ДЕРЖАВНИЙ УНІВЕРСИТЕТ ТЕЛЕКОМУНІКАЦІЙ

Навчально-науковий інститут Інформаційних технологій

Кафедра інженерії програмного забезпечення

Пояснювальна записка

до бакалаврської роботи
на ступінь вищої освіти бакалавр
на тему: «**РОЗРОБКА ПРОФАЙЛЕРУ ПРОДУКТИВНОСТІ SQL-ЗАПИТІВ МОВОЮ C#»**

Виконав: студент 4 курсу, групи ПД–41
спеціальності
121 Інженерія програмного забезпечення
(шифр і назва спеціальності)

Бондаренко В.В
(прізвище та ініціали)

Керівник Гаманюк І.М
(прізвище та ініціали)

Рецензент
(прізвище та ініціали)

Київ – 2023

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ТЕЛЕКОМУНІКАЦІЙ

Навчально-науковий інститут Інформаційних технологій

Кафедра Інженерії програмного забезпечення
Ступінь вищої освіти - «Бакалавр»
Напрям підготовки -124 - Інженерії програмного забезпечення"

ЗАТВЕРДЖУЮ

Завідувач кафедри
Інженерії програмного забезпечення
О.В Негоденко
—‖2023 року

З А В Д А Н Н Я

НА БАКАЛАВРСЬКУ РОБОТУ СТУДЕНТУ

БОНДАРЕНКА ВОЛОДИМИРА ВАЛЕРІЙОВИЧА
(прізвище, ім'я, по батькові)

1. Тема роботи: «Розробка профайлеру продуктивності SQL-запитів мовою C#»

Керівник роботи Гаманюк Ігор Миколайович, старший викладач кафедри
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом вищого навчального закладу від —24 ‖лютого 2023 року №26.

2. Строк подання студентом роботи 01 червня 2023 року
3. Вхідні дані до роботи:
   науково-технічна література, пов'язана з аналізом продуктивності SQL-запитів, мова програмування C#, .NET6, система управління базами даних SqlServer, система управління базами даних PostgreSQL.
4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити).
   4.1 Аналіз існуючих систем для проведення навантажувального тестування
   4.2 Вибір засобів розробки
   4.3 Розробка програмного продукту
   4.5 Тестування застосунку

5. Перелік графічного матеріалу
   5.1. Мета, об'єкт та предмет дослідження
   5.2. Задачі дипломної роботи
   5.3. Аналіз аналогів
   5.4. Вимоги до програмного забезпечення
   5.5. Програмні засоби реалізації
   5.6. Діаграма прецедентів

5.7. Діаграма діяльності
5.8. Діаграма класів
5.9. Діаграма послідовності
5.10. Діаграма станів для моделювання роботи меню
5.11. Екранні форми
5.12. Демонстрація роботи
5.13. Апробація результатів дослідження
5.14. Висновки

6. Дата видачі завдання 25 лютого 2023 року

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів бакалаврської роботи	Строк виконання етапів роботи	Примітка
1	Підбір науково-технічної літератури	25.02.2023 - 26.02.2023	Виконано
2	Аналіз існуючих систем	27.02.2023 – 28.02.2023	Виконано
3	Підбір засобів розробки	01.03.2023 – 02.03.2023	Виконано
4	Визначення вимог	03.03.2023 – 04.03.2023	Виконано
5	Визначення архітектури застосунку	05.02.2023 – 07.02.2023	Виконано
6	Розробка застосунку	08.02.2023 – 20.04.2023	Виконано
7	Вступ, висновки, реферат	01.05.2023 – 05.05.2023	Виконано
8	Розробка демонстраційних матеріалів	06.05.2023 – 21.05.2023	Виконано
9	Попередній захист роботи	22.05.2023	Виконано
10	Подання роботи в деканат	01.06.2023	Виконано

Студент _____ Бондаренко В.В
(підпис) (прізвище та ініціали)

Керівник роботи _____ Гаманюк І.М
підпис) (прізвище та ініціали)

РЕФЕРАТ

Текстова частина бакалаврської роботи: 59с., містить 48 рис., 1 табл., 7 дод., 9 джерел.

SQL, SQLSERVER, POSTGRESQL, .NET, НАВАНТАЖУВАЛЬНЕ ТЕСТУВАННЯ.

Об’єкт дослідження – процес тестування застосунку.

Предмет дослідження – тестування запитів SQL.

Мета роботи – покращення працездатності застосунку шляхом використання профайлеру продуктивності SQL-запитів.

Методи дослідження – робота виконувалась по аналогії з існуючим рішенням для навантажувального тестування – SQLQueryStress.

Результатом даної роботи є програмне рішення, що дозволяє поєднувати декілька СУБД в рамках одної системи, на яких можна проводити навантажувальне тестування SQL-запитів. Також в ході реалізації застосунку було розроблено сценарну модель виконання навантаження, яка робить реалізацію більш гнучкою та дає можливість розширювати список сценаріїв навантаження.

Результати виконаної роботи можуть бути використані розробниками для проведення навантажувального тестування SQL команд для таких СУБД: SqlServer, PostgreSQL.

В подальших дослідженнях слід розширити перелік СУБД для яких можна проводити навантажувальне тестування, перелік сценаріїв навантаження, ґрунтуючись на отриманих відгуках користувачів. Також необхідно розробити можливість способу взаємодії з застосунком через CLI (інтерфейс командного рядка), виконуючи команди через термінал з використанням інтерактивних запитів командного рядка. Крім того, слід додати можливість вивантажувати отримані результати в вихідні файли: excel, word, txt, для забезпечення можливості збереження результатів тестування та їх аналіз в подальшому.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

База даних - це впорядкований набір структурованої інформації або даних, які зазвичай зберігаються в електронному вигляді комп'ютерної системи. База даних зазвичай управляється системою управління базами даних (СУБД).

СУБД (система управління базами даних) - спеціалізована програма (частіше комплекс програм), призначена для організації і ведення бази даних.

Фреймворк - програмна платформа, що визначає структуру програмної системи; програмне забезпечення, що полегшує розробку та об'єднання різних компонентів великого програмного проекту.

SQL - декларативна мова програмування, що використовується для створення, модифікації та управління даними в реляційній базі даних, що керується відповідною системою управління базами даних.

Кросплатформність (міжплатформність) — здатність програмного забезпечення працювати з кількома апаратними платформами або операційними системами.

Модульне тестування, іноді блочне тестування або юніт-тестування - процес у програмуванні, що дозволяє перевірити на коректність окремі модулі вихідного коду програми, набори з одного або більше програмних модулів разом із відповідними керуючими даними, процедурами використання та обробки.

Мануальне тестування – частина процесу тестування на етапі контролю якості у процесі розробки програмного забезпечення. Воно проводиться тестувальником без використання програмних засобів для перевірки програми або сайту шляхом моделювання дій користувача.

Microsoft Store - компонент операційної системи Windows, починаючи з Windows 8, призначений для покупки та завантаження Metro/UWP-застосунків, ігор, фільмів, теми робочого столу.

ВСТУП

Проблеми продуктивності виконання SQL команд турбує розробників з того часу, як компанією Oracle представила першу комерційно доступну реалізацію SQL у 1979-му році. Багато проблем продуктивності та швидкодії з того часу були вирішені, тим не менш, проблеми з продуктивністю все ще є звичайною справою, та інструменти, які допомагають ці проблеми вирішувати є популярними серед спільноти розробників [1].

Проблеми, які можуть впливати час виконання запитів можуть бути наступні:

- Запит написаний не неефективним чином та містить вузькі місця, які впливають на час виконання.
- Відсутність індексування
- Проблеми на рівні обладнання
- Проблеми масштабованості бази даних

Всі наведені приклади, зазвичай, починають створювати проблеми при збільшенні навантаження на систему, коли сервер баз даних може обробляти циклічні запити, які надходять від великої кількості користувачів одночасно.

Тестування продуктивності SQL-запитів є надзвичайно корисною практикою для розробників, яка може принести значну користь в процесі тестування написаних запитів. Воно може використовуватися для різних цілей, таких як оцінка продуктивності виконання запитів, планування ресурсів та виявлення проблем з продуктивністю. Стрес-тестування запиту, також відоме як навантажувальне тестування, полягає у виконанні запитів з високою частотою з різних з'єднань або потоків до бази даних. Його головною метою є визначення того, наскільки добре сервер бази даних впорається з високим рівнем навантаження, що створюється під час виконання запиту підчас стрес-тестування. Навантажувальне тестування передбачає більше, ніж просте виконання запитів до сервера баз даних. Крім цього, таке тестування має надавати розробникам статистику щодо продуктивності виконання навантаження, таку як фактичний час виконання запиту, середній час

виконання команд, медіана, дисперсія, кількість читань з бази даних, процесорний час, тощо. Статистика може бути представлена у вигляді числових значень, діаграм або графіків. В ході тестування важливо аналізувати дані, отримані підчас виконання кожного сценарію [2].

Стрес-тестування дозволяє виявити та виправити можливі проблеми, такі як:

– Помилки в програмному забезпеченні, які можуть виникати при великому навантаженні;
– Проблеми зі швидкістю роботи системи, що можуть впливати на її продуктивність;
– Проблеми зі шкалою системи, що можуть призвести до перевантаження бази даних або серверів.

Підчас проведення навантажувального тестування розробнику необхідно переконатися, що SQL-команда вірно функціонує за умов високого навантаження і не призводить до витоку пам'яті або системних збоїв. Крім того, розробник повинен зосередитися на оптимізації запиту та покращенні продуктивності запиту, якщо це необхідно [2].

Об’єкт дослідження – процес тестування застосунку.

Предмет роботи – тестування запитів SQL.

Мета роботи – покращення працездатності застосунку шляхом використання профайлеру продуктивності SQL-запитів. Для досягнення цієї мети необхідно розв’язати наступні задачі:

– Провести огляд існуючих рішень, що надають можливості проведення навантажувального тестування. Описати переваги та недоліки різних програмних рішень та обґрунтувати актуальність створення нового програмного засобу для вирішення поставленої задачі.
– Необхідно вибрати інструментарій для розробки програми, зокрема мову програмування та додаткові інструменти, що сприятимуть процесу розробки. Обґрунтувати вибір засобів.
– Прийняти архітектурні рішення щодо структури проекту.

– Визначити інтерфейс взаємодії користувача з застосунком.
– Розробити програмне забезпечення. Описати впроваджений функціонал.
– Виконати тестування застосунку: забезпечити модульне тестування всіх сценаріїв навантаження та провести мануальне тестування всього застосунку.

Робота виконувалась по аналогії з існуючим рішенням для навантажувального тестування – SQLQueryStress. На основі цього застосунку був проведений аналіз недоліків та було розроблено рішення, яким чином ці недоліки можна усунути при реалізації даної роботи. Також був проведений аналіз декількох інших систем для визначення функціональних вимог.

Результати виконаної роботи можуть бути використані розробниками для проведення навантажувального тестування SQL команд для таких СУБД: SqlServer, PostgreSQL.

1 АНАЛІЗ ІСНУЮЧИХ СИСТЕМ ДЛЯ ПРОВЕДЕННЯ НАВАНТАЖУВАЛЬНОГО ТЕСТУВАННЯ

1.1 SQLQueryStress

SQLQueryStress є простим і легким інструментом для тестування продуктивності SQL-запитів, який призначений для випробування навантаження окремих запитів. Він дозволяє використовувати рандомізацію вхідних параметрів, щоб перевірити повторюваність кешування, і має базові можливості для звітування про використання ресурсів сервера [3]. На рисунку 1.1.1 показано головний екран програми.

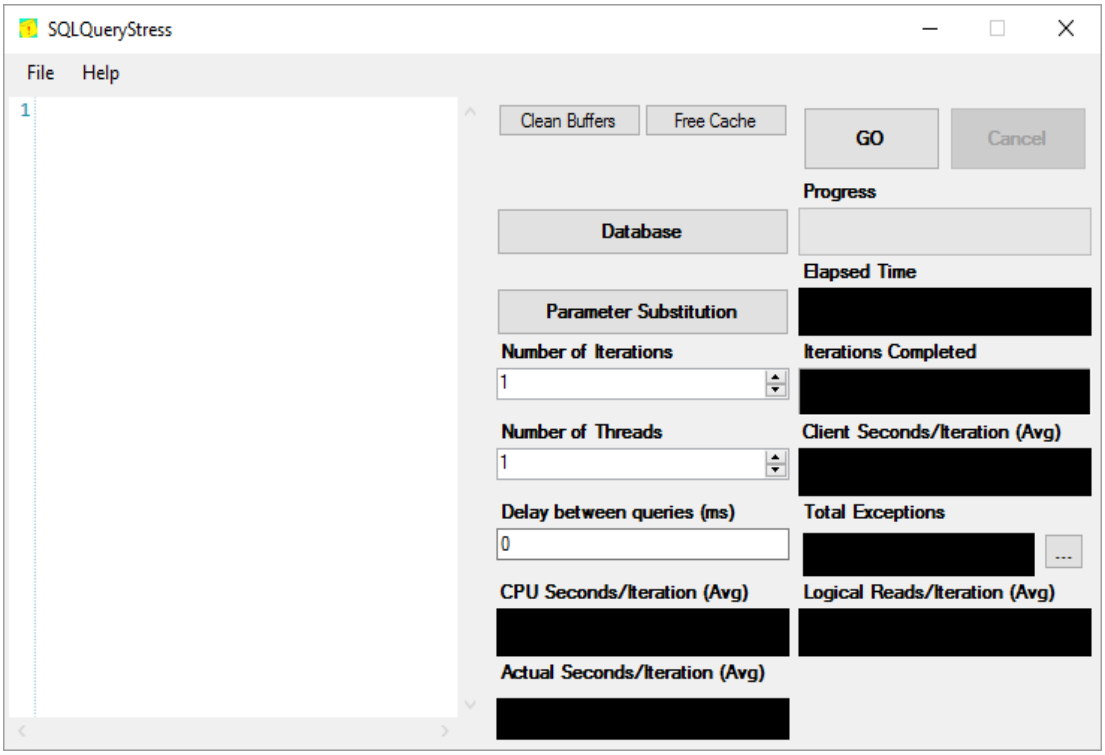


Рисунок 1.1.1 — SQLQueryStress (головний екран)

Область запиту – це місце, де можна ввести запит, який потрібно навантажити для перевірки. Це може бути або запит T-SQL, або виклик збереженої процедури. Також можливо включити імена змінних, які можна використовувати для динамічної підстановки параметрів.

У полі «Кількість ітерацій» можна вказати, скільки разів запит має виконуватися. В полі «Кількість потоків» можна вказати до 200 потоків, які виконуватимуть запит одночасно, щоб імітувати навантаження.

Інструмент збирає та відображає три типи статистики часу. Поле Client Seconds/Iteration (Клієнтські секунди/ітерація) відображає середній час роботи за всі ітерації, як це було записано на клієнті. Поля «Секунди процесора/ітерація» та «Фактичні секунди/ітерація» відображають статистику часу, яку повідомляє SQL Server. Перше – це середній зареєстрований час ЦП, а друге – середній загальний зареєстрований час запиту. Інша статистика, зібрана та показана інструментом — це кількість логічних читань (що є об'єднанням буферного кешу та читання диска) у полі «Логічні читання/Ітерація». Зверху головного екрана знаходяться дві кнопки: Clean Buffers, Free Cache. Вони необхідні для очистки кешу, який згенерувала СУБД [4].

З меню «Файл» на головному екрані можна увійти в діалогове вікно «Параметри», показане на рисунку 1.1.2. Ці параметри дозволяють контролювати деякі параметри тесту, щоб імітувати різні налаштування:

- Якщо змінити параметр «Тайм-аут підключення», інструмент довше чекає, перш ніж повідомити про виняткову ситуацію, якщо цільовий сервер не відповідає.
- Пул з'єднань можна вимкнути, щоб показати вплив створення та знищення нового з'єднання на кожній ітерації тесту.
- Зміна параметра «Час очікування команди» закінчить виконання запиту, якщо він не завершиться вчасно.
- Зміна параметрів «Збирати статистику вводу-виводу» та «Збирати статистику часу» призведе до того, що інструмент не збиратиме дані про час сервера (ЦП і фактичний) і статистику читань. Це зробить пробіг дещо легшим з точки зору використання ресурсів.
- Нарешті, параметр «Примусове отримання клієнтом даних» змушує інструмент циклічно переглядати всі дані, повернуті запитом, таким чином гарантуючи, що вони надсилаються через SQL Server. Якщо не

встановити цей параметр, існує ймовірність того, що більшість даних може залишатися в буфері в SQL Server, особливо з великими запитами, не створюючи реального навантаження.

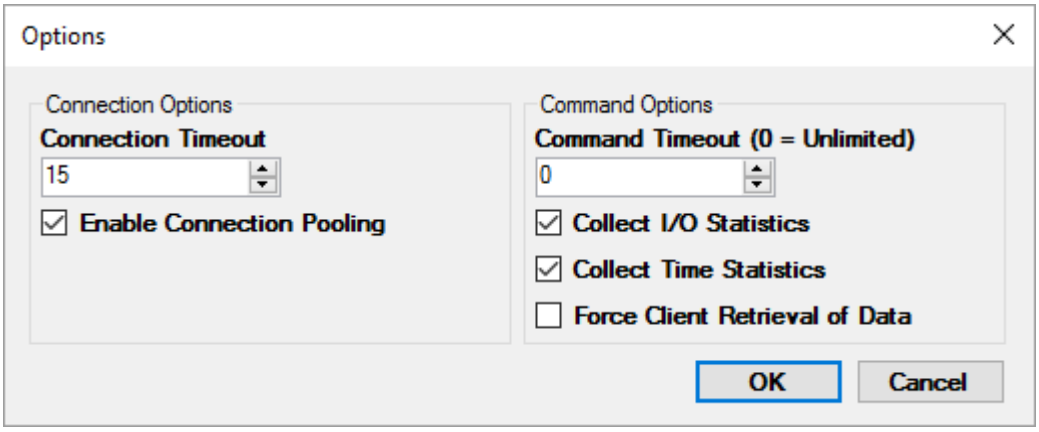


Рисунок 1.1.2 — SQLQueryStress (Екран параметрів підключення)

Плюси застосунку:

– Зручний редактор команди, який виділяє кольором ключові слова в команді, рахує рядки команди;
– Можливість вивільняти кеш/очищати буфер СУБД;
– Доволі зрозумілий інтерфейс застосунку;
– Можливість збереження/імпорту/експорту налаштувань з’єднання;
– Наявність CLI;

Мінуси застосунку:

– Відсутність можливості додавання закладок з командами, для можливості проведення навантаження для декількох команд одночасно;
– Застосунок може виконувати навантаження тільки для Microsoft SQL Server;
– Сценарії навантаження жорстко обмеженні параметрами запиту, які заповнюються (кількість ітерацій, потоків, час затримки). Розширення застосунка новими параметрами є важким через обмеженість архітектури
– Відсутність надання результату навантаження в графічному виді;

1.2 SQLTest

SQLTest - це простий у використанні інструмент для створення реального робочого навантаження для тестування. Його можна використовувати як локально, так і в хмарі [5]. На рисунку 1.2.1 показано головний екран програми.

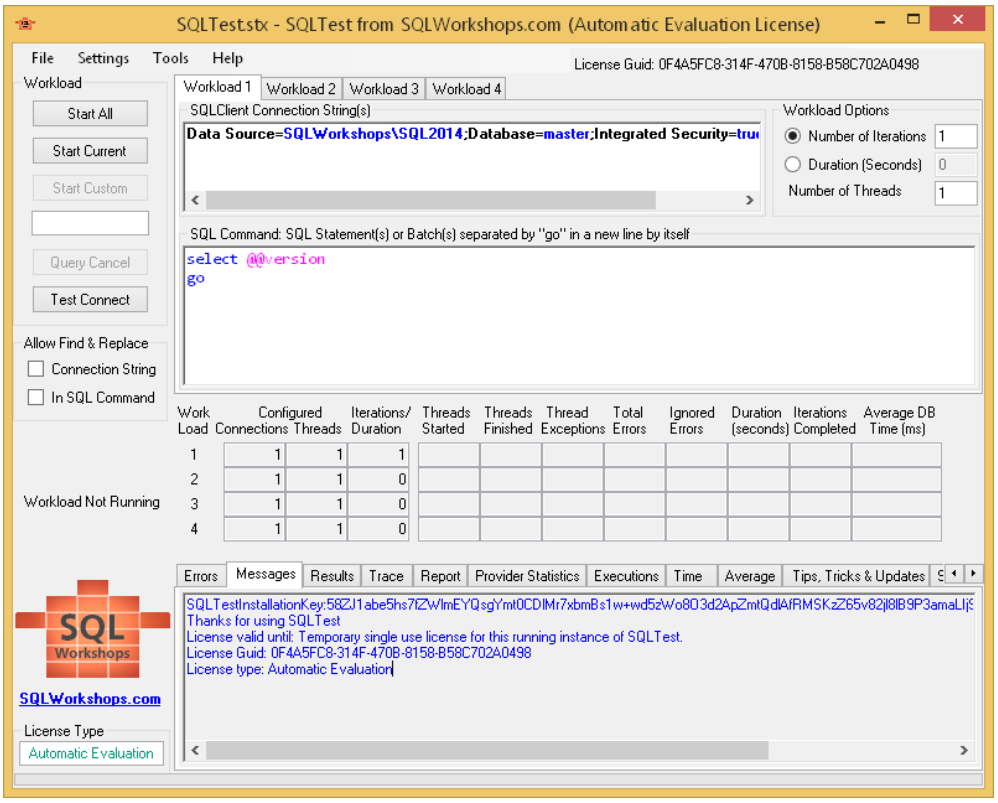


Рисунок 1.2.1 — SQLTest (головний екран)

SQLTest може генерувати виклики бази даних SQL Server, схожі на запити від реальної програми. Після налаштування тестів із використанням параметризації та функції підстановки значень тести можна динамічно масштабувати для імітації будь-якої кількості користувачів або транзакцій. Його можна використовувати для тестування продуктивності та паралельності SQL Server для вимірювання часу відповіді та аналізу проблем із блокуванням і взаємоблокуванням [5].

На відміну від SQLQueryStress, застосунок має декілька закладки для паралельного навантаження кількох команд, але їх кількість обмежена чотирма закладками.

Параметри навантаження містять вибір Duration (Seconds), який вказує не кількість ітерацій виконання, а обмеження їх виконання по часу. Але параметр, який додасть затримку між виконанням команди відсутній.

В SQLTest наявний великий перелік інформації про результат виконання навантаження, який розміщений внизу екрана. Наприклад:

– Provider Statistics - статистика виконання від провайдера;
– Trace - трейс виконання запитів;
– Results - результати, які могла повернути виконання команди;
– Time – час виконання навантаження;
– Average – середній час виконання команди;

На відміну від SQLQueryStress цей застосунок показує інформацію про виконання в режимі реального часу в таблиці, яка розташована посередині головного екрану, але не має можливості припинити навантаження вручну, після початку. SQLTest містить великий перелік параметрів налаштування, які знаходяться на екрані налаштувань.

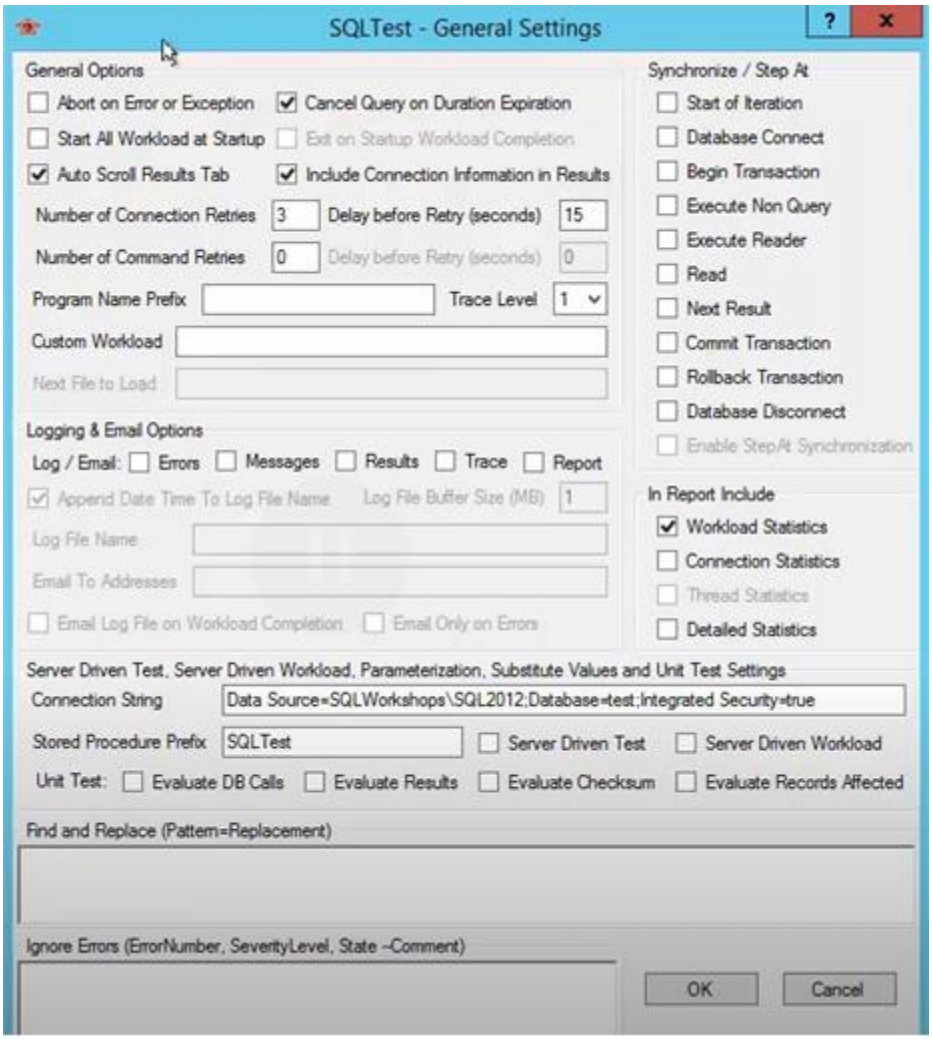


Рисунок 1.2.2 — SQLTest (екран налаштувань)

З параметрів, які слід відзначити, можна назвати:

– Logging Options – дозволяє писати логи виконання навантаження в вказаний файл;
– Можливість включати в звітність по навантаженню різні види статистики: Workload Statistics, Connection Statistics, Thread Statistics, Detailed Statistics;
– Ignore Errors – можна вказати номер помилки, які потрібно ігнорувати при навантаженні;

Плюси застосунку:

– Редактор команди, який виділяє кольором ключові слова в команді, рахує рядки команди;
– Наявність закладок, що дозволяють проводити тестування декількох запитів;

– Велика кількість різноманітної статистики, яка надається;
– Можливість отримати логи по навантаженню у вказаний файл;

Мінуси застосунку:

– Застосунок може виконувати навантаження тільки для Microsoft SQL Server;
– Сценарії навантаження жорстко обмеженні параметрами запиту, які заповнюються (кількість ітерацій, потоків, обмеження по часу);
– Розширення застосунку новими параметрами є важким через обмеженість архітектури;
– Відсутність надання результату навантаження в графічному виді;

1.3 Порівняння існуючих програм-аналогів

Для порівняння наведених програм-аналогів наведена таблиця 1.3.1.

Таблиця 1.3.1 – Порівняння існуючих програм-аналогів

	SQLQueryStress	SQLTest
Можливості додавання закладок з командами	Ні	Так
Можливість вибору сценаріїв навантаження	Ні	Ні
Надання результату навантаження в графічному виді	Ні	Ні
Редактор SQL команд	Так	Так
Можливість вивільняти кеш/очищати буфер СУБД	Так	Ні

Продовження таблиці 1.3.1 – Порівняння існуючих програм-аналогів

Наявність CLI	Так	Ні
Можливість навантаження різних СУБД	Ні	Ні

Перелік функціональних вимог до програмної системи, створений на основі аналізу існуючих систем:

- Можливості додавання закладок з командами;
- Можливість вибору сценаріїв навантаження;
- Сценарії навантаження, які потрібно реалізувати в рамках даної роботи: Sequential (послідовний), Parallel (паралельний), SequentialWithDelay (послідовний, з заданим часом затримки між запитами), SequentialWithTimeLimit (послідовний, з заданим обмеженням в часі виконання);
- Наявність редактору SQL команд;
- Можливість вивільняти кеш/очищати буфер СУБД;
- Можливість навантаження різних СУБД;
- СУБД для яких потрібно реалізувати в рамках даної роботи: SqlServer, PostgreSQL;
- Можливість перервати виконання навантаження;
- Програма повинна збирати статистику по трьом значенням: Elapsed Time (час виконання команди), CPU Time (процесорний час), Logical Reads (кількість читань з диску), та надавати їх у вигляді наступних чисельних показників: Total (загальна сума), Average (середнє арифметичне), Median (медіана), Standard Deviation (середньоквадратичне відхилення);
- Надання результату навантаження як в чисельному, так і в графічному виді;

2 ВИБІР ЗАСОБІВ РОЗРОБКИ

Було вирішено розробити десктопний застосунок використовуючи C#, як мову програмування та .NET стек.

Інтерфейс застосунку буде розроблений за допомогою WPF, використовуючи фреймворк MvvmCross, який передбачає використання шаблону проектування архітектури MVVM для побудови користувацького інтерфейсу. MVVM означає "Model-View-ViewModel" і є архітектурним шаблоном для розробки програмного забезпечення. У цій архітектурі модель (Model) представляє дані та бізнес-логіку застосунку, представлення (View) відображає інтерфейс користувача, а в'ю-модель (ViewModel) діє як посередник між моделлю та представленням, забезпечуючи обмін даними та змінами між ними. MVVM дозволяє відокремлювати бізнес-логіку від користувацького інтерфейсу та спрощує тестування та розширення застосунку. Він часто використовується у розробці програмного забезпечення для платформи .NET, таких як WPF, Silverlight і Xamarin. На рисунку 2.1 зображено взаємодію компонентів шаблона проектування MVVM.

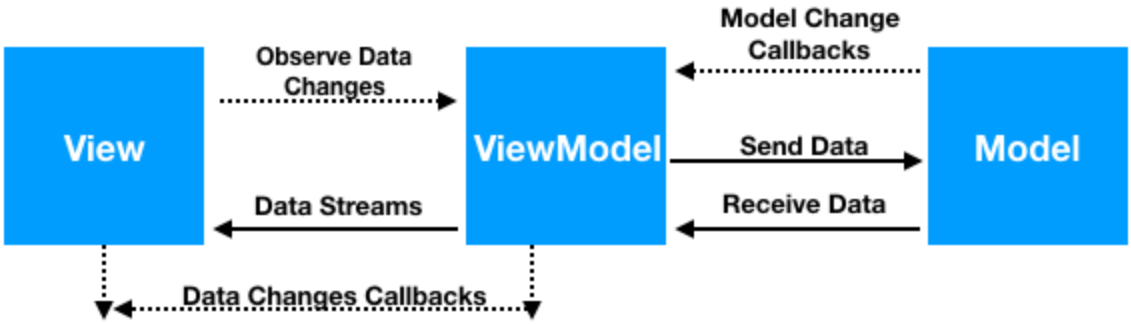


Рисунок 2.1 – Ілюстрація шаблону MVVM

MvvmCross — це кросплатформна платформа MVVM, яка дозволяє розробникам створювати потужні кросплатформні програми. Він підтримує Xamarin.iOS, Xamarin.Android, Xamarin.Mac, Xamarin.Forms, Universal Windows

Platform (UWP) і Windows Presentation Framework (WPF) [9]. На рисунку 2.2 зображено логотип MvvmCross.

Рисунок 2.2 – Логотип MvvmCross

Для реалізації бізнес-логіки застосунку було вирішено використовувати так звану Onion architecture (Архітектура Луковиці). Це архітектурний шаблон, який покладається на принципи інверсії залежностей та декомпозиції за рівнями, з метою облегшення тестування та зберігання програмного забезпечення зі зменшеною залежністю від зовнішніх фреймворків та бібліотек.

У даній архітектурі програмне забезпечення поділяється на кілька рівнів абстракції, з яких кожен рівень залежить від наступного вищого рівня. Ці рівні включають зазвичай Domain Model (модель домену), Application Logic (логіка застосунку), та Infrastructure (інфраструктура) [6].

Відповідно до цієї архітектури, вся бізнес-логіка та правила застосунку розміщуються на найнижчому рівні - у моделі домену. Логіка застосунку керує взаємодією між рівнями, тоді як інфраструктура відповідає за доступ до даних та зовнішніх служб.

Ця архітектура дає змогу дотримуватися принципу «Чистої архітектури», що дозволяє зберігати програмне забезпечення незалежним від фреймворків та залежностей ззовні, що робить його більш гнучким, масштабованим та підтримуваним. На рисунку 2.3 зображено архітектуру луковиці.

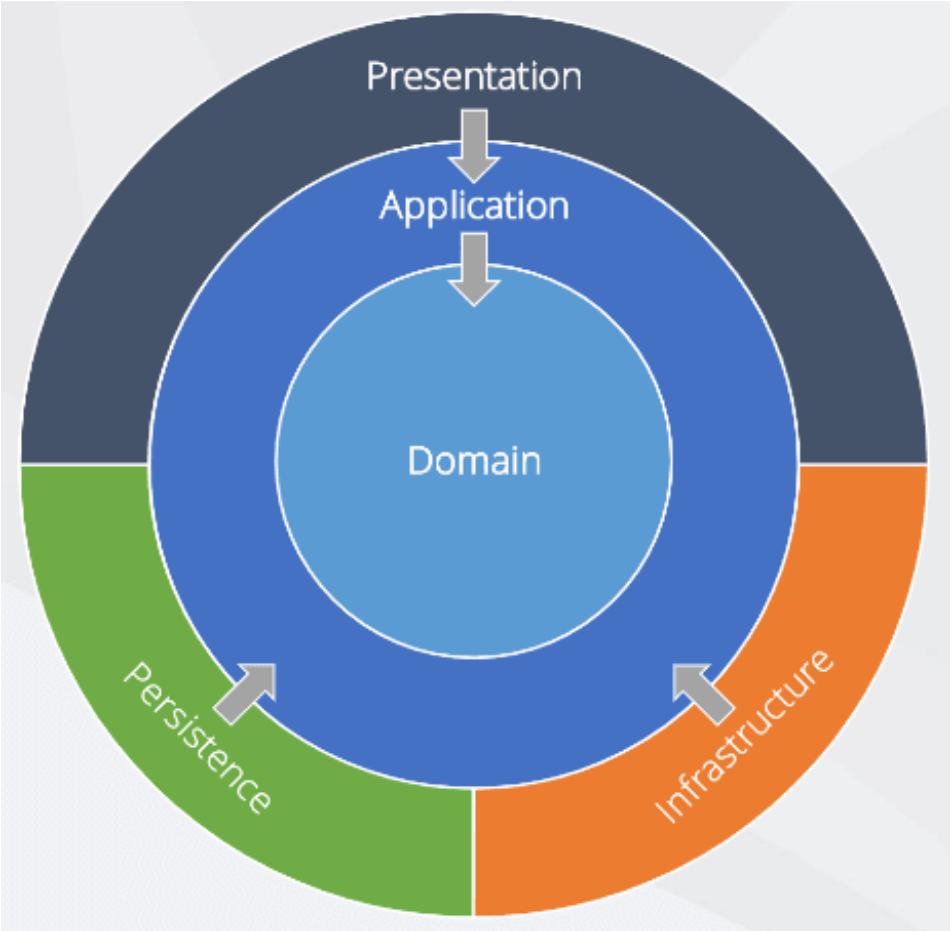


Рисунок 2.3 – Архітектура луковиці

Нижче наведено обґрунтування використання засобів розробки.

2.1 Середовище розробки .NET

Середовище розробки .NET - це платформа, яка була розроблена компанією Microsoft для створення програмного забезпечення на мовах програмування C#, F# та інших. Основною складовою середовища розробки .NET є інтегрована середовище розробки (IDE) - Visual Studio, що надає інструменти для створення, редагування та налагодження програмного коду. Також є безкоштовний IDE - Visual Studio Code. Це середовище розробки має велику бібліотеку класів, що називається .NET Framework, яка містить готові реалізації багатьох стандартних задач, таких як робота з мережевими протоколами, робота з базами даних, робота з файловою системою та інше. У .NET є можливість використання об'єктно-

орієнтованого програмування, що забезпечує зручність і підтримку кодування на декількох мовах програмування. Крім того, .NET має можливість розгортання програмного забезпечення на різних платформах, таких як Windows, macOS, Linux та інших, що зменшує залежність від конкретної операційної системи [7].

.NET надає різноманітні інструменти та бібліотеки для створення програм. Деякі з переваг .NET включають:

– Сумісність із різними платформами: .NET розроблено для роботи з кількома операційними системами, включаючи Windows, Linux і macOS. Це означає, що розробники можуть створювати програми, які можуть працювати на широкому діапазоні пристроїв;
– Взаємодія мов: .NET підтримує кілька мов програмування, таких як C#, VB.NET і F#. Розробники можуть писати код мовою, яка їм найбільш зручна;
– Надійні бібліотеки та інфраструктури: .NET надає широкий спектр готових бібліотек і інфраструктур, які можна використовувати для спрощення завдань розробки. До них належать інструменти для доступу до даних, роботи в мережі та розробки інтерфейсу користувача.
– Безпека: .NET містить вбудовані функції безпеки, які допомагають захистити програми від поширених загроз, таких як впровадження SQL і міжсайтовий сценарій;
– Просте розгортання: програми .NET можна легко розгортати за допомогою таких інструментів, як технологія Microsoft ClickOnce, яка спрощує процес розповсюдження оновлень програмного забезпечення для кінцевих користувачів;
– Висока продуктивність: застосунки .NET скомпільовані до рідного коду, що може призвести до більш високої продуктивності порівняно з інтерпретованими мовами, такими як JavaScript;

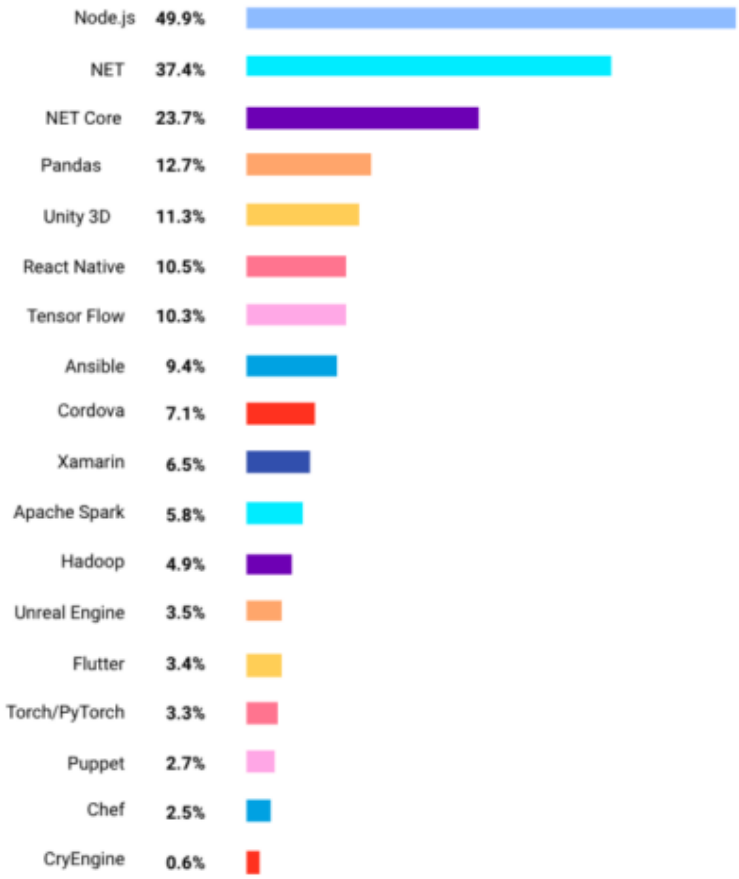


Рисунок 2.1.1 – Статистика використання середовищ розробки

Технології .NET стрімко розвивається, і над нею працюють тисячі розробників по всьому світу.

2.2 WPF

WPF (Windows Presentation Foundation) — це графічна підсистема для відтворення інтерфейсів користувача в програмах на базі Windows. Вона була представлена корпорацією Майкрософт як частина .NET Framework 3.0, і надає уніфікований API для створення настільних програм із насиченим інтерактивним інтерфейсом користувача [8].

WPF дозволяє розробникам створювати програми з сучасним і візуально привабливим інтерфейсом користувача, використовуючи декларативну модель програмування. Він надає широкий спектр елементів керування, панелей макета та механізмів прив’язки даних, які дозволяють розробникам створювати складні

та адаптивні інтерфейси користувача. WPF також підтримує анімацію, 3D-графіку та мультимедіа [8].

Ще однією перевагою WPF є його підтримка XAML, мови розмітки, яка дозволяє розробникам визначати елементи інтерфейсу користувача та їхні властивості декларативним способом, окремо від коду програми. Це дає змогу чітко розділити проблеми між інтерфейсом користувача та логікою програми, що полегшує підтримку та оновлення програми.

Загалом, WPF є потужною та гнучкою платформою для створення сучасних настільних програм із багатим інтерактивним інтерфейсом користувача.

Переваги WPF, які переконали використовувати його в проекті:

- Підтримка між платформами: програми WPF можна запускати на будь-якій платформі, яка підтримує .NET, включаючи Windows, macOS і Linux;
- Розширений інтерфейс користувача: WPF пропонує широкий спектр графічних можливостей і дозволяє розробникам створювати візуально привабливі та багаті інтерфейси користувача для своїх програм;
- WPF використовує шаблон Model-View-ViewModel (MVVM), який сприяє розділення логіки між інтерфейсом користувача, даними та поведінкою програми. Це полегшує підтримку та тестування програм;
- XAML: WPF використовує XAML (Extensible Application Markup Language), що є мовою на основі XML для розробки інтерфейсів користувача. XAML надає декларативний синтаксис для опису інтерфейсу користувача та поведінки програми;
- Прив’язка даних: WPF надає потужні можливості зв’язування даних, які дозволяють розробникам прив’язувати дані з різних джерел до елементів керування інтерфейсу користувача. Це дозволяє розробникам створювати динамічні та інтерактивні інтерфейси користувача, які реагують на зміни базових даних;

2.3 Фреймворк MvvmCross

MvvmCross — це кросплатформний фреймворк MVVM, який дозволяє розробникам створювати потужні кросплатформні програми [9]. Функції високого рівня, які надає MvvmCross:

- Шаблон архітектури MVVM;
- Система навігації;
- Прив'язка даних;
- Підтримка особливостей платформи;
- Інверсія контейнера керування та механізм впровадження залежностей;
- Багато плагінів для загальних функцій;
- Помічники модульних тестів;
- Повна гнучкість;

Переваги MvvmCross, які надихнули використовувати його, як MVVM фреймворк в цьому проекті:

- Легко почати використовувати;
- Має багато варіантів використання і в той же час, якщо прийняте рішення обмежити варіанти використання у проекті він може працювати “Прямо із коробки”;
- Кросплатформність;
- Користується попитом у спільноти. Має більше 5 мільйонів завантажень в NuGet Packages;
- Багата документація;

2.4 Додаткові компоненти, утиліти та бібліотеки

В ході розробки застосунку також були використані наступні засоби для зручного вирішення задач, які були поставленні:

- Бібліотека MathNet.Numerics - це чисельна основа проекту Math.NET,

метою якого є надання методів і алгоритмів для чисельних обчислень в інженерії та щоденному використанні;

– ScottPlot.WPF - це пакет, який надає елемент керування WPF для інтерактивного відображення графіків, діаграм, дашбордів у програмах WPF. З його допомогою було реалізована можливість надання статистики по навантажувальному тестуванню в графічному виді;
– AutoMapper - дозволяє проектувати одну модель на іншу, що дозволяє скоротити обсяги коду та спростити програму;
– Serilog – бібліотека для логування;
– AvalonEdit - текстовий редактор на основі WPF;
– xUnit – фреймворк, для модульного тестування;
– Moq – фреймворк, для імітації роботи об’єктів;
– Newtonsoft.Json – гнучкий серіалізатор JSON для перетворення між об’єктами .NET і JSON.

3 РОЗРОБКА ПРОГРАМНОГО ПРОДУКТУ

Для моделювання та ілюстрації потоку дій у застосунку, який проводить навантажувальне тестування SQL запитів приведено діаграму діяльності на рисунку 3.1, яка дає змогу зрозуміти алгоритм дій, які проводяться підчас такого тестування та перейти до визначення функціональних вимог.

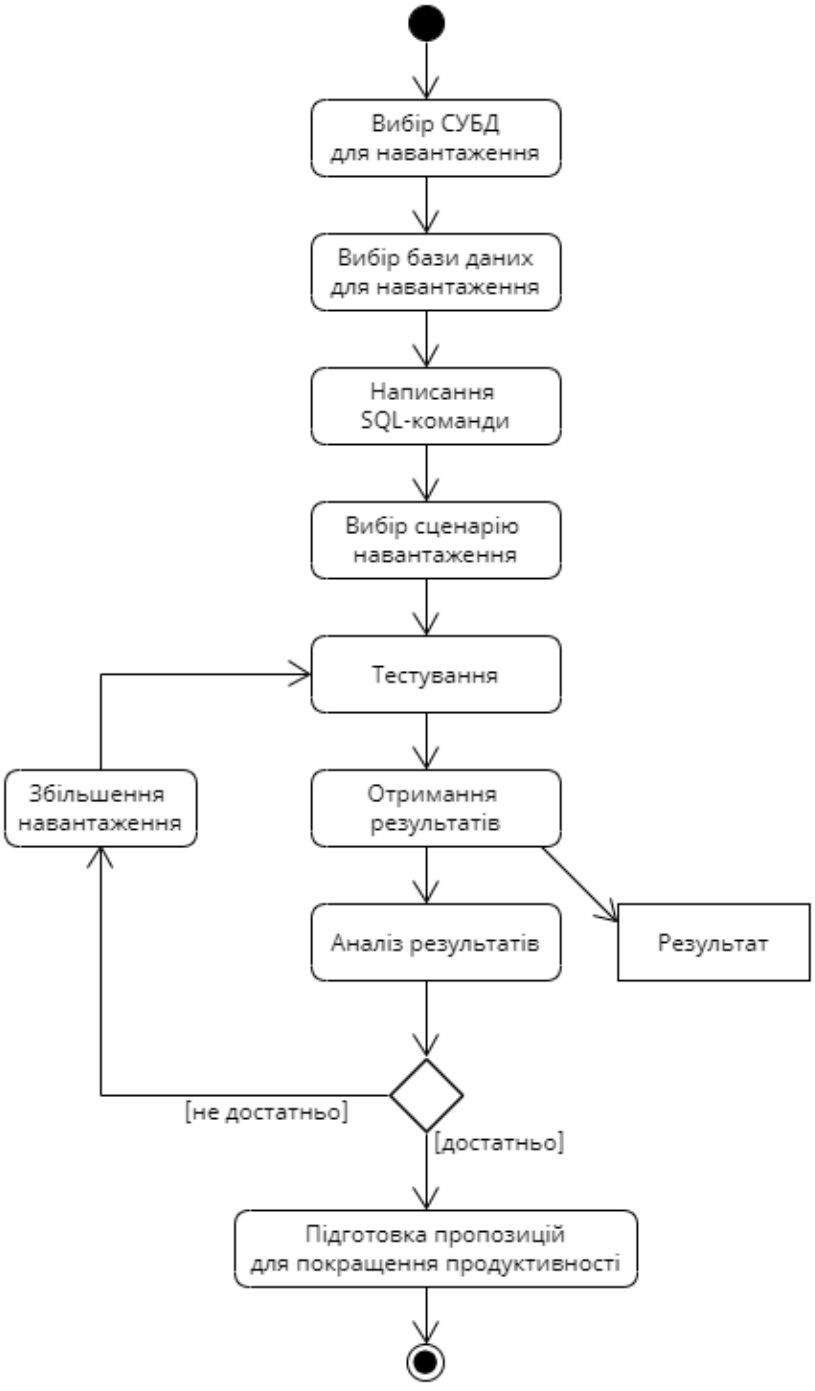


Рисунок 3.1 – Моделювання потоку дій (діаграма діяльності)

3.1 Визначення функціональних вимог

На основі наданого вище образу проекту, з метою попереднього визначення функціональності застосунку, що розробляється, наведено діаграму прецедентів на рисунку 3.1.1.

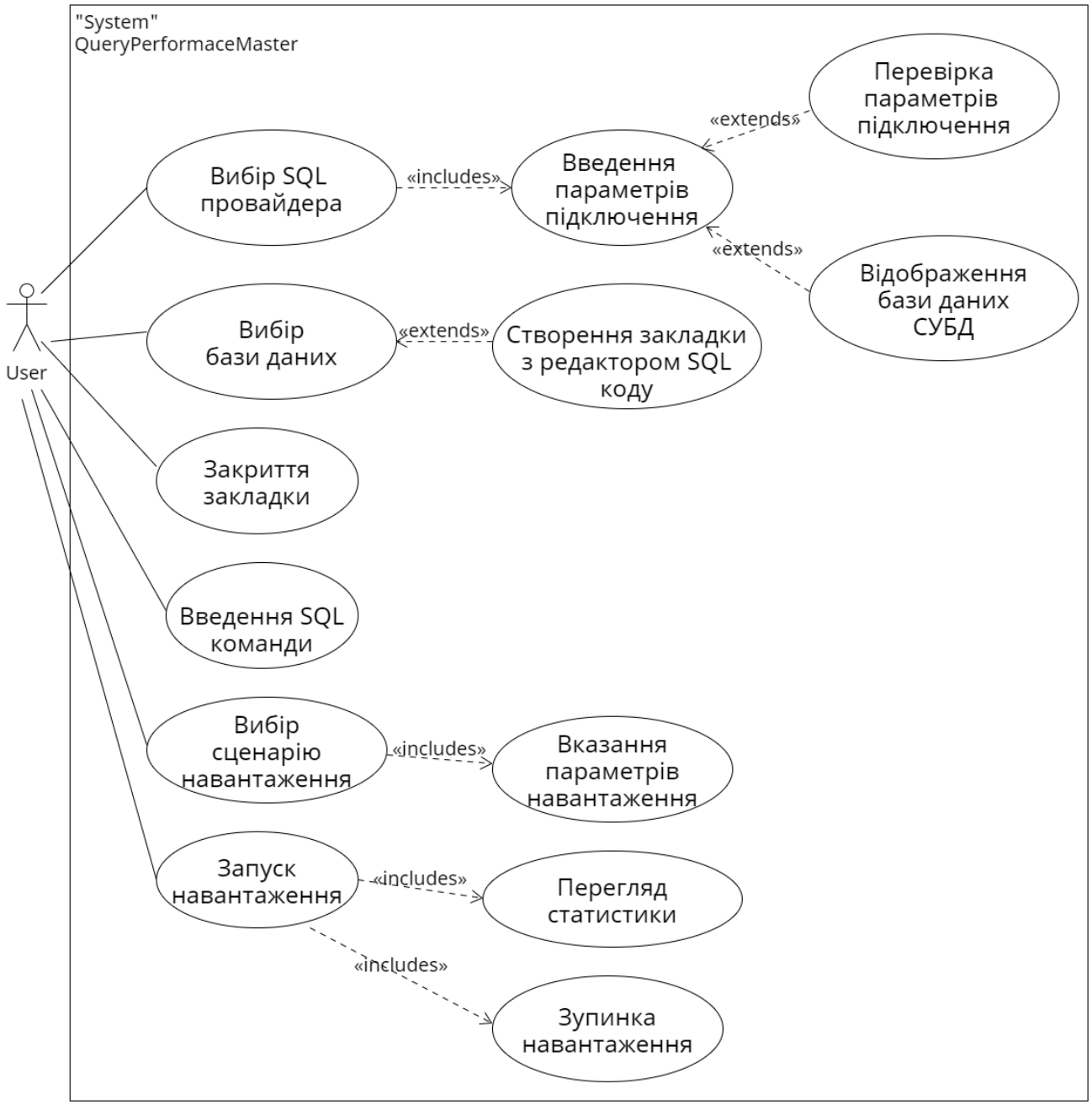


Рисунок 3.1.1 – Моделювання функціональності (діаграма прецедентів)

Вербальний опис усіх варіантів використання системи:

a) Прецедент: Вибір SQL провайдера.

Рамки: система “QueryPerformanceMaster”.

Рівень: “Задача, визначена користувачем”.

Виконавець: Користувач (розробник).

Передумови: Застосунок запущено.

Результат: Відкривається вікно для введення параметрів підключення.

Основний успішний сценарій:

1) З лівого боку екрана користувач бачить Tree View (елемент графічного інтерфейсу для ієрархічного відображення інформації) списку провайдерів СУБД (SqlServer, PostgreSQL).
2) Користувач намагається розгорнути вузол з обраним провайдером.
3) Відкривається вікно для введення параметрів підключення.

Спеціальні вимоги: Можливість розширення списку провайдерів.

Список технологій і типів даний: .NET 6, WPF.

Частота використання: при потребі.

Відкриті питання: Провести аналіз інших СУБД, які можна додати по переліку провайдерів.

b) Прецедент: Введення параметрів підключення

Рамки: система “QueryPerformanceMaster”.

Рівень: “Задача, визначена користувачем”.

Виконавець: Користувач (розробник).

Передумови: SQL провайдер був обраний.

Результат: Застосунок завантажує список баз даних на основі ведених параметрів підключення.

Основний успішний сценарій:

1) Користувач вводить параметри підключення.
2) Користувач зберігає введені параметри підключення.
3) Система перевіряє введені параметри підключення.
4) Система завантажує перелік баз даних провайдера.

Розширення (альтернативні потоки):

1) Перевірка параметрів підключення:
    a. При введені неправильних значень система пропонує ввести значення заново.
    b. При введені правильних значень система завантажує перелік баз даних.
2) Відображення баз даних.

Список технологій і типів даний: .NET 6, WPF.

Частота використання: при потребі.

c) Прецедент: Вибір бази даних

Рамки: система “QueryPerformanceMaster”.

Рівень: “Задача, визначена користувачем”.

Виконавець: Користувач (розробник).

Передумови: Параметри підключення були збережені та перелік баз даних був завантажений.

Результат: Створення закладки з редактором SQL коду.

Основний успішний сценарій:

1) Користувач натискає на обраній базі даних ліву кнопку миші та бачить кнопку “Create Query” і натискає її.
2) Система створює закладку з редактором SQL коду для обраною бази даних.

Розширення (альтернативні потоки):

1) Створення закладки з редактором SQL коду.

Спеціальні вимоги: Можливість додавання декількох закладок для різних баз даних різних СУБД.

Список технологій і типів даний: .NET 6, WPF.

Частота використання: при потребі.

d) Прецедент: Запуск навантаження

Рамки: система “QueryPerformanceMaster”.

Рівень: “Задача, визначена користувачем”.

Виконавець: Користувач (розробник).

Передумови:

1) Сценарій навантаження обрано.
2) Параметри навантаження вказано.
3) Параметри підключення введено.

Результат: Проведення навантаження та надання статистики.

Основний успішний сценарій:

1) Користувач починає навантаження.
2) Система проводить навантажувальне тестування з вказаними параметрами та збирає статистику.
3) Система надає зібрану статистику користувачу після проведення навантажувального тестування.

Розширення (альтернативні потоки):

1) Відображення прогресу виконання навантаження.
2) Виконання навантаження та збір статистики.
3) Відкриття вікна з зібраною статистикою по навантаженню.

Список технологій і типів даний: .NET 6, WPF.

Частота використання: при потребі.

Для моделювання послідовності дій при виконанні навантажувального тестування застосунку та ілюстрації повідомлень, якими обмінюються об’єкти та компоненти приведений рисунок 3.1.2.

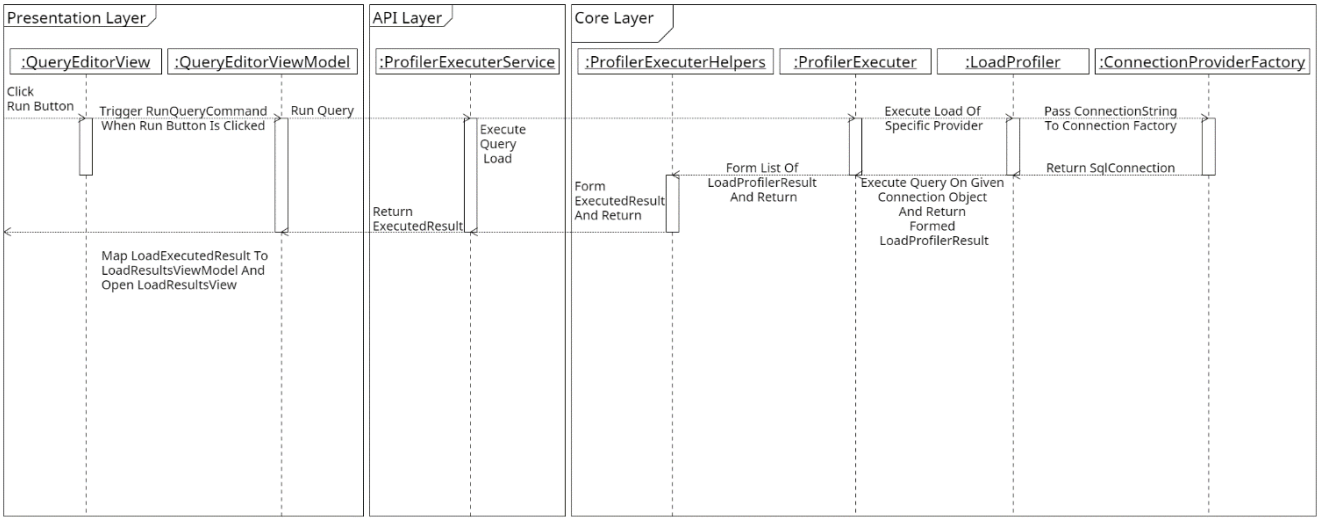


Рисунок 3.1.2 – Моделювання послідовності при виконанні навантажувального тестування (Діаграма послідовності)

3.2 Визначення не функціональних вимог

Не функціональні вимоги застосунку:

– Локалізація інтерфейсу – англійська;
– Операційна система – Windows;
– Мінімальна версія операційної системи - Windows 10 версії 1809 (10.0.17763.0);
– Застосунок повинен мати змогу пакуватись у формат MSIX для публікації в Microsoft Store;

3.3 Визначення архітектури ядра застосунку

В рамках реалізації принципів чистої архітектури, структура ядра застосунку була розбита на три бібліотеки. Продемонстровано на рисунку 3.3.1.

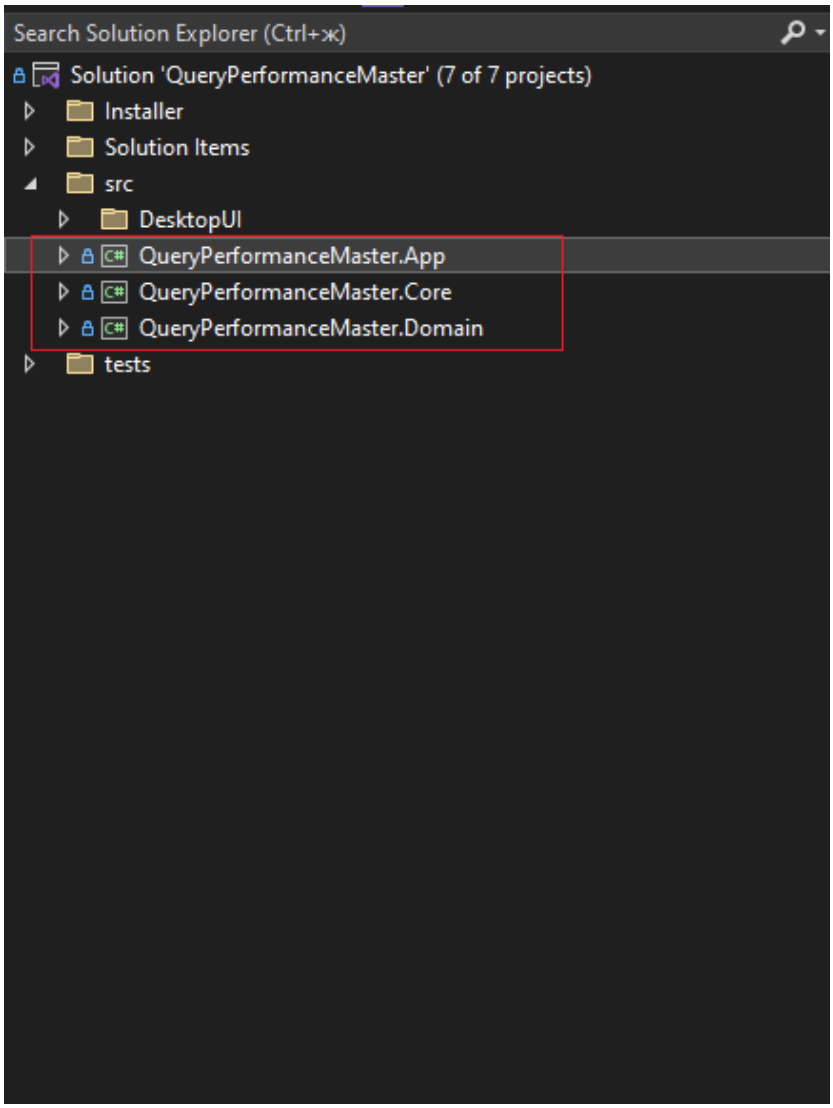


Рисунок 3.3.1 – Архітектура ядра

– QueryPerformanceMaster.App – містить абстрактну частину застосунку;
– QueryPerformanceMaster.Core – містить інфраструктурну частину застосунку;
– QueryPerformanceMaster.Domain – містить загальну частину застосунку;

Для виконання задачі суміщення декількох СУБД на яких може проводитись навантажувальне тестування для декількох сценаріїв, було введено дві центральні сутності:

– LoadProfiler (далі профайлер) – сутність, задача якої є отримати команду запиту, виконати його для СУБД до якої вона належить та зібрати статистику по виконанню;
– ProfilerExecuter (далі сценарій навантаження) – сутність, яка представляє собою сценарій виконання запиту. Вона має якимось заданим чином викликати LoadProfiler для виконання запиту;

Для реалізації профайлерів було створено інтерфейс ILoadProfiler, який визначає специфікацію сутності профайлера, в даному випадку – це метод ExecuteQueryLoadAsync, який виконує запит. В рамках даної роботи було реалізовано два види профайлера – MsSqlLoadProfiler і PostgreSqlLoadProfiler, для навантаження запитів SqlServer і PostgreSQL відповідно. Їх реалізація надається в додатку А.

Для вирішення проблеми того, яким чином профайлери будуть створювати підключення до бази даних та виконувати запит вводиться нова сутність – ConnectionProvider. Для введення цієї сутності в ядро було створено узагальнений інтерфейс IConnectionProvider<T>, який приймає в параметр T клас того API, яке буде виконувати з’єднання на запит. Реалізовувати цей інтерфейс буде абстрактний клас ConnectionProviderBase<T>, який вже будуть наслідувати конкретні реалізації та перевизначати метод для створення SQL з’єднання. Таких реалізацій буде дві – MsSqlConnectionProvider і PostgreSqlConnectionProvider, для створення з’єднання з SqlServer і PostgreSQL відповідно. Для створення екземплярів класів було створено два фабричних класи. На рисунку 3.3.2 зображено зв'язок між сутностями LoadProfiler і ConnectionProvider. Також їх реалізація надається в додатку Б.

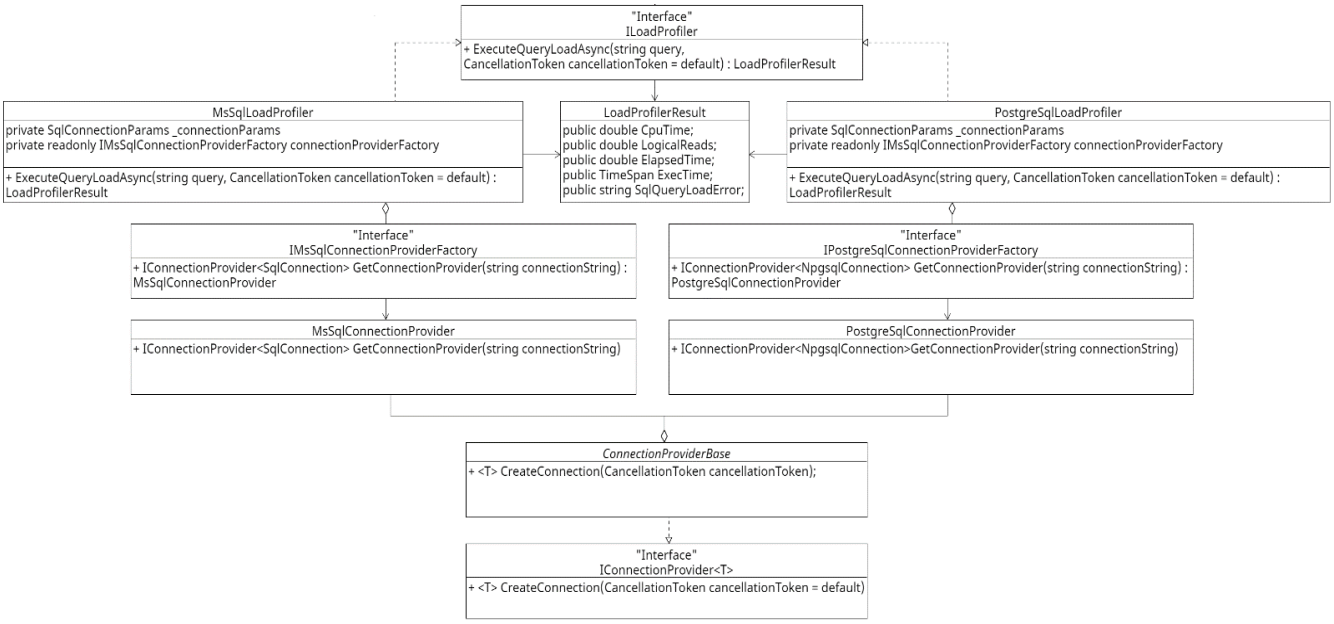


Рисунок 3.3.2 – Ілюстрація зв’язку між LoadProfiler і ConnectionProvider

Для розробки сценаріїв навантаження було створено реалізовано чотири сценарії навантаження:

– ParallelProfilerExecuter – сценарій, який передбачає паралельне виконання навантаження в задану кількість потоків та ітерацій;
– SequentialProfilerExecuter – сценарій, який передбачає послідовне виконання в задану кількість ітерацій;
– SequentialProfilerExecuterWithDelay - сценарій, який передбачає послідовне виконання в задану кількість ітерацій та заданим часом затримки між запитами;
– SequentialProfilerExecutorWithTimeLimit - сценарій, який передбачає послідовне виконання в задану кількість ітерацій та обмеженням по часу виконання;

Для вирішення задачі можливості заміни профайлеру та сценарію незалежно один від одного, було використано шаблон проектування Міст (Bridge), який дозволяє відокремити абстракцію від реалізації таким чином, щоб і абстракцію, і реалізацію можна було змінювати незалежно один від одного. На рисунку 3.3.3 зображено ілюстрацію цього шаблону. Реалізацію сценаріїв наведено в додатку В.

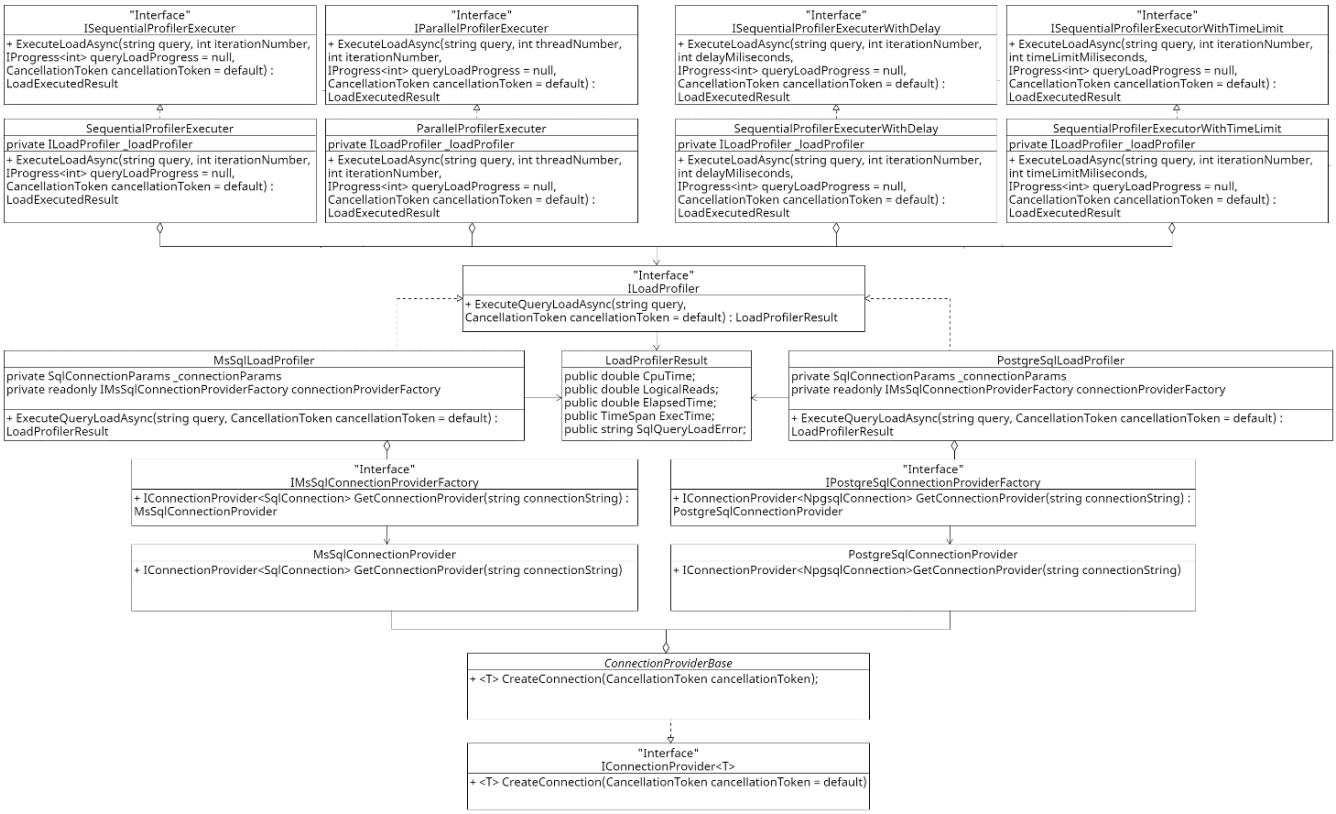


Рисунок 3.3.3 – Реалізація шаблону Bridge

На рисунку 3.3.3 видно, що кожна реалізація сценарію реалізує свій інтерфейс - це необхідно тому, що список параметрів методу, який виконує навантаження у кожного сценарію різний, тому реалізовувати один інтерфейс вони не можуть. Таке рішення веде за собою наступну проблему – потрібно звести всі реалізації сценаріїв в одному місці, щоб ініціювати виконання навантаження можна було одним методом. Для вирішення цієї задачі було введено клас ProfilerExecuterService, який має по переданим параметрам визначити сценарій навантаження та викликати ініціювання навантаження в необхідній реалізації сценарію. Для отримання реалізацій сценаріїв для кожного сценарію було створено фабричні класи, які створюють об’єкт класу сценарію. Це потрібно для того, щоб винести створення екземплярів класів в окремий клас. Фабричний клас також буде створено і для профайлерів. На рисунку 3.3.4 зображено діаграму класів для демонстрації повної картини зв’ язків між класами ядра.

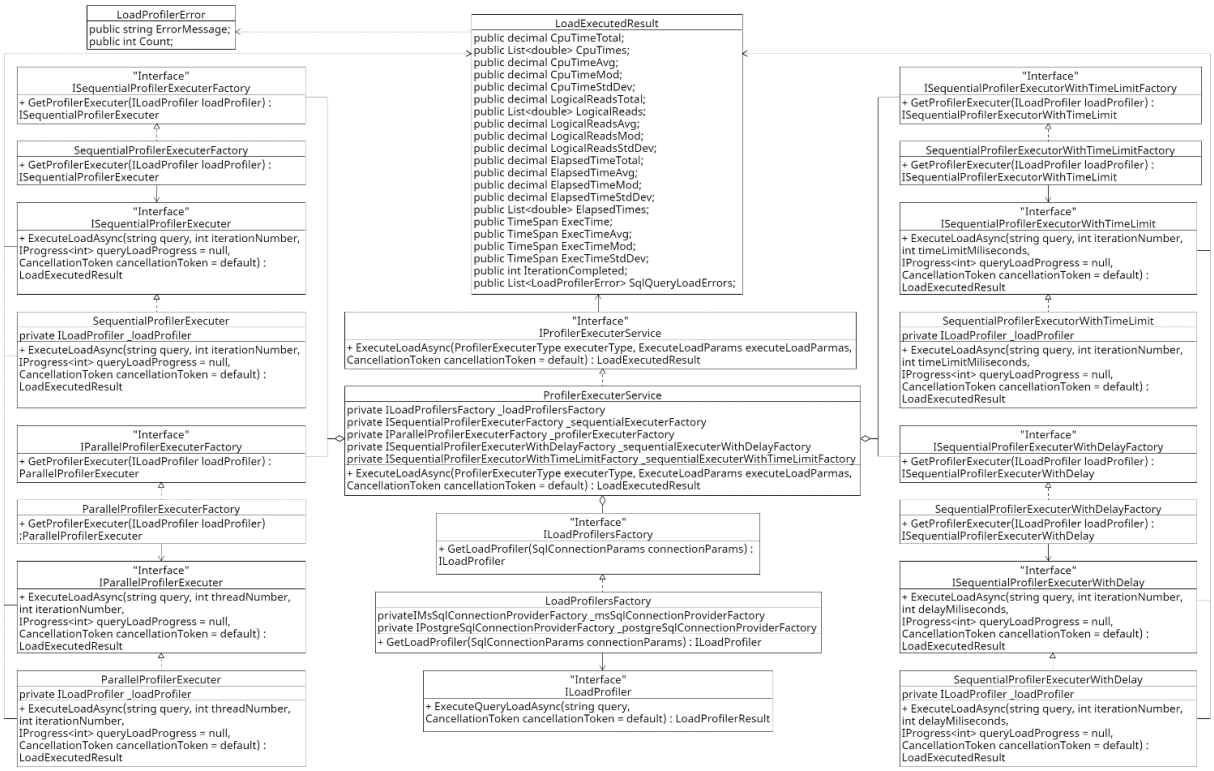


Рисунок 3.3.4 – Діаграма класів

3.4 Визначення структури допоміжних класів застосунку

Для вирішення задач роботи з параметрами підключення, таких як:

– отримання рядка підключення з заповненого об’єкту параметрів підключення та навпаки;

– коригування створеного рядка підключення ;

було створено наступну ієрархію класів, які виконують цю роботу. Проілюстровано на рисунку 3.4.1. Також їх реалізацію наведено в додатку Г.

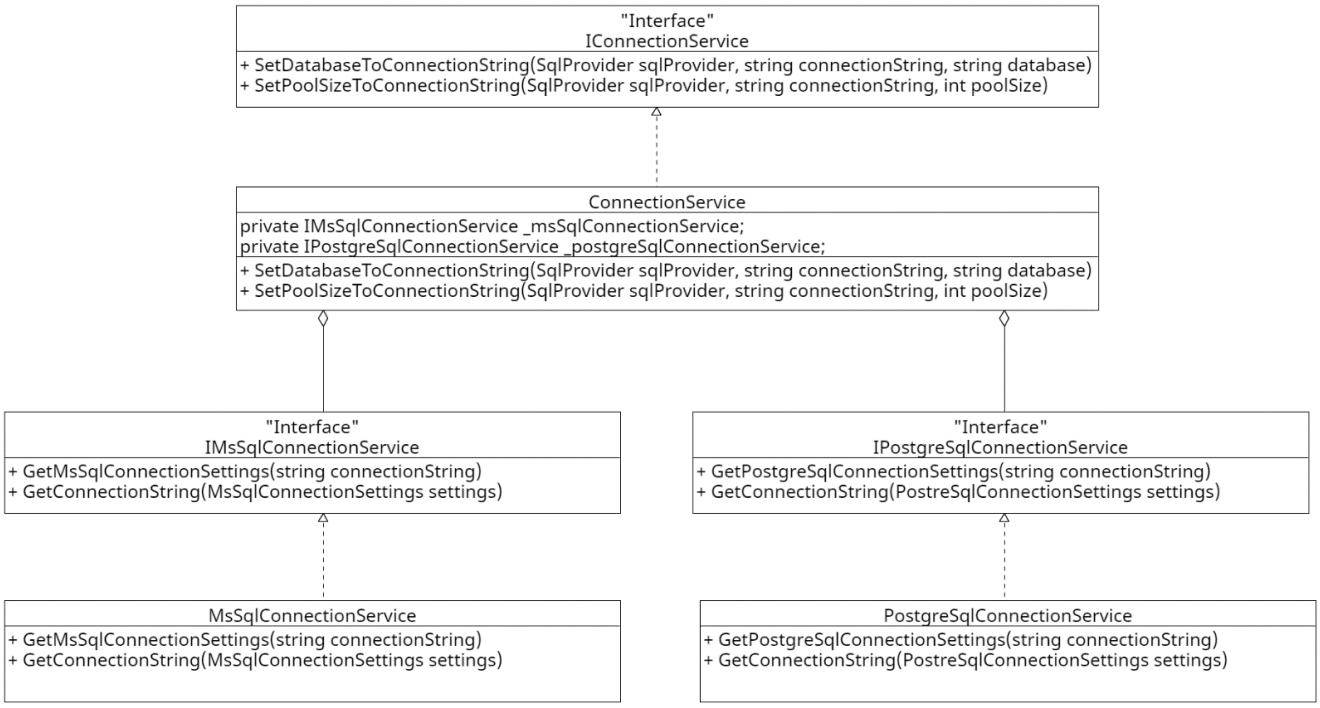


Рисунок 3.4.1 – Діаграма класів для роботи з параметрами підключення

Для вирішення задач з отримання переліку баз даних СУБД, очистки кешу та буферу, було створено ієрархію класів, яку наведено на рисунку 3.4.2. Також їх реалізацію наведено в додатку Д.

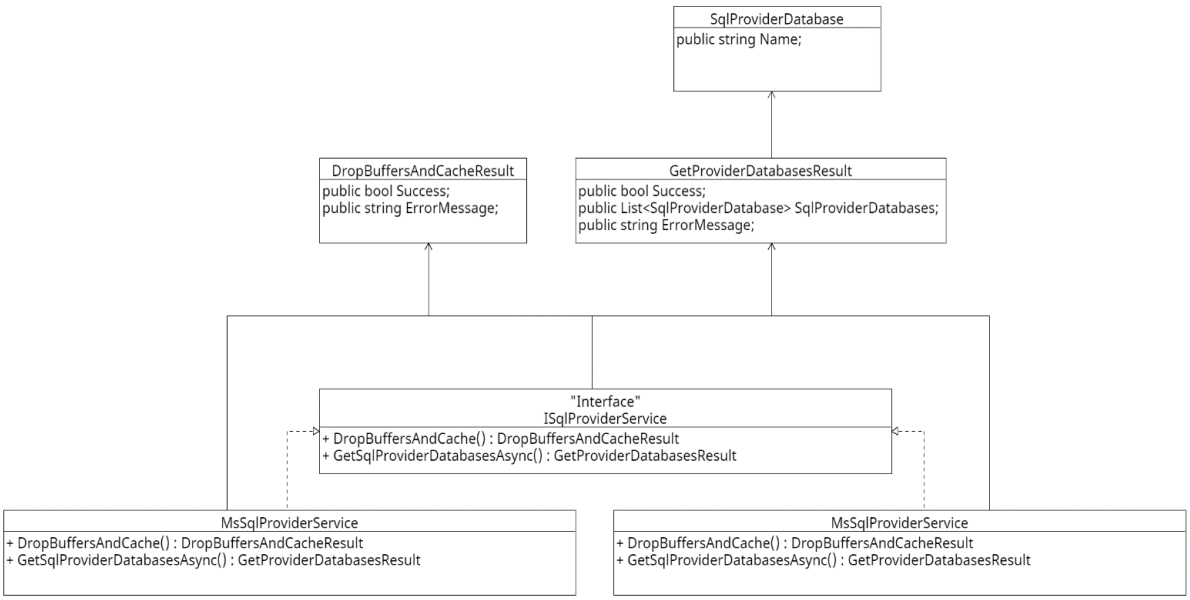


Рисунок 3.4.2 – Діаграма класів для роботи з СУБД

3.5 Розробка візуальної частини застосунку

В рамках реалізації шаблону проектування MVVM використовуючи MVVM фреймворк MvvmCross, структура візуальної частини застосунку була розбита на дві бібліотеки. Продемонстровано на рисунку 3.5.1.

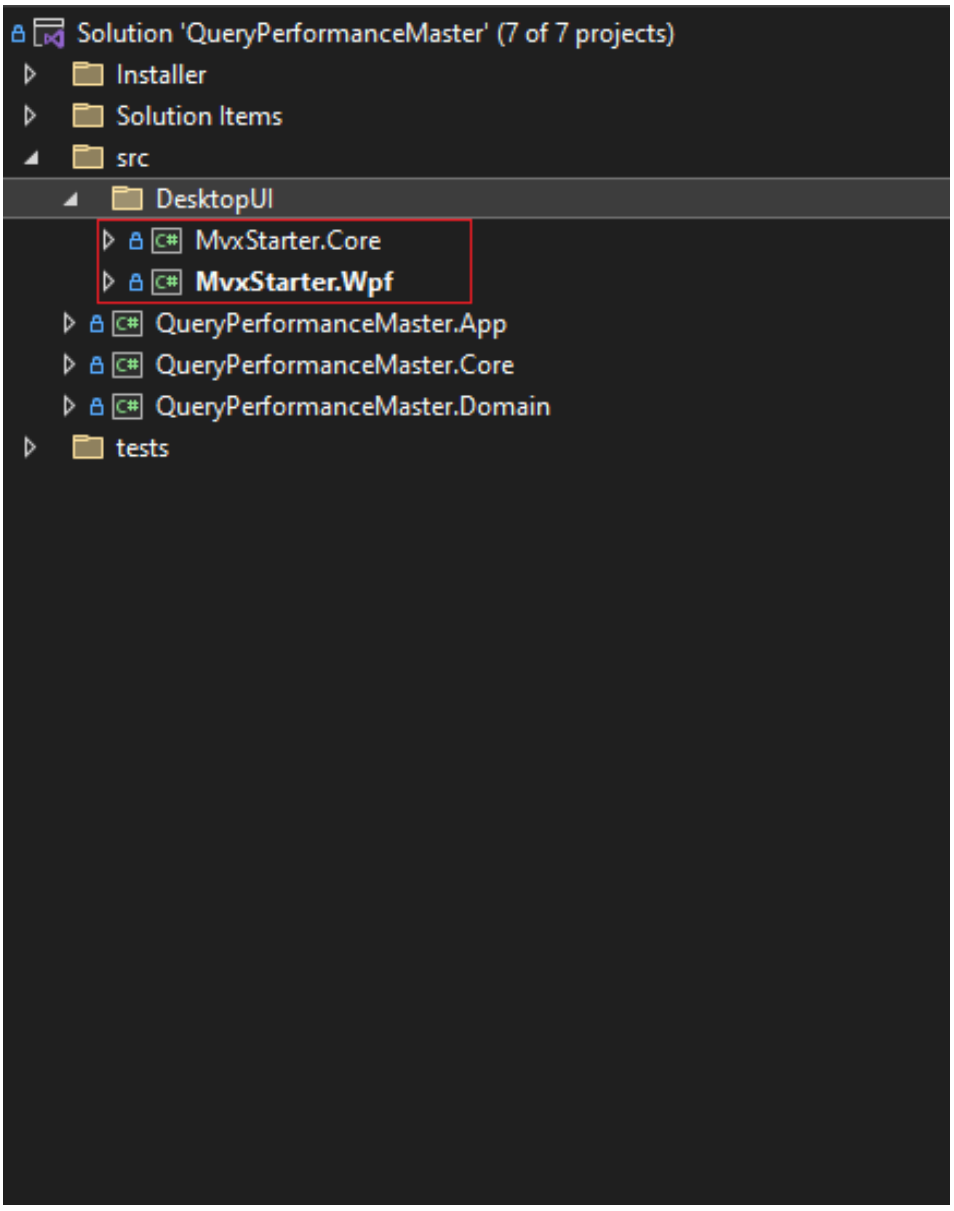


Рисунок 3.5.1 – Архітектура візуальної частини

– MvxStarter.Core – містить інфраструктурну частину;
– MvxStarter.Wpf – містить візуальну частину;

Для моделювання меню, ґрунтуючись на поставлених функціональних вимогах, було розроблено діаграму станів, яка відображує поставлені

функціональні вимоги, яка демонструє послідовність станів і переходів, які в сукупності характеризують поведінку застосунку. Зображено на рисунку 3.5.2.

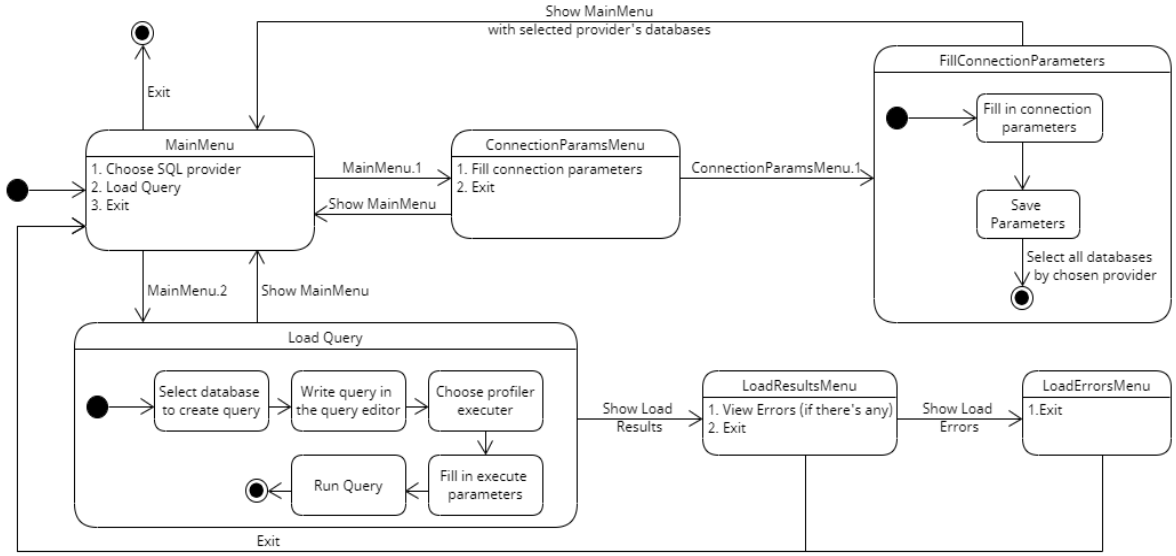


Рисунок 3.5.2 – Діаграма станів

На основі діаграми станів були розроблені екрани застосунку. На рисунку 3.5.3 зображено головний екран застосунку, на якому знаходяться всі елементи, які передбачені функціональними вимогами.

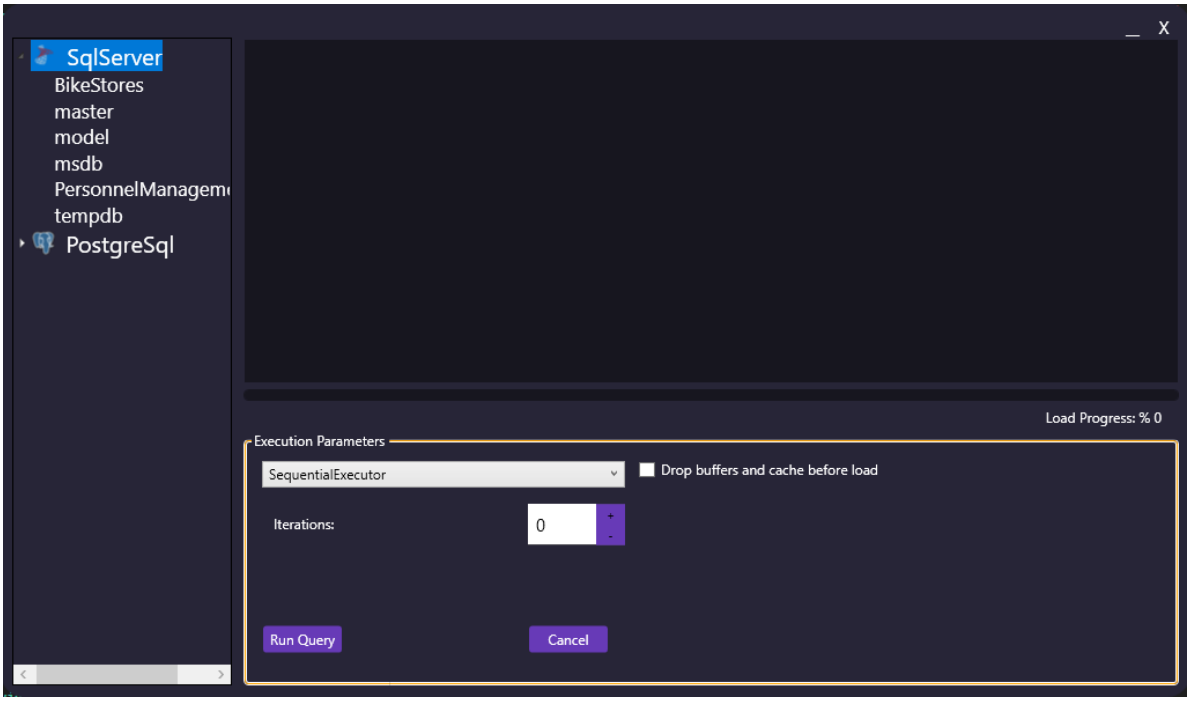


Рисунок 3.5.3 – Головний екран застосунку

З лівого боку екрану розташований TreeView в якому містяться список провайдерів СУБД, які на даний момент реалізовані в застосунку (цей список може бути розширений при необхідності). При спробі розгорнути кореневий вузол, якщо параметри підключення не були вказані раніше, то відкриється відповідний екран для їх вводу. З правому нижньому кутку розташований сектор, який відповідає за виконання навантаження. На ньому є випадаючий список в якому вибирається сценарій навантаження. Справа від випадаючого списку розташована ознака, який визначає чи потрібно очищувати буфер та кеш бази даних перед виконанням навантажувального тестування. Внизу під випадаючим списком розташовані параметри навантаження, такі як: кількість ітерацій, кількість потоків, час затримки між запитами, час обмеження виконання навантаження. В нижній частину екрану розташовані дві кнопки: “Run Query” та “Cancel”, які відповідають за початок тестування та відміну виконання тестування. Посередні екрану розташований прогрес бар, який відображає відсоток виконаної роботи при виконанні навантаження.

Рисунок 3.5.4 демонструє головний екран с двома доданими закладками для введення SQL запитів для баз даних SqlServer і PostgreSQL.

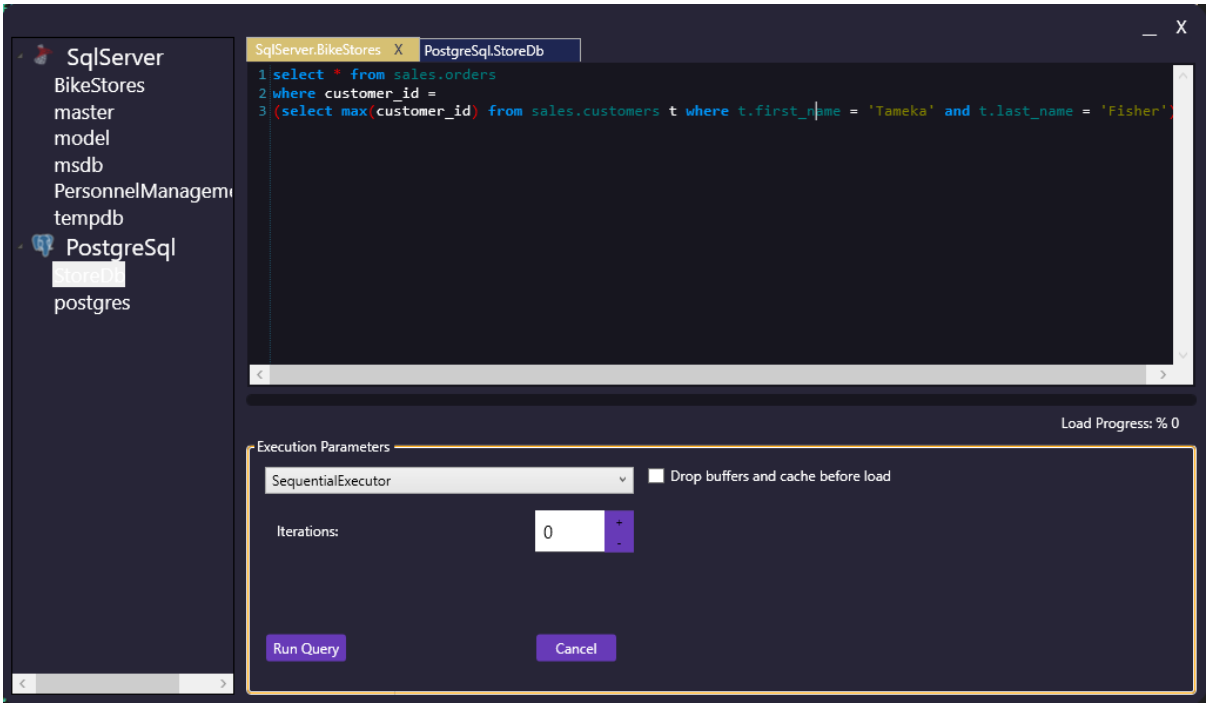


Рисунок 3.5.4 – Головний екран (демонстрація закладок)

Для введення параметрів підключення до СУБД було реалізовано два екрани. Один для SqlServer, другий для PostgreSQL. Демонстрацію цих екранів зображено на рисунках 3.5.5 і 3.5.6 відповідно.

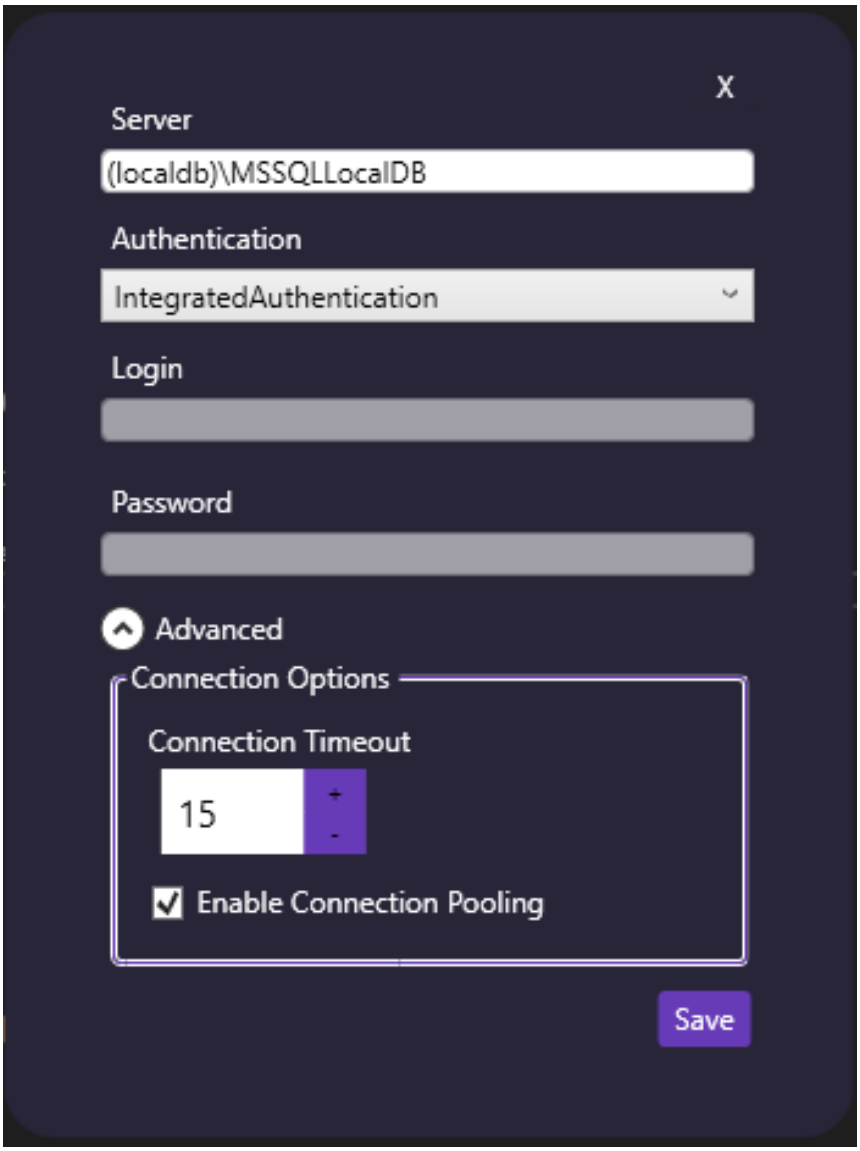


Рисунок 3.5.5 – Екран параметрів підключення SqlServer

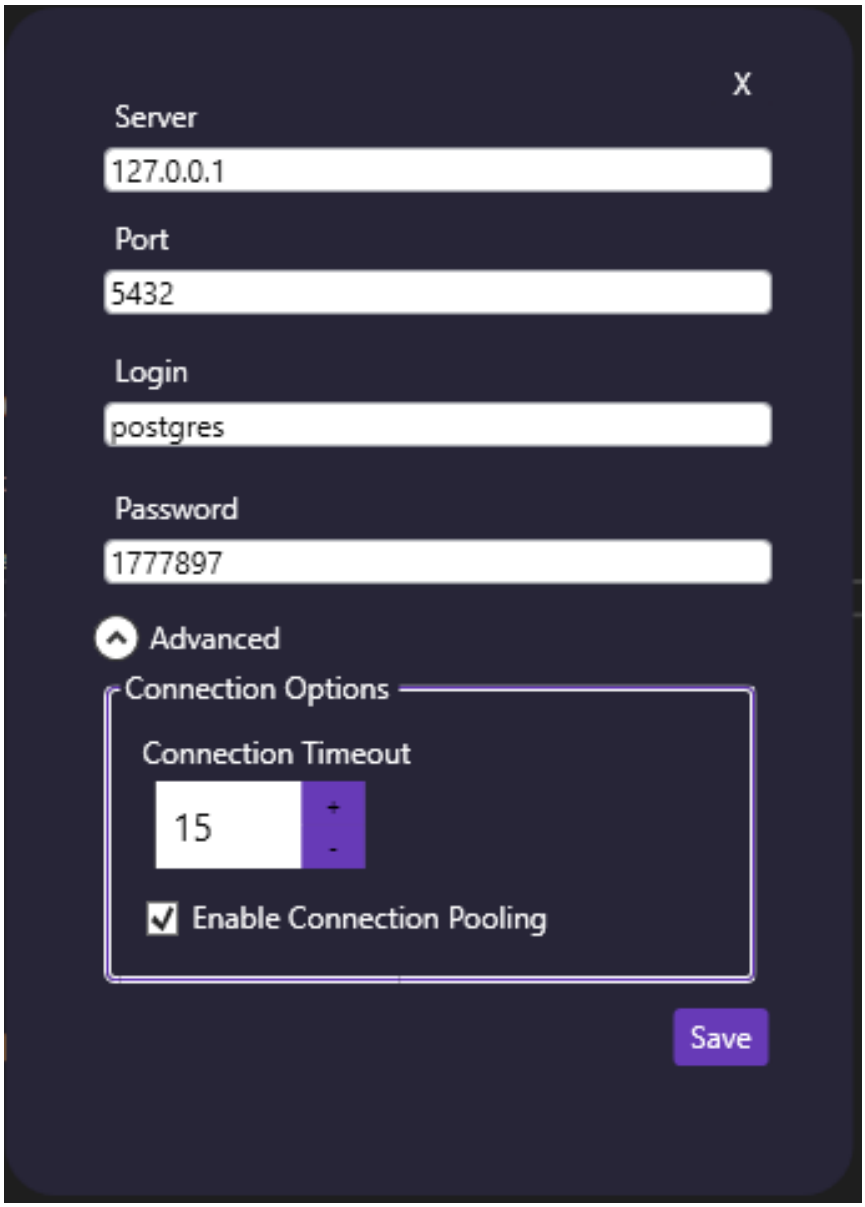


Рисунок 3.5.6 – Екран параметрів підключення PostgreSQL

Після успішного виконання навантаження відкривається вікно з зібраною статистикою по виконанню навантаження. Статистика доступна у двох форматах: в виді числових значень та у графічному виді. На рисунках 3.5.7 та 3.5.8 зображений екран статистики виконання навантаження.

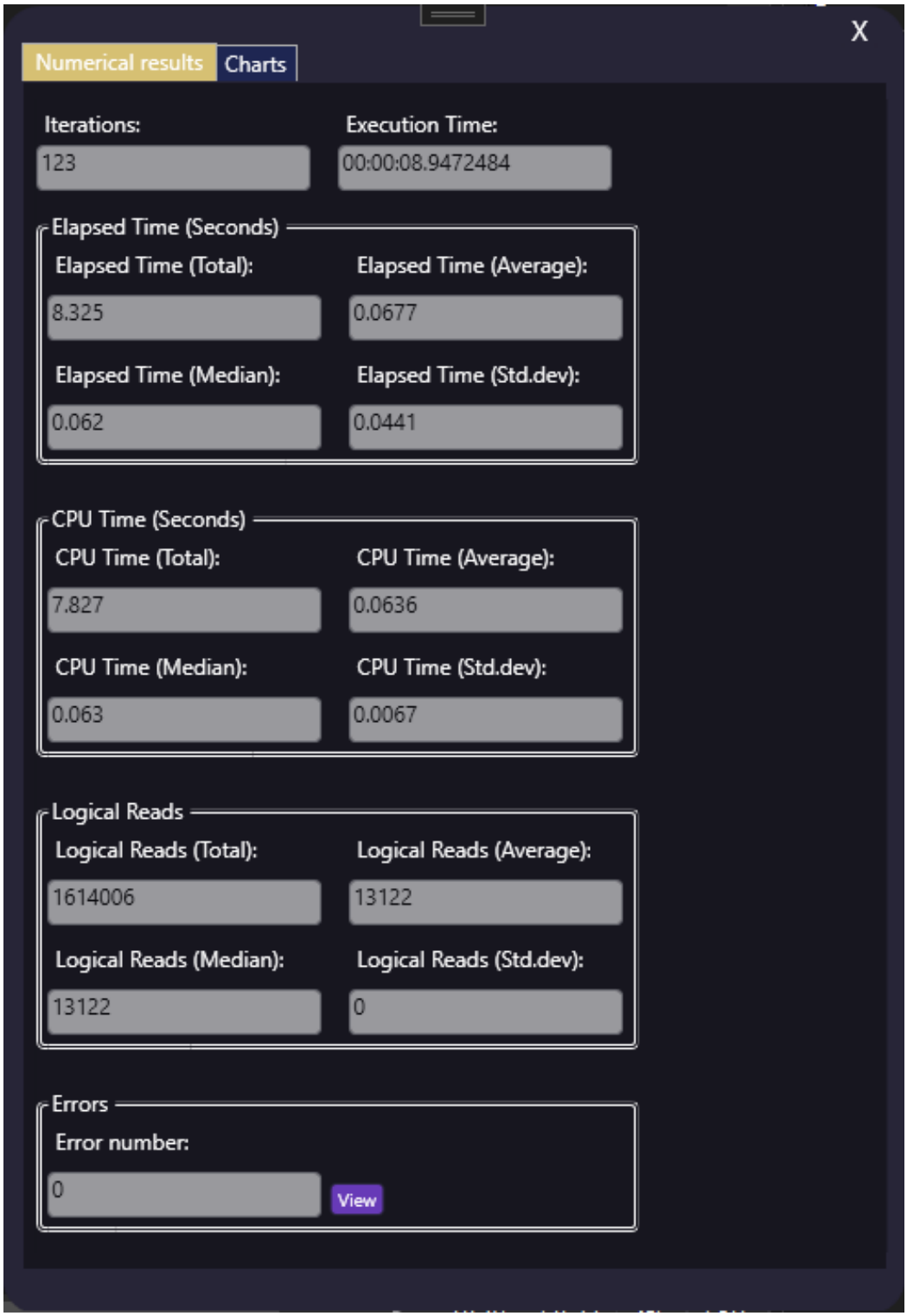


Рисунок 3.5.7 – Екран статистики навантаження у чисельному форматі

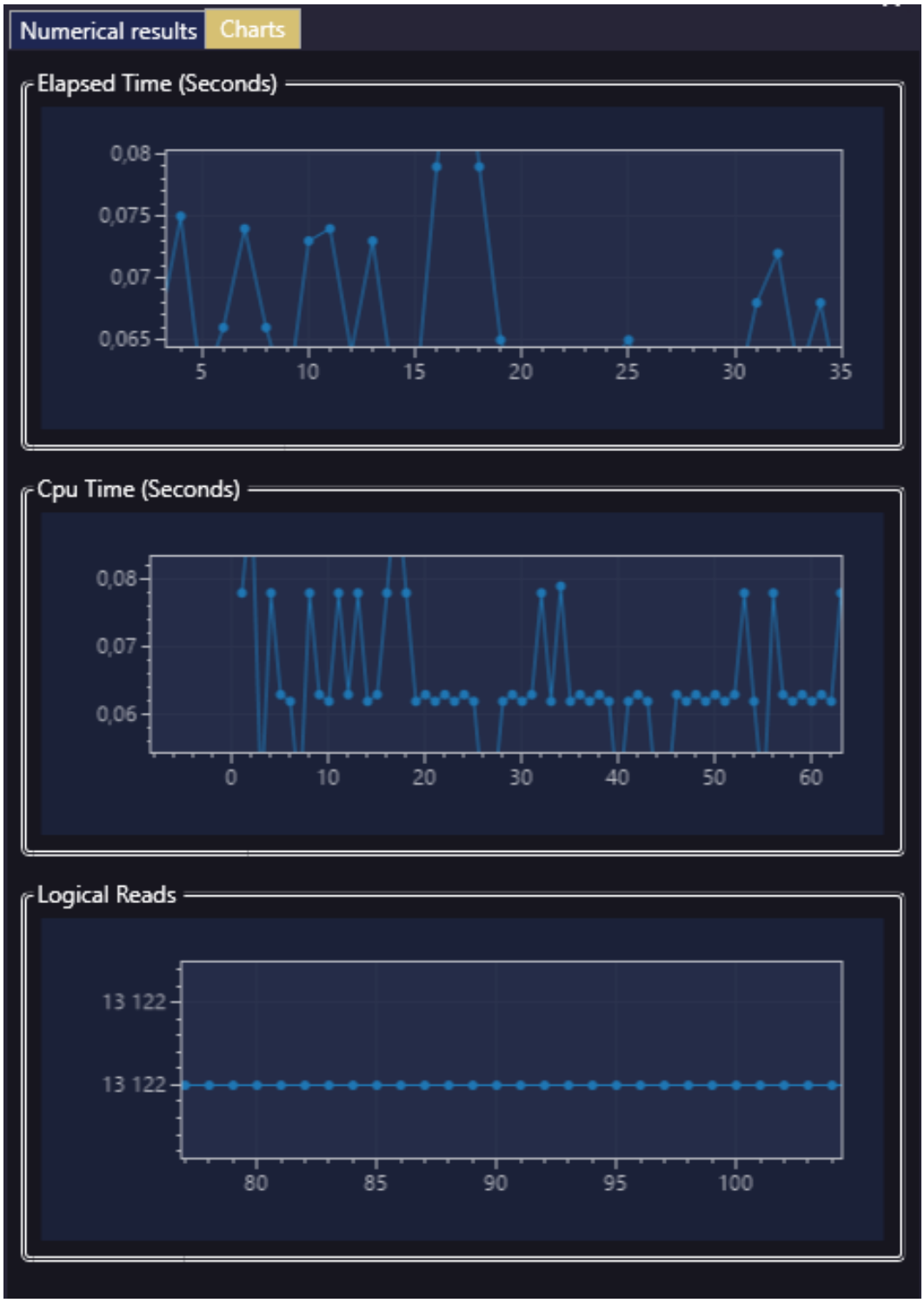


Рисунок 3.5.8 – Екран статистики навантаження у графічному форматі

3.6 Визначення компонентів застосунку

На основі проведеного визначення структури застосунку, наведено діаграму компонентів для відображення компонентів застосунку та зв’язків між ними на рисунку 3.6.1.

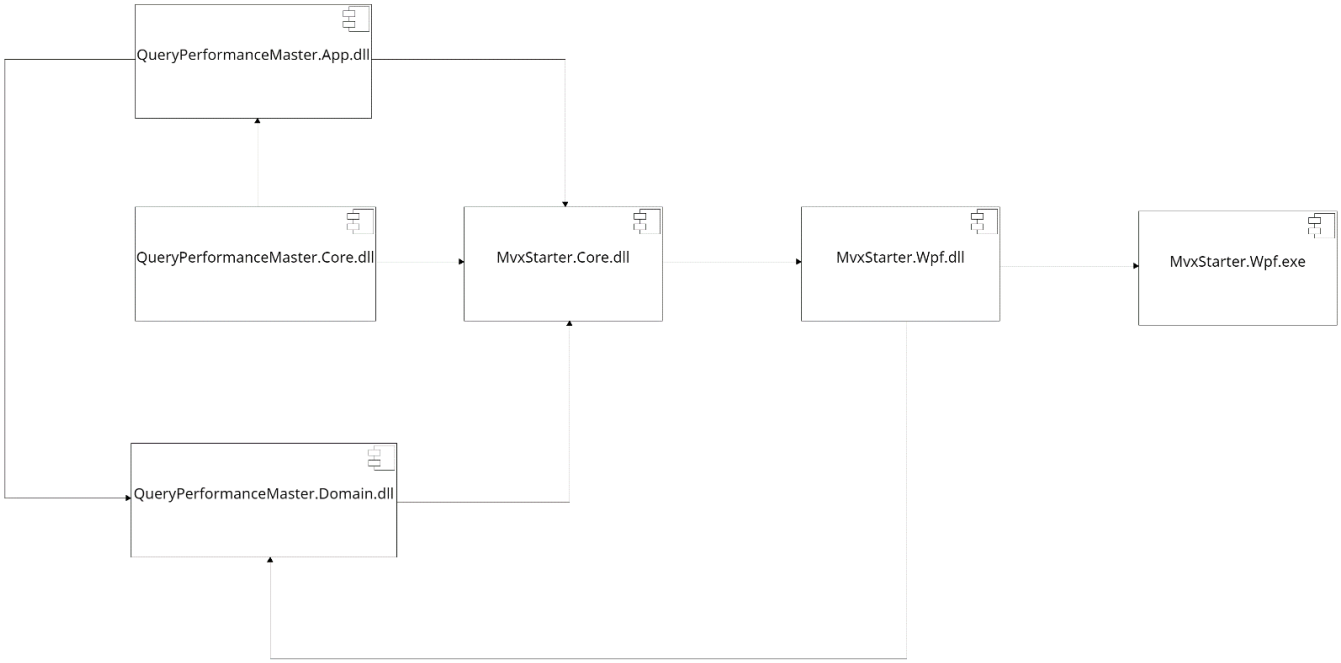


Рисунок 3.6.1 – Моделювання зав’язків компонентів (діаграма компонентів)

Нижче наведено роз’яснення компонентів, які наведено в діаграмі:

– MvxStarter.Wpf.exe – файл, який запускає застосунок;
– MvxStarter.Wpf.dll – бібліотека, яка містить код та дані візуальної частини застосунку;
– MvxStarter.Core.dll - бібліотека, яка зводить у собі інфраструктурну та абстрактну частину застосунку, для використання візуальною частиною. Також містить бізнес-логіку форм;
– QueryPerformanceMaster.App.dll – бібліотека, яка містить абстрактну частину застосунку;
– QueryPerformanceMaster.Domain.dll - бібліотека, яка містить частину застосунку, яка є загальною для всього застосунку;

Ґрунтуючись на визначених компонентах застосунку, розроблена діаграма розгортання застосунку, яка наведена на рисунку 4.6.2.

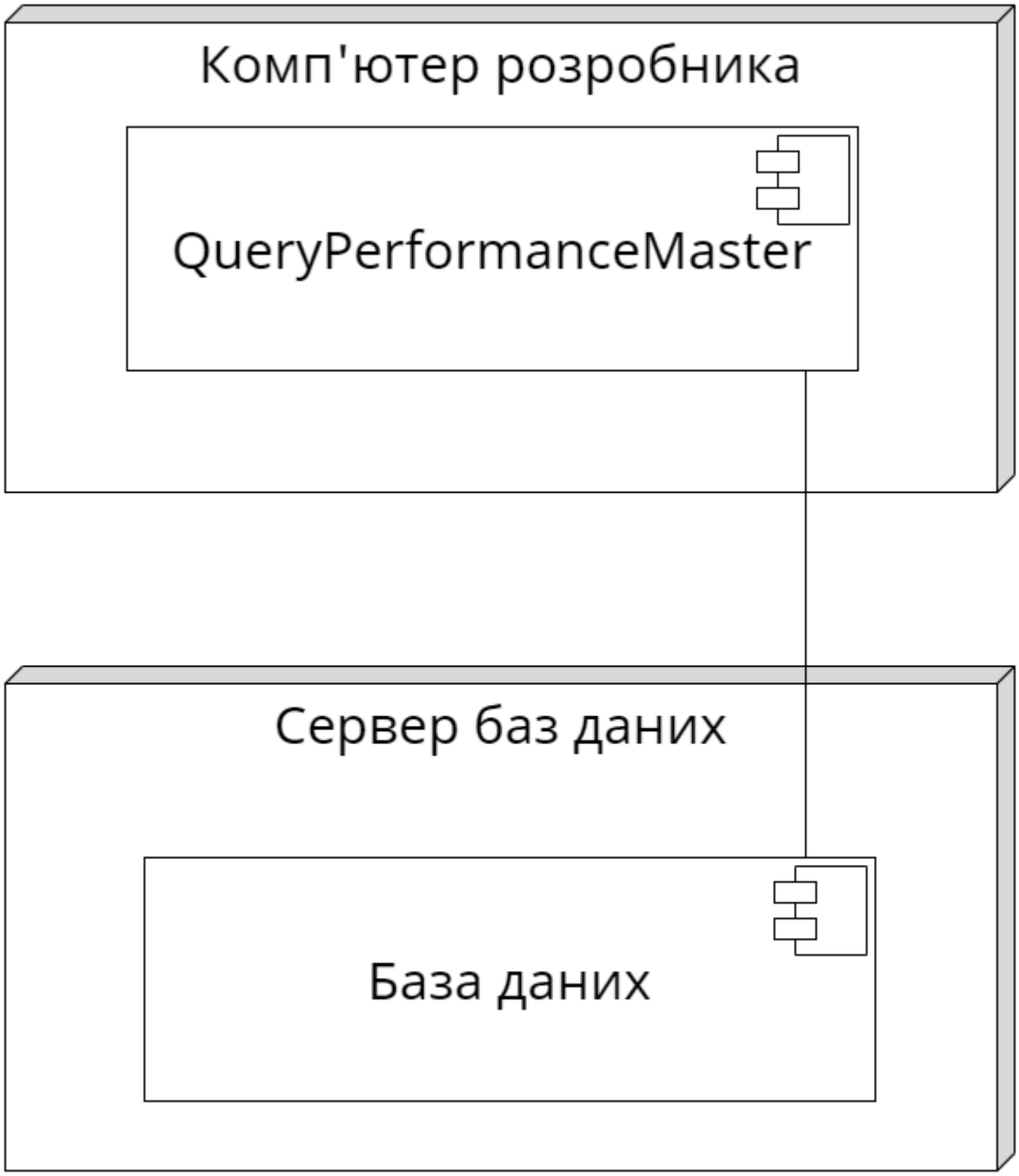


Рисунок 3.6.2 – Діаграма розгортання застосунку

4 ТЕСТУВАННЯ ЗАСТОСУНКУ

В рамках тестування функціоналу програмного застосунку, який зображений на рисунку 4.1, для оцінки програмного застосунку, з метою визначення, чи відповідає він заданим вимогам і чи працює за призначенням, було проведено два види тестування:

– Модульне тестування (Unit testing): тестування окремих компонентів або модулів програми за допомогою програмних методів. Для окремих компонентів будуть написані модульні тести.
– Мануальне тестування: весь заявлений функціонал буде протестований вручну.

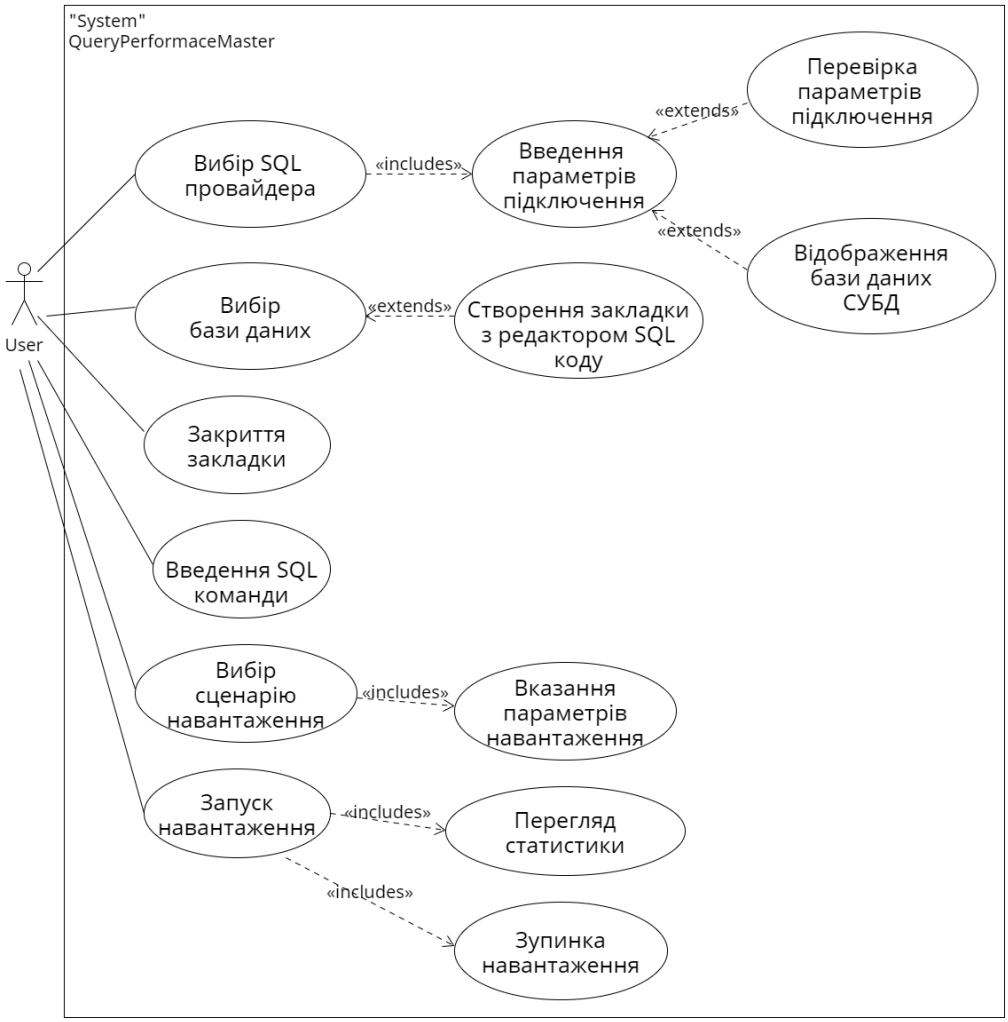


Рисунок 4.1 – Діаграма взаємодії користувача з застосунком

4.1 Модульне тестування сценаріїв навантаження

Для реалізація модульного тестування інфраструктурної частини застосунку був доданий проект QueryPerformanceMaster.Core.UnitTests, який, використовуючи фреймворк xUnit.net, реалізує модульні тести для сценаріїв виконання навантаження. На рисунку 4.1.1 зображено структуру проекту, який містить модульні тести.

При написані модульних тестів виникла необхідність використання фреймворка, який дозволяє імітувати або емулювати якусь функціональність або створювати мок-обєкти, в якості такого було вирішено використовувати фреймворк Moq, як найпопулярніший на платформі .NET.

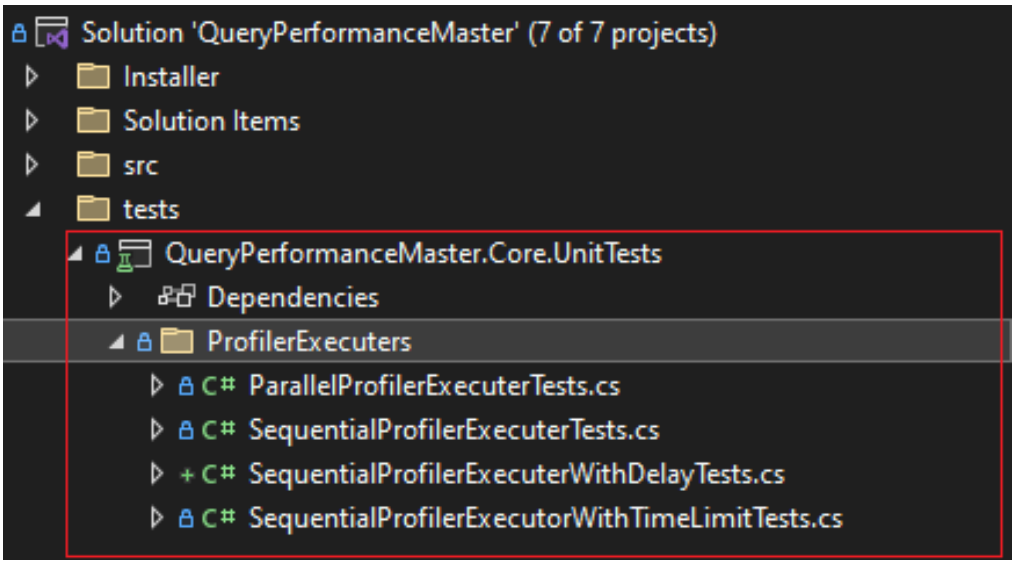


Рисунок 4.1.1 – Проект для модульного тестування

В рамках реалізації модульних тестів було передбачено чотири класи, які виконують модульне тестування чотирьох сценаріїв навантаження. Кожний клас містить дві функції: тестування без переривання навантаження та з перериванням. В додатку Е наведено реалізацію модульних тестів.

Після запуску тестування можна переконатись, що всі тести пройшли перевірку. Результат тестування зображено на рисунку 4.1.2.

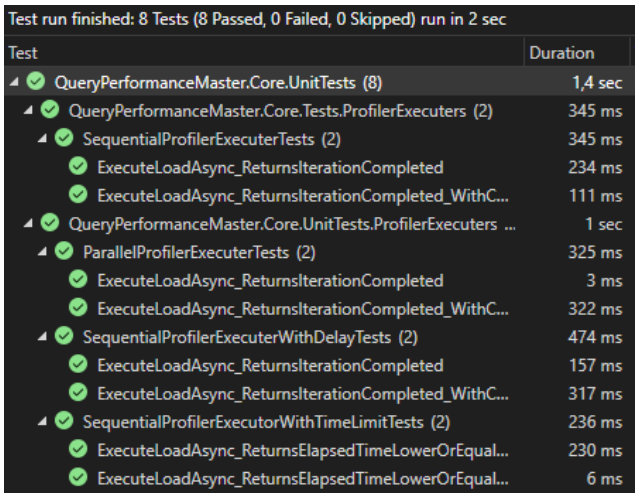


Рисунок 4.1.2 – Виконання тестів

4.2 Мануальне тестування застосунку

В рамках мануального тестування застосунку було проведено тестування всіх сценаріїв використання застосунку, які були заявлені при визначені функціональних вимог.

Перший прецедент, потрібно протестувати – це закриття/згортання головного вікна застосунку. Перевіряємо цей функціонал та переконуємося, що він працює. Зображено на рисунку 4.2.1.

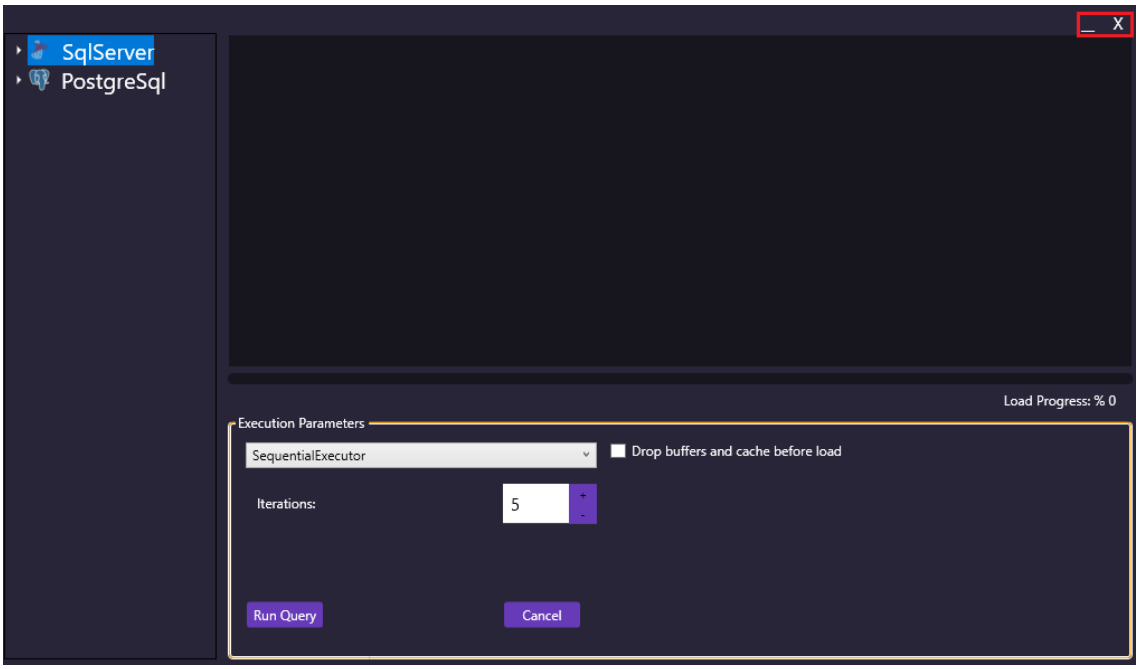


Рисунок 4.2.1 – Тестування першого прецеденту

Наступний прецедент, який необхідно протестувати – підключення до СУБД SQL Server. Для цього намагаємось розгорнути вузол під назвою SqlServer, бачимо вікно підключення до СУБД, яке зображено на рисунку 4.2.2.

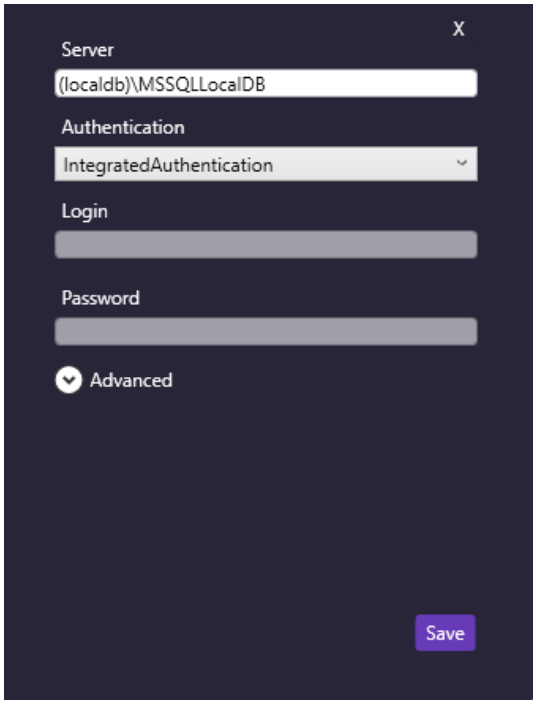


Рисунок 4.2.2 – Вікно підключення до SqlServer

Намагаємось підключитись за допомогою перевірки автентичності Windows (IntegratedAuthentication). Переконуємося, що застосунок зміг підключитись до СУБД та отримати перелік баз даних. Наведено на рисунку 4.2.3.

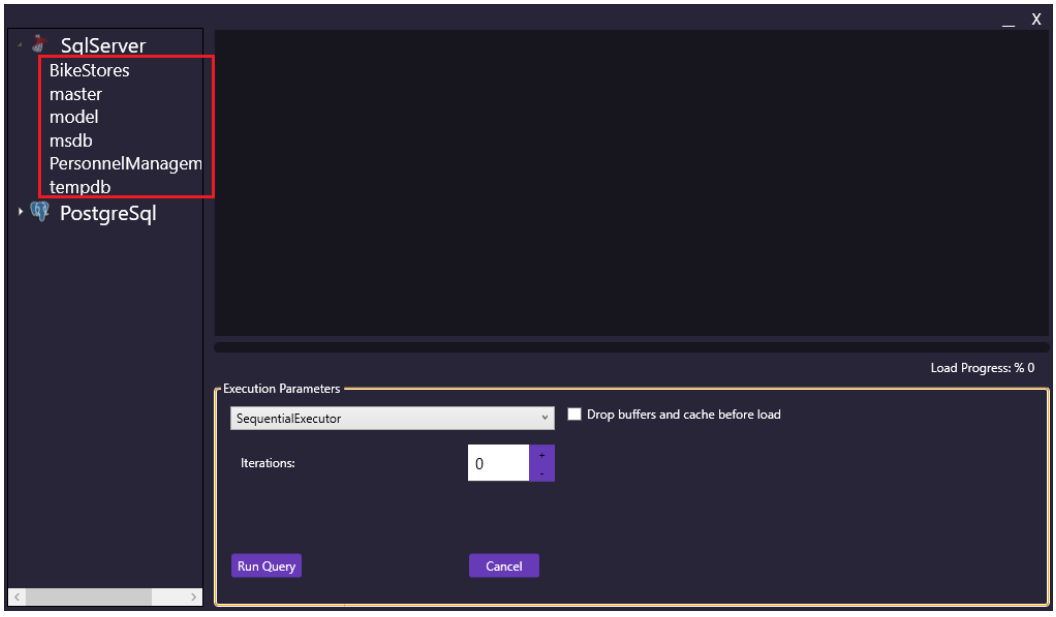


Рисунок 4.2.3 – Успішне підключення до SqlServer

Спробуємо зайти в параметри підключення до SqlServer та спробуємо підключитись до СУБД за допомогою перевірки автентичності SqlServer (SqlServerAuthentication). Наведено на рисунку 4.2.4.

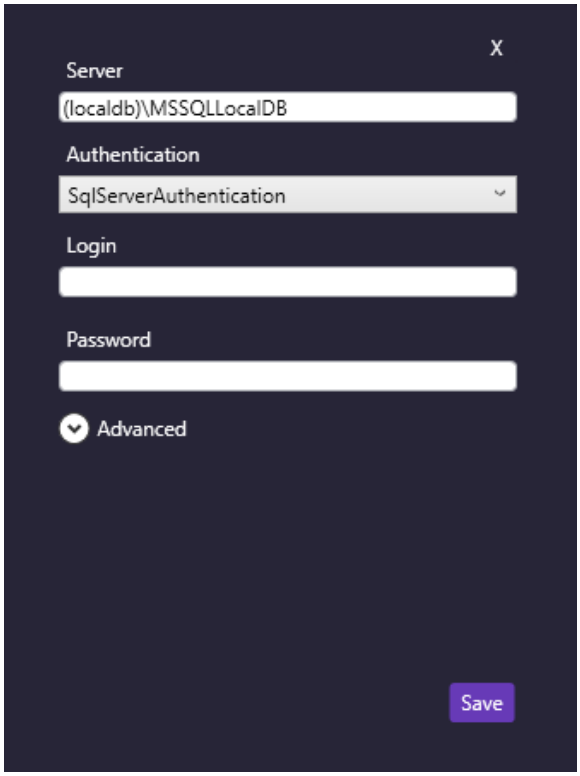


Рисунок 4.2.4 – Вікно підключення до SqlServer (SqlServerAuthentication)

Для цього створимо нового користувача через Microsoft SQL Server Management Studio. Зображено на рисунку 4.2.5.

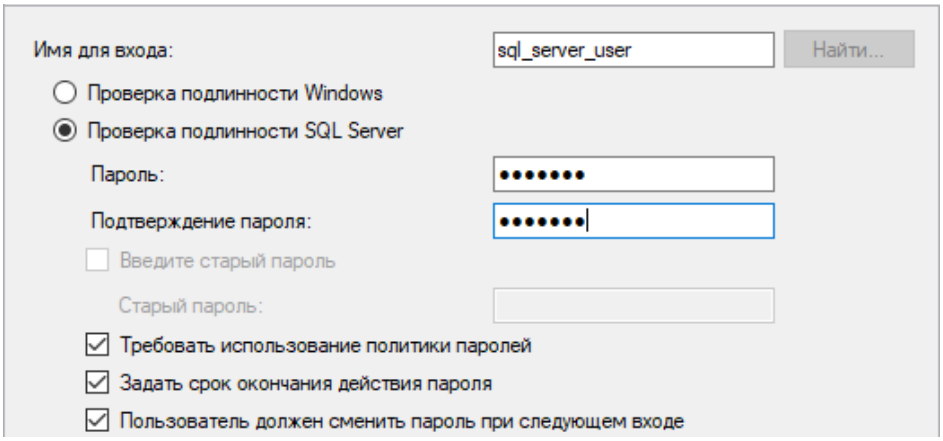


Рисунок 4.2.5 – Створення користувача в MSSQLMS

Намагаємось під’ єднатись до SqlServer з обліковими даними нового користувача. Екран підключення від нового користувача зображено на рисунку 4.2.6.

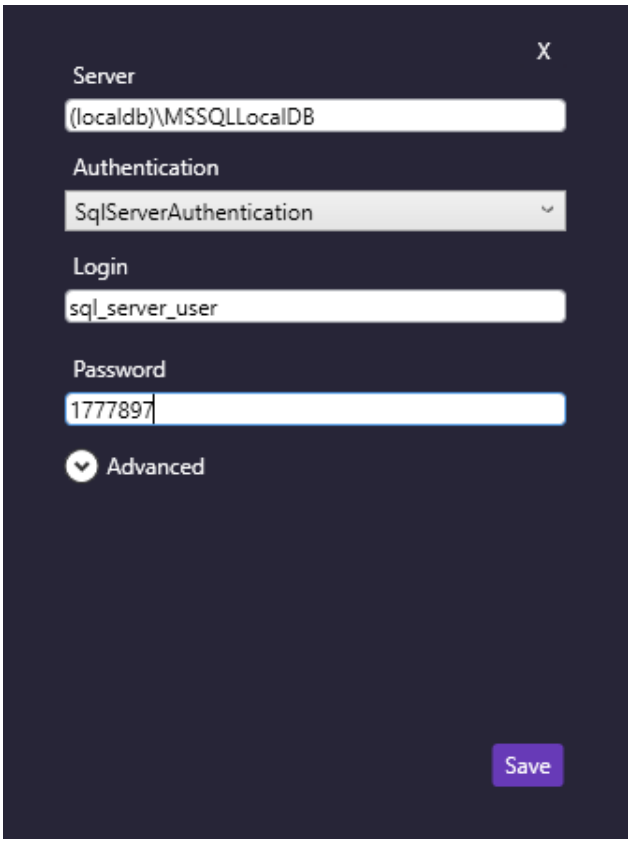


Рисунок 4.2.6 – Параметри підключення SqlServer від імені нового користувача

Переконуємось, що під’єднання пройшло успішно і бази даних були отримані. Успішне підключення до SqlServer від імені нового користувача зображено на рисунку 4.2.7.

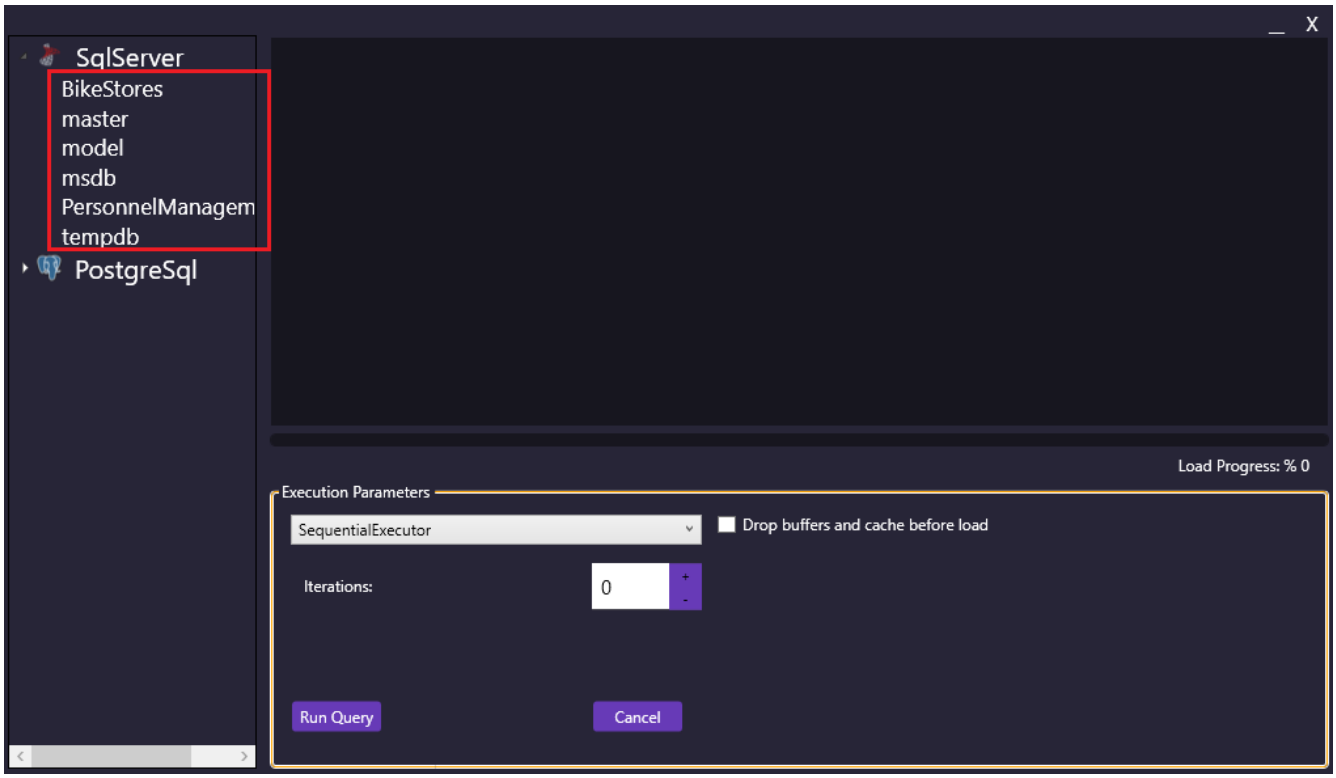


Рисунок 4.2.7 – Успішне підключення до PostgreSQL
від імені нового користувача

Наступний прецедент, який буде протестований – створення та виконання сценаріїв навантаження команди. Для цього в базу даних Bike Store було додано перелік таблиць, які продемонстровано на рисунку 4.2.8, для яких буде написано SQL команду та виконані сценарії навантаження.

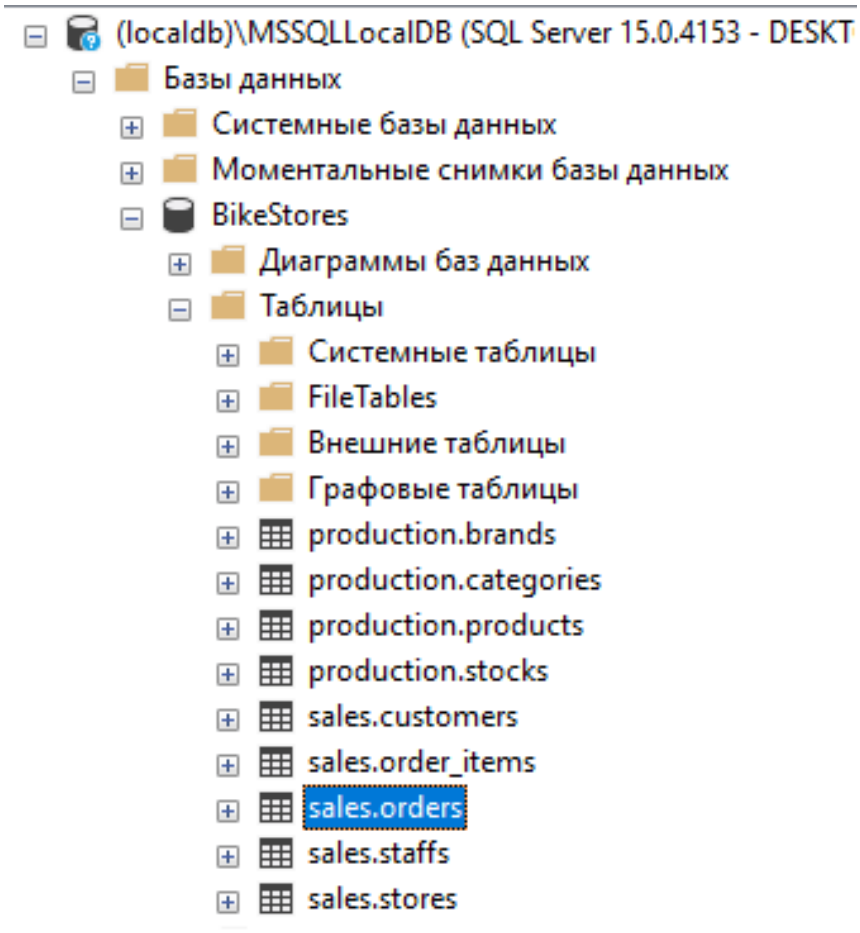


Рисунок 4.2.8 – Перелік таблиць бази даних BikeStore

Для перевірки виконання сценаріїв навантаження було написано наступну команду:

```
select * from sales.orders
where customer_id =
(select max(customer_id) from sales.customers t where t.first_name = 'Tameka'
and t.last_name = 'Fisher')
```

Яка вибирає id користувача з таблиці Customers по прізвищу та імені, і по цьому користувачу шукає замовлення в таблиці Orders. Вкажемо цю команду в редактор команди на закладці та перевіримо роботу сценаріїв навантаження. На рисунку 4.2.9 зображений редактор команд з вказаною командою.

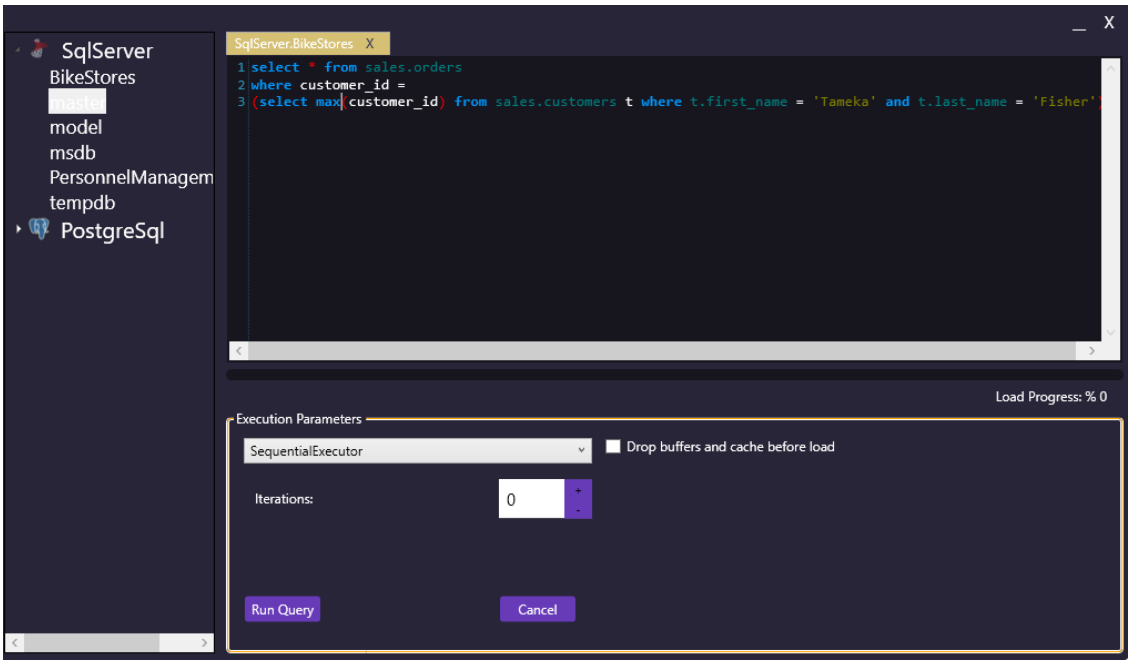


Рисунок 4.2.9 – Команда вказана в редакторі

Спершу спробуємо перевірити роботу послідовного сценарію навантаження, для цього у випадаючому списку обираємо варіант SequentialExecutor. Бачимо, що з’явився один параметр навантаження – Iterations (кількість ітерацій). Вказуємо значення та починаємо навантаження. Підчас навантаження переконуємось, що прогрес бар сигналізує про відсоток завершеної роботи. Зображено на рисунку 4.2.10.

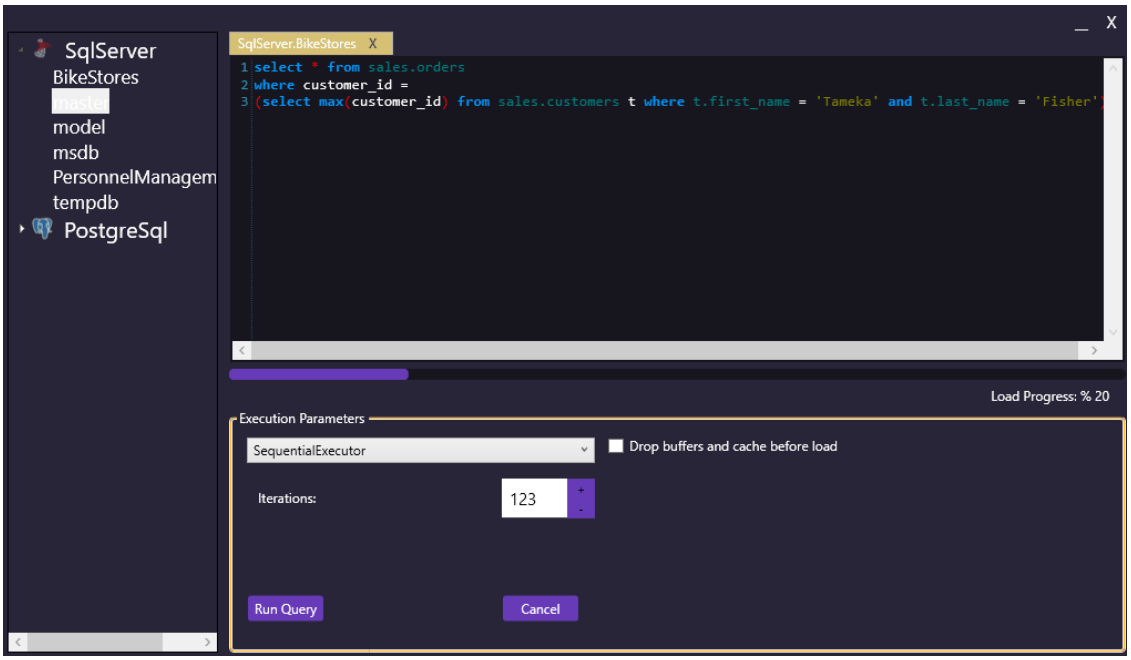


Рисунок 4.2.10 – Головний екран підчас навантаження

Після того, як шкала виконання дійшла до кінця, бачимо що відкрилось вікно з результатом виконання навантаження. На рисунку 4.2.11 зображено екран статистики.

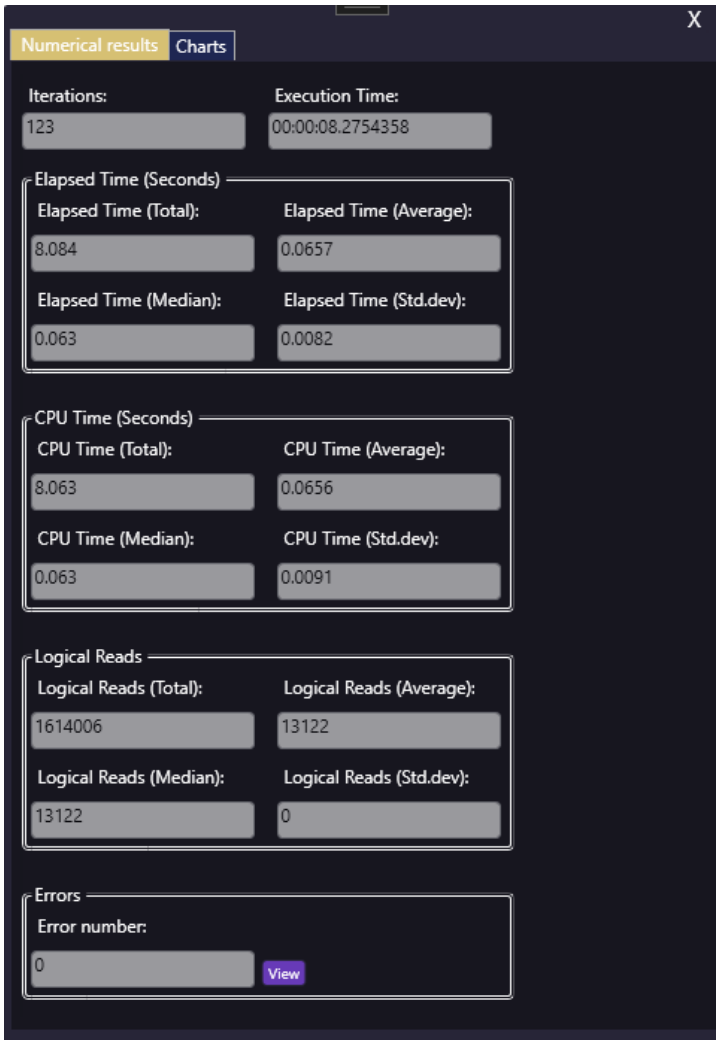


Рисунок 4.2.11 – Екран статистики навантаження. Числові значення

Далі аналізуємо показані дані. Зверху вікна бачимо два параметра: Iterations (кількість ітерацій), Execution Time (клієнтський час виконання). Робимо висновок з цих двох параметрів, що заявлені 123 ітерації виконались за 8.27 секунд.

Перший сектор з назвою Elapsed Time (Seconds), який містить дані про час виконання команд в секундах:

– Elapsed Time (Total) - містить загальний час виконання всіх команд в секундах, який повернула СУБД після виконання команди, отримане значення – 8.084 секунд.
– Elapsed Time (Average) – містить середнє арифметичне значення часу

виконання команд, отримане значення – 0.00656 секунд.

- Elapsed Time (Median) – містить медіану часу виконання команд. Отримане значення – 0.063.
- Elapsed Time (Std.dev) – містить середнє квадратичне відхилення часу виконання команд, отримане значення – 0.0082. По малому отриманому значенню робимо висновок, що міра розсіювання у наборі числових даних знаходиться у нормі.

Посередині вікна знаходиться сектор з назвою CPU Time (Seconds), який містить інформацію про процесорний час при виконанні команд в секундах:

- CPU Time (Total) - містить процесорний час виконання всіх команд в секундах, який повернула СУБД після виконання команди, отримане значення – 8.063 секунд.
- CPU Time (Average) – містить середнє арифметичне значення процесорного часу виконання команд, отримане значення – 0.00656 секунд.
- CPU Time (Median) – містить медіану процесорного часу виконання команд. Отримане значення – 0.063.
- CPU Time (Std.dev) – містить середнє квадратичне відхилення процесорного часу виконання команд, отримане значення – 0.0091. Дане значення свідчить про згуртованість числового набору навколо середнього значення, що свідчить про достовірність отриманих даних.

Наступний сектор з назвою Logical Reads (читання з диску), який містить кількість читань з бази даних, яку довелось зробити при виконанні навантаження:

- Logical Reads (Total) - містить кількість читань з диску всіх команд, який повернула СУБД після виконання команди.
- Logical Reads (Average) – містить середнє арифметичне значення кількості читань з диску виконаних команд.
- Logical Reads (Median) – містить медіану кількості читань з диску виконаних команд.

– Logical Reads (Std.dev) – містить середнє квадратичне відхилення кількості читань з диску виконаних команд.

В даному випадку кожна команда повернула одне значення кількості читань з диску – 13122, тому дані по середньому значенню, медіані та стандартному відхиленню є однаковими.

Далі перевіримо вміст закладки з графічним відображенням значень, переходимо на наступну закладу. На рисунку 4.2.12 зображений вміст закладки з графічним відображенням статистики.

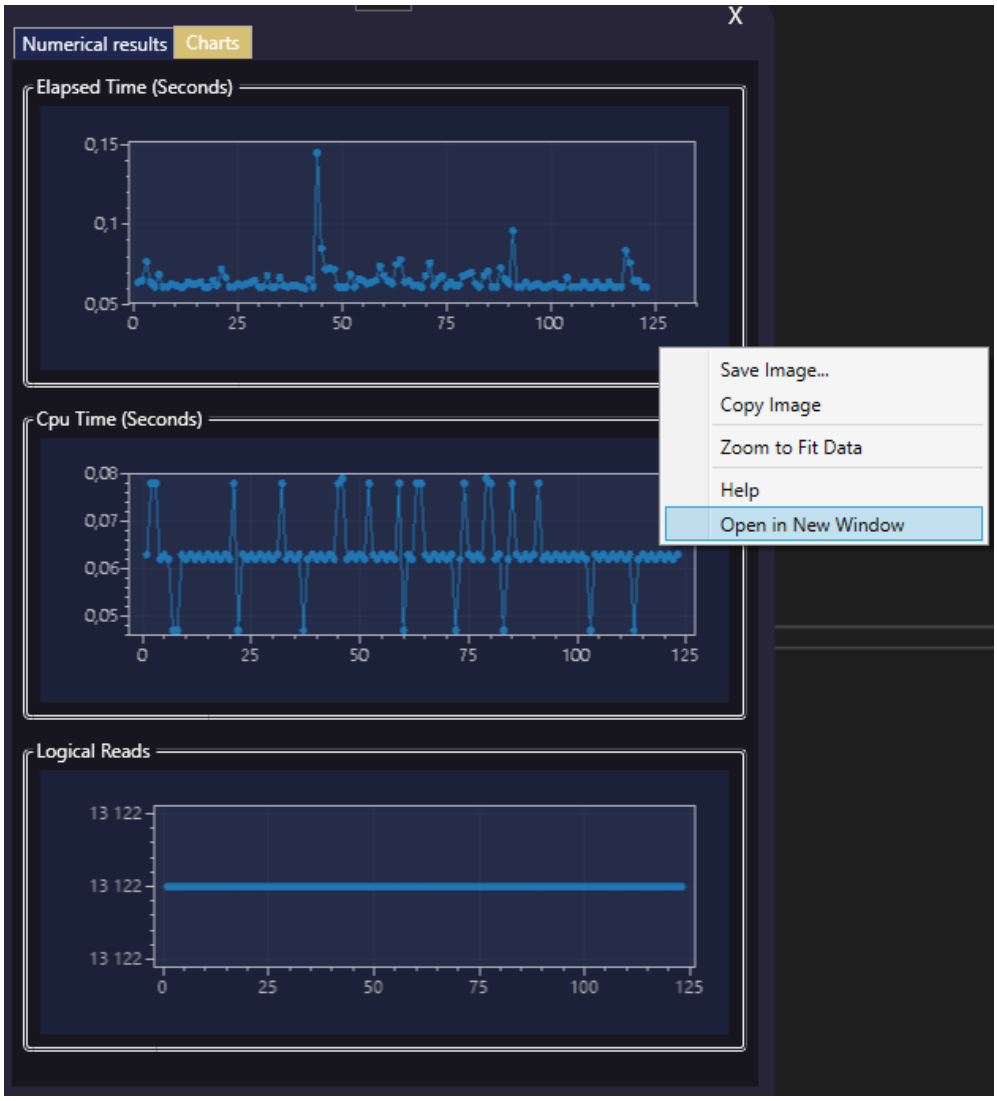


Рисунок 4.2.12 – Екран статистики навантаження. Графіки

При переході на закладку бачимо три графіка, які показують відображення трьох зібраних значень підчас проведення навантаження. Через натиск правої кнопки миші на графіку можна розкрити графік на весь екран для зручного аналізу

даних. При перегляді графіків можна проаналізувати отримані значення на кожній ітерації. На рисунку 4.2.13 зображений графік по часу виконання команд в секундах.

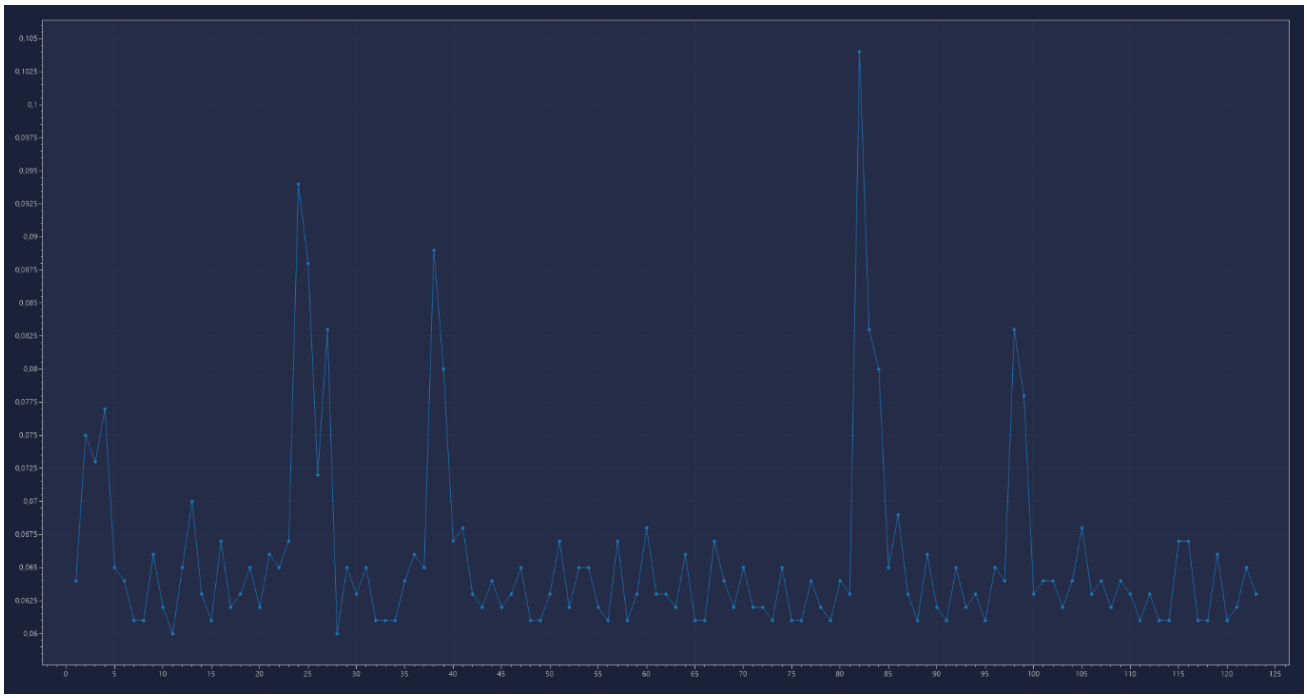


Рисунок 4.2.13 – Графік по часу виконання команд

Наступний функціонал, який необхідно протестувати – навантажування PostgreSQL. Для цього намагаємось підключитись до PostgreSQL. В головному екрані бачимо елемент, який зображений на рисунку 4.2.14.

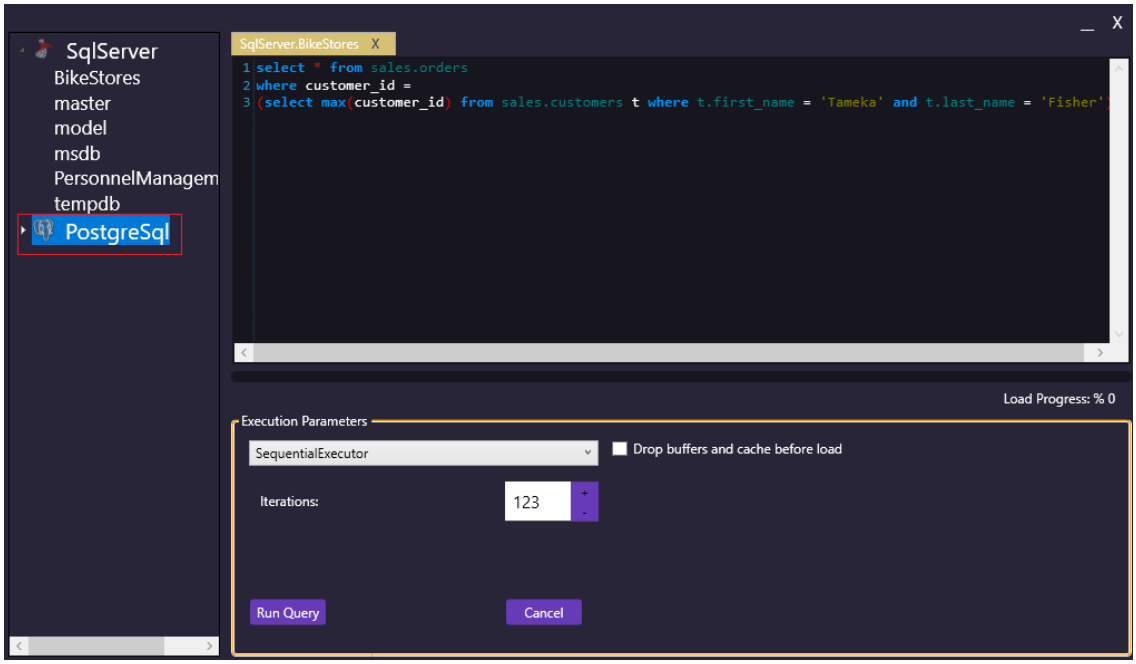


Рисунок 4.2.14 – PostgreSQL в головному екрані

Як і в випадку з SqlServer – намагаємось розгорнути ієрархічний елемент та бачимо вікно параметрів підключення до PostgreSQL, яке показане на рисунку 4.2.15.

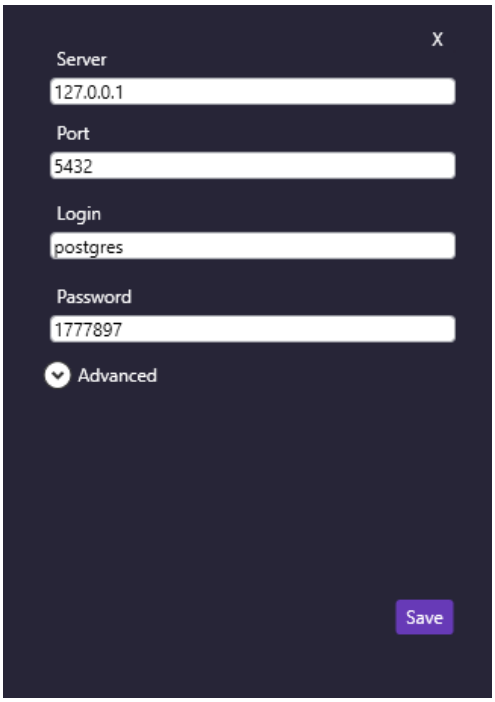


Рисунок 4.2.15 – Параметри підключення до PostgreSQL

Намагаємось зберегти налаштувань підключення та переконуємось, що збереження пройшло успішно і перелік баз даних був отриманий.

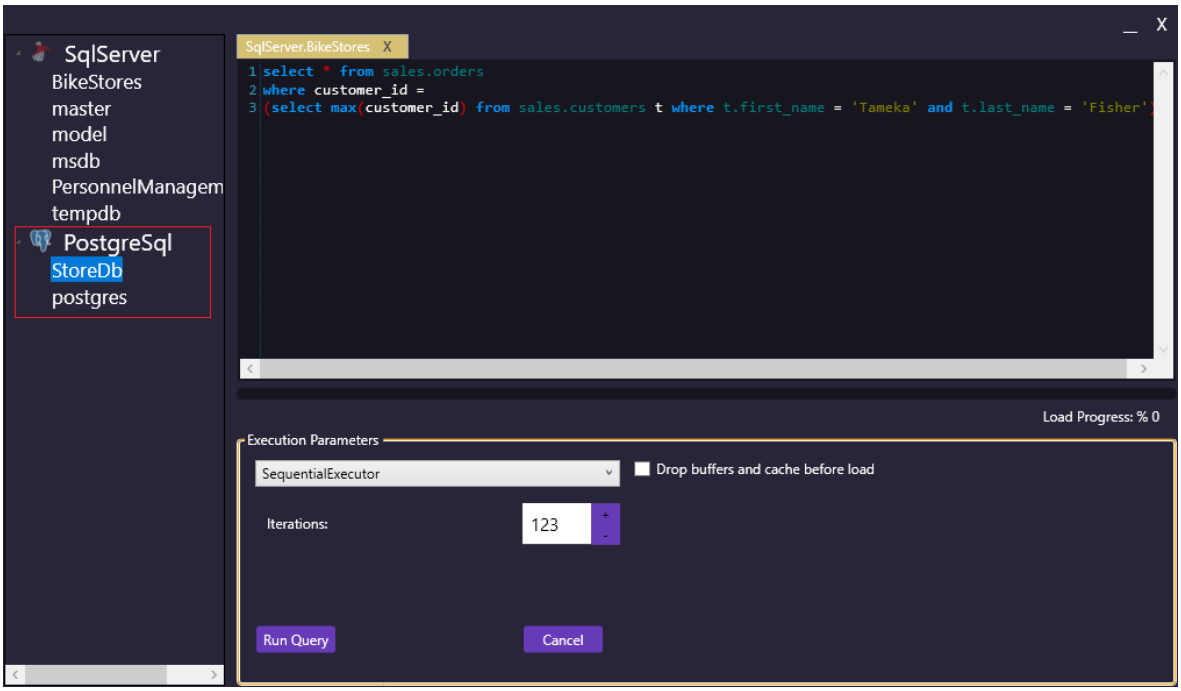


Рисунок 4.2.16 – Перелік баз даних PostgreSQL

В рамках перевірки навантажувального тестування PostgreSQL було створено базу даних StoreDb, яку бачимо серед переліку баз даних PostgreSQL та декілька таблиць, які зображені на рисунку 4.2.17.

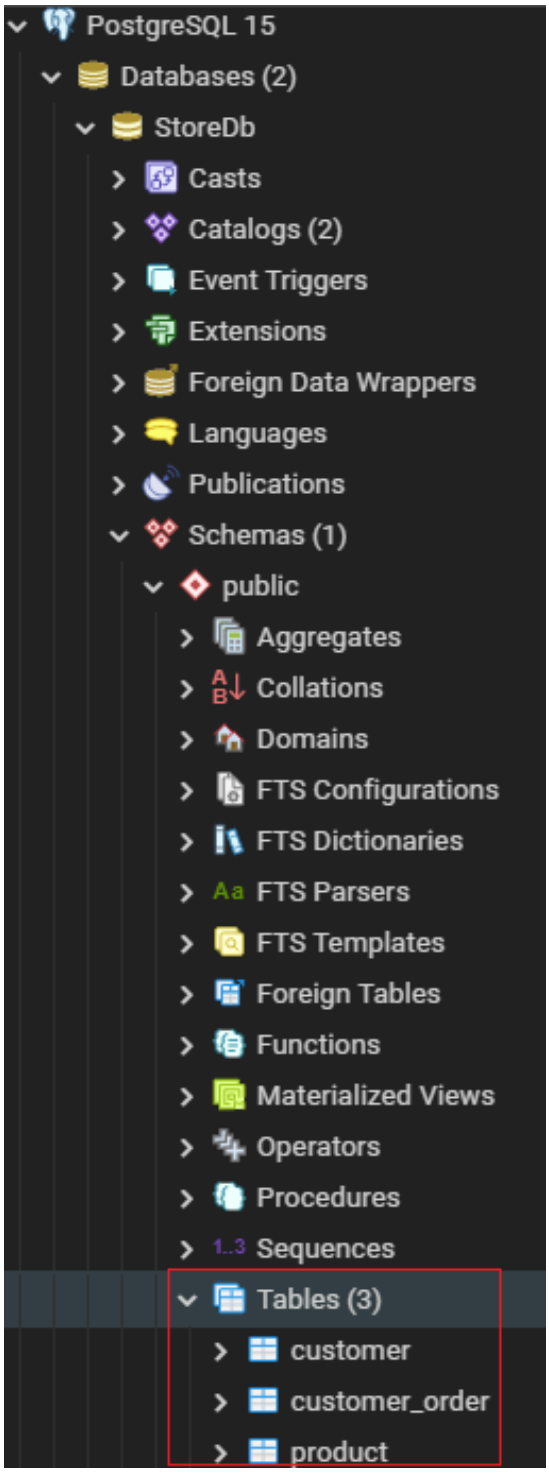


Рисунок 4.2.17 – Цільова база даних PostgreSQL

Зображені на рисунку 4.2.17 таблиці будуть використані для навантажувального тестування цієї СУБД. Для цього напишемо наступну команду:

```
select id from customer_order where product_id =
    (select id from product where product_name = 'product 54')
```

Ця команда вибирає id продукту по назві в таблиці Product, та по знайденому продукту шукає замовлення користувачів з таблиці Customer_Order.

Для тестування цієї команди створимо закладку для бази даних StoreDb та запишемо вказану команду в редактор команди, який знаходиться на закладці.

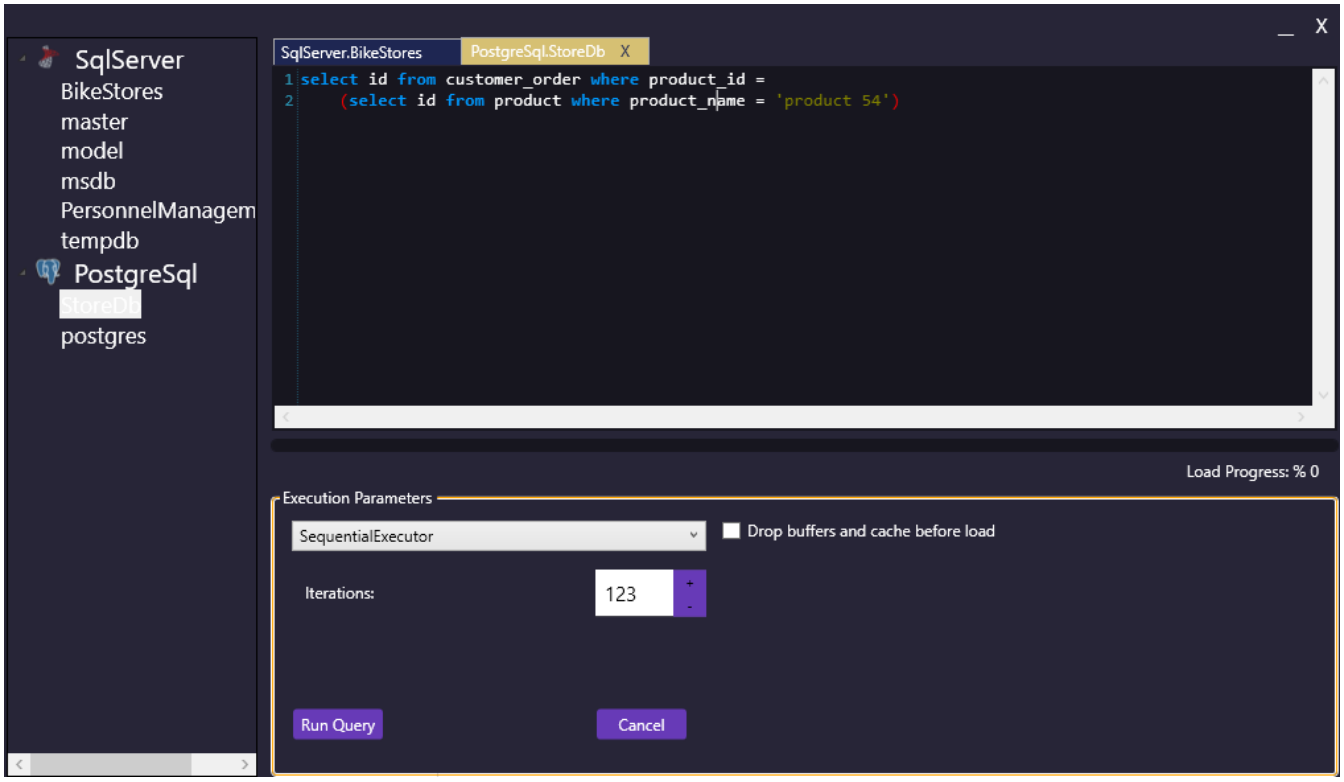


Рисунок 4.2.18 – Створена команда для навантаження PostgreSQL

Виконуємо навантажувальне тестування для цієї команди та отримуємо результат, який зображено на рисунку 4.2.19.

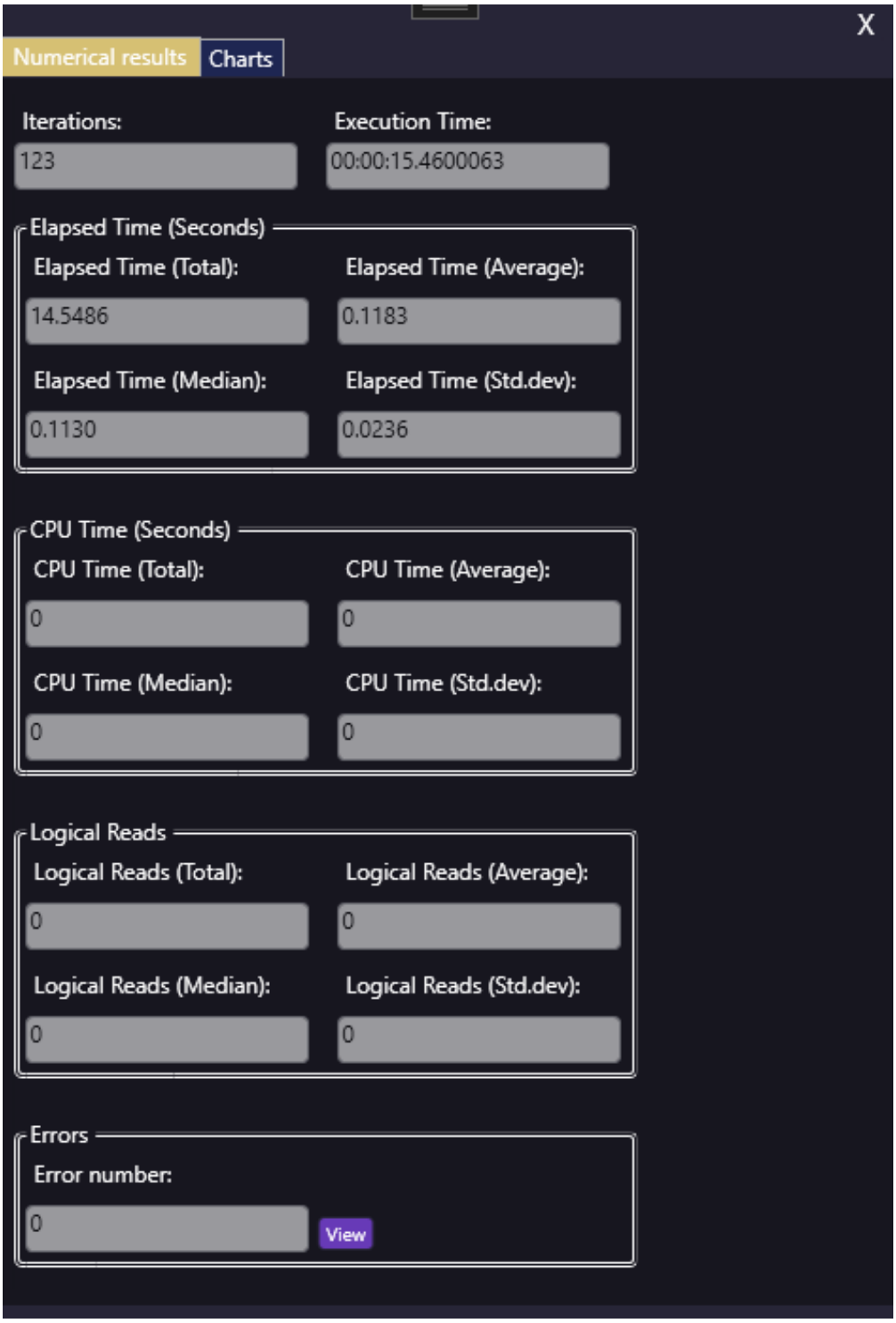


Рисунок 4.2.19 – Результат навантаження PostgreSQL

На рисунку 4.2.19 бачимо, що окрім часу виконання команд більше статистики через СУБД надано не було.

Таким чином було протестовані всі описано прецеденти і визначено, що розроблене програмне забезпечення готове до експлуатації.

ВИСНОВКИ

Таким чином, розроблено програмний продукт для проведення навантажувального тестування SQL запитів, який має можливість виконувати навантаження для різних СУБД і може розширювати їх список, та поєднує у собі всі сценарії навантаження з проаналізованих програмних рішень та може розширювати список сценаріїв, при потребі користувачів. В ході виконання роботи було отримано наступні результати:

1) Проведено аналіз існуючих програмних рішень по проведенню навантажувального тестування, який дав інформацію про те, які недоліки існуючі рішення мають та показав актуальність створення програмного продукту, побудованого на основі «Чистої архітектури», яка покладається на принципи інверсії залежностей та декомпозиції за рівнями.
2) Вибрано засоби розробки застосунку: .NET стек, що складається з наступних компонентів:
   a) Загальні компоненти:
      – мова програмування: C# 10;
      – версія платформи: .NET 6;
      – Архітектура: Onion architecture (Архітектура Луковиці);
   b) Компоненти візуальної частини:
      – Система побудови клієнтських застосунків: WPF;
      – Архітектура клієнтської частини: MVVM;
      – MVVM фреймворк: MvvmCross;
   c) Компоненти, які дозволяють інсталювати застосунок:
      – формат пакетів програм Windows: MSIX;
   d) Компоненти, які дозволяють проводити модульне тестування:
      – фреймворк, для модульного тестування: xUnit;
      – фреймворк, для імітації роботи об’єктів: Moq;
3) Розроблено архітектурні рішення для побудови ядра застосунку та клієнтської частини.

4) Розроблено програмне забезпечення.
5) Написано модульні тести для перевірки правильної роботи сценаріїв навантаження.
6) Проведено мануальне тестування застосунку.
7) Продемонстровано тестові приклади проведення навантажувального тестування за допомогою розробленого застосунку.
8) Результати тестування підтвердили працездатність застосунку та відповідність поставленим вимогам.

Результати виконаної роботи можуть бути використані розробниками для проведення навантажувального тестування SQL команд для таких СУБД: SqlServer, PostgreSQL.

В подальших дослідженнях слід розширити перелік СУБД для яких можна проводити навантажувальне тестування, перелік сценаріїв навантаження, ґрунтуючись на отриманих відгуках користувачів. Також необхідно розробити можливість способу взаємодії з застосунком через CLI (інтерфейс командного рядка), виконуючи команди через термінал з використанням інтерактивних запитів командного рядка. Крім того, слід додати можливість вивантажувати отримані результати в вихідні файли: excel, word, txt, для забезпечення можливості збереження результатів тестування та їх аналіз в подальшому.

ПЕРЕЛІК ПОСИЛАНЬ

1. Markus Winand, “SQL Performance Explained Everything Developers Need to Know about SQL Performance”, 2012. – 6 с.

2. SQL Performance Testing Tools. [Електронний ресурс]. — Режим доступу: https://www.dnsstuff.com/sql-performance-testing.

3. SqlQueryStress Documentation [Електронний ресурс] — Режим доступу: https://github.com/ErikEJ/SqlQueryStress/wiki.

4. SQL Query Stress Tool. [Електронний ресурс]. — Режим доступу: https://www.mssqltips.com/sqlservertip/2730/sql-query-stress-tool/.

5. SQLTest Documentation [Електронний ресурс] — Режим доступу: https://www.sqltest.org/.

6. Understanding Onion Architecture. [Електронний ресурс]. — Режим доступу: https://www.codeguru.com/csharp/understanding-onion-architecture/.

7. .NET Documentation. [Електронний ресурс]. — Режим доступу: https://learn.microsoft.com/ru-ru/dotnet/framework/get-started/overview.

8. WPF .NET Documentation. [Електронний ресурс]. — Режим доступу: https://learn.microsoft.com/ru-ru/dotnet/desktop/wpf/overview/?view=netdesktop-7.0.

9. MvvmCross Documentation. [Електронний ресурс]. — Режим доступу: https://www.mvvmcross.com/documentation/getting-started/getting-started.

Додаток А

Реалізація MsSqlLoadProfiler і PostgreSqlLoadProfiler

1. MsSqlLoadProfiler:

```
internal class MsSqlLoadProfiler : ILoadProfiler
    {
        private readonly SqlConnectionParams _connectionParams;
        private readonly IMsSqlConnectionProviderFactory _connectionProviderFactory;
        private static readonly Regex _queryLogicalReads = new Regex(
            @"(?:Table (\'\w{1,}\'|'#\w{1,}\'|'##\w{1,}\'). Scan count \d{1,}, logical
reads )(\d{1,})", RegexOptions.Compiled);
        private static readonly Regex _queryTimes =
                new Regex(
                    @"(?:SQL Server Execution Times:|SQL Server parse and compile
time:)(?:\s{1,}CPU time = )(\d{1,})(?: ms,\s{1,}elapsed time = )(\d{1,})",
                    RegexOptions.Compiled);

        private const string _statisticsCommand = "SET STATISTICS IO ON; SET STATISTICS
TIME ON;";

        public MsSqlLoadProfiler(SqlConnectionParams connectionParams,
            IMsSqlConnectionProviderFactory connectionProviderFactory)
        {
            _connectionParams = connectionParams;
            _connectionProviderFactory = connectionProviderFactory;
        }

        public async Task<LoadProfilerResult> ExecuteQueryLoadAsync(string query,
            CancellationToken cancellationToken = default)
        {
            var sqlQueryLoadResult = new LoadProfilerResult();
            var isTaskCancelled = false;

            cancellationToken.Register(() =>
            {
                isTaskCancelled = true;
            });

            SqlInfoMessageEventHandler infoMessageHandler = (sender, e) =>
            {
                foreach (SqlError erorr in e.Errors)
                {
                    var matches = _queryLogicalReads.Split(erorr.Message);

                    if (matches.Length > 1)
                    {
                        sqlQueryLoadResult.LogicalReads += Convert.ToInt32(matches[2],
CultureInfo.InvariantCulture);
                        continue;
                    }

                    matches = _queryTimes.Split(erorr.Message);

                    if (matches.Length > 1)
                    {
                        sqlQueryLoadResult.CpuTime += Convert.ToInt32(matches[1],
CultureInfo.InvariantCulture);
                        sqlQueryLoadResult.ElapsedTime += Convert.ToInt32(matches[2],
CultureInfo.InvariantCulture);
                    }
```

```
                }
            };

            var sw = new Stopwatch();
            var connectionProvider =
_connectionProviderFactory.GetConnectionProvider(_connectionParams.ConnectionString);
            SqlConnection sqlConnection = null;

            try
            {
                sqlConnection = await
connectionProvider.CreateConnection(cancellationToken);

                await using var statCommand = sqlConnection.CreateCommand();
                statCommand.CommandText = _statisticsCommand;
                statCommand.Connection.InfoMessage += infoMessageHandler;
                await statCommand.ExecuteNonQueryAsync(cancellationToken);

                await using var cmd = sqlConnection.CreateCommand();
                cmd.CommandText = query;

                sw.Start();

                await cmd.ExecuteNonQueryAsync(cancellationToken);

                sw.Stop();
            }
            catch (Exception ex)
            {
                if (!isTaskCancelled)
                {
                    sqlQueryLoadResult.SqlQueryLoadError = ex.Message;
                }
                sw.Stop();
            }
            finally
            {
                if (sqlConnection != null)
                {
                    sqlConnection.InfoMessage -= infoMessageHandler;
                    await sqlConnection.CloseAsync();
                }
            }

            sqlQueryLoadResult.ExecTime = sw.Elapsed;

            sw.Reset();

            return sqlQueryLoadResult;
        }
        }
```

2. PostgreSqlLoadProfiler:

```
internal class PostgreSqlLoadProfiler : ILoadProfiler
    {
        private readonly SqlConnectionParams _connectionParams;
        private readonly IPostgreSqlConnectionProviderFactory _connectionProviderFactory;
        private static readonly Regex _queryLogicalReads = new Regex(
            @"Buffers: shared read=(\d{1,})?",
            RegexOptions.Compiled);
        private static readonly Regex _queryTimes =
```

```
                new Regex(
                    @"Execution Time: (\d+\.\d+)? ms",
                    RegexOptions.Compiled);

        private const string _statisticsCommand = "EXPLAIN (ANALYZE, BUFFERS)";

        public PostgreSqlLoadProfiler(SqlConnectionParams connectionParams,
            IPostgreSqlConnectionProviderFactory connectionProviderFactory)
        {
            _connectionParams = connectionParams;
            _connectionProviderFactory = connectionProviderFactory;
        }

        public async Task<LoadProfilerResult> ExecuteQueryLoadAsync(string query,
CancellationToken cancellationToken = default)
        {
            var sqlQueryLoadResult = new LoadProfilerResult();
            var isTaskCancelled = false;

            cancellationToken.Register(() =>
            {
                isTaskCancelled = true;
            });

            var sw = new Stopwatch();
            var connectionProvider =
_connectionProviderFactory.GetConnectionProvider(_connectionParams.ConnectionString);
            NpgsqlConnection sqlConnection = null;

            try
            {
                sqlConnection = await
connectionProvider.CreateConnection(cancellationToken);

                await using var cmd = sqlConnection.CreateCommand();
                cmd.CommandText = string.Concat(_statisticsCommand, " ", query);

                sw.Start();

                var reader = cmd.ExecuteReader();

                while (await reader.ReadAsync(cancellationToken))
                {
                    var textReader = await reader.GetTextReaderAsync(0,
cancellationToken);
                    var queryPlanRow = await textReader.ReadLineAsync();

                    var matches = _queryTimes.Split(queryPlanRow);
                    if (matches.Length > 1)
                    {
                        sqlQueryLoadResult.ElapsedTime += Convert.ToDouble(matches[1],
CultureInfo.InvariantCulture);
                        continue;
                    }

                    matches = _queryLogicalReads.Split(queryPlanRow);
                    if (matches.Length > 1)
                    {
                        sqlQueryLoadResult.LogicalReads += Convert.ToDouble(matches[1],
CultureInfo.InvariantCulture);
                        continue;
                    }
                }

                sw.Stop();
```

```
            }
            catch (Exception ex)
            {
                if (!isTaskCancelled)
                {
                    sqlQueryLoadResult.SqlQueryLoadError = ex.Message;
                }
                sw.Stop();
            }
            finally
            {
                if (sqlConnection != null)
                {
                    await sqlConnection.CloseAsync();
                }
            }

            sqlQueryLoadResult.ExecTime = sw.Elapsed;

            sw.Reset();

            return sqlQueryLoadResult;
        }
       }
```

3. ILoadProfiler:

```
public interface ILoadProfiler
    {
        Task<LoadProfilerResult> ExecuteQueryLoadAsync(string query, CancellationToken
cancellationToken = default);
       }
```

Додаток Б

Реалізація ConnectionProvider

1. IConnectionProvider:

```
public interface IConnectionProvider<T>
    {
        Task<T> CreateConnection(CancellationToken cancellationToken = default);
    }
```

2. ConnectionProviderBase:

```
public abstract class ConnectionProviderBase<T> : IConnectionProvider<T>
    {
        public abstract Task<T> CreateConnection(CancellationToken
cancellationToken);
    }
```

3. PostgreSqlConnectionProvider:

```
internal class PostgreSqlConnectionProvider : ConnectionProviderBase<NpgsqlConnection>
    {
        private readonly string _connectionString;

        public PostgreSqlConnectionProvider(string connectionString)
        {
            _connectionString = connectionString;
        }

        public async override Task<NpgsqlConnection> CreateConnection(CancellationToken
cancellationToken = default)
        {
            var sqlConnection = new NpgsqlConnection(_connectionString);
            await sqlConnection.OpenAsync(cancellationToken);
            return sqlConnection;
        }
    }
```

4. PostgreSqlConnectionProviderFactory:

```
public class PostgreSqlConnectionProviderFactory : IPostgreSqlConnectionProviderFactory
    {
        public IConnectionProvider<NpgsqlConnection> GetConnectionProvider(string
connectionString)
        {
            var connectionProvider = new PostgreSqlConnectionProvider(connectionString);
            return connectionProvider;
        }
    }
```

5. MsSqlConnectionProvider:

```
public class MsSqlConnectionProvider : ConnectionProviderBase<SqlConnection>
    {
        private readonly string _connectionString;
```

```
        public MsSqlConnectionProvider(string connectionString)
        {
            _connectionString = connectionString;
        }

        public async override Task<SqlConnection> CreateConnection(CancellationToken
cancellationToken = default)
        {
            var sqlConnection = new SqlConnection(_connectionString);
            await sqlConnection.OpenAsync(cancellationToken);
            return sqlConnection;
        }
          }
```

6. MsSqlConnectionProviderFactory:

```
public class MsSqlConnectionProviderFactory : IMsSqlConnectionProviderFactory
    {
        public IConnectionProvider<SqlConnection> GetConnectionProvider(string
connectionString)
        {
            var connectionProvider = new MsSqlConnectionProvider(connectionString);
            return connectionProvider;
        }
    }
```

Додаток В

Реалізація сценаріїв навантаження

1. SequentialProfilerExecuter:

```
public class SequentialProfilerExecuter : ISequentialProfilerExecuter
    {
        private readonly ILoadProfiler _loadProfiler;

        public SequentialProfilerExecuter(ILoadProfiler loadProfiler)
        {
            _loadProfiler = loadProfiler;
        }

        public async Task<LoadExecutedResult> ExecuteLoadAsync(string query, int
iterationNumber,
            IProgress<int> queryLoadProgress = null, CancellationToken cancellationToken
= default)
        {
            var loadProfilerResult = new List<LoadProfilerResult>();

            for (int i = 1; i <= iterationNumber; i++)
            {
                queryLoadProgress?.Report((i * 100) / iterationNumber);

                var loadResult = await _loadProfiler.ExecuteQueryLoadAsync(query,
cancellationToken);
                loadProfilerResult.Add(loadResult);

                if (cancellationToken.IsCancellationRequested)
                {
                    return ProfilerExecuterHelpers.FillLoadExecutedResult(i,
loadProfilerResult);
                }
            }

            return ProfilerExecuterHelpers.FillLoadExecutedResult(iterationNumber,
loadProfilerResult);
        }
          }

public interface ISequentialProfilerExecuter
    {
        Task<LoadExecutedResult> ExecuteLoadAsync(string query, int iterationNumber,
            IProgress<int>? queryLoadProgress = null, CancellationToken cancellationToken
= default);
          }
```

2. ParallelProfilerExecuter

```
public class ParallelProfilerExecuter : IParallelProfilerExecuter
    {
        private readonly ILoadProfiler _loadProfiler;

        public ParallelProfilerExecuter(ILoadProfiler loadProfiler)
        {
            _loadProfiler = loadProfiler;
        }
```

```
        public async Task<LoadExecutedResult> ExecuteLoadAsync(string query, int
threadNumber, int iterationNumber,
            IProgress<int> queryLoadProgress = null, CancellationToken cancellationToken
= default)
        {
            var tasks = new List<Task<List<LoadProfilerResult>>>();

            for (int i = 1; i <= threadNumber; i++)
            {
                tasks.Add(ExecuteQueryLoad(query, iterationNumber,
                    threadNumber * iterationNumber, cancellationToken,
queryLoadProgress));
            }

            var completedTasks = await Task.WhenAll(tasks);
            var loadProfilerResult = completedTasks.SelectMany(x => x).ToList();

            return
ProfilerExecuterHelpers.FillLoadExecutedResult(loadProfilerResult.Count,
loadProfilerResult);
        }

        private int _currentLoadProgress;
        private async Task<List<LoadProfilerResult>> ExecuteQueryLoad(string query, int
iterationNumber,
            int finalIterationNumber, CancellationToken cancellationToken, IProgress<int>
queryLoadProgress = null)
        {
            var loadProfilerResult = new List<LoadProfilerResult>();
            for (int i = 1; i <= iterationNumber; i++)
            {
                queryLoadProgress?.Report((++_currentLoadProgress * 100) /
finalIterationNumber);

                var loadResult = await _loadProfiler.ExecuteQueryLoadAsync(query,
cancellationToken);
                loadProfilerResult.Add(loadResult);

                if (cancellationToken.IsCancellationRequested)
                {
                    return loadProfilerResult;
                }
            }
            return loadProfilerResult;
        }
      }

public interface IParallelProfilerExecuter
    {
        Task<LoadExecutedResult> ExecuteLoadAsync(string query, int threadNumber, int
iterationNumber,
            IProgress<int>? queryLoadProgress = null, CancellationToken cancellationToken
= default);
          }
```

3. SequentialProfilerExecuterWithDelay:

```
public class SequentialProfilerExecuterWithDelay : ISequentialProfilerExecuterWithDelay
    {
        private readonly ILoadProfiler _loadProfiler;
```

```
        public SequentialProfilerExecuterWithDelay(ILoadProfiler loadProfiler)
        {
            _loadProfiler = loadProfiler;
        }

        public async Task<LoadExecutedResult> ExecuteLoadAsync(string query, int
iterationNumber, int delayMiliseconds,
            IProgress<int> queryLoadProgress = null, CancellationToken cancellationToken
= default)
        {
            var loadProfilerResult = new List<LoadProfilerResult>();

            for (int i = 1; i <= iterationNumber; i++)
            {
                queryLoadProgress?.Report((i * 100) / iterationNumber);

                await Task.Delay(delayMiliseconds, cancellationToken);
                var loadResult = await _loadProfiler.ExecuteQueryLoadAsync(query,
cancellationToken);
                loadProfilerResult.Add(loadResult);

                if (cancellationToken.IsCancellationRequested)
                {
                    return ProfilerExecuterHelpers.FillLoadExecutedResult(i,
loadProfilerResult);
                }
            }

            return ProfilerExecuterHelpers.FillLoadExecutedResult(iterationNumber,
loadProfilerResult);
        }
    }

public interface ISequentialProfilerExecuterWithDelay
    {
        Task<LoadExecutedResult> ExecuteLoadAsync(string query, int iterationNumber, int
delayMiliseconds,
            IProgress<int>? queryLoadProgress = null, CancellationToken cancellationToken
= default);
    }
```

4. SequentialProfilerExecutorWithTimeLimit

```
public class SequentialProfilerExecutorWithTimeLimit :
ISequentialProfilerExecutorWithTimeLimit
    {
        private readonly ILoadProfiler _loadProfiler;

        public SequentialProfilerExecutorWithTimeLimit(ILoadProfiler loadProfiler)
        {
            _loadProfiler = loadProfiler;
        }

        public async Task<LoadExecutedResult> ExecuteLoadAsync(string query, int
iterationNumber, int timeLimitMiliseconds,
            IProgress<int> queryLoadProgress = null, CancellationToken cancellationToken
= default)
        {
            var loadProfilerResult = new List<LoadProfilerResult>();

            for (int i = 1; i <= iterationNumber; i++)
            {
                queryLoadProgress?.Report((i * 100) / iterationNumber);
```

```
                var loadResult = await _loadProfiler.ExecuteQueryLoadAsync(query,
cancellationToken);

                if (cancellationToken.IsCancellationRequested)
                {
                    return ProfilerExecuterHelpers.FillLoadExecutedResult(i,
loadProfilerResult);
                }

                loadProfilerResult.Add(loadResult);

                var elapsedTime = loadProfilerResult.Sum(x => x.ElapsedTime) +
loadResult.ElapsedTime;
                if (elapsedTime >= timeLimitMiliseconds)
                {
                    return ProfilerExecuterHelpers.FillLoadExecutedResult(i,
loadProfilerResult);
                }
            }

            return ProfilerExecuterHelpers.FillLoadExecutedResult(iterationNumber,
loadProfilerResult);
        }
           }

public interface ISequentialProfilerExecutorWithTimeLimit
    {
        Task<LoadExecutedResult> ExecuteLoadAsync(string query, int iterationNumber, int
timeLimitMiliseconds,
            IProgress<int>? queryLoadProgress = null, CancellationToken cancellationToken
= default);
           }
```

Додаток Г

Реалізація класів для вирішення задач роботи з параметрами підключення

1. MsSqlConnectionService:

```
public class MsSqlConnectionService : IMsSqlConnectionService
    {
        public MsSqlConnectionSettings GetMsSqlConnectionSettings(string
connectionString)
        {
            var builder = new SqlConnectionStringBuilder(connectionString);
            return new MsSqlConnectionSettings
            {
                Server = builder.DataSource,
                IntegratedAuth = builder.IntegratedSecurity,
                ApplicationIntent = builder.ApplicationIntent,
                Login = builder.UserID,
                Password = builder.Password,
                Database = builder.InitialCatalog,
                ConnectTimeout = builder.ConnectTimeout,
                MaxPoolSize = builder.MaxPoolSize,
                EnablePooling = builder.Pooling
            };
        }

        public string GetConnectionString(MsSqlConnectionSettings settings)
        {
            var build = new SqlConnectionStringBuilder
            {
                DataSource = settings.Server,
                IntegratedSecurity = settings.IntegratedAuth,
                ApplicationName = "QueryPerformanceMaster",
                ApplicationIntent = settings.ApplicationIntent,
                TrustServerCertificate = true
            };
            if (!settings.IntegratedAuth && !settings.AzureMFA)
            {
                build.UserID = settings.Login;
                build.Password = settings.Password;
            }

            if (!string.IsNullOrWhiteSpace(settings.Database))
            {
                build.InitialCatalog = settings.Database;
            }

            if (settings.ConnectTimeout != 0)
            {
                build.ConnectTimeout = settings.ConnectTimeout;
            }

            if (settings.MaxPoolSize != 0)
            {
                build.MaxPoolSize = settings.MaxPoolSize;
                build.MinPoolSize = settings.MaxPoolSize;
            }

            build.Pooling = settings.EnablePooling;

            return build.ConnectionString;
```

```
        }
             }

public interface IMsSqlConnectionService
    {
        string GetConnectionString(MsSqlConnectionSettings settings);
        MsSqlConnectionSettings GetMsSqlConnectionSettings(string connectionString);
             }
```

2. PostgreSqlConnectionService:

```
public class PostgreSqlConnectionService : IPostgreSqlConnectionService
    {
        public PostreSqlConnectionSettings GetPostgreSqlConnectionSettings(string
connectionString)
        {
            var builder = new NpgsqlConnectionStringBuilder(connectionString);
            return new PostreSqlConnectionSettings
            {
                Server = builder.Host,
                Port = builder.Port,
                MaxPoolSize= builder.MaxPoolSize,
                ConnectTimeout = builder.Timeout,
                Database= builder.Database,
                EnablePooling = builder.Pooling,
                Login = builder.Username,
                Password= builder.Password
            };
        }

        public string GetConnectionString(PostreSqlConnectionSettings settings)
        {
            var build = new NpgsqlConnectionStringBuilder
            {
                Host = settings.Server,
                ApplicationName = "QueryPerformanceMaster",
                TrustServerCertificate = true,
                Username = settings.Login,
                Password = settings.Password
            };

            if (!string.IsNullOrWhiteSpace(settings.Database))
            {
                build.Database = settings.Database;
            }

            if (settings.ConnectTimeout != 0)
            {
                build.Timeout = settings.ConnectTimeout;
            }

            if (settings.MaxPoolSize != 0)
            {
                build.MaxPoolSize = settings.MaxPoolSize;
                build.MinPoolSize = settings.MaxPoolSize;
            }

            build.Pooling = settings.EnablePooling;

            return build.ConnectionString;
        }
    }
```

```
public interface IPostgreSqlConnectionService
    {
        string GetConnectionString(PostreSqlConnectionSettings settings);
        PostreSqlConnectionSettings GetPostgreSqlConnectionSettings(string
connectionString);
    }
```

3. ConnectionService:

```
public class ConnectionService : IConnectionService
    {
        private readonly IMsSqlConnectionService _msSqlConnectionService;
        private readonly IPostgreSqlConnectionService _postgreSqlConnectionService;

        public ConnectionService(IMsSqlConnectionService msSqlConnectionService,
IPostgreSqlConnectionService postgreSqlConnectionService)
        {
            _msSqlConnectionService = msSqlConnectionService;
            _postgreSqlConnectionService = postgreSqlConnectionService;
        }

        public string SetDatabaseToConnectionString(SqlProvider sqlProvider, string
connectionString, string database)
        {
            var resultConnectionString = string.Empty;

            if (sqlProvider == SqlProvider.SqlServer)
            {
                var settings =
_msSqlConnectionService.GetMsSqlConnectionSettings(connectionString);
                settings.Database = database;

                resultConnectionString =
_msSqlConnectionService.GetConnectionString(settings);
            }
            else if (sqlProvider == SqlProvider.PostgreSql)
            {
                var settings =
_postgreSqlConnectionService.GetPostgreSqlConnectionSettings(connectionString);
                settings.Database = database;

                resultConnectionString =
_postgreSqlConnectionService.GetConnectionString(settings);
            }

            return resultConnectionString;
        }

        public string SetPoolSizeToConnectionString(SqlProvider sqlProvider, string
connectionString, int poolSize)
        {
            var resultConnectionString = string.Empty;

            if (sqlProvider == SqlProvider.SqlServer)
            {
                var settings =
_msSqlConnectionService.GetMsSqlConnectionSettings(connectionString);
                settings.MaxPoolSize = poolSize;

                resultConnectionString =
_msSqlConnectionService.GetConnectionString(settings);
```

```
            }
            else if (sqlProvider == SqlProvider.PostgreSql)
            {
                var settings =
_postgreSqlConnectionService.GetPostgreSqlConnectionSettings(connectionString);
                settings.MaxPoolSize = poolSize;

                resultConnectionString =
_postgreSqlConnectionService.GetConnectionString(settings);
            }

            return resultConnectionString;
        }
    }


public interface IConnectionService
    {
        string SetDatabaseToConnectionString(SqlProvider sqlProvider, string
connectionString, string database);
        string SetPoolSizeToConnectionString(SqlProvider sqlProvider, string
connectionString, int poolSize);
    }
```

Додаток Д

Реалізація класів для вирішення задач з отримання переліку баз даних СУБД, очистки кешу та буферу

1. ISqlProviderService:

```
public interface ISqlProviderService
    {
        Task<DropBuffersAndCacheResult> DropBuffersAndCache();
        Task<GetProviderDatabasesResult> GetSqlProviderDatabasesAsync();
    }
```

2. MsSqlProviderService:

```
internal class MsSqlProviderService : ISqlProviderService
    {
        private readonly IMsSqlConnectionProviderFactory _connectionProviderFactory;
        private readonly string _connectionString;

        public MsSqlProviderService(IMsSqlConnectionProviderFactory
connectionProviderFactory, string connectionString)
        {
            _connectionProviderFactory = connectionProviderFactory;
            _connectionString = connectionString;
        }

        public async Task<DropBuffersAndCacheResult> DropBuffersAndCache()
        {
            const string query =
                @"CHECKPOINT
                DBCC DROPCLEANBUFFERS
                DBCC FREEPROCCACHE
                ";

            var connectionProvider =
_connectionProviderFactory.GetConnectionProvider(_connectionString);
            SqlConnection sqlConnection = null;

            try
            {
                sqlConnection = await connectionProvider.CreateConnection();
                using var cmd = sqlConnection.CreateCommand();
                cmd.CommandText = query;

                await cmd.ExecuteNonQueryAsync();
            }
            catch (Exception ex)
            {
                return new DropBuffersAndCacheResult(false)
                {
                    ErrorMessage = ex.Message
                };
            }
```

```
            finally
            {
                if (sqlConnection != null)
                {
                    await sqlConnection.CloseAsync();
                }
            }

            return new DropBuffersAndCacheResult();
        }

        public async Task<GetProviderDatabasesResult> GetSqlProviderDatabasesAsync()
        {
            const string query = "SELECT databases.name FROM sys.databases WHERE
databases.state = 0 ORDER BY databases.name";

            var connectionProvider =
_connectionProviderFactory.GetConnectionProvider(_connectionString);
            SqlConnection sqlConnection = null;
            var databases = new List<SqlProviderDatabase>();

            try
            {
                sqlConnection = await connectionProvider.CreateConnection();
                using var cmd = sqlConnection.CreateCommand();
                cmd.CommandText = query;

                var reader = await cmd.ExecuteReaderAsync();
                while (reader.Read())
                {
                    databases.Add(new SqlProviderDatabase { Name = (string)reader[0] });
                }
            }
            catch(Exception ex)
            {
                return new GetProviderDatabasesResult(false)
                {
                    ErrorMessage = ex.Message
                };
            }
            finally
            {
                if (sqlConnection != null)
                {
                    await sqlConnection.CloseAsync();
                }
            }

            return new GetProviderDatabasesResult { SqlProviderDatabases = databases };
        }
    }
```

3. PostgreSqlProviderService:

```
internal class PostgreSqlProviderService : ISqlProviderService
    {
        private readonly IPostgreSqlConnectionProviderFactory _connectionProviderFactory;
        private readonly string _connectionString;

        public PostgreSqlProviderService(IPostgreSqlConnectionProviderFactory
connectionProviderFactory, string connectionString)
```

```
        {
            _connectionProviderFactory = connectionProviderFactory;
            _connectionString = connectionString;
        }

        public async Task<DropBuffersAndCacheResult> DropBuffersAndCache()
        {
            const string query = "DISCARD ALL";

            var connectionProvider =
_connectionProviderFactory.GetConnectionProvider(_connectionString);
            NpgsqlConnection sqlConnection = null;

            try
            {
                sqlConnection = await connectionProvider.CreateConnection();
                using var cmd = sqlConnection.CreateCommand();
                cmd.CommandText = query;

                await cmd.ExecuteNonQueryAsync();
            }
            catch (Exception ex)
            {
                return new DropBuffersAndCacheResult(false)
                {
                    ErrorMessage = ex.Message
                };
            }
            finally
            {
                if (sqlConnection != null)
                {
                    await sqlConnection.CloseAsync();
                }
            }

            return new DropBuffersAndCacheResult();
        }

        public async Task<GetProviderDatabasesResult> GetSqlProviderDatabasesAsync()
        {
            const string query = "SELECT datname FROM pg_database WHERE datistemplate =
false ORDER BY datname";

            var connectionProvider =
_connectionProviderFactory.GetConnectionProvider(_connectionString);
            NpgsqlConnection sqlConnection = null;
            var databases = new List<SqlProviderDatabase>();

            try
            {
                sqlConnection = await connectionProvider.CreateConnection();
                using var cmd = sqlConnection.CreateCommand();
                cmd.CommandText = query;

                var reader = await cmd.ExecuteReaderAsync();
                while (reader.Read())
                {
                    databases.Add(new SqlProviderDatabase { Name = (string)reader[0] });
                }
            }
            catch (Exception ex)
            {
                return new GetProviderDatabasesResult(false)
                {
```

```
                    ErrorMessage = ex.Message
                };
            }
            finally
            {
                if (sqlConnection != null)
                {
                    await sqlConnection.CloseAsync();
                }
            }

            return new GetProviderDatabasesResult { SqlProviderDatabases = databases };
        }
    }
```

4. SqlProviderServiceFactory:

```
public class SqlProviderServiceFactory : ISqlProviderServiceFactory
    {
        private readonly IMsSqlConnectionProviderFactory _msSqlconnectionProviderFactory;
        private readonly IPostgreSqlConnectionProviderFactory
_postgreSqlconnectionProviderFactory;

        public SqlProviderServiceFactory(IMsSqlConnectionProviderFactory
connectionProviderFactory,
            IPostgreSqlConnectionProviderFactory postgreSqlconnectionProviderFactory)
        {
            _msSqlconnectionProviderFactory = connectionProviderFactory;
            _postgreSqlconnectionProviderFactory = postgreSqlconnectionProviderFactory;
        }

        public ISqlProviderService GetSqlProviderService(SqlConnectionParams
connectionParams)
        {
            switch (connectionParams.SqlProvider)
            {
                case SqlProvider.SqlServer:
                    return new MsSqlProviderService(_msSqlconnectionProviderFactory,
connectionParams.ConnectionString);
                case SqlProvider.PostgreSql:
                    return new
PostgreSqlProviderService(_postgreSqlconnectionProviderFactory,
connectionParams.ConnectionString);
            }

            throw new NotImplementedException();
        }
    }


public interface ISqlProviderServiceFactory
    {
        ISqlProviderService GetSqlProviderService(SqlConnectionParams connectionParams);
    }
```

Додаток Е

Реалізація модульних тестів для перевірки роботи сценаріїв навантаження

1. ParallelProfilerExecuterTests:

```
public class ParallelProfilerExecuterTests
    {
        private Mock<ILoadProfiler> _loadProfilerMock = new();
        private IParallelProfilerExecuter _parallelProfilerExecuter;

        public ParallelProfilerExecuterTests()
        {
            _parallelProfilerExecuter = new
ParallelProfilerExecuter(_loadProfilerMock.Object);
        }

        [Fact]
        public async Task ExecuteLoadAsync_ReturnsIterationCompleted()
        {
            var cmd = "cmd";
            int iterationNumber = 10;
            int threadNumber = 10;

            _loadProfilerMock
                .Setup(x => x.ExecuteQueryLoadAsync(cmd, CancellationToken.None))
                .ReturnsAsync(new LoadProfilerResult
                {
                    CpuTime = 1,
                    ElapsedTime = 1,
                    ExecTime = TimeSpan.FromMilliseconds(1),
                    LogicalReads = 2
                });

            var loadResult = await _parallelProfilerExecuter.ExecuteLoadAsync(cmd,
threadNumber, iterationNumber);

            Assert.Equal(iterationNumber * threadNumber, loadResult.IterationCompleted);
        }

        [Fact]
        public void ExecuteLoadAsync_ReturnsIterationCompleted_WithCancellation()
        {
            var cmd = "cmd";
            int iterationNumber = 10;
            int threadNumber = 10;
            var cts = new CancellationTokenSource();

            _loadProfilerMock
                .Setup(x => x.ExecuteQueryLoadAsync(cmd, cts.Token))
                .ReturnsAsync(new LoadProfilerResult
                {
                    CpuTime = 1,
                    ElapsedTime = 1,
                    ExecTime = TimeSpan.FromMilliseconds(1),
                    LogicalReads = 2
                });
            LoadExecutedResult? loadResult = null;
```

```
            var execTask = Task.Run(async () =>
            {
                await Task.Delay(100);
                loadResult = await _parallelProfilerExecuter.ExecuteLoadAsync(cmd,
threadNumber, iterationNumber, cancellationToken: cts.Token);
            });
            var cancelTask = Task.Run(() =>
            {
                cts.Cancel();
            });
            Task.WaitAll(execTask, cancelTask);

            Assert.True(loadResult?.IterationCompleted < iterationNumber * threadNumber);
        }
          }
```

2. SequentialProfilerExecuterTests:

```
public class SequentialProfilerExecuterTests
    {
        private Mock<ILoadProfiler> _loadProfilerMock = new();
        private ISequentialProfilerExecuter _sequentialProfilerExecuter;

        public SequentialProfilerExecuterTests()
        {
            _sequentialProfilerExecuter = new
SequentialProfilerExecuter(_loadProfilerMock.Object);
        }

        [Fact]
        public async Task ExecuteLoadAsync_ReturnsIterationCompleted()
        {
            var cmd = "cmd";
            int iterationNumber = 10;

            _loadProfilerMock
                .Setup(x => x.ExecuteQueryLoadAsync(cmd, CancellationToken.None))
                .ReturnsAsync(new LoadProfilerResult
                {
                    CpuTime = 1,
                    ElapsedTime = 1,
                    ExecTime = TimeSpan.FromMilliseconds(1),
                    LogicalReads = 2
                });

            var loadResult =  await _sequentialProfilerExecuter.ExecuteLoadAsync(cmd,
iterationNumber);

            Assert.Equal(iterationNumber, loadResult.IterationCompleted);
        }

        [Fact]
        public void ExecuteLoadAsync_ReturnsIterationCompleted_WithCancellation()
        {
            var cmd = "cmd";
            int iterationNumber = 1000;
            var cts = new CancellationTokenSource();

            _loadProfilerMock
                .Setup(x => x.ExecuteQueryLoadAsync(cmd, cts.Token))
                .ReturnsAsync(new LoadProfilerResult
                {
```

```
                    CpuTime = 1,
                    ElapsedTime = 1,
                    ExecTime = TimeSpan.FromMilliseconds(1),
                    LogicalReads = 2
                });
            LoadExecutedResult? loadResult = null;

            var execTask = Task.Run(async () =>
            {
                await Task.Delay(100);
                loadResult = await _sequentialProfilerExecuter.ExecuteLoadAsync(cmd,
iterationNumber, cancellationToken: cts.Token);
            });
            var cancelTask = Task.Run(() =>
            {
                cts.Cancel();
            });
            Task.WaitAll(execTask, cancelTask);

            Assert.True(loadResult?.IterationCompleted < iterationNumber);
        }
    }
```

3. SequentialProfilerExecuterWithDelayTests:

```
public class SequentialProfilerExecuterWithDelayTests
    {
        private Mock<ILoadProfiler> _loadProfilerMock = new();
        private ISequentialProfilerExecuterWithDelay _sequentialProfilerExecuter;

        public SequentialProfilerExecuterWithDelayTests()
        {
            _sequentialProfilerExecuter = new
SequentialProfilerExecuterWithDelay(_loadProfilerMock.Object);
        }

        [Fact]
        public async Task ExecuteLoadAsync_ReturnsIterationCompleted()
        {
            var cmd = "cmd";
            int iterationNumber = 10;
            int delayMiliseconds = 10;

            _loadProfilerMock
                .Setup(x => x.ExecuteQueryLoadAsync(cmd, CancellationToken.None))
                .ReturnsAsync(new LoadProfilerResult
                {
                    CpuTime = 1,
                    ElapsedTime = 1,
                    ExecTime = TimeSpan.FromMilliseconds(100),
                    LogicalReads = 2
                });

            var loadResult = await _sequentialProfilerExecuter.ExecuteLoadAsync(cmd,
iterationNumber, delayMiliseconds);

            Assert.True(loadResult?.IterationCompleted == iterationNumber);
        }

        [Fact]
        public void ExecuteLoadAsync_ReturnsIterationCompleted_WithCancellation()
```

```
        {
            var cmd = "cmd";
            int iterationNumber = 1000;
            int delayMiliseconds = 100;
            var cts = new CancellationTokenSource();

            _loadProfilerMock
                .Setup(x => x.ExecuteQueryLoadAsync(cmd, cts.Token))
                .ReturnsAsync(new LoadProfilerResult
                {
                    CpuTime = 1,
                    ElapsedTime = 1,
                    ExecTime = TimeSpan.FromMilliseconds(1),
                    LogicalReads = 2
                });
            LoadExecutedResult? loadResult = null;

            var execTask = Task.Run(async () =>
            {
                loadResult = await _sequentialProfilerExecuter.ExecuteLoadAsync(cmd,
iterationNumber, delayMiliseconds, cancellationToken: cts.Token);
            });
            var cancelTask = Task.Run(async () =>
            {
                await Task.Delay(100);
                cts.Cancel();
            });
            Task.WaitAll(execTask, cancelTask);

            Assert.True(loadResult?.IterationCompleted < iterationNumber);
        }
    }
```

4. SequentialProfilerExecutorWithTimeLimitTests:

```
public class SequentialProfilerExecutorWithTimeLimitTests
    {
        private Mock<ILoadProfiler> _loadProfilerMock = new();
        private ISequentialProfilerExecutorWithTimeLimit _sequentialProfilerExecuter;

        public SequentialProfilerExecutorWithTimeLimitTests()
        {
            _sequentialProfilerExecuter = new
SequentialProfilerExecutorWithTimeLimit(_loadProfilerMock.Object);
        }

        [Fact]
        public async Task ExecuteLoadAsync_ReturnsElapsedTimeLowerOrEqualThanLimit()
        {
            var cmd = "cmd";
            int iterationNumber = 200;
            int timeLimit = 100;

            _loadProfilerMock
                .Setup(x => x.ExecuteQueryLoadAsync(cmd, CancellationToken.None))
                .ReturnsAsync(new LoadProfilerResult
                {
                    CpuTime = 1,
                    ElapsedTime = 1,
                    ExecTime = TimeSpan.FromMilliseconds(1),
                    LogicalReads = 2
```

```
                });

            var loadResult = await _sequentialProfilerExecuter.ExecuteLoadAsync(cmd,
iterationNumber, timeLimit);

            Assert.True(loadResult.ElapsedTimeTotal <= timeLimit);
        }

        [Fact]
        public void
ExecuteLoadAsync_ReturnsElapsedTimeLowerOrEqualThanLimit_WithCancellation()
        {
            var cmd = "cmd";
            int iterationNumber = 100;
            int timeLimit = int.MaxValue;
            var cts = new CancellationTokenSource();

            _loadProfilerMock
                .Setup(x => x.ExecuteQueryLoadAsync(cmd, cts.Token))
                .ReturnsAsync(new LoadProfilerResult
                {
                    CpuTime = 1,
                    ElapsedTime = 1,
                    ExecTime = TimeSpan.FromMilliseconds(1),
                    LogicalReads = 2
                });
            LoadExecutedResult? loadResult = null;

            var execTask = Task.Run(async () =>
            {
                loadResult = await _sequentialProfilerExecuter.ExecuteLoadAsync(cmd,
iterationNumber, timeLimit, cancellationToken: cts.Token);
            });
            var cancelTask = Task.Run(() =>
            {
                cts.Cancel();
            });
            Task.WaitAll(execTask, cancelTask);

            Assert.True(loadResult?.IterationCompleted < iterationNumber);
        }
    }
```

Додаток Є

Презентація

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ТЕЛЕКОМУНІКАЦІЙ

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

РОЗРОБКА ПРОФАЙЛЕРУ ПРОДУКТИВНОСТІ SQL-ЗАПИТІВ МОВОЮ C#

Виконав студент 4 курсу

групи ПД-41
Бондаренко Володимир Валерійович
Керівник роботи

старший викладач кафедри Гаманюк Ігор Михайлович

Київ – 2023

МЕТА, ОБ’ЄКТ ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

- **Мета роботи**

  Покращення працездатності застосунку шляхом використання профайлеру продуктивності SQL-запитів.

- **Об’єкт дослідження**

  Процес тестування застосунку.

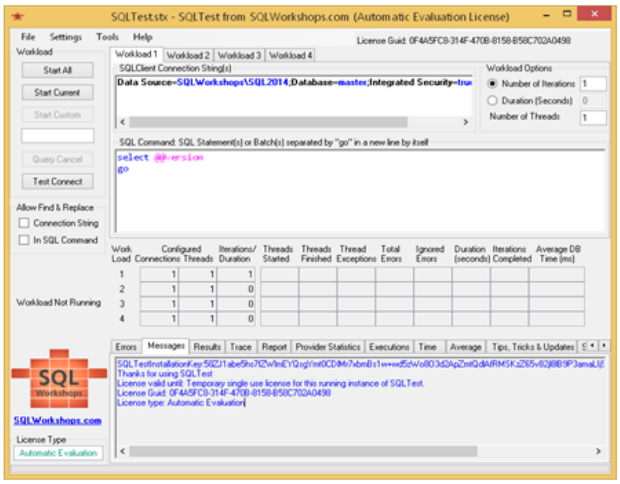
- **Предмет дослідження**

  Тестування запитів SQL.

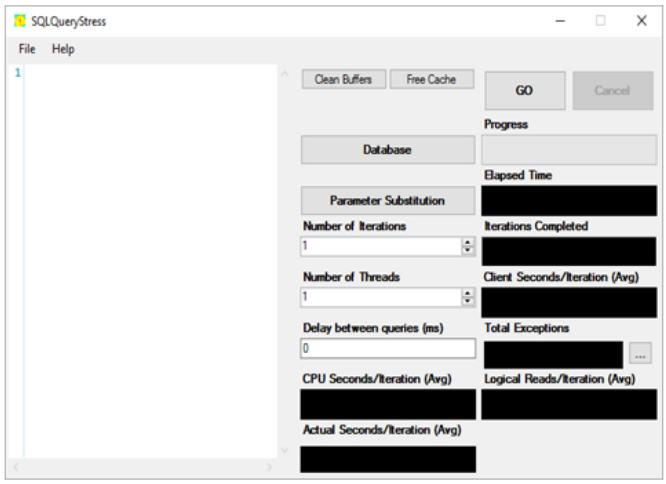
ЗАДАЧІ ДИПЛОМНОЇ РОБОТИ

1. Проведення огляду існуючих рішень, що надають можливості проведення навантажувального тестування. Описати переваги та недоліки різних програмних рішень та обґрунтувати актуальність створення нового програмного засобу для вирішення поставленої задачі.
2. Вибір інструментарію для розробки програми, зокрема мову програмування та додаткові інструменти, що сприятимуть процесу розробки. Обґрунтувати вибір засобів.
3. Визначення та реалізація функціональних вимог.
4. Визначення та реалізація нефункціональних вимог.
5. Проектування діяльності користувача.
6. Прийняття архітектурних рішень щодо структури проекту.
7. Визначення взаємодії класів.
8. Визначення станів застосунку.
9. Визначення інтерфейсу взаємодії користувача з застосунком.
10. Проектування розгортання застосунку.
11. Розробка програмного забезпечення
12. Виконання тестування застосунку: забезпечити модульне тестування всіх сценаріїв навантаження та провести мануальне тестування всього застосунку.

АНАЛІЗ АНАЛОГІВ



SQLQueryStress



SQLTest

АНАЛІЗ АНАЛОГІВ

	SQLQueryStress	SQLTest	QueryPerformanceMaster
Можливості додавання закладок з командами	Ні	Ні	Так
Можливість вибору сценаріїв навантаження	Ні	Ні	Так
Надання результату навантаження в графічному виді	Ні	Ні	Так
Редактор SQL команд	Так	Так	Так
Можливість вивільняти кеш/очищати буфер СУБД	Так	Ні	Так
Наявність CLI	Так	Ні	Ні
Можливість навантаження різних СУБД	Ні	Ні	Так

5

ВИМОГИ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Функціональні вимоги:

- Можливості додавання закладок з командами,
- Можливість вибору сценаріїв навантаження,
- Сценарії навантаження, які потрібно реалізувати в рамках даної роботи: Sequential (послідовний), Parallel (паралельний), SequentialWithDelay (послідовний, з заданим часом затримки між запитами), SequentialWithTimeLimit (послідовний, з заданим обмеженням в часі виконання),
- Наявність редактору SQL команд,
- Можливість вивільняти кеш/очищати буфер СУБД,
- Можливість навантаження різних СУБД,
- СУБД для яких потрібно реалізувати в рамках даної роботи: SqlServer, PostgreSQL,
- Можливість перервати виконання навантаження,
- Програма повинна збирати статистику по трьом значенням: Elapsed Time (час виконання команди), CPU Time (процесорний час), Logical Reads (кількість читань з диску), та надавати їх у вигляді наступних чисельних показників: Total (загальна сума), Average (середнє арифметичне), Median (медіана), Standard Deviation (середньоквадратичне відхилення),
- Надання результату навантаження як в чисельному, так і в графічному виді

Нефункціональні вимоги:

- Локалізація інтерфейсу: англійська,
- Операційна система: Windows,
- Мінімальна версія операційної системи: Windows 10 версії 1809 (10.0.17763.0),
- Застосунок повинен мати змогу пакуватись у формат MSIX для публікації в Microsoft Store

6

ПРОГРАМНІ ЗАСОБИ РЕАЛІЗАЦІЇ

Microsoft WPF

Система для побудови клієнтських застосунків WPF

Microsoft .NET

Платформа .NET, мова програмування C#

icsharpcode/ AvalonEdit

Бібліотека для створення текстових редакторів на основі WPF

MVVM CROSS

MVVM фреймворк

MSIX

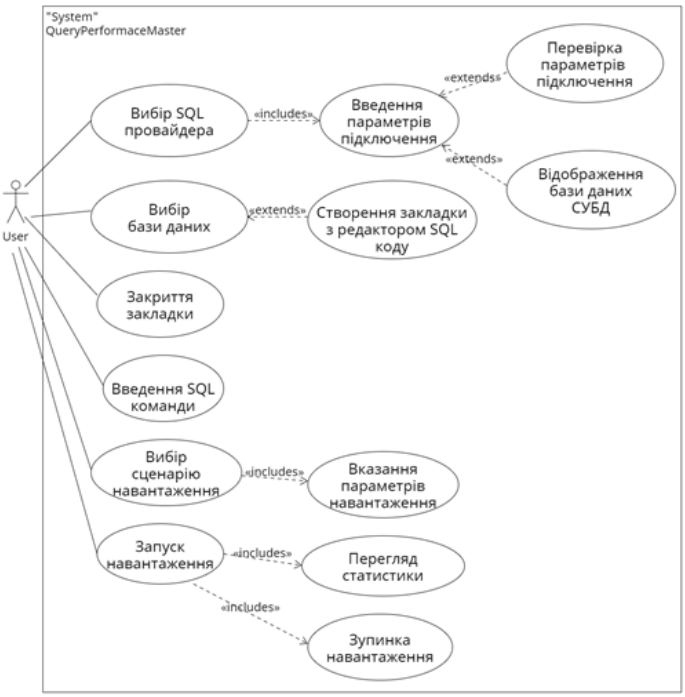
Формат пакування Windows застосунків

ScottPlot.NET

Бібліотека, яка дає можливість інтерактивного відображення графіків, діаграм.

7

ДІАГРАМА ПРЕЦЕДЕНТІВ

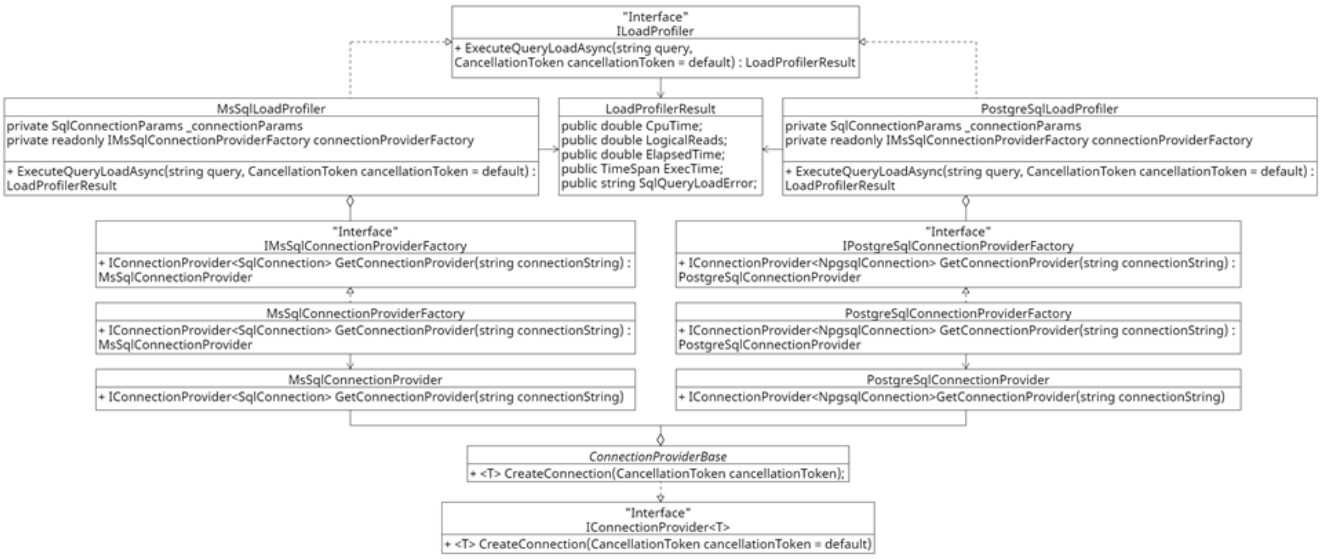


8

ДІАГРАМА ДІЯЛЬНОСТІ

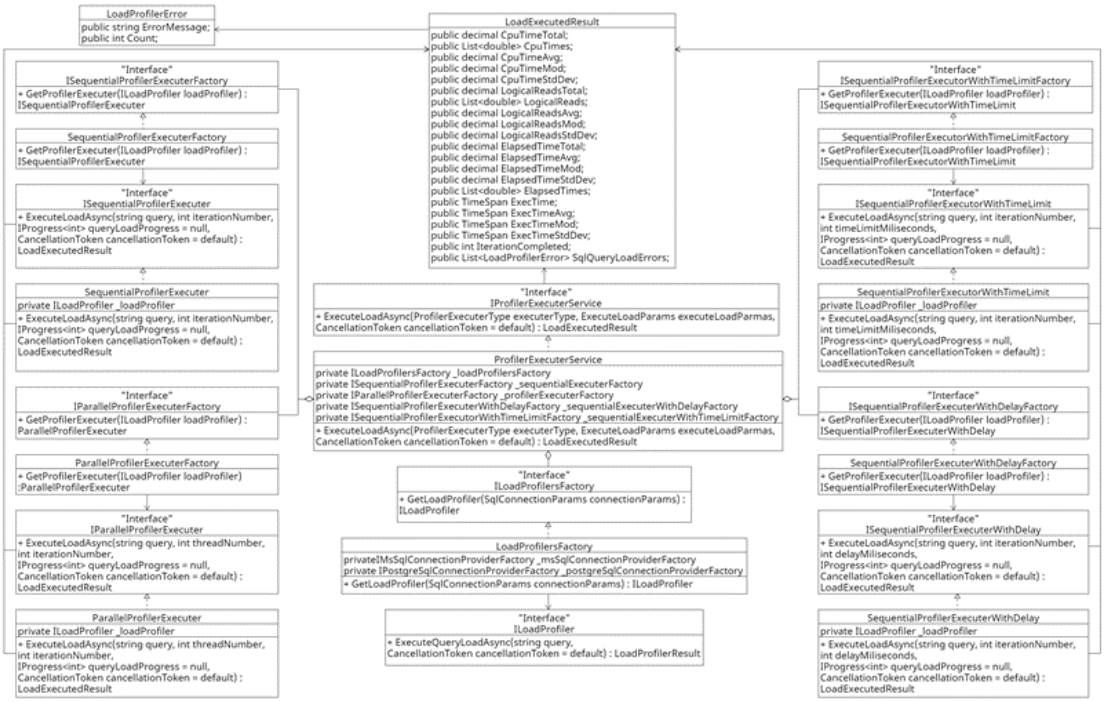
9

ДІАГРАМА КЛАСІВ



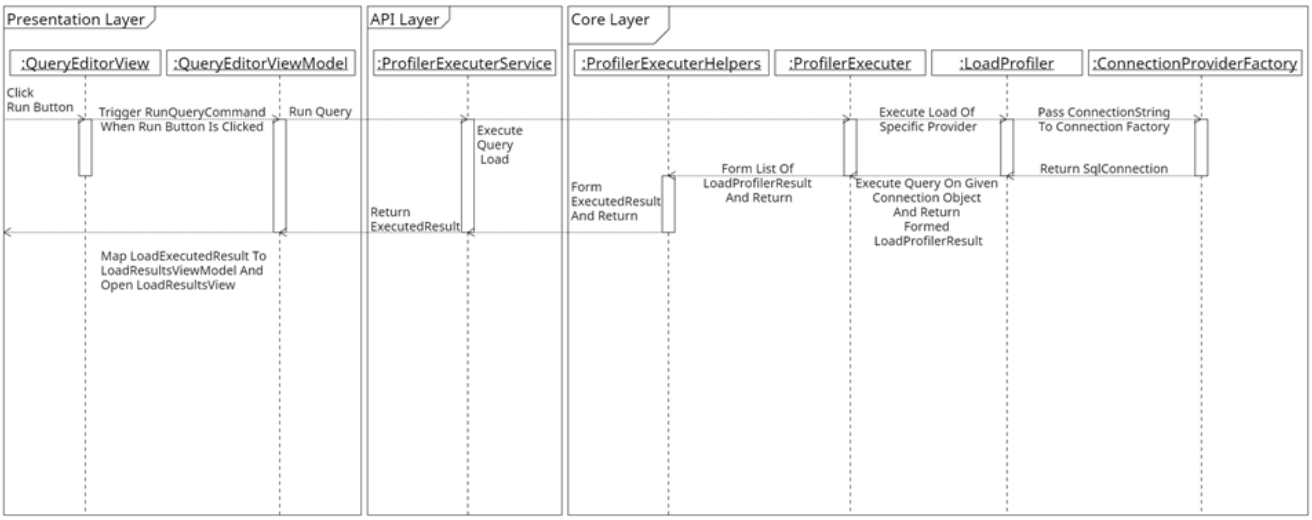
10

ДІАГРАМА КЛАСІВ



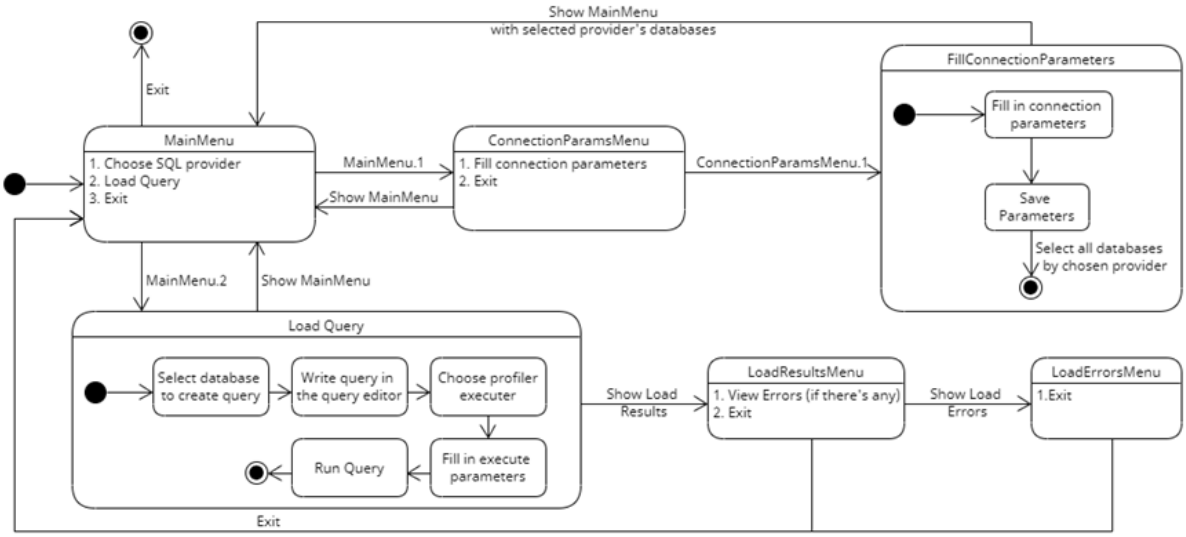
11

ДІАГРАМА ПОСЛІДОВНОСТІ



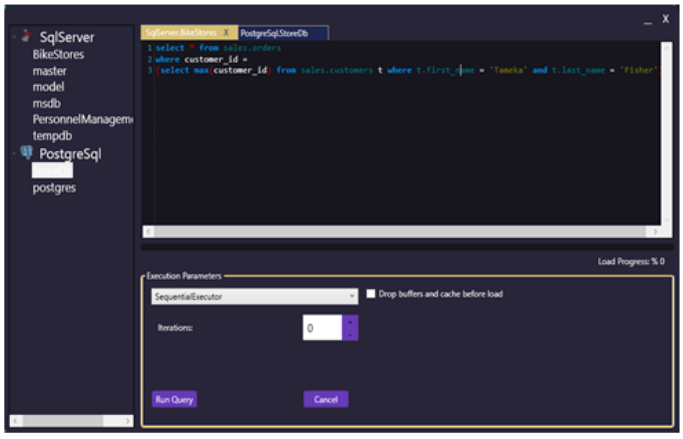
12

ДІАГРАМА СТАНІВ ДЛЯ МОДЕЛЮВАННЯ РОБОТИ МЕНЮ

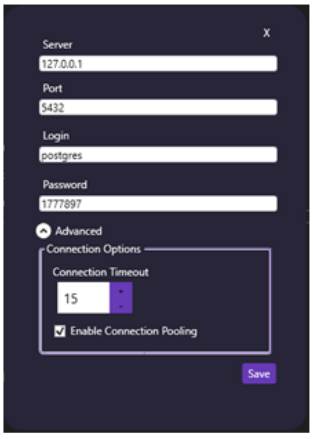


13

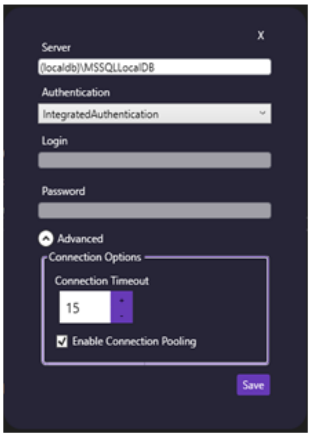
ЕКРАННІ ФОРМИ



Головний екран застосунку



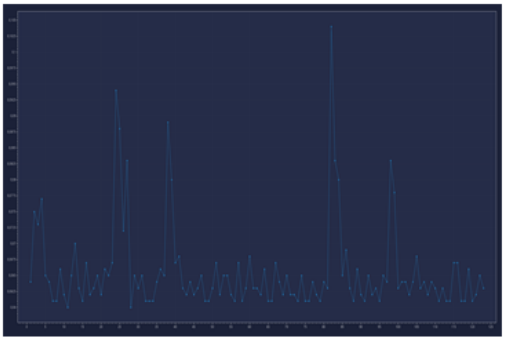
Екран параметрів підключення PostgreSql



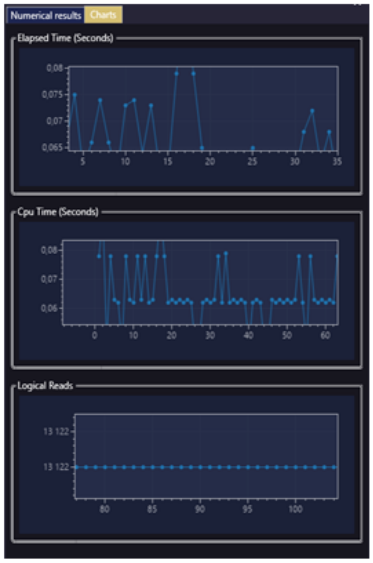
Екран параметрів підключення SqlServer

ЕКРАННІ ФОРМИ

Екран статистики навантаження у чисельному форматі



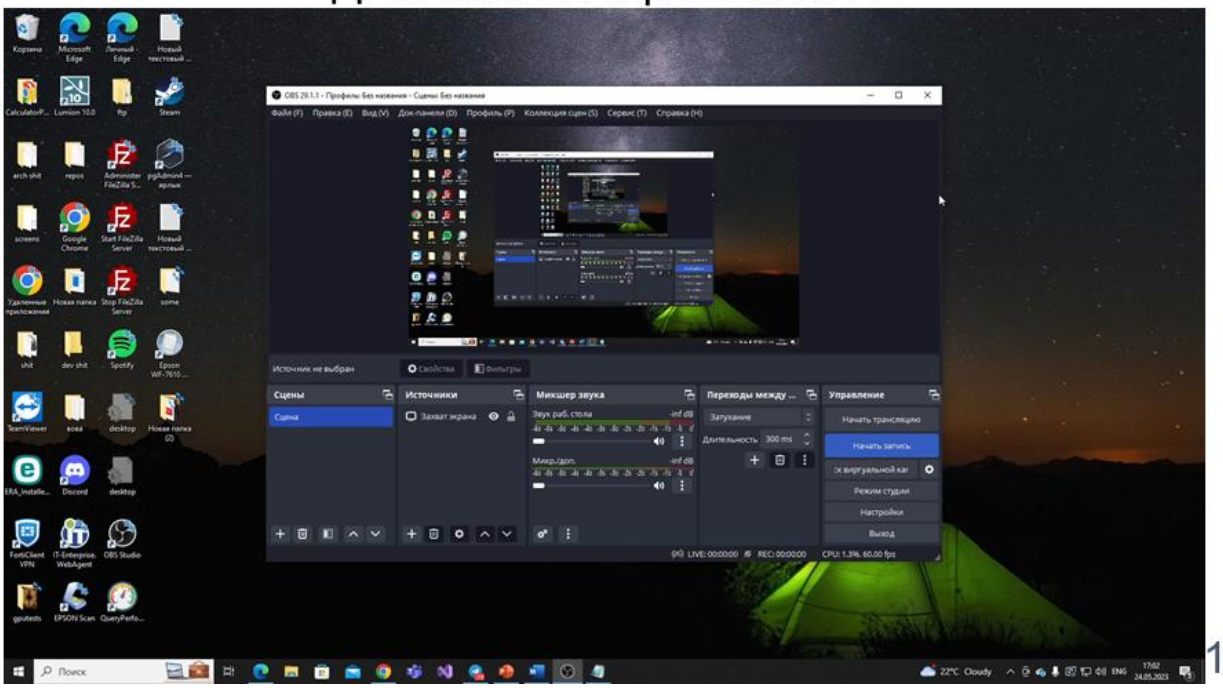
Графік по часу виконання команд



Екран статистики навантаження у графічному форматі

15

ДЕМОНСТРАЦІЯ РОБОТИ



16

АПРОБАЦІЯ РЕЗУЛЬТАТІВ ДОСЛІДЖЕННЯ

1. Бондаренко В.В. Розробка профайлеру продуктивності SQL-запитів за допомогою стрес-тестування / В.В Бондаренко, І.М Гаманюк // Застосування програмного забезпечення в інфокомунікаційних технологіях: Матеріали всеукраїнської науково-технічної конференції кафедри інженерії програмного забезпечення Державного університету телекомунікацій. Збірник тез. 20.04.2023, ДУТ, м. Київ — К.: ДУТ, 2023. — С. 98.
2. Бондаренко В.В. Використання бібліотеки SCOTPLOT.WPF для створення графіків при розробці профайлеру продуктивності SQL-запитів за допомогою стрес-тестування / В.В Бондаренко, І.М Гаманюк // Застосування програмного забезпечення в інфокомунікаційних технологіях: Матеріали всеукраїнської науково-технічної конференції кафедри інженерії програмного забезпечення Державного університету телекомунікацій. Збірник тез. 20.04.2023, ДУТ, м. Київ — К.: ДУТ, 2023. — С. 101.

ВИСНОВКИ

В ході виконання роботи було отримано наступні результати:

1. Проведено аналіз існуючих програмних рішень по проведенню навантажувального тестування.
2. Вибрано засоби розробки застосунку.
3. Визначено та реалізовано функціональні вимоги.
4. Визначено та реалізовано нефункціональні вимоги.
5. Спроектовано діяльність користувача.
6. Прийнято архітектурні рішення щодо структури проекту.
7. Визначено взаємодію класів.
8. Визначено стани застосунку.
9. Визначено інтерфейс взаємодії користувача з застосунком.
10. Спроектовано розгортання застосунку.
11. Розроблено програмне забезпечення.
12. Написано модульні тести для перевірки правильної роботи сценаріїв навантаження.
13. Проведено мануальне тестування застосунку.
14. Продемонстровано тестові приклади проведення навантажувального тестування за допомогою розробленого застосунку.
15. Результати тестування підтвердили працездатність застосунку та відповідність поставленим вимогам.