МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ВОЛИНСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ЛЕСІ УКРАЇНКИ
Кафедра комп'ютерних наук та кібербезпеки

На правах рукопису

КУРОТИЧ АНАТОЛІЙ ОЛЕКСАНДРОВИЧ

РОЗРОБКА СИСТЕМИ АВТОМАТИЗОВАНОГО ГЕНЕРУВАННЯ ER-ДІАГРАМ ДЛЯ РЕЛЯЦІЙНИХ БАЗ ДАНИХ

Спеціальність: 122 Комп'ютерні науки
Освітньо-професійна програма: Комп'ютерні науки та інформаційні технології
Робота на здобуття освітнього ступеня «магістр»

Науковий керівник:
Булатецька Леся Віталіївна,
кандидат фізико-математичних наук, доцент
кафедри комп'ютерних наук та кібербезпеки

РЕКОМЕНДОВАНО ДО ЗАХИСТУ
Протокол №_____
засідання кафедри комп'ютерних наук
та кібербезпеки
від _____ 2025 р.
Завідувач кафедри

(_____)___Гришанович Т. О.

ЛУЦЬК – 2025

ЗМІСТ

ВСТУП

Актуальність теми. Розробка та підтримка програмного забезпечення (ПЗ), що використовує реляційні бази даних, залишається одним із ключових завдань для інженерів вже понад 40 років – із часу появи перших комерційних реляційних систем управління базами даних (СУБД). Однією з основних тенденцій розвитку цієї галузі є постійне зростання складності структури баз даних, що відбувається паралельно з ускладненням ПЗ загалом. Це зумовлює потребу в нових інструментах для їх аналізу, проєктування та візуалізації.

Ключовим елементом таких інструментів є модель «сутність-зв'язок» (ER-модель), яка забезпечує графічне представлення структури бази даних, включаючи таблиці, поля та зв'язки між ними, і є де-факто стандартом у галузі розробки ПЗ. Процес створення ER-діаграм є важливою складовою частиною проєктування та супроводу сучасних інформаційних систем.

Незважаючи на широке використання ER-діаграм серед інженерів, процес їх створення часто відбувається в ручному режимі. Даний підхід схильний до помилок ймовірність яких зростає разом зі збільшенням складності структур. У комплексних динамічних системах ручне створення діаграм втрачає доцільність оскільки це потребує значних часових витрат та часто не відображає актуального стану системи.

Для полегшення створення діаграм існують інструменти, що реалізують підхід «diagrams as code», де діаграми формуються на основі текстового опису (domain specific language). Такі інструменти мають значний потенціал для автоматизації який було описано в статі «Optimizing the process of ER diagram creation with PlantUML» [1]. Серед найпоширеніших можна виділити PlantUML [2] та Mermaid [3].

Метою роботи є розробка програмного рішення автоматичної генерації ER-діаграм для реляційних баз даних на основі рядка підключення (connection string).

Для досягнення мети потрібно виконати наступні завдання:

- провести аналіз наявних рішень для генерації діаграм із визначенням їх переваг, недоліків та обмежень;
- визначити релевантні технології (СУБД, мова програмування), та бібліотеки для підключення до бази даних, створення текстових шаблонів;
- розробити архітектуру ПЗ з акцентом на модульність і розширюваність, що забезпечить підтримку різних типів СУБД та форматів текстових описів діаграм;
- реалізувати прототип з підтримкою однієї СУБД та двох форматів текстових описів (PlantUML, Mermaid);
- протестувати рішення на реальних базах даних.

Об'єкт дослідження: процес дизайну та візуалізації структури реляційних баз даних.

Предмет дослідження: технології та архітектурні підходи реалізації програмного засобу для автоматичної побудови ER-діаграм.

Публікації:

1. A. O. Kurotych, L. V. Bulatetska, Optimizing the process of ER diagram creation with PlantUML, CEUR Workshop Proceedings (2025) 47–57. URL: https://ceur-ws.org/Vol-3917/paper12.pdf
2. Куротич, А., Булатецька, Л., & Онищук, О. Відновлення зв'язків між сутностями в схемах баз даних за допомогою PlantUML та LLM. Електронне фахове наукове видання «*Кібербезпека: освіта, наука, техніка*». 2025. 1(29), 152–160. https://doi.org/10.28925/2663-4023.2025.29.847
3. Куротич А., Булатецька Л. Дослідження можливостей plantuml для покращення побудови ER-діаграм *Проблеми комп'ютерних наук, програмного моделювання та безпеки цифрових систем*.: матеріали II міжнар. науково-практ.конф. Луцьк, 9–11 червн. 2025 р. Луцьк, 2025. С. 188-189. URL: https://apcssm.vnu.edu.ua/index.php/conf/article/view/161

РОЗДІЛ І
МЕТОДОЛОГІЯ ТА ІНСТРУМЕНТИ ПОБУДОВИ ER-ДІАГРАМ ДЛЯ РЕЛЯЦІЙНИХ БАЗ ДАНИХ

1.1. Особливості реляційних баз даних

Реляційні бази даних спроєктовані на основі реляційної моделі запропонованої у науковій роботі «Реляційна модель даних для великих спільних сховищ даних» британським вченим Едгаром Коддом у 1970 році [4]. У цій моделі дані організовані у вигляді таблиць та зв'язків між ними. Кожен рядок є окремим записом, а кожен стовпець представляє окремий атрибут.

Для ідентифікації унікальних записів на рівні таблиць в таких базах даних використовуються первинні ключі (ПК, PK, Primary Key), які зазвичай являють собою значення одного поля. Найпоширенішими типами даних, що використовуються в якості ПК є ціле число (Integer) та uuid (Universally Unique Identifier). ПК, які складаються більш ніж з одного поля називаються композитними первинними ключами – сукупність кількох полів які разом утворюють унікальний ідентифікатор.

Зовнішні ключі (FK, Foreign keys) використовуються для реалізації зв'язку між таблицями. Значення у стовпці дочірньої таблиці (foreign key) обов'язково має відповідати наявному значенню у стовпці з первинним ключем (Primary Key) батьківської таблиці [5-8]. Загальноприйняті такі типи зв'язків: один до одного, один до багатьох, багато до багатьох.

Один до одного (One-to-one) – кожен запис в таблиці А відповідає тільки одному запису в таблиці Б. Найчастіше це являється зв'язком типу: «Один до нуля або одного», адже чиста реалізація відношення «один до одного» у реляційних базах даних часто неможлива або є значно ускладненою. Приклад такого зв'язку: Автомобіль – Технічний паспорт.

Один до багатьох (One-to-many) – кожен запис в таблиці А може відповідати кільком записам в таблиці Б, але кожен запис в таблиці Б відповідає

лише одному запису в таблиці А. Наприклад: Категорія – Товар, кожна категорія містить багато товарів, але товар завжди належить до однієї категорії.

Багато до багатьох (Many-to-many) — кілька записів у таблиці А може відповідати декільком записам в таблиці Б. Наприклад: Актори – Фільми. Особливістю такого типу зв'язку в реляційних базах даних є те, що для його реалізації потрібно створити третю (проміжну) таблицю.

1.2. Сутність-зв'язок (Entity-Relationship, ER) модель: роль, складові та застосування

ER-модель була запропонована у науковій публікації «The entity-relationship model – toward a unified view of data» тайвансько-американським ученим Пітером Ченом у 1976 році [9]. В анотації своєї праці П. Чен зазначає, що запроваджено спеціальний діаграмний метод як інструмент для проєктування баз даних, та що модель «Сутність-зв'язок» може бути використана як основа для уніфікації різних підходів до подання даних: мережевої моделі, реляційної моделі та моделі множин сутностей (*«A special diagrammatic technique is introduced as a tool for database design. ... The entity-relationship model can be used as a basis for unification of different views of data: the network model, the relational model, and the entity set model»*) [9]. Наведений опис можна розглядати як повне та вичерпне визначення ER-моделі.

1.2.1. Компоненти ER-моделі

До базових компонентів ER-моделі належать сутності, атрибути та зв'язки. Сутність (Entity) – об'єкт або концепт, який являється важливою складовою системи. Наприклад: продукт, клієнт. На діаграмі зображається як прямокутник. Сутність може бути сильною (існує самостійно) або слабкою (не може існувати без іншої сутності). Атрибут (Attribute) – характеристика сутності. Одна сутність може мати безліч атрибутів. Наприклад: клієнт може мати наступні характеристики: Ім'я, прізвище, дата народження. Зв'язок (Relationship) – зв'язок

між сутностями. Зв’язок має наступні характеристики: кардинальність (показує максимальну кількість екземплярів сутності) та модальність (показує чи є зв’язок між сутностями обов’язковим). Кожен зв’язок має дві пари значень кардинальність і модальність, одну для кожного боку [5-8].

1.2.2. Рівні абстракції на прикладі роботи з базами даних

Рівні абстракцій описують, наскільки детально представлено модель даних [8]. Є три основні рівні (табл.1.).

Таблиця 1.1

Порівняння рівнів абстракцій

Характ.	Концептуальний	Логічний	Фізичний
Мета	Продемонструвати загальну модель предметної області без технічних деталей	Описати детальну структуру даних без прив’язки до конкретної СУБД	Повністю готова модель для реалізації в конкретній СУБД
Цільові користувачі	Архітектори ПЗ, бізнес-аналітики, замовники	Архітектори ПЗ, розробники	Розробники, тестувальники
Атрибути	немає	назви та ключі	Точні типи даних та обмеження
Зв’ язки	Показані в загальному виду. Без кардинальності та модальності	Точний тип зв’язків з кардинальністю та модальністю.	Точний тип зв’язків з кардинальністю та модальністю.
Типи даних	відсутні	Узагальнені (Integer, String)	Конкретні (bigint, varchar(255))
Обмеження	відсутні	Загальні (PK,FK)	Конкретні (NOT NULL, CHECK)

1. Концептуальний – найвищий рівень абстракції, використовується щоб показати загальну структуру та зміст предметної області. Містить сутності (назви таблиць) та зв’язки без вказання їх кардинальності та модальності, не містить атрибутів.

2. Логічний – детально описує структуру даних, але без конкретної реалізації. Є середнім рівнем абстракції, на відміну від концептуального рівня включає атрибути, первинні та зовнішні ключі, але без конкретної реалізації (під певну СУБД).

3. Фізичний – описує структуру готову для створення в конкретній СУБД з вказаними конкретними типами даних, обмеженнями, індексами, тригерами.

1.2.3. Нотації

Нотація в даному контексті – це спосіб візуального представлення ER-моделей. Нотації різняться між собою набором символів, фігур та правил які використовуються для візуалізації ER-моделі (створення ER-діаграми) [9-11]. Таблиця 1.2 містить найпопулярніші види нотацій та їх головні відмінності.

Таблиця 1.2

Популярні нотації та їх особливості

Нотація	**Відмінності**
Chen Notation	- сутності зображають у вигляді прямокутників з назвою (сутності) всередині; - атрибути у вигляді овалів; - зв'язки – у вигляді ліній та ромбів між сутності; - модальність в оригінальній праці відсутня;
Crow's Foot Notation (Information Engineering)	- сутності зображають у вигляді прямокутників з атрибутами всередині; - кардинальність та модальність зв'язків у форматі «Вороняча лапка»; - модальність зображають завдяки символу «|» або «O»;
Barker's notation	- мутності у вигляді прямокутників - атрибути групуються за допомогою наступних символів: «#» - унікальне поле, «*» - обов'язкове поле, «o» - опціональне поле - модальність зображають завдяки пунктирній або суцільній лінії

На рис. 1.1 зображена класична модель «Клієнт – замовлення», де чітко видно особливості кожної нотації зазначеної в табл. 1.2.

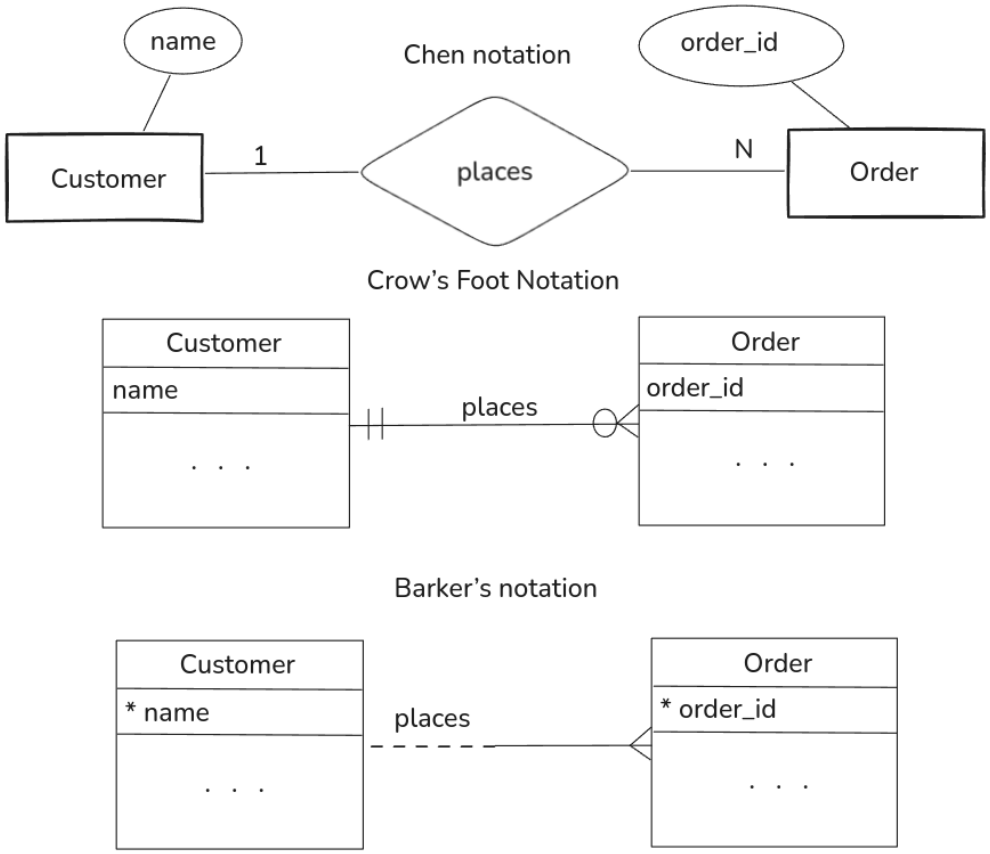

Рисунок 1.1. – Модель «Клієнт - замовлення»

1.3. Підходи до генерації та роботи з ER-діаграмами

Слід виділити два основні підходи до генерації діаграм: «Діаграма як код» (Diagram as Code) та «Ручне візуальне редагування» (Manual Visual Editing).

«Діаграма як код» – це підхід при якому візуальне представлення діаграми генерується на основі спеціального тексту Domain-Specific language (DSL). При реалізації підходу «Ручне візуальне редагування» ПЗ зазвичай не являється спеціалізованим для побудови конкретного типу діаграми, а побудова самої діаграми відбувається за допомогою графічних інструментів. [2-3], [12-15].

1.3.1. Огляд інструментів «Діаграма як код»

Серед даних інструментів можна виділити два найпопулярніші у своєму сегменті: PlantUML та Mermaid. PlantUML реалізовано на основі класичної моделі клієнт-сервер. Клієнт має вигляд розділеної веб сторінки. Де з одної сторони користувач вводить текстовий опис діаграми, а з другої бачить результат. Серверна частина відповідає за валідацію та перетворення текстового опису в візуальний (png або svg формат). Mermaid в свою чергу має лише клієнтську частину яка відповідає за весь процес.

Також слід зазначити що всі вони являються ПЗ з відкритим кодом який можна найти на ресурсі github.com. Що сприяє наявності підтримки від спільноти розробників, та надає можливість їх дослідження. [2-3]

PlantUML. Пропонує найширші можливості серед аналогів. Має велику кількість документації структурованої у вигляді книги в pdf форматі [2], але у зв'язку зі значною кількістю можливостей, документація не є вичерпною і не здатна покрити всі аспекти. З унікальних особливостей можна виділити наступні.

Створення функцій та процедур які зручно використовувати:

- для створення повторюваних елементів діаграм (однакові блоки, ноти, стрілки);
- для автоматизації створення однотипних елементів;
- в якості ключових елементів для створення модулів.

Механізм підключення зовнішніх файлів з кодом (модулів). Власні модулі доцільно створювати, коли:

- однаковий код повторюється в кількох діаграмах (наприклад, одні й ті самі стилі, кольори, форми);
- є набір користувацьких функцій/процедур для автоматизації;
- потрібно структурувати великий проєкт із багатьох діаграм;
- кілька людей працюють над одними діаграмами (щоб ділитись спільним кодом).

Підтримка інтеграції користувацьких графічних ресурсів під час генерації діаграм. PlantUML дозволяє додавати зовнішні графічні ресурси (іконки,

логотипи, зображення) до діаграм. Це не є обов'язковою частиною UML, але дуже корисна можливість для створення візуально виразніших і зрозуміліших схем, особливо у великих проєктах. Інтеграція користувацьких зображень корисна, коли потрібно:

- використати логотипи компаній чи продуктів – для діаграм архітектури або інтеграції систем;
- візуально відрізнити різні типи компонентів – наприклад, сервери, бази даних, API, мобільні додатки;
- покращити сприйняття інформації – зрозумілі іконки полегшують читання діаграми нетехнічними учасниками;
- налаштувати фірмовий стиль – у корпоративній документації, презентаціях, дипломах тощо.

Серед недоліків можна виділити найменш читабельний синтаксис. Тому аналіз текстового опису діаграми вимагає найбільше зусиль. На рисунку 1.2 зображена робота з PlantUML клієнтом, де зверху текстовий опис, а знизу згенерована діаграма.

Mermaid. Відносно новий інструмент, створений у 2014 році, головною відмінністю якого є відсутність серверної частини. Через те що вся логіка реалізована на стороні клієнта, являється найкращим вибором для тих кому важлива в першу чергу безпека даних. Mermaid [3] працює повністю в браузері завдяки JavaScript-рушію, що спрощує розгортання: не потрібно ставити додаткові сервери чи плагіни. Це робить Mermaid зручним для інтеграції у статичні сайти, наприклад GitHub [16]. Відсутність серверної частини – не тільки плюс, а й обмеження: складні діаграми чи конфігуруванні стилі в деяких випадках важче реалізувати, оскільки вся логіка обмежена можливостями браузера. На рисунку 1.3 зображена робота з Mermaid клієнтом, де ліворуч текстовий опис, а праворуч згенерована діаграма.

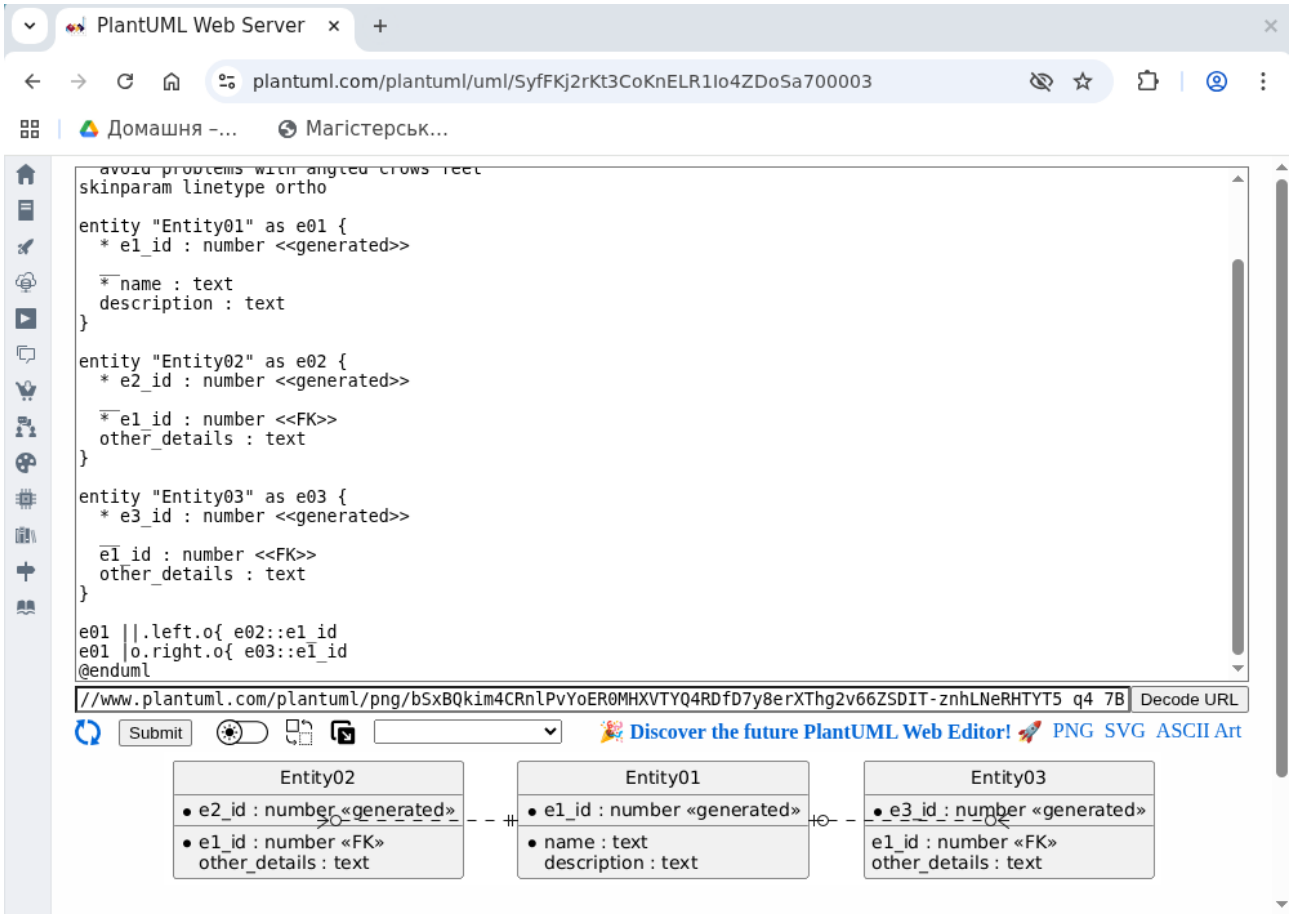

Рисунок 1.2. – Робота з веб клієнтом PlantUML

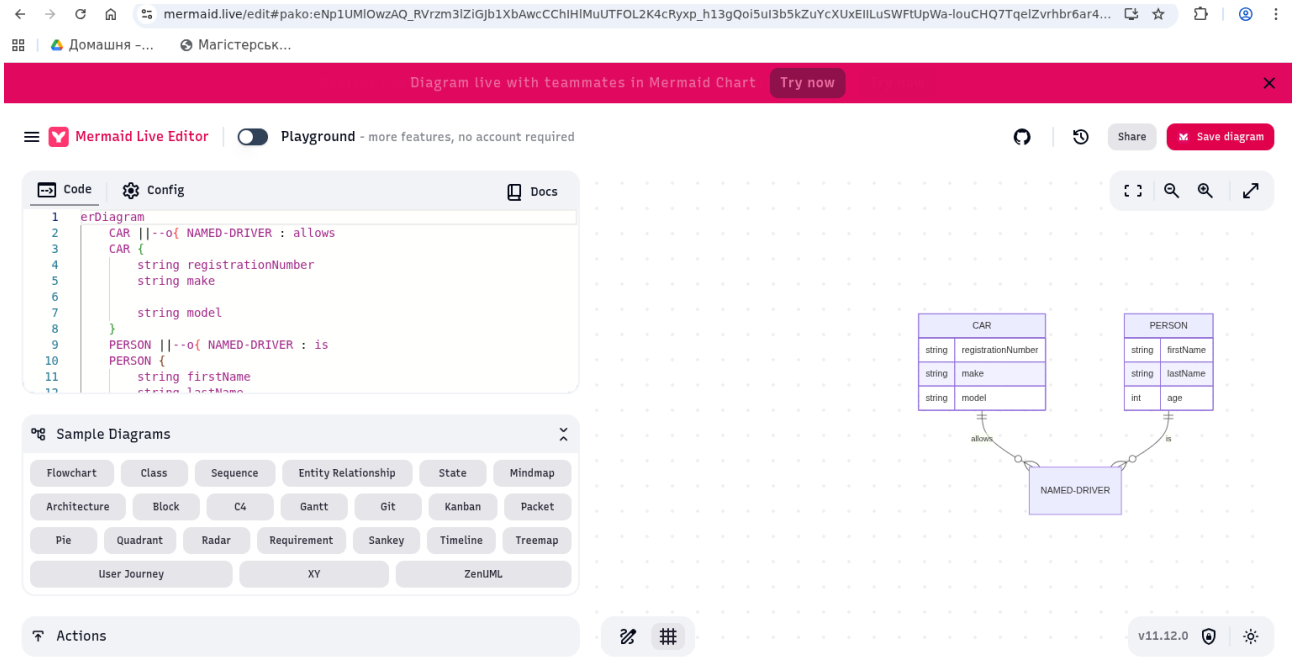

Рисунок 1.3. – Робота з веб клієнтом Mermaid

Саме легкість інтеграції з web сайтами зіграло рішучу роль в набранні популярності mermaid. На рисунку 1.4 зображено mermaid діаграму, текстовий опис якої знаходиться в файлі README.txt який розробники використовують як вхідну точку для роботи з їх репозиторіями та який відображається на головній сторінці репозиторію користувача на ресурсі Github.

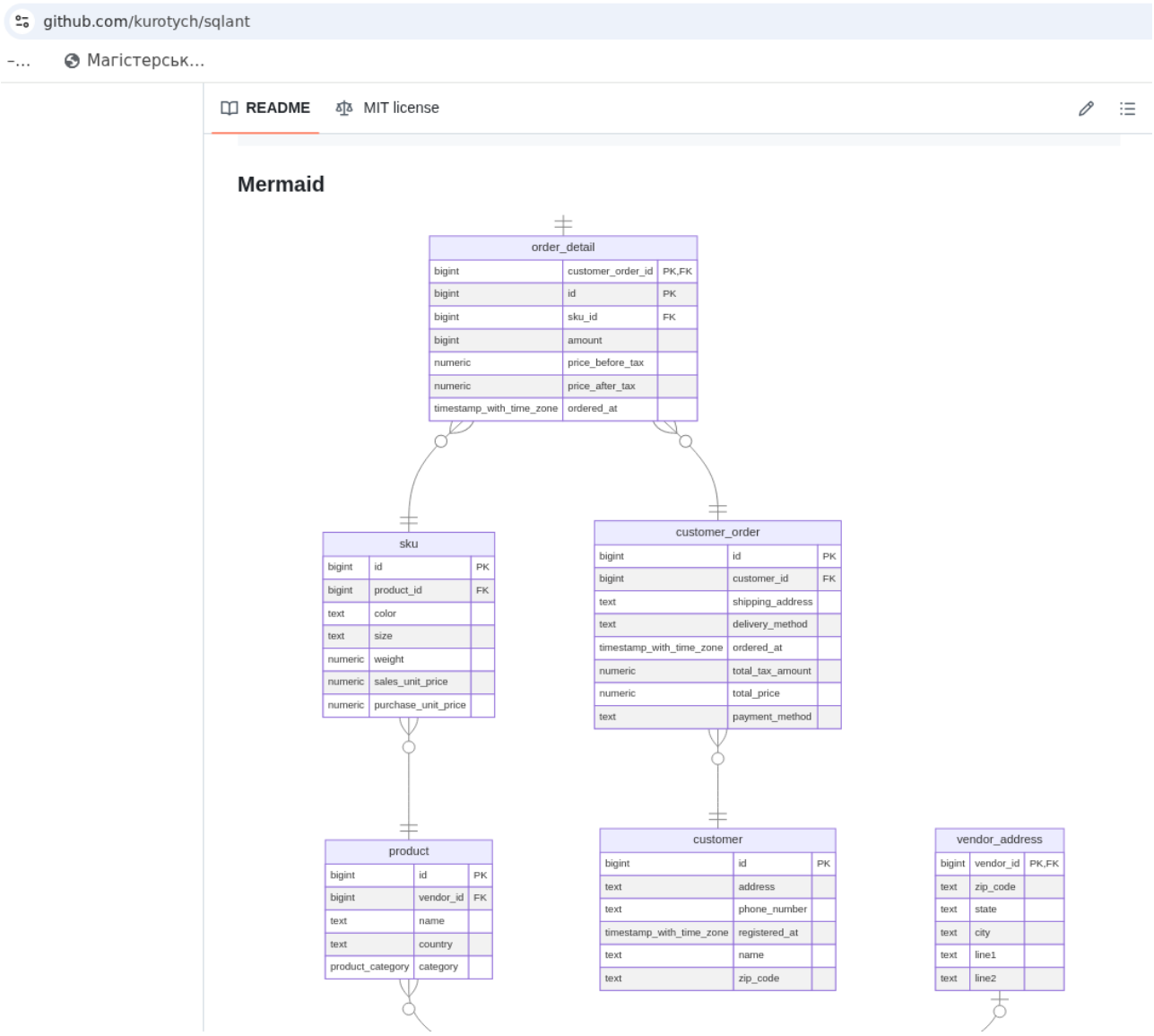

Рисунок 1.4. – Інтеграція Mermaid з Github

Таблиця 1.3 містить порівняння Mermaid та PlantUML по критеріям за допомогою яких показано ключову різницю між інструментами.

Таблиця 1.3

Порівняння Mermaid та PlantUML

Критерій	**PlantUML**	**Mermaid**
Документація	Велика кількість документації, у вигляді pdf-книги, але через багатофункціональність не охоплює всі аспекти	Документація більш компактна, легша для освоєння
Функції та процедури	Підтримуються – для повторюваних елементів, автоматизації та створення модулів	Відсутні, акцент на простому синтаксисі
Бібліотеки та підключення зовнішніх файлів	Є можливість створювати власні бібліотеки та підключати код з інших файлів	Немає
Графічні ресурси	Можна додавати власні іконки, логотипи, зображення	Інтеграція обмежена стандартними стилями браузера
Архітектура	Має серверну частину (можна запускати локально чи віддалено)	Працює без серверної частини – повністю в браузері
Безпека	Потребує серверної обробки (можливі ризики для приватних даних)	Уся логіка на клієнті – висока безпека даних
Гнучкість стилів	Дуже гнучкий, всі елементи піддаються конфігурації	Конфігурація стилів обмежена можливостями браузера

1.3.2. Огляд інструментів з ручним візуальним редагуванням

Ручне візуальне редагування – це підхід при якому діаграми створюються за допомогою лише графічних інструментів. У цьому випадку користувач працює безпосередньо з візуальними елементами (блоками, стрілками, і т.д.) – та самостійно розміщує їх на площині діаграми. Серед найпоширеніших ресурсів які реалізують даний підхід можна виділити: diagrams.net [14] , lucidchart [15], Microsoft Visio [17]. Такий підхід має низку особливостей:

- інтуїтивність, коли користувач відразу бачить кінцевий результат, що є зручним для початківців;

- гнучкість, оскільки можна створити унікальний вигляд діаграми, використовуючи різноманітні інструменти форматування, кольори та стилі;
- швидкий старт, адже немає потреби вивчати спеціальний синтаксис чи мови опису діаграм.

Разом з тим існують і недоліки:

- важко підтримувати, оскільки зміна структури або внесення правок у великі проєкти може займати багато часу;
- мала повторюваність, адже під час створення кількох схожих діаграм доводиться редагувати кожну вручну;
- обмежена автоматизація, бо неможливо застосувати функції повторного використання коду чи генерування діаграм із текстового опису.

На рисунку 1.5 зображений інтерфейс для роботи з diagrams.net [14]. Ліворуч находяться готові форми які користувач може використати для побудови діаграми, по центрі екран для відображення діаграм і праворуч додаткові налаштування, наприклад: розмір шрифту.

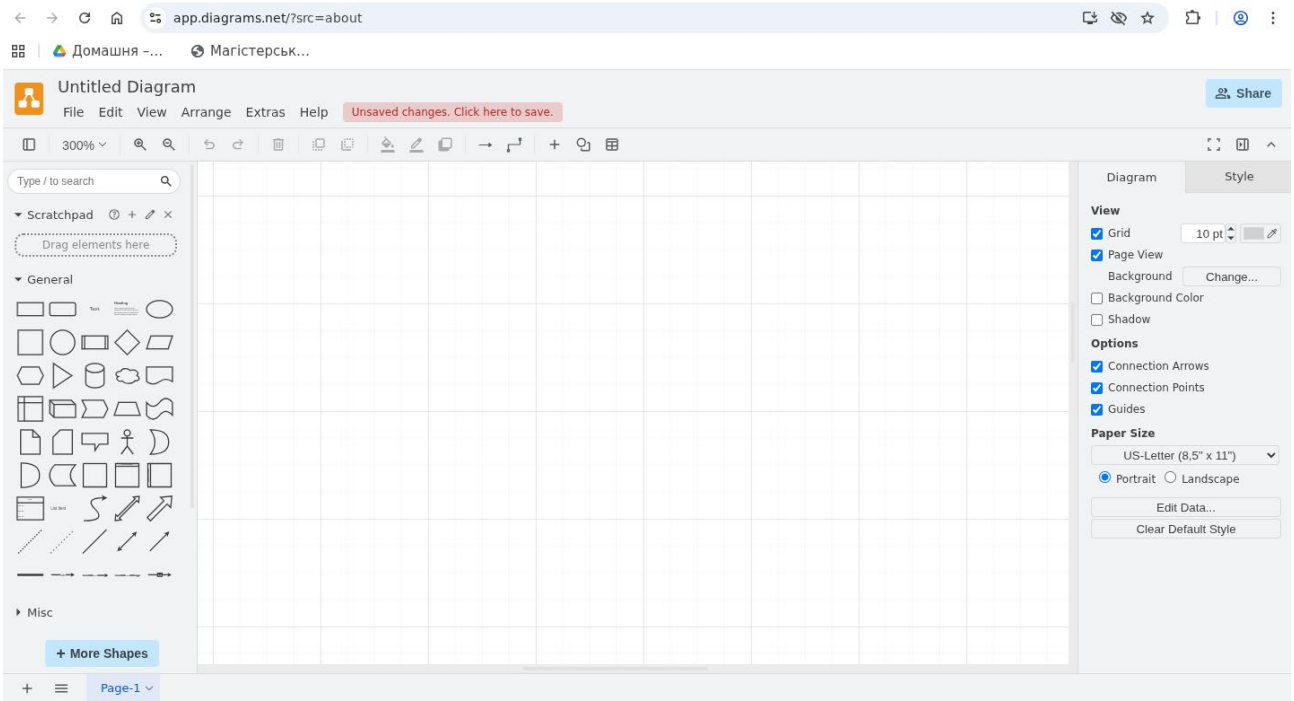

Рисунок 1.5 – Інтерфейс для роботи з diagrams.net

1.3.3. Порівняння підходів «Діаграма як код» та «Ручне візуальне редагування»

Спираючись на інформацію подану в попередніх розділах створена таблиця 1.4 яка містить порівняння підходів.

Таблиця 1.4

Порівняння підходів «Діаграма як код» та «ручне візуальне редагування»

Критерій	**Діаграма як код**	**Ручне візуальне редагування**
Складність використання	Вимагає знання синтаксису DSL	Зазвичай інтуїтивний «drag-and-drop» інтерфейс
Масштабова-ність	Простіша підтримка діаграм оскільки редагування вимагає зміни лише текстового опису.	Важче підтримувати (редагувати) великі діаграми.
Автоматизація	Можливість використання модулів, фукнцій, шаблонів. Використання спеціального ПЗ (наведеного в пункті 1.7 «Аналіз існуючих рішень») для автоматичної генерації та використання в CI/CD процесах.	Зазвичай відсутня, все редагується в ручну.
Гнучкість стилів	Кастомізація залежить від ПЗ	Кастомізація залежить від ПЗ, але зазвичай більше можливостей.
Швидкість створення діаграм	Довший початковий процес (вивчення специфічного синатксису), але швидше створення діаграм в подальшому.	Легко створити перший варіант без спеціальних знань, але повільно масштабувати.
Співпраця	Підтримка контролю версій (git), текстові файли можна редугвати і проводити процес перевірки коректності без наявності спеціального ПЗ.	Спільна робота лише через онлайн редактори.
Сфера застосування	Технічна документація.	Навчання, презентації, діаграми які не потребують постійної підтримки.
Інтеграція з LLM (Large Language Model)	Дуже добре інтегрується оскільки LLM чудово розуміє текстові патерни.	Мінімальна, оскільки штучний інтелект не вміє змінювати діаграми в візуальному форматі.

В якості доповнення до критерію «Інтеграція з LLM» слід звернути уваги на роботу «RECONSTRUCTING ENTITY RELATIONSHIPS IN DATABASE SCHEMAS WITH PLANTUML AND LLMS.» [18], яка описує як за допомогою діаграми описаної в форматі PlantUML, LLM Claude вдало відновлює втрачені зв'язки між таблицями.

Підсумовуючи, можна зазначити, що підхід з ручним візуальним редагування краще підходить для невеликих задач, таких як, навчання та візуалізації для нетехнічних користувачів. Тоді як «Діаграма як код» підходить у випадках коли потрібно забезпечити автоматизацію, довготривалу підтримку та можливість інтеграції з LLM.

1.4. Отримання метаданих в реляційних баз даних (на прикладі PostgreSQL)

PostgreSQL – це СУБД з відкритим кодом головними особливостями якої є надійність та наявність широких можливостей відсутніх в аналогів. PostgreSQL має вбудовану підтримку JSON та XML форматів та надає можливість користувачу створювати власні типи даних. Має широке застосування при розробці вебдодатків та аналізу даних. [19-20]

Кожна база даних в PosgreSQL може мати одну чи більше схем (schemas) [21], в схемах в свою чергу зберігаються як таблиці, так і інші типи об'єктів, такі як: типи даних, функції, оператори (у PosgreSQL можна навіть створювати власні оператори). В одній схемі два об'єкти не можуть мати однакове ім'я. По замовчуванню коли користувач створює базу даних, створюються службові схеми, та схема з назвою «public» де зберігатимуться об'єкти створені користувачем.

Для того щоб провести огляд можливості отримання метаданих реляційних баз даних на прикладі PostgreSQL, також слід дати визначення терміну «метадані реляційних баз даних».

Метадані реляційних баз даних – це дані про самі дані, тобто опис структури та організації бази, а не її вміст. До метаданих в контексті реляційних баз даних належать:

- таблиці: назви, простори імен (схеми);
- стовпці: імена, типи даних, обмеження NOT NULL, значення за замовчуванням;
- ключі: первинні (PRIMARY KEY), зовнішні (FOREIGN KEY);
- індекси: інформація про побудовані індекси для пришвидшення пошуку в тому числі унікальні індекси;
- представлення (views): збережені SQL-запити, які поводяться як таблиці;

Даний перелік не являється вичерпним і може варіюватись залежно від типу СУБД).

В PostgreSQL ці метадані зберігаються у наступних службових схемах:

- information_schema – стандартизований набір подань (SQL-стандарт), що описує структуру БД;
- pg_catalog – системний каталог PostgreSQL із розширеною інформацією (наприклад, про індекси, специфічні типи даних тощо).

1.5. Роль ER діаграм в життєвому циклі проєктування баз даних

В життєвому циклі проєктування баз даних (DBLC, Database life cycle) зазвичай описують наступні етапи:

1. початкове дослідження предметної області;
2. проєктування бази даних;
3. реалізація та завантаження;
4. тестування та оцінювання;
5. експлуатація;
6. супровід та розвиток.

ER діаграми використовуються продовж 2-6 етапів включно. На етапі 1 ER діаграми не створюють, оскільки цей етап відповідає за початкове дослідження

на якому проводиться збір вимог. Етап 2 являється ключовим в використанні ER діаграм, саме тут відбувається створення сутностей, атрибутів, зв'язків тощо. Для цього використовують концептуальну та логічну модель. На етапі 3 готова логічна модель являється довідковим матеріалом для розробників, які створюють фізичну модель та вже безпосередньо реалізовують запити створення сутностей за допомогою DDL (Data Definition Language). Порядок створення фізичної моделі та DDL запитів може варіюватись, в пункті 1.7 описані рішення які можуть на основі рядка підключення до баз даних згенерувати фізичну модель. За допомогою яких спочатку можуть бути створені та виконані DDL запити, а потім автоматично згенеровані діаграми. На етапі 4 логічна та фізичні моделі являються як контрольний інструмент, перевіряється на скільки реалізація відповідає задуму проєктувальника. Перевіряється чи всі зв'язкі були реалізовані, чи немає зайвих або пропущених атрибутів. На етапі 5 ER діаграми можуть використовуватись як частина документації. Етап 6, при зміні вимог ERD оновлюють разом зі змінами в БД, що допомагає узгодженості між членами команди.

ERD є ключовим елементом на етапі проєктування, але також являється частиною інших етапів.

1.6. CLI-додатки: еволюція, роль та особливості використання

CLI (Command Line Interface) або консольна утиліта – це тип програмних рішень для взаємодії з комп'ютером за допомогою текстових команд які користувач виконує в терміналі. Почав набувати популярності в 1960-70-х роках. В 1960-х відбулась поява перших терміналів, а в 1970-х набули популярності UNIX системи де командний рядок став основним способом взаємодії з операційною системою. У свою чергу, UNIX системи дали значний поштовх масовому поширенню персональних комп'ютерів. На початку 1990-х років з'явились перші операційні системи з графічними інтерфейсами (Windows 3.x, Macintosh), які почали поступово витісняти CLI додатки з побутових ПК. У 2000-

х роках завдяки популяризації використання операційної системи GNU/Linux на серверах, яка була розроблена Лінусом Торвальдсом (ядро) та Річардом Столменом (програмна оболонка), CLI пережив нову хвилю розвитку.

Особливістю CLI є відсутність графічного інтерфейсу (GUI), що може бути негативно сприйнято недосвідченими користувачами, однак текстовий інтерфейс є одним з ключових моментів, що робить цей тип застосунків зручним для індивідуального налаштування, автоматизації процесів і інтеграції у сценарії командного рядка.

Другим ключовим моментом є підтримка механізму «конвеєр» (pipe), який дозволяє результат однієї команди (утиліти) передавати на вхід другій, що робить можливим комбінацію функціональності декількох незалежних програм без створення допоміжних файлів чи інших механізмів комунікації. В Unix-подібних системах оператор «конвеєр» позначається як вертикальна лінія «|». Наприклад: наступна комбінація «ls | grep ".txt"», є результатом використання двох програм ls та grep, де результатом буде вивід лише тих файлів в директорії які закінчуються на «.txt»

Наступним важливим елементом який сприяє автоматизації є «Аліас» (Alias). Аліас – це спосіб запису (часто використовуваних) довгих команд, або комбінацій в короткому вигляді. Наприклад щоб створити аліас «lst» який виконуватиме ту ж дію що попередня комбінація потрібно до shell оболонки додати наступний запис: «alias lst='ls | grep ".txt"'». В емпіричному дослідженні «An empirical investigation of command-line customization» [32] було розглянуто понад два мільйони аліасів та виділені найпопулярніші в відкритому доступі на ресурсі «Github», що вказує нам на їх розповсюджене використання.

Отже, CLI-додатки проходять свою еволюцію від початку появи обчислювальної техніки до найновіших розробок в області штучного інтелекту, таких як claude code [33]. Вони залишаються ключовим вибором серед серверних адміністраторів та розробників ПЗ займаючи основну частину в CI/CD процесах, адміністрації серверів, аналізі даних, в інструментах розробки (компілятори, системи управління версіями, засоби контейнеризації тощо). Мають значний

потенціал для подальшого розвитку забезпечуючи основу для систем керування, збірки та розгортання програмного забезпечення.

1.7. Аналіз існуючих рішень

Planter. Консольна утиліта (CLI) з відкритим кодом та ліцензією MIT [22-23], написана за допомоги мови програмування GoLang [24]. Генерує ER діаграми у форматі PlantUML для PostgreSQL баз даних приймаючи на вхід рядок підключення. Перша версія (0.1) була випущена у 2017 році, що було реакцією на включення підтримки ER діаграм до PlantUML. Підтримує одну базу даних та один формат діаграм, а саме PostgreSQL та PlantUML відповідно. Має ряд недоліків з яких можна виділити наступні:

- потребує встановленого golang компілятора для збірки та запуску (немає готових бінарних релізів);
- обмежена підтримка баз даних та нотацій (по одній) з відсутньою (зручною) можливістю додати підтримку нової без значного втручання в наявну базу коду;
- відсутність активної підтримки розробником (на момент написання даної роботи останні зміни в програмний продукт вносились рік назад);
- відсутність додаткових опцій (конфігурації вигляду діаграм, підтримки інших нотацій, і т.д.);
- відсутня можливість генерації діаграм концептуального рівня.

Plant_erd. Plant_erd це також, як попереднє рішення, консольна утиліта (CLI) з відкритим кодом написана за допомоги мови програмування GoLang. Plant_erd підтримує два типи діаграм: PlantUML та Mermaid та чотири СУБД: Sqlite3, MySQL, PostgreSQL. Oracle. [25]

Розповсюджується в якості скомпільованих бінарних файлів завантажених на сервіс github, або коду за допомогою якого користувач може скомпілювати рішення самостійно. З недоліків можна виділити:

- відсутність підтримки матеріалізованих представлень (materialized views) для PostgreSQL;
- відсутність будь-якої стилізації та спроб покращення вигляду діаграм;
- відсутня підтримка визначення обмежень (NOT NULL, Unique, etc);
- відсутня підтримка типу даних Enum (з вище перелічених СУБД enum підтримується лише в PostgreSQL);
- відсутня можливість генерації діаграм концептуального рівня.

РОЗДІЛ ІІ
ПРОЄКТУВАННЯ ТА РОЗРОБКА CLI ДОДАТКУ «SQLANT» ДЛЯ АВТОМАТИЗАЦІЇ ГЕНЕРАЦІЇ ER ДІАГРАМ

2.1. Постановка задачі, призначення та вимоги до програмного засобу

Інженери ПЗ та решта ІТ спеціалістів (дата аналітиків, системних адміністраторів, бізнес-аналітиків тощо), що взаємодіють з базами даних часто потребують наявності якісних діаграм, що відображаються актуальний стан бд проєкту над яким вони працюють. Актуальність стану (відповідність діаграми та структури) в середніх і великих проєктах досягається виключно за допомогою автоматизації. В підсумку до розділу 1.3 було зазначено що автоматизації краще підлягає підхід «Діаграма як код». У зв’язку з цим прийнято рішення розробити програмний продукт який сприятиме подальшому розвитку автоматизації генерації ER діаграм використовуючи вище зазначений підхід.

Постановка задачі та призначення. Розробити CLI (Command Line Interface) додаток для генерації текстового опису ER діаграм у форматі PlantUML та Mermaid на основі рядка підключення до бази даних PostgreSQL.

Призначення цього продукту — сприяти автоматизації генерації ER-діаграм і підвищувати їх якість, що, своєю чергою, позитивно вплине на процес розробки програмного забезпечення (ПЗ) загалом.

Оскільки спеціалісти та команди різних напрямків зазвичай використовують персоналізований набір програмних інструментів та операційних систем, кінцевий результат (зображення діаграми) повинен бути універсальним і не потребувати спеціального ПЗ для роботи. Враховуючи мету продукту, додаток повинен реалізувати базову функціональність та усунути частину недоліків рішень розглянутих в розділі 1.7 «Аналіз існуючих рішень». Під час розробки використати сучасні інструменти та підходи.

В додатку А описано вичерпне технічне завдання для цього продукту.

2.2. Вибір моделі розробки програмного засобу «Sqlant»

Враховуючи специфіку проєкту, а саме вимогу поступового нарощування нового функціоналу (підтримки різних баз даних та форматів діаграм) було обрано ітераційно-інкрементну модель яка сприяє поетапному створенню і супроводу продукту, адже після кожної ітерації отримується готова версія продукту, готового до використання.

Для проєкту «sqlant» ця модель є найбільш доцільною. Оскільки:

- вимоги до функціональності визначаються поступово (продукт був готовий до використання з першої версії (0.1.0) починаючи з якої під час роботи з ПЗ аналізуються можливості які слід впровадити в наступних версіях [26]);
- необхідно створювати та перевіряти прототипи (підтримку різних СУБД і нотацій), що дає змогу перевіряти працездатність ідей, алгоритмів та вигляд згенерованих діаграм на ранніх етапах, знижуючи ризики виникнення помилок у фінальних версіях;
- кожна нова можливість (наприклад, робота з views, яка була додана в версії 0.6.0, або формат Mermaid, доданий в версії 0.2.0) реалізовується як окремий інкремент без впливу на основний код [26];
- після кожної ітерації здійснюється ручне та автоматизоване тестування, що підвищує стабільність та якість продукту;
- модель добре узгоджується з принципами безперервної інтеграції (CI/CD) і модульною архітектурою розробленого ПЗ.

Застосування ітераційно-інкрементної моделі було вдалим рішенням оскільки це дозволило отримати робочий продукт на кожному етапі розробки, яка триває. Дана модель забезпечила гнучкість при внесенні змін, знизила ризики при розробці та впровадженні нових версій та підвищила загальну якість програмного забезпечення.

2.3. Опис проєкту «Sqlant»

«Sqlant» – це ПЗ з відкритим кодом яке розповсюджується за ліцензією MIT (Massachusetts Institute of Technology) [27], головною ціллю якого є генерація ER діаграм. На вхід подається рядок підключення до бази даних (connection string), а результатом є формалізований текстовий опис (PlantUML, Mermaid) виведений в термінал, за допомогою якого відбувається генерація графічного зображення. Менш значущою, але також важливою частиною є те що sqlant можна використати в якості бібліотеки мови програмування Rust, що дає змогу використовувати код програми, для аналізу структури бази даних, чи генерації діаграм в інших проєктах. [26-29]

Саме модульна архітектура, яка зображена на рисунку 2.1, та можливості мови програмування Rust дали змогу з легкістю підтримувати два типи розповсюдження як окремий виконуваний файл (CLI-додаток) та як бібліотека.

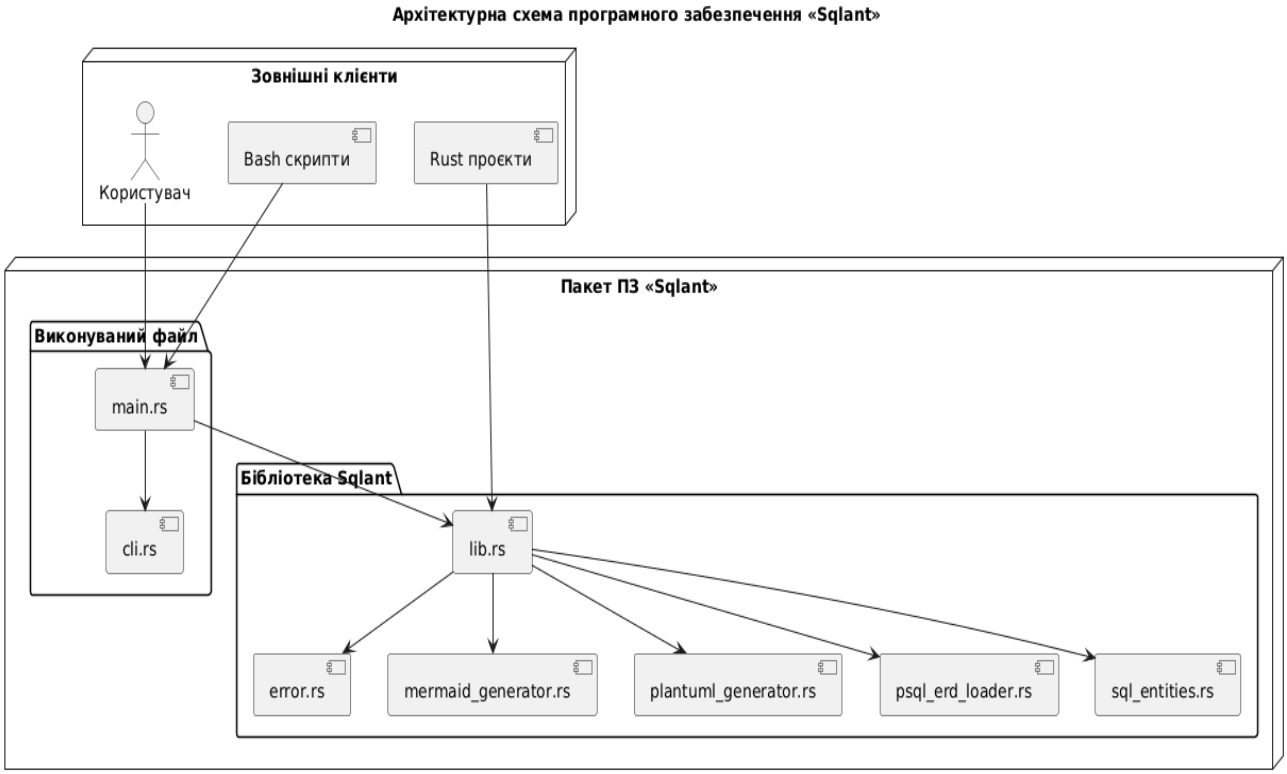

Рисунок 2.1. – Архітектурна схема ПЗ «Sqlant»

На рисунку 2.1 зображений пакет ПЗ «Sqlant» який складається з двох компонентів, а саме: виконуваний файл та бібліотека. Блок «Зовнішнє середовище вокристання» доданий для наглядності і розуміння як та ким або чим використовується ПЗ.

2.3.1. Бібліотека Sqlant

Бібліотека являється ядром пакету ПЗ і містить основну логіку програми. Головним, корінним файлом бібліотеки є lib.rs, завдяки якому пакетний менеджер Cargo [30] розпізнає код в директорії src як бібліотеку. Файл являється «точкою входу» в бібліотеку де зазвичай декларуються й експортуються модулі та описується публічний API. Тут також може знаходитись код такий як і в інших файлах (функції, структури, типи тощо). Кожен модуль в середині бібліотеки доступний для її членів які мають змогу використовувати його публічні елементи. lib.rs окрім опису модулів містить наступні елементи:

- структура GeneratorConfigOptions з полями: not_null, draw_enums, draw_legend, inline_puml_lib, conceptual_diagram, що відповідає за налаштування генерації діаграм і подається на вхід генератора;
- Trait (аналог інтерфейсів у Java або C#) ViewGenerator визначає інтерфейс з однією функцією «generate», яку повинен реалізувати кожен модуль-генератор та яка відповідає за генерацію PlantUML, Mermaid тощо;
- функція «lookup_parser» та її реалізація, що на основі рядка підключення визначає тип модуля потрібного для роботи з БД;
- перелічення (enum) GeneratorType містить підтримувані типи текстового опису діаграм (PlantUML та Mermaid);
- функція «get_generator» та її реалізація, що на вхід отримує перелічення GeneratorType, повертає об’єкт типу ViewGenerator.

Окрім файлу lib.rs бібліотека містить 5 модулів які мають доступ до публічних елементів один одного.

Error.rs, модуль обробки помилок. Містить лише одне перелічення SqlantError. Призначений для централізованого опису помилок які можуть

виникнути під час використання бібліотеки. За допомогою бібліотеки «thiserror» [31] виконує автоматичне перетворення помилок з інших бібліотек у внутрішній тип SqlantError;

Sql_entities.rs, модуль який містить представлення sql сутностей. Він містить структури даних для роботи з таблицями (struct Table), представленнями (struct View), полями (struct TableColumn) та їх обмеженнями (enum ColumnConstraints). Також містить структуру SqlERData в якій є 4 поля: tables, foreign_keys, enums та views. Trait SqlERDataLoader містить функцію load_erd_data результатом якої є SqlERData. SqlERDataLoader являється інтерфейсом для модулів які відповідають за підключення до бази даних та отримання інформації про її структуру;

Psql_erd_loader.rs, завантажувач метаданих для PostgreSQL бази даних. Відповідає за отримування інформації про структуру бази даних (таблиці, колонки, зв'язки) з системних таблиць PostgreSQL. Імплементує інтерфейс SqlERDataLoader;

Mermaid_generator.rs/Planuml_generator.rs, ці два модулі реалізують генерацію ER-діаграм у різних форматах приймаючи на вхід структуру SqlERData. Обидва реалізовують інтерфейс ViewGenerator.

Бібліотека sqlant об'єднує ключові модулі системи — від встановлення з'єднання з БД та завантаження метаданих, до генерації діаграм різних нотацій. В ній забезпечене чітке розділення відповідальностей між модулями та уніфікований інтерфейс публічного API. Завдяки модульності та використанню інтерфейсів (trait) бібліотека легко розширюється підтримуючи додавання нових генераторів (ViewGenerator) та завантажувачів (SqlERDataLoader).

2.3.2. Виконуваний Файл (CLI)

Виконуваний файл складається з бібліотеки «Sqlant» та двох модулів Cli.rs і Main.rs. Cli.rs, модуль який відповідає за парсинг аргументів командного рядка, тоді як Main.rs є точкою входу в програму і містить головну функцію «main» де

відбувається використання модуля Cli.rs, бібліотеки, та виведення результату в термінал.

На рисунку 2.2 зображена діаграма послідовності (Sequence Diagram), яка демонструє процес використання утиліти (або виконуваного файлу) «sqlant» та її місце в процесі генерації ER діаграм у форматі PlantUML. На діаграмі зображено 4 елементи:

- User – користувач, який запускає програму з командного рядка;
- Sqlant – консольна утиліта розроблена в рамках магістерської;
- DB – база даних на основі якої генерується діаграма;
- Plantuml.com – онлайн сервіс який генерує діаграми в форматі png/svg на основі текстового опису (PlantUML).

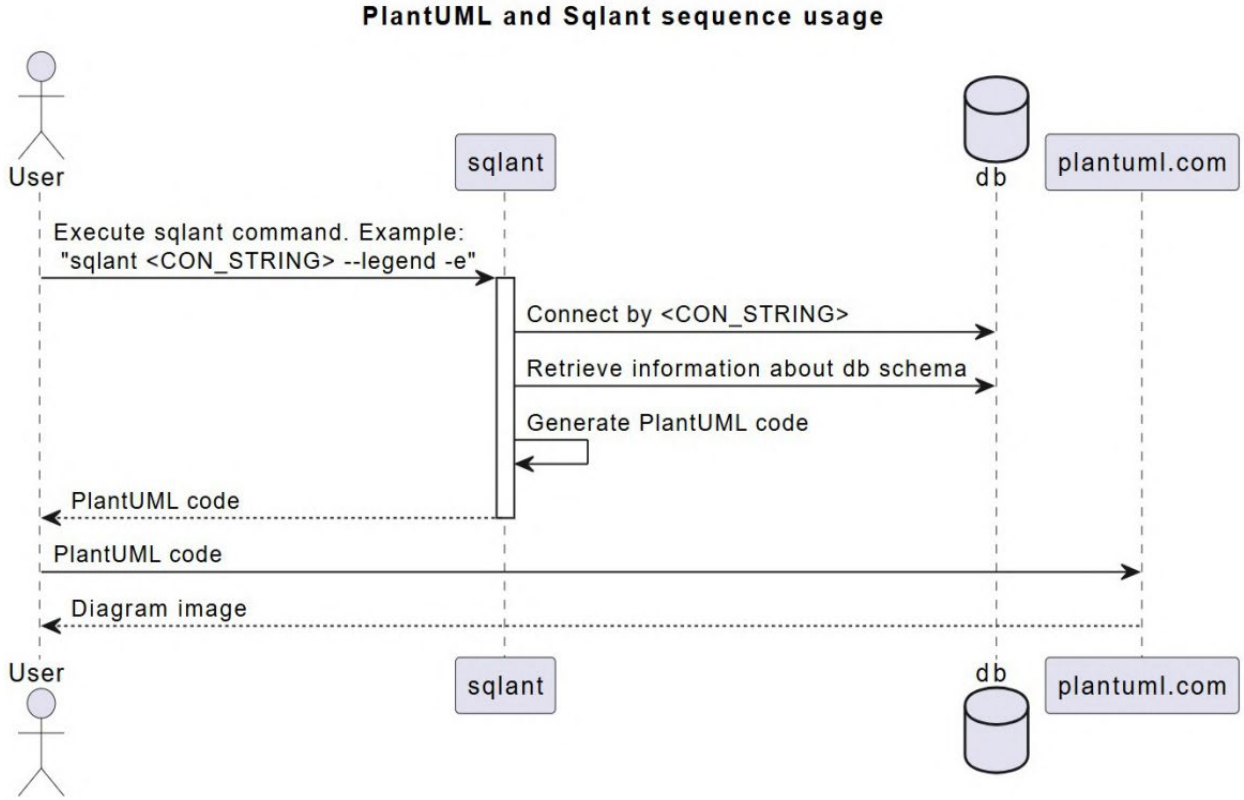

Рисунок 2.2. – Діаграма послідовності (Sequence Diagram) використання «Sqlant» та PlantUML

User виконує команду sqlant передаючи стрічку з’єднання до БД, та конфігураційні параметри (опційно), sqlant з’єднується з БД та отримує інформацію (метадані) про структуру БД, на основі отриманих даних sqlant генерує текст у форматі PlantUML, далі User використовує цей текст для генерації фінального зображення.

В статті «RECONSTRUCTING ENTITY RELATIONSHIPS IN DATABASE SCHEMAS WITH PLANTUML AND LLMS.» [18] було продемонстровано можливість розширення даного сценарію включивши крок використання LLM як інструмента для відновлення пропущених зв’язків між таблицями. Що може мати позитивний вплив на архітектуру БД користувачів.

2.3.3. PlantUML бібліотека

В PlantUML існує підтримка бібліотек по аналогії з бібліотеками у звичайних мовах програмування (наприклад Python), які мають поділ на стандартні та користувацькі. Це надає змогу створювати власні функції та процедури, а також можливість організовувати їх в бібліотеки, що використовується для покращення зручності читання (PlantUML) коду зменшуючи його об’єм. Стандартна бібліотека PlantUML детально розглянута в довідковому посібнику [2, розд. 27].

Бібліотеки містять процедури та функції які відрізняються між собою тим, що функції повертають код (результат) який буде використаний для рендерингу, а процедури виконують операції відразу і не повертають результат. Функції мають поділ на вбудовані (built-in) та користувацькі. Виклик вбудованих функцій повинен починатись символом «%» в довідковому посібнику [2, розд. 25.12] наведена таблиця що містить близько 50 записів, з переліченими назвами вбудованих функцій та їх описом. Також надається функціонал вбудованих директив препроцесора [2, розд. 25] (подібно до макросів в мові програмування C) виклик яких починається символом «!». Список використаних директив препроцесора для розробки PlantUML бібліотеки яка являється частиною програмного пакета «Sqlant», наведено в таблиці 2.1.

Таблиця 2.1

Опис використаних директив

Директива	**Опис**
!function, !endfunction	Визначення початку та закінчення користувацької функції
!return	Повернення результату функції
!local	Визначення локальної змінної
!if,!elseif,!endif	Реалізація умовної логіки
!procedure, !endprocedure	Визначення початку та закінчення користувацької процедури
!foreach,!endfor	Визначення початку та закінчення циклу

У зв’язку з потребою покращення вигляду діаграм спочатку був розроблений PlantUML код (прототип) який покращував дану характеристику. Після розробки коду було виділено блоки які повторювались та на їх основі розроблено функції та процедури. Щоб не підключати всі функції та процедури в кожен PlantUML код діаграми, було прийнято рішення розробити бібліотеку яка підключатиметься за допомогою директиви «!include» та веб адреси самого файлу бібліотеки.

Бібліотека складається з трьох функцій та двох процедур. Функції мають як обов’язкові параметри, так і опціональні (з визначеними значеннями по замовчуванню).

Функція Column повертає код для генерації колонки. Обов’язкові параметри функції: ім’я та тип. Опціональні мають булевий тип який відповідає за наступні характеристики колонок: первинний ключ, зовнішній ключ, обмеження (Not Null).

Функція Table повертає код для генерації верхньої частини (шапки) таблиці. Приймає один обов’язковий вхідний параметр: ім’я таблиці.

Функція View, аналогічна функціїї Table, але для представлення (view). Обов’язковий параметр: ім’я представлення. Опціональний: чи являється представлення матеріалізованим (materialized).

Процедура Enum виконує код для генерації типу даних Enum. Приймає два обов’язкові параметри, а саме ім’я перелічення та самі значення перелічень.

Процедура add_legend. Виконує код для генерації легенди. Легенда використовується для пояснення елементів діаграми.

На рисунку 2.3 зображено вплив вище перерахованих функцій та процедур на рендеринг діаграми.

Розробка PlantUML бібліотеки допомогла покращити загальний вигляд діаграм зробивши їх більш інформативними та покращити якість коду абстрагуючи складні елементи в зрозумілі для будь-якого користувача SQL баз даних функції та процедури.

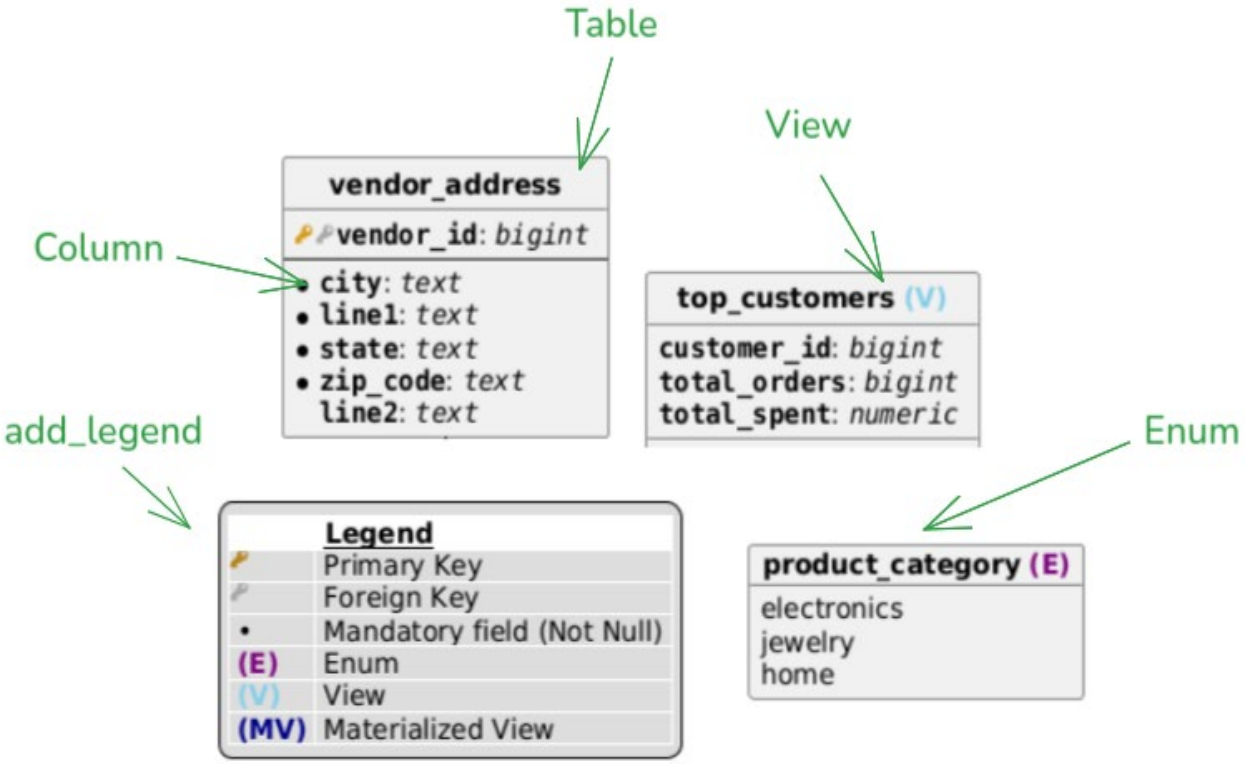

Рисунок 2.3. – Опис елементів діаграми відповідно до функціональності бібліотеки.

2.4. Засоби розробки «Sqlant»

2.4.1. Вибір мови програмування

Однією з головних ідей при розробці «Sqlant» було забезпечити простий процес інсталяції з підтримкою всіх популярних ОС, тобто зробити програмний засіб максимально портативним. Тому стало зрозуміло що потрібна компільована мова програмування з підтримкою статичного лінкування бібліотек.

Статичне лінкування (static linking) – це процес коли весь код потрібний для роботи програми включно з бібліотеками компілюється в один виконуваний файл. Що забезпечує високу портативність, але збільшує розмір виконуваного файлу, та час компіляції програми. З популярних мов програмування, які підтримують статичне лінкування можна виділити: C, C++, Rust, Go. З цих чотирьох мов програмування офіційну підтримку центрального репозиторію пакетів (який використовується для розповсюдження програм та/чи бібліотек) має лише Rust, а саме crates.io [34] з пакетним менеджером cargo [53]. Особливістю cargo також є те що він виконує не лише функцію пакетного менеджера, а й багато інших, таких як: інструмент для форматування коду (cargo fmt), аналізатор коду (cargo clippy), засіб запуску тестів (cago test) тощо.

Rust відносно молода мова програмування зі статичною типізацією, розроблена командою з компанії Mozilla перша стабільна версія якої вийшла у 2015 році [35-36]. Головний фокус якої є безпека програм та швидкодія. Має дуже широку сферу застосування від вбудованих систем (embedded systems) до веб сервісів, настільних застосунків та використовується великими технічними компаніями такими як Microsoft, Amazon тощо [37-38].

2.4.2. Програмні бібліотеки

За час свого існування Rust спільнота розробила велику кількість бібліотек опублікованих на crates.io, що й дало змогу розробити проєкт у відносно короткі терміни. Вибір кожної бібліотеки яка використовуватиметься при розробці

повинен бути усвідомленим. Особливо у випадку використання статичної лінковки де кожна бібліотека впливає на вихідний розмір програми. Нижче представлено список в алфавітному порядку і короткий опис бібліотек які використовуються в пакеті ПЗ «Sqlant».

*Async-trait***.** Використовується разом з trait SqlERDataLoader. Потрібний для користувачів бібліотеки щоб забезпечити можливість використання в асинхронних оточеннях. Даний функціонал не було заплановано в перших версіях, й було додано користувачем ресурсу Github з нікнеймом jayvdb в пулл реквесті номер 27 [39].

Clap. Забезпечує роботу з CLI аргументами. Створення команд, їх опис, та парсинг [40].

Native-tls. Кроссплатформена TLS/SSL імплементація. Забезпечує SSL з'єднання (sslmode=require в стрічці підключення). Одна з самих більших за обсягом бібліотек. Підтримка SSL з'єднання була реалізована в версії 0.5.0, де розмір виконуваного файлу суттєво збільшився, з 1.64MB до 6.27MB [41].

Postgres-native-tls. Адаптер який з'єднує native-tls та tokio-postgres [42].

Serde. Фреймворк для операцій серіалізації та десералізації структур, потрібен для роботи з tinytemplate [43].

Thiserror. Використовується для більш зручної та лаконічної роботи з помилками. Створення власних типів помилок, перетворення існуючих типів на власні автоматично (за допомогою оператора «?») [44].

Tinytemplate. Мінімалистична система шаблонізації тексту. Викороистовується для генерації PlantUML/Mermaid коду [45].

Tokio. Використовується для асинхронних та багато поточних операцій. Потрібен для функціонування tokio-postgres [46].

Tokio-postgres. Асинхронний PostgreSQL клієнт, використовується для з'єднання і операцій з БД [47].

2.4.3. Допоміжні засоби розробки

Розробка даного програмного продукту не вимагає спеціальних середовищ, але вірний підбір допоміжних інструментів розробки допоміг спростити процес створення програмного продукту.

В якості інструменту для написання коду використовується текстовий редактор Neovim [48]. Neovim – це термінальний текстовий редактор який являється відгалуженням (fork) іншого популярного редактора Vim [49]. Головна відмінність якого є підтримка плагінів написаних мовою програмування Lua [50], та підтримка LSP (Language Server Protocol) [51] який був створений для розділення логіки текстових редакторів та інструментів аналізу коду. Це дає можливість зробити повноцінну заміну IDE з величезною можливістю конфігурації та швидкодією при невеликому використанні обчислювальних ресурсів.

Оскільки маніпуляції з базою даних під час розробки, а саме підключення та редагування структури в цілях тестування являються досить простими, консольного клієнта бази даних PostgreSQL, який має назву «psql» [52], цілком достатньо. Для локального розгортання СУБД PostgreSQL використовується система контейнеризації Docker, де за допомогою одної команди можна з легкістю розгорнути потрібну версію баз даних, а після використовуючи файл міграції та psql створити потрібну структуру.

З метою контролю версій використано ПЗ git включно з механізмом створення міток (git tag). Платформа Github для хостингу коду та командної розробки. За допомогою Github Actions реалізовано підхід безперервної інтеграції (Continuous Integration) а саме: автоматизацію тестів, збірки, та статичного аналізу коду.

2.5. Особливості програмної реалізації ПЗ «Sqlant»

Блок-схема процесу генерації ER-діаграми з відповідними частинами коду головної функції «main» в програмному засобі «Sqlant» зображена на рис. 2.4, де

кожен блок зв’язаний з відповідною частиною коду. В даному розділі розглянемо особливості реалізації наступних процесів: завантаження метаданих схеми, побудова внутрішньої ER-моделі, генерація текстового опису діаграми.

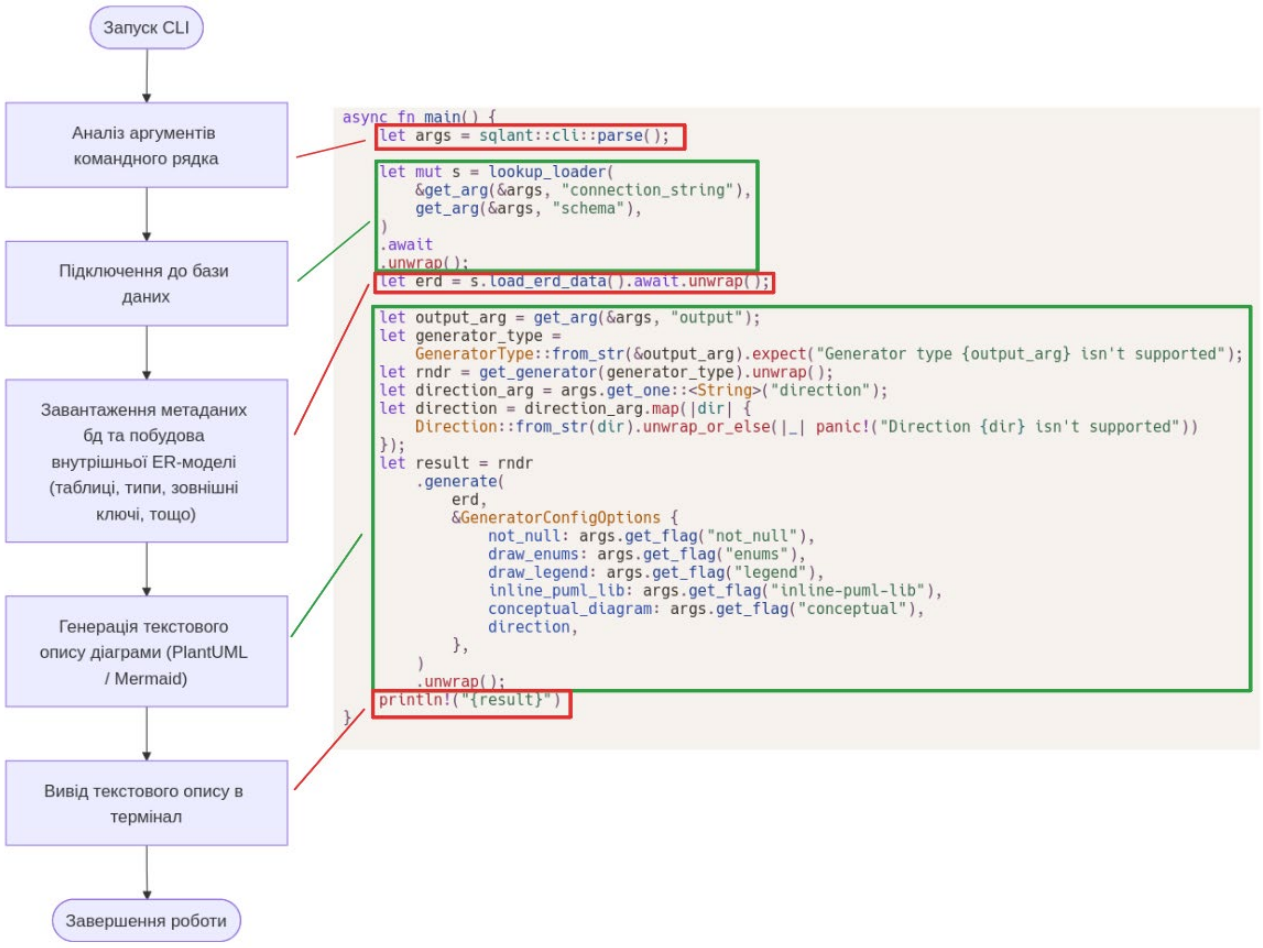

Рисунок 2.4. – Блок-схема та головна функція (точка входу) CLI-компонента програмного засобу «Sqlant».

2.5.1. Завантаження метаданих схеми та побудова внутрішньої ER-моделі

Для того щоб завантажити метадані схеми (назва таблиць, поля, їх особливості тощо) потрібно приєднатись до бази даних. Для встановлення зв’язку з базою даних використовується рядок підключення (connection string) в форматі URI (Uniform Resource Identifier), який є єдиним обов’язковим параметром для роботи програми.

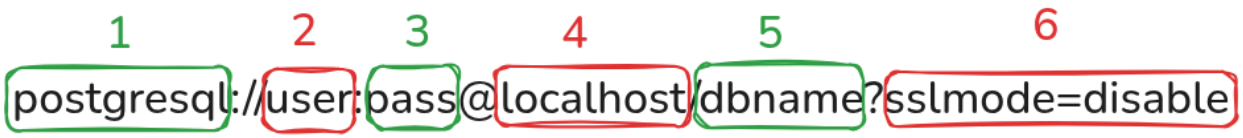

Рисунок 2.5. – приклад рядка підключення до бази даних PostgreSQL

На рисунку 2.5 зображено 6 компонентів рядка підключення. Далі наведено опис кожного компоненту.

1. Схема (protocol) яка вказує який драйвер (тип СУБД) використовується.
2. Ім’я користувача від імені якого буде виконано підключення.
3. Пароль користувача.
4. Адрес хоста, де запущена база даних (може містити також номер порту на якому працює сервер).
5. Назва бази даних до якої відбувається підключення.
6. Після знаку «?» можуть міститися додаткові параметри підключення розділених амперсандом.

На основі значення компонента 1 відбувається пошук відповідного завантажувача (SqlERDataLoader) метаданих. Оскільки наразі наявна лише підтримка PostgreSQL, функція відразу повертає структуру типу PostgreSqlERDLoader. Структури які реалізують інтерфейс SqlERDataLoader відповідають за підключення до бази даних та отримання метаданих схеми. PostgreSqlERDLoader (який реалізує інтерфейс SqlERDataLoader) приймає в конструкторі (метод new) повне значення рядка підключення та назву схеми бази даних. Під час створювання об’єкта також відбувається встановлення зв’язку з базою даних. Код конструктора зображено на рис. 2.6.

Після того як об’єкт створено та зв’язок з базою даних встановлено викликається метод load_erd_data в якому відбувається завантаження метаданих БД, та побудова внутрішньої моделі яка представлена структурою SqlERData.

Для завантаження метаданих використовуються SQL запити, які знаходяться в файлі psql_erd_loader.rs. Запити зберігаються в статичних, константних змінних та використовують інформацію з наступних таблиць схеми

information_schema: tables, pg_matviews, pg_attribute, pg_class, pg_type, pg_constraint.

```rust
impl PostgreSqlERDLoader {
    pub async fn new(
        connection_string: &str,
        schema_name: String,
    ) -> Result<PostgreSqlERDLoader, SqlantError> {
        let connector = TlsConnector::builder()
            .danger_accept_invalid_certs(true)
            .build()?;
        let connector = MakeTlsConnector::new(connector);

        let (client, connection) = tokio_postgres::connect(connection_string, connector).await?;
        tokio::spawn(async move {
            if let Err(e) = connection.await {
                eprintln!("connection error: {}", e);
            }
        });
        Ok(PostgreSqlERDLoader {
            client,
            schema_name,
            pks: BTreeMap::new(),
            fks: BTreeMap::new(),
        })
    }
```

Рисунок 2.6. – Код конструктора структури PostgreSqlERDLoader

Всі запити приймають по одному вхідному параметру який під час виконання підставляється замість комбінації символа «$1». Відносно простими запитами код яких зображено на рис. 2.7 можна вважати наступні: GET_TABLES_LIST_QUERY, GET_MATERIALIZED_VIEWS та GET_ENUM_VALUES.

GET_TABLES_LIST_QUERY. Повертає список таблиць та нематеріалізованих представлень (view), в якості вхідного параметра використовується ім’я схеми.

GET_MATERIALIZED_VIEWS. Повертає список матеріалізованих представлень, в якості вхідного параметра використовується ім’я схеми.

GET_ENUM_VALUES. Повертає список значень перелічення, в якості вхідного параметра використовується pg_type.oid, що є унікальним ідентифікатором типу. Цей ідентифікатор отримується за допомоги запиту GET_COLUMNS_BASIC_INFO_QUERY.

```
static GET_TABLES_LIST_QUERY: &str = r#"
SELECT trim(both '"' from table_name) as table_name, table_type
FROM information_schema.tables
WHERE table_schema = $1
ORDER BY table_name;
"#;

static GET_MATERIALIZED_VIEWS: &str = r#"
SELECT trim(both '"' from matviewname) AS matview_name
FROM pg_matviews
WHERE schemaname = $1
ORDER BY matviewname;
"#;

static GET_ENUM_VALUES: &str = r#"
SELECT enumlabel
FROM pg_enum
JOIN pg_type ON pg_enum.enumtypid = pg_type.oid
WHERE pg_type.oid = $1
ORDER BY pg_enum;
"#;
```

Рисунок 2.7. – Код запитів GET_TABLES_LIST_QUERY, GET_MATERIALIZED_VIEWS та GET_ENUM_VALUES.

Після виконання запиту GET_TABLES_LIST_QUERY, для отримання переліку полів (колонок) та їх характеристик виконується запит GET_COLUMNS_BASIC_INFO_QUERY (рис. 2.8), який приймає в якості вхідного параметра список назв таблиць отриманих за допомогою запиту GET_TABLES_LIST_QUERY. Даний запит агрегує інформацію від трьох таблиць: pg_attribute, pg_class, pg_type. Оскільки видалені колонки за допомогою конструкції ALTER TABLE ... DROP COLUMN не видаляються фізично з системного каталогу їх потрібно виключити з результату за допомогою умови NOT attisdropped. Також таблиця може містити системні (службові) колонки які повинні бути виключені з фінального результату за допомогою додаткової умови attnum > 0. Разом ці фільтри дають тільки актуальні, не видалені, користувацькі колонки таблиці які містять інформацію про кожну колонку для кожної з таблиць, а саме: назва колонки, її унікальний ідентифікатор (attnum), тип, характеристика Not Null, назва таблиці якій належить, ідентифікатор типу.

```
static GET_COLUMNS_BASIC_INFO_QUERY: &str = r#"
SELECT attname                                  AS col_name,
       attnum                                   AS col_num,
       pga.atttypid::regtype::name              AS datatype,
       attnotnull                               AS not_null,
       relname                                  AS table_name,
       pg_type.oid                              AS column_type_oid,
       typtype
FROM   pg_attribute pga
INNER JOIN pg_class
   ON pg_class.oid = pga.attrelid
INNER JOIN pg_type
   ON pga.atttypid::regtype::oid = pg_type.oid
WHERE relname =  any($1)
AND    NOT attisdropped
AND    attnum  > 0
ORDER  BY relname, col_name;
"#;
```

Рисунок 2.8. – Код запиту GET_COLUMNS_BASIC_INFO_QUERY.

Оскільки GET_COLUMNS_BASIC_INFO_QUERY не повертає інформації про первинні та вторинні ключі (теоретично це можна було б реалізувати в одному запиті, але такий запит вийшов би невиправдано складним) було розроблено ще два додаткові запити, а саме GET_PKS_QUERY та GET_FOREIGN_KEYS_QUERY зображені на рис. 2.9, які отримують дані з таблиці pg_constraint, приймаючи в якості вхідного параметра назву схеми, та повертаючи інформацію про всі наявні ключі відповідного типу фільтруючи за параметром contype де значення f відповідає за зовнішні ключі, а значення p за первинні. У двох запитах останнім полем є результат функції pg_get_constraintdef, дане поле використовується лише в цілях налагодження програми та повертає зрозумілий для людини опис обмеження, наприклад: FOREIGN KEY (hotspot_pubkey_id) REFERENCES hotspot_pubkey(id). Результатом запиту GET_PKS_QUERY є: назва таблиці, назва первинного ключа та масив порядкових номерів колонок. Масив використовується у зв’язку з тим що первинні ключі можуть складатися з більш ніж одного поля, такі первинні ключі називаються композитними. Результатом запиту GET_FOREIGN_KEYS_QUERY є: назва таблиці яка має вторинний ключ, назва

таблиці на яку вона посилається, відповідні номери колонок та унікальна назва обмеження.

```
static GET_FOREIGN_KEYS_QUERY: &str = r#"
SELECT trim(both '"' from conrelid::regclass::name)  AS source_table_name,
       trim(both '"' from confrelid::regclass::name) AS target_table_name,
       conname                   AS foreign_key_name,
       conkey                    AS source_column_nums,
       confkey                   AS target_columns_nums,
       pg_get_constraintdef(oid) -- just for debugging purposes
FROM   pg_constraint
WHERE  contype = 'f'
AND    connamespace = to_regnamespace($1)::oid
ORDER  BY source_table_name;
"#;

static GET_PKS_QUERY: &str = r#"
SELECT trim(both '"' from conrelid::regclass::name) as table_name,
       conname                   AS primary_key_name,
       conkey                    AS pk_columns_nums,
       pg_get_constraintdef(oid) -- just for debugging purposes
FROM   pg_constraint
WHERE  contype = 'p'
AND    connamespace = to_regnamespace($1)::oid
ORDER  BY table_name;
"#;
```

Рисунок 2.9. – Код запитів GET_PKS_QUERY та GET_FOREIGN_KEYS_QUERY.

Підсумовуючи, результатом виконання вище перелічених запитів та парсингу їхніх результатів у функції load_erd_data є структура SqlERData яка містить побудовану внутрішню модель бази даних.

2.5.2. Генерація текстового опису діаграми

Sqlant підтримує генерацію текстового опису у двох форматах: PlantUML та Mermaid, в даному розділі буде розглянуто генерацію текстового опису PlantUML оскільки вона являється більш функціональною.

Генерація відбувається на основі даних які знаходяться в структурі SqlERData та налаштувань переданих за допомогою опційних параметрів

командного рядка. Ключовим елементом в процесі генерації тексту являється бібліотека «TinyTemplate».

TinyTemplate належить до типу бібліотек які називають «шаблонізаторами» (template engine). Такі бібліотеки використовуються для генерації текстів на основі шаблонів та даних. Головною особливістю саме цієї бібліотеки є легкість використання та мінімальна кількість залежностей що впливає на фінальний розмір програми [45]. На рисунку 2.10 зображено код шаблонів які знаходяться в файлі plantuml_generator.rs. Де головним шаблоном являється «PUML_TEMPLATE», який в загальній формі описує результат. В подвійних фігурних дужках містяться ті значення які будуть підставлені в процесі виконання функції render (рендерингу) та умовні конструкції if/endif які спрацьовують залежно від наявності значень.

```
static PUML_TEMPLATE: &str = "@startuml\n\n\
{{ if direction }}{ direction }\n{{ endif }}\
    hide circle\n\
    hide empty members\n\
    skinparam linetype ortho\n\n\
    {puml_lib}\n\n\
    {{ for ent in entities}}{ent}\n{{ endfor }}\n\
    {{ for view in views}}{view}\n{{ endfor }}\n\
    {{ for fk in foreign_keys}}{fk}\n{{ endfor }}\n\
    {{ for e in enums}}{e}\n{{ endfor }}{legend}\n@enduml";

static ENTITY_TEMPLATE: &str =
    "table({name}) \\{\n{pks} {{ if pks }} ---\n{{ endif }}{fks}{nns}{others}}\n";

static VIEW_TEMPLATE: &str =
    "view({name}{{ if materialized}}, $materialized=true{{ endif }}) \\{\n{columns}}\n";

static COLUMN_TEMPLATE: &str = "  column({col.name}, \"{col.datatype}\"{{ if is_pk }},
    $pk=true{{ endif }}{{ if is_fk }}, $fk=true{{ endif }}{{if is_nn}}, $nn=true{{ endif }})\n";

static REL_TEMPLATE: &str = "{source_table_name} {{ if is_zero_one_to_one }}|o--||{{else}}}o--||{{ endif }} {target_table_name}\n";

static ENUM_TEMPLATE: &str =
    "enum({name}, \"{{ for v in values}}{{if @last}}{v}{{else}}{v}, {{ endif }}{{ endfor }}\")\n";
```

Рисунок 2.10. – Код для шаблонів «TinyTemplate»

Значення в одинарних фігурних дужках puml_lib не являється частиною шаблону tinytemplate, а являється частиною звичайного шаблону для типу даних str, вбудованого в мову програмування Rust. Залежно від переданого командного аргументу puml_lib набуває значення посилання на розроблену PlantUML бібілотеку, або вбудованого коду всієї бібліотеки в діаграму. Це розроблено для того щоб надати користувачу вибір:

- зменшити об’єм тексту, але зробити генерацію діаграми залежною від наявності інтернету та доступу до бібліотеки;
- збільшити об’єм тексту, але зробити PlantUML код незалежним від сторонніх бібліотек.

За генерацію текстового опису відповідають структури які реалізують інтерфейс ViewGenerator який містить функцію generate на вхід якій передається структура SqlERData та GeneratorConfigOptions, результатом якої є текстовий опис діаграми в форматі String готовий до використання. В даному випадку такою структурою є PlantUmlDefaultGenerator.

GeneratorConfigOptions (рис. 2.11) містить параметри для конфігурації фінального текстового опису, а саме:

- Not_null, візуальне виділення поля яке обов’язкове для заповнення (в PlantUmlDefaultGenerator завжди true);
- Draw_enums, відображення типів перечислення;
- Draw_legend, відображення легенди;
- Include_puml_lib, тип використання бібліотеки (додати в код діаграми, або під’єднати динамічно);
- Conceptual_diagram, генерація концептуальної діаграми яка містить лише назву таблиць та їх зв’язки;
- Direction, опціональний параметр який впливає на розміщення елементів в діаграмі, можливі значення: tb (top to bottom), bt(bottom to top), lr (left to right), rl (right to left).
-

```rust
pub struct GeneratorConfigOptions {
    pub not_null: bool,
    pub draw_enums: bool,
    pub draw_legend: bool,
    pub inline_puml_lib: bool,
    pub conceptual_diagram: bool,
    pub direction: Option<Direction>,
}
```

Рисунок 2.11. – GeneratorConfigOptions

Фінальний результат текстового опису виводиться на екран (в термінал) і використовується для генерації зображення. Приклад (частини) згенерованого текстового опису в форматі PlantUML є фінальним результатом роботи утиліти «Sqlant» та діаграми згенерованої за допомогою ресурсу planttext.com зображено на Рисунку 2.12.

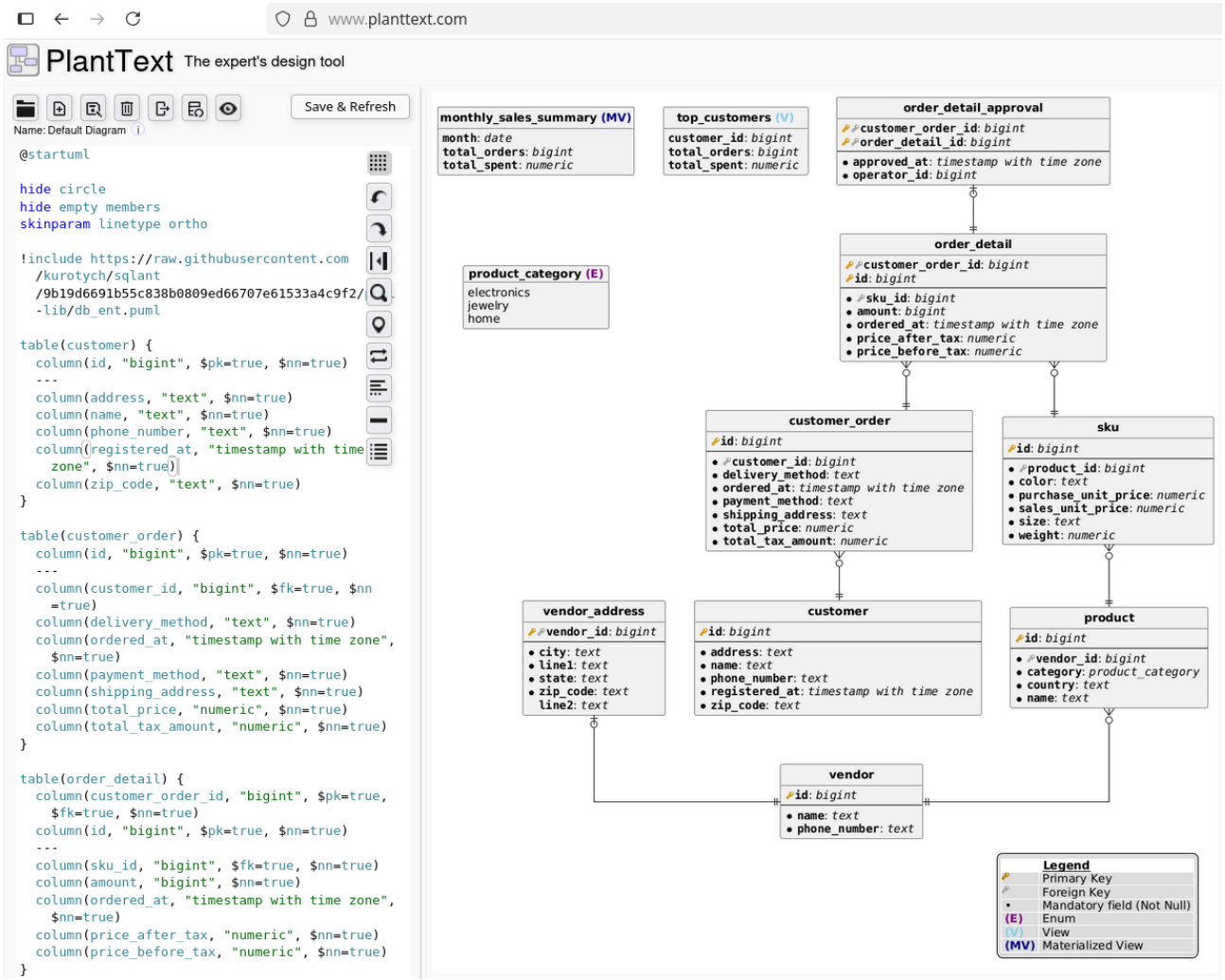

Рисунок 2.12. – Діаграми згенерована за допомогою ресурсу planttext.com

2.6. Організація тестування та налагодження програмного засобу

Тестування ПЗ «Sqlant» можна розділити на те яке піддається автоматизації (автоматичне), та ручне.

Зазвичай частиною репозиторію де знаходиться код програми являються модульні та інтеграційні тести які є складником автоматичного тестування.

Різниця між ними полягає в масштабі тестування, де модульні тестують лише один ізольований модуль, а інтеграційні співпрацю декількох компонентів системи. Інтеграційні та модульні тести являються частиною репозиторію з кодом, написані за допомогою мови Rust, знаходяться в спеціальний директорії tests та запускаються за допомогою команди cargo test. На ранніх етапах (на якому являється даний проєкт) надається перевага інтеграційним тестам над модульними, оскільки код продукту часто переписується, а разом з ним виникає постійна потреба в переписуванні модульних тестів, а оскільки інтеграційні працюють на вищому рівні абстракції потреба в їх переписуванні виникає рідше.

В нашому випадку інтеграційні тести покривають частину коду яка відповідає за завантаження даних з БД та побудови внутрішньої моделі результатом чого є структура SqlERData. Для налаштування структури БД перед запуском інтеграційних тестів було розроблено два файли з форматом sql які містять DDL запити для створення її структури. Основний файл містить SQL код для створення всіх компонентів які підтримує утиліта, а саме: таблиці, первинні та вторинні ключі, користувацькі типи, представлення, матеріалізовані представлення. Перед запуском тестів потрібно виконати код даного файлу щоб встановити структуру готову до тестування й встановити змінну середовища CON_STRING зі значенням яке містить рядок підключення до БД, що використовуватиметься для тестування. На рис. 2.13 зображено код одного з інтеграційних тестів проєкту. Даний тест створює внутрішнє представлення бази даних та перевіряє відповідність значення поля enum в SqlERData до очікуваного (створеного за допомогою тестового файлу SQL).

Наступним типом тестів які вдалось написати та автоматизувати є системні тести які в нашому випадку перевіряють що скомпільований виконуваний файл sqlant (в режимі випуску), генерує коректний PlantUML та Mermaid код. Під коректним мається на увазі що дані системи в змозі інтерпретувати його без помилок. Для забезпечення середовища тестування у випадку з PlantUML потрібно встановити сервер локально, це можна зробити за допомогою докер контейнера, для запуску тестів використовувався образ plantuml/plantuml-

server:tomcat-v1.2025.2. У випадку з Mermaid потрібно встановити консольну утиліту за допомогою пакетного менеджера npm [54-55].

```
async fn load_erd() -> SqlERData {
    let con_string = env::var("CON_STRING").unwrap();
    let mut parser = lookup_loader(&con_string, "public".to_string())
        .await
        .unwrap();
    parser.load_erd_data().await.unwrap()
}

#[tokio::test]
async fn enums() {
    let sql_er_data: SqlERData = load_erd().await;
    let mut expected_hash_map = BTreeMap::new();
    expected_hash_map.insert(
        "product_category".to_string(),
        vec![
            "electronics".to_string(),
            "jewelry".to_string(),
            "home".to_string(),
        ],
    );
    assert_eq!(sql_er_data.enums, expected_hash_map);
}
```

Рисунок 2.13. – Код інтеграційного тесту enums

В якості основної системи процесу безперервної інтеграції було обрано CircleCI [56], який надає безкоштовний пакет з можливістю використовувати кастомізовані середовища (образи) для тестування.

Повністю автоматизувати тестування в рамках звичайних засобів тестування неможливо, оскільки кінцевий результат являє собою графічне представлення у форматах png або svg. Зовнішній вигляд діаграми являється критерієм скоріш суб’єктивним де відсутні характеристики які можливо було б виміряти в автоматичному режимі. Тому фінальним етапом тестування являється повторення кроків користувача в ручному режимі зображених на рис. 2.2. на змодельованій або реальній базі даних.

2.7 Аналіз продуктивності та рекомендації щодо практичного використання

Для вимірювання продуктивності та споживання ресурсів ПЗ «Sqlant» було проведено експеримент з використанням утиліти «GNU Time» [57]. Було згенеровано базу даних яка містить 1000 таблиць, в кожній таблиці 10 полів та кожна таблиця містить 1 зовнішній ключ. Виконавши команду /usr/bin/time -v sqlant postgresql://sql:sql@localhost:5434/test1000 отримано результати зображені на Рисунку 2.14, а саме: загальний час виконання 0.08 секунди, а пікове використання RAM 14784 кілобайт, що приблизно 14.5МБ.

Для інструмента аналізу схеми з 1000 таблиць використання RAM в межах 15МБ являється досить незначним обсягом, а час виконання приблизний одній десятій секунди свідчить про високу ефективність алгоритму. Такі результати підтверджують що застосунок здатен працювати з великими, комплексними базами даних без потреби в виділені додаткових ресурсів.

```
Command being timed: "sqlant postgresql://sql:sql@localhost:5434/test1000"
User time (seconds): 0.05
System time (seconds): 0.00
Percent of CPU this job got: 66%
Elapsed (wall clock) time (h:mm:ss or m:ss): 0:00.08
Average shared text size (kbytes): 0
Average unshared data size (kbytes): 0
Average stack size (kbytes): 0
Average total size (kbytes): 0
Maximum resident set size (kbytes): 14784
Average resident set size (kbytes): 0
Major (requiring I/O) page faults: 0
Minor (reclaiming a frame) page faults: 2635
Voluntary context switches: 566
Involuntary context switches: 1
Swaps: 0
File system inputs: 0
File system outputs: 0
Socket messages sent: 0
Socket messages received: 0
Signals delivered: 0
Page size (bytes): 4096
Exit status: 0
```

Рисунок 2.14. – Результат вимірювання продуктивності та споживання ресурсів

ПЗ «Sqlant» (CLI) являється виконуваним файлом який не потребує встановлення додаткових бібліотек та використовує незначну кількість апаратних ресурсів, які здатні забезпечити будь-яка система з встановленою операційною системою (Linux, Windows, MacOS, тощо). Для успішного використання потрібно лише наявність з’єднання з базою даних та достатній рівень прав користувача (бази даних) для перегляду відповідних службових таблиць.

Головною ідеєю створення застосунку є полегшення аналізу схеми бази даних та автоматизації процесу їх документування. Як показав аналіз продуктивності ПЗ здатне працювати з великими базами даних. Моніторинг ресурсу Github демонструє, що розробники використовують «Sqlant» як частину CI процесу (а саме Bash/Shell сценаріїв) для автоматичної генерації діаграм після внесених змін в кодову базу. Також інструмент можна використовувати в навчальних цілях, показуючи здобувачам вищої освіти логічні та фізичні моделі.

Діаграми згенеровані в форматі png або svg не потребують спеціальних інструментів для роботи, тому кінцевими результатами виконання програми з легкістю можна обмінюватись між командами та їх членами. Також слід зауважити що діаграми доцільно включати у більші системи документування що дасть змогу прискорити передачу знань між учасниками проєкту.

ВИСНОВКИ

У роботі було розглянуто теоретичну компоненту реляційних баз даних з акцентом на модель сутність-зв’язок зі способами їх візуалізації. Також проведено дослідження особливостей використання та розробки консольних застосунків (CLI), та можливості отримання структури бази даних за допомогою SQL запитів використовуючи СУБД PostgreSQL. Проаналізовано існуючі рішення та розроблено консольну утиліту для автоматизації генерації діаграм з назвою «Sqlant».

Актуальність даної розробки виникла у зв’язку з потребою в автоматизації створення документації та покращення її актуальності. Одною з головних частин якісної документації є діаграми, зокрема діаграми бази даних які описують їх структуру. Підтвердженням актуальності є активність серед користувачів готового програмного продукту на репозиторії ресурсу Github, та понад 26 тисяч скачувань на ресурсі crates.io, що являється офіційним репозиторієм пакетів мови програмування Rust.

Важливим аспектом успішності став вибір та використання мови програмування Rust, що забезпечило надійність, високу продуктивність, кросплатформність та безпечне управління пам'яттю без негативного ефекту на продуктивність програми. Використання модульної архітектури дозволило забезпечити можливість використання більшої частини коду проєкту в якості бібліотеки (Rust), реалізувати якісне покриття автоматичними тестами та надало потенціал для подальшого розширення можливостей ПЗ як в напрямку підтримки різних СУБД, так і типів діаграм усуваючи наявні обмеження.

Основною мовою опису діаграм, що являється результатом виконуваного файлу було обрано PlantUML оскільки даний продукт має більше можливостей для розширення ніж його аналог Mermaid. Підтвердженням цього являється розроблена PlantUML бібліотека яка зменшує обсяг коду, та робить його більш читабельним та інтуїтивно зрозумілим. Хоча активність користувачів показує що вони також використовують Mermaid. Використання мов текстового опису

діаграм також сприяє можливостям аналізу діаграм та інтеграції з ШІ, що також робить дану працю актуальною.

Поставлене у роботі завдання виконано в повному обсязі, а розроблений програмний продукт «Sqlant» довів свою ефективність, надійність, портативність і практичне застосування серед спеціалістів.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Kurotych, Anatolii O., and Lesya V. Bulatetska. "Optimizing the process of ER diagram creation with PlantUML." *CS&SE@ SW*. 2024. URL: https://cssesw.easyscience.education/cssesw2024/CSSESW2024/paper12.pdf (дата звернення: 01.10.2025)
2. Drawing UML with PlantUML: PlantUML Language Reference Guide (Version 1.2025.0) [Електронний ресурс]. – 2025. – Режим доступу: https://pdf.plantuml.net/PlantUML_Language_Reference_Guide_en.pdf (дата звернення: 01.10.2025).
3. Mermaid official documentation [Електронний ресурс]. – Режим доступу: https://mermaid.js.org/intro/ (дата звернення: 02.10.2025).
4. Codd, E. F. A relational model of data for large shared data banks // *Communications of the ACM*. – 1970. – Vol. 13, No. 6. – P. 377–387. – DOI: https://doi.org/10.1145/362384.362685.
5. Ярцев В. П. Організація баз даних та знань: навч. посібник. – Київ: ДУТ, 2018. – 214 с. – Режим доступу: https://org2.knuba.edu.ua/pluginfile.php/197037/mod_folder/content/0/%D0%91%D0%94_%D0%94%D0%A3%D0%A2_compressed_compressed.pdf?forcedownload=1 (дата звернення: 03.10.2025).
6. Булатецька Л. В., Булатецький В. В. Мова запитів SQL: текст лекцій нормативної навчальної дисципліни. – Луцьк: СНУ імені Лесі Українки, 2018. – 92 с. – Режим доступу: https://evnuir.vnu.edu.ua/handle/123456789/17722 (дата звернення: 03.10.2025).
7. Лосєв М. Ю., Федько В. В. Бази даних: навчально-практичний посібник для самостійної роботи студентів. – Харків: ХНЕУ, 2018. – 233 с. – Режим доступу: https://repository.hneu.edu.ua/bitstream/123456789/21468/1/2018-Лосєв%20М%20Ю,%20Федько%20В%20В.pdf

8. Доценко С. І. Організація та системи керування базами даних: навч. посібник. – Харків: УкрДУЗТ, 2023. – 117 с. – Режим доступу: http://lib.kart.edu.ua/bitstream/123456789/13596/1/Навчальний%20посібник.pdf
9. Chen, P. P. The entity-relationship model — toward a unified view of data // *ACM Transactions on Database Systems*. – 1976. – Vol. 1, No. 1. – P. 9–36. – DOI: https://doi.org/10.1145/320434.320440
10. Capuñay Puyén, R., Quiñones Martínez, P. Working Method for Conceptual Database Modeling using Crow's Foot Notation and ER-Assistant Software to improve the process of database design in Database Management Courses at a University of Trujillo // *LACCEI International Multi-Conference*. – 2022. – DOI: 10.18687/LACCEI2022.1.1.176.
11. Mamayev, R. Barker's Notation // *Pro JPA 2 in Java EE 8*. – Berkeley, CA: Apress, 2018. – P. 15–34. – – DOI: 10.1007/978-1-4302-6590-0_2.
12. Hnatkowska, Bogumiła, and Mateusz Cebinka. "Activity diagram generation based on use-case textual specification." *Computing and Informatics* 40.4 (2021): 772-795.
13. Shin, Sun-Joo, Oliver Lemon, and John Mumma. "Diagrams." (2001).
14. Diagrams.net (Draw.io). Онлайн-сервіс для побудови діаграм [Електронний ресурс]. – Режим доступу: https://app.diagrams.net (дата звернення: 02.10.2025).
15. Lucidchart. Онлайн-платформа для побудови діаграм та візуалізації даних [Електронний ресурс]. – Режим доступу: https://www.lucidchart.com (дата звернення: 02.10.2025).
16. GitHub. Платформа для спільної розробки програмного забезпечення [Електронний ресурс]. – Режим доступу: https://github.com (дата звернення: 02.10.2025).
17. Microsoft Corporation. Microsoft Visio: програмне забезпечення для побудови діаграм та блок-схем [Електронний ресурс]. – 2024. – Режим доступу: https://www.microsoft.com/visio (дата звернення: 02.10.2025).

18. Куротич, Анатолій, Леся Булатецька, and Оксана Онищук. "RECONSTRUCTING ENTITY RELATIONSHIPS IN DATABASE SCHEMAS WITH PLANTUML AND LLMS." *Електронне фахове наукове видання «Кібербезпека: освіта, наука, техніка»* 1.29 (2025): 152-160. DOI: https://doi.org/10.28925/2663-4023.2025.29.847
19. PostgreSQL Global Development Group. PostgreSQL 16 Documentation [Електронний ресурс]. – Режим доступу: https://www.postgresql.org/docs/ (дата звернення: 03.10.2025).
20. Бази даних та засоби управління. Практикум [Електронний ресурс] / уклад.: О. В. Ткачук, О. М. Гавриленко, І. В. Гриценко ; Нац. техн. ун-т України «КПІ ім. Ігоря Сікорського». – Київ: КПІ, 2019. – 217 с. – Режим доступу: https://ela.kpi.ua/items/c25320ec-1151-49c2-871b-9df486027c2f (дата звернення: 03.10.2025).
21. PostgreSQL Global Development Group. DDL-Schemas. PostgreSQL Documentation [Електронний ресурс]. – Режим доступу: https://www.postgresql.org/docs/current/ddl-schemas.html (дата звернення: 03.10.2025).
22. achiku. planter [Електронний ресурс]. – Режим доступу: https://github.com/achiku/planter (дата звернення: 03.10.2025).
23. Open Source Initiative. The MIT License (MIT) [Електронний ресурс]. – Режим доступу: https://opensource.org/licenses/MIT (дата звернення: 03.10.2025).
24. The Go Authors. The Go Programming Language [Електронний ресурс]. – Режим доступу: https://go.dev/ (дата звернення: 03.10.2025).
25. sue445. plant_erd [Електронний ресурс]. – Режим доступу: https://github.com/sue445/plant_erd (дата звернення: 03.10.2025).
26. Kurotych, Anatolii. *sqlant* [Електронний ресурс]. – Режим доступу: https://github.com/kurotych/sqlant/releases (дата звернення: 03.10.2025).
27. The MIT License / Open Source Initiative. [Електронний ресурс]. — Режим доступу: https://opensource.org/licenses/MIT (дата звернення: 18.10.2025)

28. The Rust programming language [Електронний ресурс]. – Режим доступу: https://www.rust-lang.org/ (дата звернення: 18.10.2025).
29. Open Source Initiative [Електронний ресурс]. — Режим доступу: https://opensource.org/ (дата звернення: 18.10.2025).
30. Cargo official documentation [Електронний ресурс]. – Режим доступу: https://doc.rust-lang.org/cargo/ (дата звернення: 18.10.2025).
31. thiserror [Електронний ресурс]. – Режим доступу: https://crates.io/crates/thiserror
32. Empirical software engineering [Електронний ресурс] // Springer. – Режим доступу: https://link.springer.com/content/pdf/10.1007/s10664-021-10036-y.pdf (дата звернення: 27.10.2025).
33. Claude Code [Електронний ресурс]. – Режим доступу: https://www.claude.com/product/claude-code (дата звернення: 27.10.2025).
34. Crates.io — реєстр бібліотек спільноти Rust [Електронний ресурс]. – Режим доступу: https://crates.io (дата звернення: 01.11.2025).
35. Mozilla Foundation. Mozilla [Електронний ресурс]. – Режим доступу: https://www.mozilla.org
36. The Rust Programming Language — офіційна документація мови Rust [Електронний ресурс]. – Режим доступу: https://doc.rust-lang.org/book/ (дата звернення: 01.11.2025).
37. Why AWS Loves Rust and How We'd Like to Help — чому AWS підтримує Rust і як прагне допомогти [Електронний ресурс] // AWS Open Source Blog. – Режим доступу: https://aws.amazon.com/blogs/opensource/why-aws-loves-rust-and-how-wed-like-to-help/ (дата звернення: 01.11.2025).
38. Safer Drivers, Stronger Devices — безпечніші драйвери, надійніші пристрої [Електронний ресурс] // Microsoft Tech Community. – Режим доступу: https://techcommunity.microsoft.com/blog/surfaceitpro/safer-drivers-stronger-devices/4431411 (дата звернення: 01.11.2025).
39. Kurotych A. Sqlant: Pull Request #27 [Електронний ресурс]. – Режим доступу: https://github.com/kurotych/sqlant/pull/27 (дата звернення: 01.11.2025).

40. Clap [Електронний ресурс]. – Режим доступу: https://crates.io/crates/clap (дата звернення: 01.11.2025).
41. Native-tls [Електронний ресурс]. – Режим доступу: https://crates.io/crates/native-tls (дата звернення: 01.11.2025).
42. Postgres-native-tls [Електронний ресурс]. – Режим доступу: https://crates.io/crates/postgres-native-tls (дата звернення: 01.11.2025).
43. Serde [Електронний ресурс]. – Режим доступу: https://crates.io/crates/serde (дата звернення: 01.11.2025).
44. Thiserror [Електронний ресурс]. – Режим доступу: https://crates.io/crates/thiserror (дата звернення: 01.11.2025).
45. Tinytemplate [Електронний ресурс]. – Режим доступу: https://crates.io/crates/tinytemplate (дата звернення: 01.11.2025).
46. Tokio [Електронний ресурс]. – Режим доступу: https://crates.io/crates/tokio(дата звернення: 01.11.2025).
47. Tokio-postgres [Електронний ресурс]. – Режим доступу: https://crates.io/crates/tokio-postgres (дата звернення: 01.11.2025).
48. Neovim [Електронний ресурс]. – Режим доступу: https://neovim.io/ (дата звернення: 01.11.2025).
49. Vim [Електронний ресурс]. – Режим доступу: https://www.vim.org/ (дата звернення: 01.11.2025).
50. Lua [Електронний ресурс]. – Режим доступу: https://www.lua.org/ (дата звернення: 01.11.2025).
51. Language Server Protocol: Specification v3.17 [Електронний ресурс]. – Режим доступу: https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification/
52. PostgreSQL Documentation: psql – Interactive Terminal [Електронний ресурс]. – Режим доступу: https://www.postgresql.org/docs/current/app-psql.html (дата звернення: 01.11.2025).
53. Cargo: The Rust Package Manager [Електронний ресурс]. – Режим доступу: https://doc.rust-lang.org/cargo/ (дата звернення: 01.11.2025).

54. Docker [Електронний ресурс]. — Режим доступу: https://www.docker.com/ (дата звернення: 09.11.2025)
55. NPM [Електронний ресурс]. — Режим доступу: https://www.npmjs.com/ (дата звернення: 09.11.2025)
56. CircleCI [Електронний ресурс]. — Режим доступу: https://circleci.com/ (дата звернення: 09.11.2025)
57. GNU Time. GNU Operating System. URL: https://www.gnu.org/software/time/ (дата звернення: 15.11.2025).

ДОДАТКИ

ДОДАТОК А.

Технічне завдання на розробку програмного засобу «Sqlant» — утиліти для автоматизованої генерації ER-діаграм для реляційних баз даних

1. Підстава для розробки

Розробка ведеться на підставі написання кваліфікаційної роботи.

2. Мета розробки

Метою розробки є створення програмного засобу, що автоматизує генерацію ER діаграм (Entity-Relationship Diagrams) для реляційних баз даних використовуючи рядок підключення (connection string) як вхідні дані.

3. Завдання розробки

- розробити програму яка реалізує інтерфейс командного рядка (CLI) з інтуїтивним інтерфейсом;
- забезпечити підключення до бази даних PostgreSQL;
- реалізувати аналіз структури БД;
- забезпечити генерацію ER діаграм в форматах: PlantUML, Mermaid;
- передбачити опції командного рядка для конфігурації результату, такі як: вибір схеми, додавання легенди, формат вихідного файлу, властивості полів, тощо;
- забезпечити модульну архітектуру з можливістю розширення функціональності;
- реалізувати автоматичне тестування за допомогою GithubCI;
- реалізувати наступні способи розповсюдження:
    1. бінарний файл на ресурсі Github;
    2. докер контейнер з встановленим бінарним файлом;
    3. добавити проєкт в менеджер пакетів (crates.io), для можливості установки за допомогою утиліти «cargo».

4. Функціональні вимоги

- програма повинна приймати рядок підключення у стандартному форматі для підтримуваних СУБД (PostgreSQL, SQLite, MySQL тощо);
- при помилці автентифікації або недоступності сервера система має виводити зрозуміле повідомлення про помилку;
- система повинна аналізувати схему бази даних, включно з: назвами таблиць, полями (імена, типи даних, обмеження), первинними та зовнішніми ключами;
- для PostgreSQL реалізувати підтримку View, Materialized View, та Enum;
- підтримка концептуального та фізичного рівня представлення діаграм;
- результатом виконання програми повинен бути текстовий опис ER-діаграми у форматах: PlantUML, Mermaid;
- можливість впливати на результат виконання за допомогою параметрів командного рядка (адаптація під потреби користувача);
- система повинна автоматично визначати тип зв’язку між таблицями (1:1, 1:N, M:N) на основі зовнішніх ключів.

5. Нефункціональні вимоги

- Модульна архітектура. Реалізація (код) повинна забезпечувати просте впровадження підтримки додаткових нотацій та баз даних;
- можливість використовувати код проєкту як бібліотеку мови програмування (Rust);
- Інтерфейс командного рядка повинен бути інтуїтивно зрозумілим і мати довідку (--help);
- Базова функціональність повинна бути покрита автоматичним тестуванням;
- Безперервна Інтеграція. Забезпечити автоматичну збірку, тестування та перевірку якості коду;

- Підтримка базових ОС: Linux, MacOS, Windows

6. Етапи розробки

1. Аналіз вимог і вибір інструментів.
2. Проєктування архітектури ПЗ.
3. Реалізація основного функціоналу (CLI, аналіз схеми, генерація діаграми).
4. Тестування та оптимізація.
5. Розробка CI сценаріїв.
6. Підготовка технічної документації.

ДОДАТОК Б.

Інструкція користувачу

1. Загальні відомості

Програмний засіб реалізований в якості консольного (CLI) додатку та програмної бібліотеки написаною мовою програмування Rust для генерації текстового опису ER-діаграм на основі рядка підключення до бази даних – «Sqlant».

2. Функціональне призначення

Призначений для автоматизації аналізу структури бази даних, побудови логічної та/або концептуальної моделей та їх візуального представлення.

За допомогою даного ПЗ можна виконувати наступні завдання:

- Автоматичне отримання метаданих БД: таблиці, колонки та їх обмеження, типи даних, первині та вторинні ключі з побудовою внутрішньої моделі яка відображає її структуру (для використання в якості бібліотеки);
- Генерація текстового опису діаграм в форматі PlantUML/Mermaid на основі рядка підключення до БД;
- Інтеграція автоматизованої візуалізації схем в документації програмних продуктів за допомогою CLI який зручно використовувати в побудові конвеєрів CI/CD.

3. Умови застосування програми

Користувачі використовують sqlant на таких ОС як: GNU/Linux, MacOS, Windows використання на інших ОС (наприклад FreeBSD) можливе, але не було протестоване. Виконуваний файл версії 0.7 займає 6.3 MB дискового простору. Обсяг потрібної оперативної пам’яті залежить від розміру схеми баз даних з

якою ви працюєте. Для прикладу sqlant використовує менше ніж 1 MB оперативної пам’яті для генерації текстового опису схеми яка має 10 таблиць по 5 полів.

4. Повідомлення оператору

Можливі повідомлення помилок:

- database "sqlw" does not exist. База даних яку ви передали в рядку підключення не існує;
- Connection refused. Неможливо приєднатись до сервера бази даних;
- "Schema doesn't exist". Схеми яку ви передали за допомогою опції -s не існує.

5. Опис роботи програми

Для отримання довідки про наявні можливості вашої версії програми введіть команду sqlant --help:

```
❯ ./target/debug/sqlant --help
Generate Entity Relationship diagram textual description from SQL connection string

Usage: sqlant [OPTIONS] <connection_string>

Arguments:
  <connection_string>

Options:
      --inline-puml-lib        Inline PlantUML lib into diagram code
      --legend                 Add legend to diagram (supported only for PlantUML)
  -n, --nn                     Add NOT_NULL(NN) marks (for PlantUML always true)
  -e, --en                     Draw enum types
  -s, --schema <schema>        Schema name [default: public]
  -o, --output <output>        Generate output in mermaid format [default: plantuml] [possible values: plantuml, mermaid]
      --conceptual             Create conceptual ER diagram
      --direction <direction>  Optionally set the direction of the generated diagram [possible values: tb, bt, lr, rl]
  -h, --help                   Print help
  -V, --version                Print version
```

Для отримання текстового опису діаграми з налаштуваннями по замовчуванню вистачить лише одного командного аргументу, а саме рядка підключення. По замовчуванню буде згенерований текстовий опис діаграми в форматі PlantUML.

```
❯ sqlant postgresql://sql:sql@localhost:5434/sql
@startuml

hide circle
hide empty members
skinparam linetype ortho

!include https://raw.githubusercontent.com/kurotych/sqlant/9b19d6691b55c838b0809ed66707e61533a4c9f2/puml-lib/db_ent.puml

table(customer) {
  column(id, "bigint", $pk=true, $nn=true)
  ---
  column(address, "text", $nn=true)
  column(name, "text", $nn=true)
  column(phone_number, "text", $nn=true)
  column(registered_at, "timestamp with time zone", $nn=true)
  column(zip_code, "text", $nn=true)
}

table(customer_order) {
  column(id, "bigint", $pk=true, $nn=true)
  ---
  column(customer_id, "bigint", $fk=true, $nn=true)
  column(delivery_method, "text", $nn=true)
  column(ordered_at, "timestamp with time zone", $nn=true)
  column(payment_method, "text", $nn=true)
  column(shipping_address, "text", $nn=true)
  column(total_price, "numeric", $nn=true)
  column(total_tax_amount, "numeric", $nn=true)
}
```

Анотація

Куротич А.О. – Розробка системи автоматизованого генерування ER-діаграм для реляційних баз даних.

Кваліфікаційна робота на здобуття освітнього ступеня «магістр» за спеціальністю 122 Комп'ютерні науки, освітньої програми Комп'ютерні науки та інформаційні технології. – Волинський національний університет імені Лесі Українки. – 2025 р.

У роботі розглянуто методологію та інструменти для побудови діаграм типу сутність-зв'язок включаючи опис компонентів, рівні абстракції та варіанти нотацій. Виділено два підходи до генерації ER-діаграм, де зроблено акцент на підході типу: «Діаграма як код». Продемонстровано спосіб отримання структури баз даних у PostgreSQL за допомогою SQL запитів та аналіз особливостей використання CLI (Command Line Interface) додатків. Було проведено дослідження можливостей PlantUML для відображення діаграм сутність-зв'язок у контексті реляційних баз даних, результатом чого стала PlantUML бібліотека яка спрощує процес написання діаграм та підвищує їх якість.

Кінцевим результатом роботи є пакет ПЗ «Sqlant», який складається з CLI утиліти та бібліотеки для мови програмування Rust. Основною функціональністю є автоматизована генерація текстового опису (PlantUML та Mermaid) ER-діаграм для реляційних баз даних на основі рядка підключення. Використання консольної утиліти «Sqlant» має позитивний вплив на швидкість створення діаграм та їх якість, а результат в форматі текстового опису сприяє аналізу структури баз даних за допомогою LLMs.

Розробка велась з урахуванням передових практик процесу розробки ПЗ. Безперервна інтеграція, автоматизоване тестування, та статичний аналіз коду забезпечили проведення контролю якості при кожному внесенні змін, а чітка модульна архітектура спростила підтримку та розширення функціональності.

Ключові слова: ERD, PostgreSQL, PlantUML, Rust, CLI, Sqlant

Abstract

Kurotych A.O. – Development of an automated ER diagram generation system for relational databases.

Manuscript. Qualification work for the degree of "Master" in the field of Computer Science, educational program "Computer Science and Information Technologies." – Lesya Ukrainka Volyn National University. – 2025.

The work examines the methodology and instruments for building ER diagrams including component descriptions, levels of abstraction, and notation variants. Two approaches for ER diagram generation are highlighted, with emphasis placed on the approach named “Diagram as Code”. The method of obtaining database structure in PostgreSQL using SQL queries has been demonstrated and an analysis of CLI (Command Line Interface) applications has been performed. The investigation of PlantUML ER-diagram generation capabilities in the context of relational databases has been performed, which resulted in a PlantUML library that simplifies the process of writing diagrams and enhances their quality.

The final result is the software “Sqlant”, which consists of a CLI utility and a library for the Rust programming language. Its basic functionality is the automated generation of PlantUML and Mermaid textual descriptions of ER-diagrams for relational databases based on a connection string. The use of CLI “Sqlant” has a positive impact on the speed of diagram creation and their quality, and a textual description format facilitates database analysis by LLMs.

The latest software development approaches have been taken into account during the development of the program. Continuous integration, automated testing and static code analysis ensured quality control during each commit, and a modular architecture simplified program support and functionality extension.

Keywords: ERD, PostgreSQL, PlantUML, Rust, CLI, Sqlant