

Изображение выглядит как текст, снимок экрана, число, Шрифт

Содержимое, созданное искусственным интеллектом, может быть неверным.

*Рисунок 20 –* *Вміст таблиці journal в PgAdmin* *після генерації 100000 рядків*

Ілюстрації введення пошукового запиту та результати виконання (копії SQL-запитів з початковими параметрами)

**Мета запиту 1:** для вказаного класу (s.class) отримати по кожному предмету (subject) кількість оцінок (marks\_count) та середній бал (avg\_grade), відсортовані за спаданням середнього балу.

Копія SQL-запиту

SELECT sb.name AS subject,

COUNT(j.journal\_id) AS marks\_count,

AVG(j.grade) AS avg\_grade

FROM "journal" j

JOIN "subject" sb ON j.subject\_id = sb.subject\_id

JOIN "student" s ON j.student\_id = s.student\_id

WHERE s.**class** = %s

GROUP BY sb.name

ORDER BY avg\_grade DESC;

Фрагмент controller.py — як зчитується параметр і викликається запит

**if** ch == "1":

cls = input("Введіть class (наприклад 10A): ").strip()

t0 = time.time()

rows = self.model.complex\_query\_1(cls)

t = (time.time() - t0) \* 1000

self.view.show\_rows(rows)

self.view.show\_message(f"Час виконання: {t:.2f} ms")

Метод у model.py, що виконує SQL

**def** complex\_query\_1(self, class\_value):

q = """

SELECT sb.name AS subject, COUNT(j.journal\_id) AS marks\_count, AVG(j.grade) AS avg\_grade

FROM "journal" j

JOIN "subject" sb ON j.subject\_id = sb.subject\_id

JOIN "student" s ON j.student\_id = s.student\_id

WHERE s.class = %s

GROUP BY sb.name

ORDER BY avg\_grade DESC;

"""

**with** self.conn.cursor() **as** cur:

cur.execute(q, (class\_value,))

**return** cur.fetchall()

Робота коду:

1. Контролер запитує у користувача параметр class (наприклад 10A), фіксує початковий час, викликає метод моделі і після виконання виводить рядки результату та час виконання у мілісекундах.
2. Модель збирає дані за допомогою одного SQL-запиту: JOIN між таблицями journal, subject і student, фільтр по s.class, агрегація COUNT і AVG, групування по назві предмета.
3. View (через show\_rows) форматовано виводить результати у табличному вигляді для читача.

Для коректного та однозначного тестування пошукових запитів, перед виконанням тестів було очищено базу від попередньо згенерованих тестових рядків. При цьому ручні записи, що були в базі до початку робіт, збережені (тільки journal було заново згенеровано). Такий підхід дозволяє виконувати запити над чистішим набором даних.

На Рис. 22 показано ввід параметра та результат виконання складного запиту (пункт меню 11 → варіант 1). У якості параметра було введено class = 10. Програма виконала SQL-запит, який підраховує для кожного предмета кількість оцінок (marks\_count) та середній бал (avg\_grade) для вказаного класу, після чого вивела табличний результат і час виконання див Рис. 21.

Результат (інтерпретація):

Таблиця показує три предмети з такими значеннями:

* Programming — marks\_count = 2, avg\_grade = 11
* Math — marks\_count = 1, avg\_grade = 8
* History — marks\_count = 2, avg\_grade = 6.50

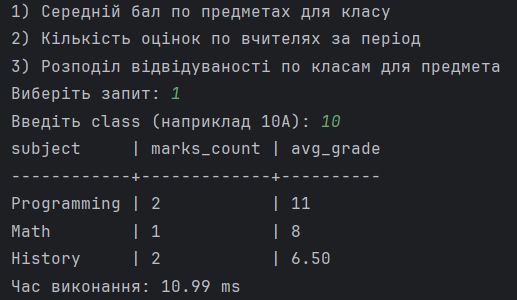
Це означає, що для класу 10 у журналі знайдено відповідні оцінки: два значення для Programming, одне для Math і два для History; AVG по кожному предмету обчислено як середнє арифметичне наявних оцінок.

Під таблицею видно Час виконання: 10.99 ms. Це час, зафіксований контролером (wall-clock для виклику complex\_query\_1), і він відображає швидкість виконання агрегованого SQL-запиту над поточною невеликою вибіркою. Для великих наборів даних (або при складніших індексах) цей час може істотно збільшуватися.

Изображение выглядит как текст, снимок экрана, Шрифт, число

Содержимое, созданное искусственным интеллектом, может быть неверным.

*Рисунок 21 –* *Вміст таблиці journal в PgAdmin* *після генерації 7 рядків*



*Рисунок 22 –* *Термінал: виконання запиту «Середній бал по предметах для класу»*

**Мета запиту 2:** для вказаного періоду (date\_from .. date\_to) підрахувати, скільки записів журналу (оцінок) має кожен вчитель — це дозволяє оцінити активність/навантаження вчителів у заданому часовому інтервалі.

Копія SQL-запиту

SELECT t.first\_name || ' ' || t.last\_name AS teacher,

COUNT(j.journal\_id) AS marks\_count

FROM "journal" j

JOIN "teacher" t ON j.teacher\_id = t.teacher\_id

WHERE j.entry\_date BETWEEN %s AND %s

GROUP BY teacher

ORDER BY marks\_count DESC

LIMIT 50;

Фрагмент controller.py — як зчитуються параметри і викликається запит

**elif** ch == "2":

d1 = input("Дата з (YYYY-MM-DD): ").strip()

d2 = input("Дата по (YYYY-MM-DD): ").strip()

t0 = time.time()

rows = self.model.complex\_query\_2(d1, d2)

t = (time.time() - t0) \* 1000

self.view.show\_rows(rows)

self.view.show\_message(f"Час виконання: {t:.2f} ms")

Метод у model.py, що виконує SQL

**def** complex\_query\_2(self, date\_from, date\_to):

q = """

SELECT t.first\_name || ' ' || t.last\_name AS teacher, COUNT(j.journal\_id) AS marks\_count

FROM "journal" j

JOIN "teacher" t ON j.teacher\_id = t.teacher\_id

WHERE j.entry\_date BETWEEN %s AND %s

GROUP BY teacher ORDER BY marks\_count DESC LIMIT 50;

"""

**with** self.conn.cursor() **as** cur:

cur.execute(q, (date\_from, date\_to))

**return** cur.fetchall()

Робота коду:

1. Контролер зчитує початкову та кінцеву дати у форматі YYYY-MM-DD, запускає вимір часу, викликає model.complex\_query\_2 і виводить результати та час виконання.
2. Модель виконує один агрегований SQL-запит з JOIN між journal і teacher, фільтром по датах, GROUP BY teacher і сортуванням за кількістю оцінок.
3. View виводить таблично teacher | marks\_count.

На Рис. 23 зображено ввід параметрів для запиту (пункт меню 11 → варіант 2) з Дата з = 2023-03-25 та Дата по = 2024-08-26. Програма повернула агрегований результат — імені вчителя (teacher) та кількість оцінок (marks\_count) за вказаний період. У прикладі видно один рядок: Oksana Melnyk | 2. Під таблицею відображено час виконання: Час виконання: 15.19 ms.

Відображений час 15.19 ms — це wall-clock, зафіксований контролером (час від початку до закінчення виклику complex\_query\_2). Це невелике значення відповідає виконанню агрегації над відносно малою вибіркою (або ефективній реалізації запиту).

Изображение выглядит как текст, снимок экрана, Шрифт

Содержимое, созданное искусственным интеллектом, может быть неверным.

*Рисунок 23 –* *Термінал: виконання запиту «Кількість оцінок по вчителях за період»*

**Мета запиту 3:** для вказаного предмета (subject.name) отримати по кожному класу підрахунок статусів відвідуваності (present, absent, late). Це дає розподіл відвідуваності по класам для обраного предмета.

Копія SQL-запиту

SELECT s.**class**, j.attendance\_status, COUNT(\*) AS cnt

FROM "journal" j

JOIN "student" s ON j.student\_id = s.student\_id

JOIN "subject" sb ON j.subject\_id = sb.subject\_id

WHERE sb.name = %s

GROUP BY s.**class**, j.attendance\_status

ORDER BY s.**class**;

Фрагмент controller.py — як зчитується параметр і викликається запит

**elif** ch == "3":

subj = input("Назва предмета: ").strip()

t0 = time.time()

rows = self.model.complex\_query\_3(subj)

t = (time.time() - t0) \* 1000

self.view.show\_rows(rows)

self.view.show\_message(f"Час виконання: {t:.2f} ms")

Метод у model.py, що виконує SQL

**def** complex\_query\_3(self, subject\_name):

q = """

SELECT s.class, j.attendance\_status, COUNT(\*) AS cnt

FROM "journal" j

JOIN "student" s ON j.student\_id = s.student\_id

JOIN "subject" sb ON j.subject\_id = sb.subject\_id

WHERE sb.name = %s

GROUP BY s.class, j.attendance\_status

ORDER BY s.class;

"""

**with** self.conn.cursor() **as** cur:

cur.execute(q, (subject\_name,))

**return** cur.fetchall()

Робота коду:

1. Контролер читає назву предмета, вимірює час і викликає модель.
2. Модель виконує JOIN між journal, student, subject, фільтрує записи за назвою предмета, групує по класу і статусу відвідуваності і рахує кількість записів.
3. View виводить результати таблично та показує час виконання.

На Рис. 24 видно, що обрано варіант 3 у меню складних запитів і введено назву предмета Programming. Програма виконала запит, який повертає для кожного класу кількість записів журналу для кожного статусу відвідуваності (present, absent, late).

Результат показує чотири рядки:

* клас 10, absent — 1 запис
* клас 10, present — 1 запис
* клас 5, late — 1 запис
* клас 6, late — 1 запис

Тобто для предмета Programming зафіксовано по одному запису кожного з наведених статусів у відповідних класах. Невеликі значення (cnt = 1) свідчать про малу вибірку.

Під таблицею видно Час виконання: 1.93 ms. Це дуже швидкий час для агрегації над невеликою вибіркою. За великих обсягів даних або при відсутності індексів час може збільшитись.

Изображение выглядит как текст, снимок экрана, Шрифт

Содержимое, созданное искусственным интеллектом, может быть неверным.

*Рисунок 24 –* *Термінал: виконання запиту «Розподіл відвідуваності по класам для предмета»*

Короткий опис функцій модуля model.py

**Фрагмент 1 — Імпорти, конфігурація та винятки**

Опис: налаштування підключення до БД, список дозволених таблиць, константи й визначення кастомних винятків ChildRowsExistError (поки є залежні дочірні записи) і ValidationError (помилки валідації).

**import** os

**import** psycopg

**from** psycopg.rows **import** dict\_row

DB = {

"dbname": os.getenv("DB\_NAME", "postgres"),

"user": os.getenv("DB\_USER", "postgres"),

"password": os.getenv("DB\_PASS", "1111"),

"host": os.getenv("DB\_HOST", "localhost"),

"port": os.getenv("DB\_PORT", "5432"),

}

ALLOWED\_TABLES = ["parents", "student", "teacher", "subject", "journal"]

ATTENDANCE\_STATUSES = ('present', 'absent', 'late')

# Виняток (є дочірні записи)

**class** ChildRowsExistError(Exception):

**def** \_\_init\_\_(self, counts):

super().\_\_init\_\_("Child rows exist")

self.counts = counts

# Виняток (помилка валідації)

**class** ValidationError(Exception):

**pass**

**Фрагмент 2 — Клас Model, PK\_MAP, ініціалізація і закриття**

Опис:

* PK\_MAP — мапа для швидкого знаходження імен PK-стовпчиків по таблицях.
* \_\_init\_\_ — встановлює з'єднання з PostgreSQL, налаштовує row\_factory щоб повертати словники.
* close — коректно закриває з'єднання.

**class** Model:

PK\_MAP = {

"parents": "parents\_id",

"teacher": "teacher\_id",

"subject": "subject\_id",

"student": "student\_id",

"journal": "journal\_id",

}

# Ініціалізація з'єднання

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

self.conn = psycopg.connect(\*\*DB)

self.conn.row\_factory = dict\_row

self.conn.autocommit = False

# Закрити з'єднання

**def** close(self):

**if** self.conn:

self.conn.close()

**Фрагмент 3 — Валідація таблиці і отримання списку стовпців/таблиць**

Опис:

* \_validate\_table — захист від роботи з небажаними таблицями.
* \_get\_columns\_list / get\_columns — дають перелік стовпців
* get\_tables — повертає список дозволених таблиць.

# Перевірка допустимої таблиці

**def** \_validate\_table(self, table):

**if** table **not** **in** ALLOWED\_TABLES:

**raise** ValueError("Невідома таблиця")

# Повернути список стовпців

**def** \_get\_columns\_list(self, table):

q = """

SELECT column\_name

FROM information\_schema.columns

WHERE table\_schema = 'public' AND table\_name = %s

ORDER BY ordinal\_position;

"""

**with** self.conn.cursor() **as** cur:

cur.execute(q, (table,))

**return** [r["column\_name"] **for** r **in** cur.fetchall()]

# Повернути список таблиць

**def** get\_tables(self):

**return** ALLOWED\_TABLES.copy()

# Повернути інформацію про стовпці таблиці

**def** get\_columns(self, table):

self.\_validate\_table(table)

q = """

SELECT column\_name, data\_type, is\_nullable

FROM information\_schema.columns

WHERE table\_schema = 'public' AND table\_name = %s

ORDER BY ordinal\_position;

"""

**with** self.conn.cursor() **as** cur:

cur.execute(q, (table,))

**return** cur.fetchall()

**Фрагмент 4 — Базові операції вибірки/перевірки існування/лістингу**

Опис:

* list\_table — перегляд даних таблиці з лімітом.
* row\_exists — швидка перевірка наявності запису (використовується для валідацій перед вставкою).
* select\_by\_pk — отримати всі колонки рядка по PK.

# Повернути рядки таблиці

**def** list\_table(self, table, limit=200):

self.\_validate\_table(table)

q = f'SELECT \* FROM "{table}" ORDER BY 1 LIMIT %s'

**with** self.conn.cursor() **as** cur:

cur.execute(q, (limit,))

**return** cur.fetchall()

# Перевірити наявність рядка за PK

**def** row\_exists(self, table, pk\_col, value):

self.\_validate\_table(table)

q = f'SELECT 1 FROM "{table}" WHERE "{pk\_col}" = %s LIMIT 1'

**with** self.conn.cursor() **as** cur:

cur.execute(q, (value,))

**return** cur.fetchone() **is** **not** None

# Повернути всі рядки за PK

**def** select\_by\_pk(self, table, pk\_col, pk\_val):

self.\_validate\_table(table)

q = f'SELECT \* FROM "{table}" WHERE "{pk\_col}" = %s;'

**with** self.conn.cursor() **as** cur:

cur.execute(q, (pk\_val,))

**return** cur.fetchall()

**Фрагмент 5 — Вставки (parents, student, teacher, subject)**

Опис: insert\_parent, insert\_student, insert\_teacher, insert\_subject — вставляють рядки в відповідні таблиці, роблять локальну валідацію (наприклад перевірка наявності батька для student) і повертають згенерований/повернутий PK. Використовуються ValidationError для логіки валідації.

# Вставка батьків

**def** insert\_parent(self, parents\_id, first\_name, last\_name, phone, email):

**if** parents\_id **is** None:

**raise** ValidationError("parents\_id обов'язковий для вставки (не можна автогенерувати).")

**with** self.conn.cursor() **as** cur:

cur.execute(

'INSERT INTO "parents"(parents\_id, first\_name, last\_name, phone, email) '

'VALUES (%s,%s,%s,%s,%s) RETURNING parents\_id;',

(parents\_id, first\_name, last\_name, phone, email)

)

row = cur.fetchone()

self.conn.commit()

**return** row["parents\_id"] **if** isinstance(row, dict) **else** row[0]

# Вставка студентів

**def** insert\_student(self, student\_id, parents\_id, first\_name, last\_name, birth\_date, class\_, email):

**if** student\_id **is** None:

**raise** ValidationError("student\_id обов'язковий для вставки (не можна автогенерувати).")

**if** parents\_id **is** **not** None **and** **not** self.row\_exists("parents", "parents\_id", parents\_id):

**raise** ValidationError("Parent with given parents\_id not found.")

**with** self.conn.cursor() **as** cur:

q = 'INSERT INTO "student"(student\_id, parents\_id, first\_name, last\_name, birth\_date, class, email) VALUES (%s,%s,%s,%s,%s,%s,%s) RETURNING student\_id;'

cur.execute(q, (student\_id, parents\_id, first\_name, last\_name, birth\_date, class\_, email))

sid = cur.fetchone()["student\_id"]

self.conn.commit()

**return** sid

# Вставка вчителя

**def** insert\_teacher(self, teacher\_id, first\_name, last\_name, email):

**if** teacher\_id **is** None:

**raise** ValidationError("teacher\_id обов'язковий для вставки (не можна автогенерувати).")

**with** self.conn.cursor() **as** cur:

q = 'INSERT INTO "teacher"(teacher\_id, first\_name, last\_name, email) VALUES (%s,%s,%s,%s) RETURNING teacher\_id;'

cur.execute(q, (teacher\_id, first\_name, last\_name, email))

tid = cur.fetchone()["teacher\_id"]

self.conn.commit()

**return** tid

# Вставка предмета

**def** insert\_subject(self, subject\_id, name):

**if** subject\_id **is** None:

**raise** ValidationError("subject\_id обов'язковий для вставки (не можна автогенерувати).")

**with** self.conn.cursor() **as** cur:

q = 'INSERT INTO "subject"(subject\_id, name) VALUES (%s,%s) RETURNING subject\_id;'

cur.execute(q, (subject\_id, name))

sid = cur.fetchone()["subject\_id"]

self.conn.commit()

**return** sid

**Фрагмент 6 — Вставка журналу з валідаціями**

Опис: insert\_journal — виконує комплексну валідацію (існування FK, коректність attendance\_status, правила щодо grade) і записує рядок у journal. Повертає journal\_id.

# Вставка журналу

**def** insert\_journal(self, journal\_id, student\_id, teacher\_id, subject\_id, entry\_date, grade, attendance\_status):

**if** journal\_id **is** None:

**raise** ValidationError("journal\_id обов'язковий для вставки (не можна автогенерувати).")

**if** **not** self.row\_exists("student", "student\_id", student\_id):

**raise** ValidationError("Student not found.")

**if** teacher\_id **is** **not** None **and** **not** self.row\_exists("teacher", "teacher\_id", teacher\_id):

**raise** ValidationError("Teacher not found.")

**if** subject\_id **is** **not** None **and** **not** self.row\_exists("subject", "subject\_id", subject\_id):

**raise** ValidationError("Subject not found.")

**if** attendance\_status **is** **not** None **and** attendance\_status **not** **in** ATTENDANCE\_STATUSES:

**raise** ValidationError("Invalid attendance\_status.")

**if** attendance\_status == 'absent':

**if** grade **is** **not** None:

**raise** ValidationError("If attendance is 'absent', grade must be NULL / not provided.")

**else**:

**if** grade **is** None:

**raise** ValidationError("For present/late attendance grade must be provided (1..12).")

**try**:

g = int(grade)

**except** Exception:

**raise** ValidationError("Grade must be integer.")

**if** g < 1 **or** g > 12:

**raise** ValidationError("Grade out of allowed range (1-12).")

**with** self.conn.cursor() **as** cur:

q = """INSERT INTO "journal"(journal\_id, student\_id, teacher\_id, subject\_id, entry\_date, grade, attendance\_status)

VALUES (%s,%s,%s,%s,%s,%s,%s) RETURNING journal\_id;"""

cur.execute(q, (journal\_id, student\_id, teacher\_id, subject\_id, entry\_date, grade, attendance\_status))

jid = cur.fetchone()["journal\_id"]

self.conn.commit()

**return** jid

**Фрагмент 7 — Оновлення, лічильники, видалення всіх рядків**

Опис:

* update\_by\_pk — динамічно будує UPDATE по словнику updates і повертає оновлений рядок.
* delete\_all — видаляє всі рядки таблиці (використовується у пункті меню 10).
* count\_rows — повертає кількість рядків у таблиці.

# Оновити рядок по PK

**def** update\_by\_pk(self, table, pk\_col, pk\_val, updates: dict):

self.\_validate\_table(table)

**if** **not** updates:

**return** None

cols = self.\_get\_columns\_list(table)

set\_parts = []

params = []

**for** k, v **in** updates.items():

**if** k **not** **in** cols:

**raise** ValueError(f"Невідомий стовпець: {k}")

set\_parts.append(f'"{k}" = %s')

params.append(v)

params.append(pk\_val)

set\_clause = ", ".join(set\_parts)

q = f'UPDATE "{table}" SET {set\_clause} WHERE "{pk\_col}" = %s RETURNING \*;'

**with** self.conn.cursor() **as** cur:

cur.execute(q, tuple(params))

row = cur.fetchone()

self.conn.commit()

**return** row

# Видалити всі рядки в таблиці

**def** delete\_all(self, table):

self.\_validate\_table(table)

q = f'DELETE FROM "{table}";'

**with** self.conn.cursor() **as** cur:

cur.execute(q)

deleted = cur.rowcount

self.conn.commit()

**return** deleted

# Порахувати рядки в таблиці

**def** count\_rows(self, table):

self.\_validate\_table(table)

q = f'SELECT COUNT(\*) AS cnt FROM "{table}";'

**with** self.conn.cursor() **as** cur:

cur.execute(q)

**return** cur.fetchone()["cnt"]

**Фрагмент 8 — Отримання FK, підрахунок дочірніх записів (preview) і лічильник дітей**

Опис:

* get\_referencing\_fks — знаходить в БД які таблиці мають FK, що посилаються на дану таблицю
* count\_children — підрахунок дочірніх рядків по FK-стовпцю.
* preview\_child\_counts — збирає інформацію про те, скільки дочірніх записів існує для даного батька (буде використано у логіці видалення).

# Отримати FK, які посилаються на цю таблицю

**def** get\_referencing\_fks(self, table):

q = """

SELECT

kcu.table\_name AS child\_table,

kcu.column\_name AS child\_column

FROM information\_schema.table\_constraints AS tc

JOIN information\_schema.key\_column\_usage AS kcu

ON tc.constraint\_name = kcu.constraint\_name AND tc.table\_schema = kcu.table\_schema

JOIN information\_schema.constraint\_column\_usage AS ccu

ON ccu.constraint\_name = tc.constraint\_name AND ccu.table\_schema = tc.table\_schema

WHERE tc.constraint\_type = 'FOREIGN KEY'

AND ccu.table\_name = %s

AND ccu.table\_schema = 'public';

"""

**with** self.conn.cursor() **as** cur:

cur.execute(q, (table,))

rows = cur.fetchall()

res = []

**for** r **in** rows:

**if** isinstance(r, dict):

res.append({'child\_table': r['child\_table'], 'child\_column': r['child\_column']})

**else**:

res.append({'child\_table': r[0], 'child\_column': r[1]})

**return** res

# Порахувати дітей

**def** count\_children(self, child\_table, fk\_column, value):

self.\_validate\_table(child\_table)

q = f'SELECT COUNT(\*) AS cnt FROM "{child\_table}" WHERE "{fk\_column}" = %s'

**with** self.conn.cursor() **as** cur:

cur.execute(q, (value,))

**return** cur.fetchone()["cnt"]

# Попередній підрахунок дочірніх записів (для видалення)

**def** preview\_child\_counts(self, table, pk\_col, pk\_val):

self.\_validate\_table(table)

counts = {}

**with** self.conn.cursor() **as** cur:

cur.execute(f'SELECT COUNT(\*) AS cnt FROM "{table}" WHERE "{pk\_col}" = %s;', (pk\_val,))

r = cur.fetchone()

parent\_exists = (r["cnt"] **if** isinstance(r, dict) **else** r[0]) > 0

counts[table] = 1 **if** parent\_exists **else** 0

fks = self.get\_referencing\_fks(table)

**for** fk **in** fks:

child = fk['child\_table']; child\_col = fk['child\_column']

cur.execute(f'SELECT COUNT(\*) AS cnt FROM "{child}" WHERE "{child\_col}" = %s;', (pk\_val,))

rr = cur.fetchone()

cnt = rr["cnt"] **if** isinstance(rr, dict) **else** rr[0]

counts[child] = cnt

**return** counts

**Фрагмент 9 — Видалення по PK**

Опис: delete\_by\_pk — безпечна операція видалення: спочатку перевіряє наявність батька та дочірніх записів (preview\_child\_counts), а потім вже коректно видаляє рядок.

# Видалення по PK

**def** delete\_by\_pk(self, table, pk\_col, pk\_val, force=False):

self.\_validate\_table(table)

counts = self.preview\_child\_counts(table, pk\_col, pk\_val)

**if** counts.get(table, 0) == 0:

**return** None, {}

child\_totals = {t: c **for** t, c **in** counts.items() **if** t != table **and** c > 0}

**if** child\_totals **and** **not** force:

**raise** ChildRowsExistError(child\_totals)

deleted\_counts = {}

**with** self.conn.cursor() **as** cur:

**try**:

cur.execute("BEGIN;")

**if** force:

fks = self.get\_referencing\_fks(table)

**for** fk **in** fks:

child = fk['child\_table']; child\_col = fk['child\_column']

cur.execute(f'DELETE FROM "{child}" WHERE "{child\_col}" = %s;', (pk\_val,))

deleted\_counts[child] = deleted\_counts.get(child, 0) + cur.rowcount

cur.execute(f'DELETE FROM "{table}" WHERE "{pk\_col}" = %s RETURNING \*;', (pk\_val,))

row = cur.fetchone()

deleted\_counts[table] = deleted\_counts.get(table, 0) + (1 **if** row **else** 0)

self.conn.commit()

**return** row, deleted\_counts

**except** Exception:

**try**:

self.conn.rollback()

**except** Exception:

**pass**

**raise**

**Фрагмент 10 — Методи генерації (parents, teacher, subject, student, journal)**

Опис: generate\_\* — використовують set-based SQL (CTE + generate\_series) для швидкої масової генерації тестових даних; generate\_journal робить двофазну операцію (INSERT потім UPDATE) для реалістичного розподілу attendance\_status і grade.

# Генерація батьків

**def** generate\_parents(self, n):

q = """

WITH maxv AS (

SELECT COALESCE(MAX(parents\_id), 0) AS m FROM "parents"

), gens AS (

SELECT m + row\_number() OVER () AS new\_id,

left(md5(random()::text),8) AS fn,

left(md5(random()::text),8) AS ln,

('+380' || (100000000 + floor(random()\*900000000)::bigint)::text) AS phone,

lower(left(md5(random()::text),8) || '@example.com') AS em

FROM maxv, generate\_series(1, %s)

)

INSERT INTO "parents"(parents\_id, first\_name, last\_name, phone, email)

SELECT new\_id, fn, ln, phone, em FROM gens;

"""

**with** self.conn.cursor() **as** cur:

cur.execute(q, (n,))

self.conn.commit()

Аналогічні методи: generate\_teachers, generate\_subjects, generate\_students

Генерація журналу (generate\_journal) — трохи складніша: вставка з підрахунку counts, а потім оновлення attendance\_status і обнулення grade для absent — див. повну реалізацію у пункті “Повний текст програми”

**Фрагмент 11 — Складні запити (complex\_query\_1/2/3)**

Опис: complex\_query\_1/2/3 — готові аналітичні запити для звітності: середній бал по предметах, кількість оцінок по вчителям за період, розподіл відвідуваності по класах. Контролер викликає їх із параметрами, а view виводить результати.

# complex\_query\_1

**def** complex\_query\_1(self, class\_value):

q = """

SELECT sb.name AS subject, COUNT(j.journal\_id) AS marks\_count, AVG(j.grade) AS avg\_grade

FROM "journal" j

JOIN "subject" sb ON j.subject\_id = sb.subject\_id

JOIN "student" s ON j.student\_id = s.student\_id

WHERE s.class = %s

GROUP BY sb.name

ORDER BY avg\_grade DESC;

"""

**with** self.conn.cursor() **as** cur:

cur.execute(q, (class\_value,))

**return** cur.fetchall()

# complex\_query\_2

**def** complex\_query\_2(self, date\_from, date\_to):

q = """

SELECT t.first\_name || ' ' || t.last\_name AS teacher, COUNT(j.journal\_id) AS marks\_count

FROM "journal" j

JOIN "teacher" t ON j.teacher\_id = t.teacher\_id

WHERE j.entry\_date BETWEEN %s AND %s

GROUP BY teacher ORDER BY marks\_count DESC LIMIT 50;

"""

**with** self.conn.cursor() **as** cur:

cur.execute(q, (date\_from, date\_to))

**return** cur.fetchall()

# complex\_query\_3

**def** complex\_query\_3(self, subject\_name):

q = """

SELECT s.class, j.attendance\_status, COUNT(\*) AS cnt

FROM "journal" j

JOIN "student" s ON j.student\_id = s.student\_id

JOIN "subject" sb ON j.subject\_id = sb.subject\_id

WHERE sb.name = %s

GROUP BY s.class, j.attendance\_status

ORDER BY s.class;

"""

**with** self.conn.cursor() **as** cur:

cur.execute(q, (subject\_name,))

**return** cur.fetchall()

Повний текст програми

Програмний код проєкту (модулі model.py, controller.py, view.py та main.py) має значний обсяг і через зручність перегляду в звіті не наводиться повністю. Повна версія коду доступна в репозиторії на GitHub: <https://github.com/snizhana-ponamarova/databases>

Висновки

У ході виконання роботи реалізовано консольний додаток для роботи з реляційною базою даних PostgreSQL за шаблоном MVC. Модуль Model забезпечує повний набір операцій CRUD (вставка, читання, оновлення, видалення), включно з валідацією вхідних даних та перевірками цілісності (перевірка існування FK, правила для полів grade/attendance\_status). Контролер і view забезпечують зручний інтерфейс для введення параметрів і відображення результатів.

Реалізовано механізми масової генерації тестових даних (generate\_\*) з використанням наборних SQL-операцій (CTE + generate\_series), що забезпечує високу швидкість наповнення таблиць. Проведено контрольний тест генерації 100000 записів та аналіз результатів: операція була виконана успішно, результати (кількість рядків, часи виконання, спостереження по NULL/грейдам) занесено у звіт. Також додано набір аналітичних запитів (complex\_query\_1/2/3) для отримання корисних зведень (середній бал по предметах, навантаження вчителів, розподіл відвідуваності).

Підсумовуючи: реалізований функціонал відповідає вимогам розграхунково-графічної роботи— забезпечує роботу з БД, зберігає референційну цілісність, дає можливість генерації й аналізу даних і надає очевидні повідомлення та захисти для користувача.

Контакти:

Репозиторій Git: <https://github.com/snizhana-ponamarova/databases>