Вінницький національний технічний університет
(повне найменування вищого навчального закладу)

Факультет інформаційних технологій та комп'ютерної інженерії
(повне найменування інституту, назва факультету (відділення))

Кафедра програмної інженерії
(повна назва кафедри (предметної, циклової комісії))

МАГІСТЕРСЬКА КВАЛІФІКАЦІЙНА РОБОТА

на тему:

«Розробка методів і програмних засобів для підвищення продуктивності веб-сервісів під час критичних навантажень на систему»

Виконав: студент 2-го курсу, групи 1ПІ-20м спеціальності 121 – Інженерія програмного забезпечення
(шифр і назва напряму підготовки, спеціальності)

_____Свіжак В.В.
(прізвище та ініціали)

Керівник: к.т.н., доцент каф. ПЗ

_____Романюк О.В.
(прізвище та ініціали)

«____» _____ 2021 р.

Опонент: д.т.н., професор каф. КН

_____Іванчук Я.В.
(прізвище та ініціали)

«____» _____ 2021 р.

Допущено до захисту

Завідувач кафедри ПЗ

д.т.н., проф. Романюк О.Н.
(прізвище та ініціали)

«____» _____ 2021 р.

Вінниця ВНТУ - 2021 рік

Вінницький національний технічний університет
Факультет інформаційних технологій та комп'ютерної інженерії
Кафедра програмного забезпечення
Рівень вищої освіти ІІ-й (магістерський)
Галузь знань 12 – Інформаційні технології
Спеціальність 121 – Інженерія програмного забезпечення
Освітньо-професійна програма – Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ
Завідувач кафедри ПЗ
Романюк О. Н.
« 13 » вересня 2021 р.

З А В Д А Н Н Я
НА МАГІСТЕРСЬКУ КВАЛІФІКАЦІЙНУ РОБОТУ
СТУДЕНТУ

Свіжак Віктору Вячеславовичу

1. Тема роботи – розробка методів і програмних засобів для підвищення продуктивності веб-сервісів під час критичних навантажень на систему.

Керівник роботи: Романюк Оксана Володимирівна, к.т.н., доцент кафедри ПЗ, затверджені наказом вищого навчального закладу від « 24 » вересня 2021 р. № 277.

2. Строк подання студентом роботи
1 грудня 2021 р.

3. Вихідні дані до роботи: базові методи підвищення продуктивності веб-сервісів – методи автоматизованого аналізу результатів стрес-тестування «Abstract Genetic Algorithm»; система керування базами даних – Microsoft SQL Server; мова запитів – T-SQL; вхідні дані для оптимізації – SQL-запит; режими роботи – з підключенням до БД та без підключення; спосіб підключення до БД – рядок підключень ADO.NET; вихідні дані – аналіз результату стрес-тестування.

4. Зміст розрахунково-пояснювальної записки: вступ; аналіз стану питання та постановка задач дослідження, розробка структури системи та методів підвищення продуктивності веб-сервісів, розробка програмних засобів, тестування системи, економічна частина, висновки, список використаних джерел, додатки.

5. Перелік графічного матеріалу: мета, об’єкт та предмет дослідження; завдання дослідження; наукова новизна одержаних результатів; порівняння з аналогами; метод агрегації результатів стрес-тестування; методу аналізу результатів стрес-тестування; схема алгоритму алгоритму аналізу результатів стрес-тестування; схема алгоритму пошуку аномалій в запиті та їх вирішення; архітектура системи; тестування системи; економічна частина; апробація та публікації; висновки.

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
1-4	Романюк О.В. к.т.н, доцент кафедри ПЗ		
5	Ратушняк О.Г., к.т.н, доцент кафедри ЕПВМ		

7. Дата видачі завдання ___14 вересня 2021 р._____

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів магістерської кваліфікаційної Роботи	Строк виконання етапів роботи	Примітка
1	Аналіз стану питання та постановка задач дослідження	15.09. 2021 – 29.09.2021	Вик.
2	Розробка структури системи та методів підвищення продуктивності веб-сервісів	30.09.2021 – 14.10.2021	Вик.
3	Розробка програмних засобів	15.10.2021 – 29.10.2021	Вик.
4	Тестування системи	30.10.2021 – 13.11.2021	Вик.
5.	Економічна частина	14.11.2021 – 30.11.2021	Вик.

Студент _____ **Свіжак В. В.**

(підпис) (прізвище та ініціали)

Керівник магістерської кваліфікаційної роботи _____ **Романюк О. В.**

(підпис) (прізвище та ініціали)

АННОТАЦІЯ

УДК 004.42

Свіжак В. В. Розробка методів і програмних засобів для підвищення продуктивності веб-сервісів під час критичних навантажень на систему. Магістерська кваліфікаційна робота зі спеціальності 121 - інженерія програмного забезпечення, освітня програма – інженерія програмного забезпечення. Вінниця: ВНТУ, 2021, 132с.

На укр. мові. Бібліогр.: 32 назв; рис.: 42; табл.: 11.

У магістерській кваліфікаційній роботі розроблено методи для автоматизованого аналізу та агрегації результатів стрес-тестування, а також програмний засіб, побудований на мікросервісній архітектурі, для проведення тестування та аналізу отриманих результатів, що дозволило підвищити продуктивність веб-сервісів в умовах високого навантаження на систему.

Розглянуто особливості факторів, що знижують продуктивність веб-сервісів, а також обґрунтована доцільність розробки. Виконана розробка методів для агрегації та аналізу результатів стрес-тестування, а також методу пошуку аномалій в SQL запиті. Розроблено програмний додаток для проведення тестування, а також проведення автоматизованого аналізу результатів на основі розроблених методів.

Система розроблена на мові програмування C#. Серверна частина розроблена за допомогою фреймворку ASP.NET Core. В якості СУБД використовується MS SQL.

Ключові слова: продуктивність, оптимізація, веб-сервіс, стрес-тестування, SQL запит.

ABSTRACT

UDC 004.42

Svizhak V. Development of methods and software to increase the productivity of web services during critical loads on the system. Master's thesis in specialty 121 - software engineering, educational program - software engineering. Vinnytsia: VNTU, 2021, 132p.

In Ukrainian language. Bibliography: 32 titles; pictures: 42; tables: 11.

In the master's qualification work developed methods for automated analysis and aggregation of stress test results, as well as software built on microservice architecture for testing and analysis of the results, which increased the productivity of web services under high load on the system.

The peculiarities of the factors that reduce the productivity of web services are considered, as well as the expediency of development is substantiated. Methods for aggregation and analysis of stress test results, as well as a method for finding anomalies in SQL query have been developed. A software application has been developed for testing, as well as automated analysis of results based on the developed methods.

The system is developed in the C # programming language. The server part is developed using the ASP.NET Core framework. MS SQL is used as a database.

Keywords: performance, optimization, web service, stress testing, SQL query.

ЗМІСТ

ВСТУП

Обґрунтування вибору теми дослідження.

Ефективність роботи веб-сервісів напряму впливає на рівень задоволеності клієнтів. Так, до формування негативного досвіду використання веб-сервісів і, як наслідок, відтоку клієнтів можуть призводити такі фактори як: складний інтерфейс; помилки, що виникають під час роботи веб-сервісу; обмежена функціональність; тривала обробка інформації, що вводиться; довге завантаження даних.

Кожен з цих факторів здатний привести до відмови від використовуваного програмного продукту на користь конкурентів. Згідно зі статистикою, 23% мобільних додатків видаляються після отримання першого досвіду взаємодії (user experience) [1].

Більшість недоліків веб-сервісів можна виявити на стадії тестування додатку. Але на цій стадії дуже важко оцінити продуктивність веб-сервісу. В загальному оцінюється швидкість відгуку сервісу під час конкретних запитів, але у виробничому середовищі навантаження на сервери значно відрізняються від тестового середовища. У таких випадках часто виникає ефект «вузького місця».

«Вузьке місце» виникає, коли потужність програми або комп'ютерної системи обмежена одним компонентом, наприклад, горловина пляшки, що уповільнює загальний потік води. «Вузьке місце» має найнижчу пропускну здатність з усіх частин шляху транзакції.

Тому, розробники систем намагаються уникати «вузьких місць» та спрямовувати зусилля на пошук та видалення існуючих.

Серед методів реалізації подібних задач найчастіше використовують різні види стрес-тестування, для перевірки як веде себе система при критичних навантаженнях [2]. Однак потрібно провести аналіз отриманих результатів.

Відомий підхід до реалізації подібних задач – це використання методу «Abstract Genetic Algorithm», який запропонував Чжень Мін (Джек) Цзян [3]. Але даний підхід не враховує можливих обмежень у апаратній частині сервера, саме

тому даний підхід потрібно модифікувати для застосування на практиці в програмній реалізації.

Таким чином задача розробки методів і програмних засобів для підвищення продуктивності веб-сервісів під час критичних навантажень на систему є досить актуальною.

Зв'язок роботи з науковими програмами, планами, темами. Робота виконувалась відповідно до плану науково-дослідних робіт кафедри програмного забезпечення.

Мета та завдання дослідження. Метою роботи є підвищення продуктивності веб-сервісів під час високих навантажень на систему.

Основними задачами дослідження є:

– провести аналіз існуючих методів діагностування продуктивності веб-сервісів;

– розробити метод аналізу результатів стрес-тестування;

– розробити метод агрегації результатів стрес-тестування;

– розробити сервіс для створення користувацьких сценаріїв тестування;

– розробити сервіс автоматичного пошуку вузьких місць у апаратній частині;

– розробити клієнтську частину для проведення тестування веб-сервісів;

– провести тестування системи.

В результаті виконання перелічених завдань буде отримано систему, яка в повному обсязі зможе допомогти виявити проблеми, що впливають на продуктивність веб-сервісів під час критичних навантажень.

Об'єкт дослідження – процес підвищення продуктивності веб-сервісів.

Предмет дослідження – методи та засоби підвищення продуктивності веб-сервісів під час критичних навантажень.

Методи дослідження. У процесі досліджень використовувались: теорія чисел та чисельних методів для розробки моделей проведення розрахунку результатів тестування, теорія алгоритмів для розробки алгоритмів і програмних

модулів, теорія баз даних для розробки структури бази даних, комп'ютерне моделювання для аналізу та перевірки отриманих теоретичних положень.

Наукова новизна одержаних результатів.

1. Подальшого розвитку отримав метод автоматизованого аналізу результатів стрес-тестування «Abstract Genetic Algorithm», в якому, на відміну від класичного методу, використано додаткову оцінювальну функцію автоматичного пошуку вузьких місць у апаратній частині сервера, що дало можливість збільшити точність пошуку сценаріїв з низькою продуктивністю.

2. Подальшого розвитку отримав метод агрегації результатів стрес-тестування, який, на відміну від існуючих, зберігає окремо результати стрес тестування та їх аналізу, що дало можливість підвищити ефективність проведення аналізу результатів стрес-тестування.

3. Удосконалено метод оптимізації SQL-запитів у середовищі MS SQL, який, на відміну від відомих алгоритмів, передбачає зменшення кількості операцій читання, що дозволило зменшити час виконання запитів в середньому на 20%.

Практична цінність отриманих результатів. Практична цінність одержаних результатів полягає в тому, що на основі отриманих в магістерській кваліфікаційній роботі теоретичних положень запропоновано алгоритми та програмні засоби для підвищення продуктивності роботи веб-сервісів в умовах критичного навантаження.

Особистий внесок здобувача. Усі наукові результати, викладені у магістерській кваліфікаційній роботі, отримані автором особисто. У роботах, опублікованих у співавторстві, здобувачу належить дослідження шляхів підвищення продуктивності роботи веб-сервісів під час критичних навантажень [4], програмний додаток для аналізу продуктивності веб-сервесів в умовах критичних навантажень [5], дослідження підвищення продуктивності виконання SQL-запитів [6], програмний додаток для дослідження продуктивності SQL-запитів [7].

Апробація матеріалів магістерської кваліфікаційної роботи. Результати роботи доповідалися на науково-технічній конференції: міжнародна

інтернет-конференція «Перспективні напрямки наукових досліджень XXI століття» – Харків 2021, Всеукраїнська науково-практична Інтернет-конференція «Електронні інформаційні ресурси: створення, використання, доступ» – Вінниця 2021, «Молодь в науці: дослідження, проблеми, перспективи» (Вінниця, 2020), XLIX Науково-технічна конференція факультету інформаційних технологій та комп'ютерної інженерії (Вінниця, 2020).

Публікації. Основні результати досліджень опубліковано в 4 наукових працях, у матеріалах конференцій.

Структура та обсяг роботи. Магістерська кваліфікаційна робота складається зі вступу, п’яти розділів, висновків, списку літератури, що містить 32 найменувань, 4 додатки. Робота містить 42 ілюстрацій, 11 таблиць. У першому розділі виконано аналіз стану питання та постановку задач досліджень. У другому розділі було розроблено структуру системи, методи та алоритми підвищення продуктивності веб-сервісів. У третьому розділі було здійснено розробку програмних засобів та компонентів системи. У четвертому розділі було проведено тестування роботи системи. У п’ятому розділі проведено економічне обґрунтування доцільності та економічної вигоди даного проекту. У додатках міститься технічне завдання на роботу, лістинг коду, літистинг коду модульних тестів та ілюстративний матеріал до захисту роботи.

1 АНАЛІЗ СТАНУ ПИТАННЯ ТА ПОСТАНОВКА ЗАДАЧ ДОСЛІДЖЕННЯ

1.1 Аналіз стану питання

Більшість недоліків веб-сервісів можна виявити на стадії тестування додатку. Але на цій стадії дуже важко оцінити продуктивність веб-сервісу. В загальному оцінюється швидкість відгуку сервісу під час конкретних запитів, але у виробничому середовищі навантаження на сервери значно відрізняються від тестового середовища. У таких випадках часто виникає ефект «вузького місця».

«Вузьке місце» виникає, коли потужність програми або комп'ютерної системи обмежена одним компонентом, наприклад, горловина пляшки, що уповільнює загальний потік води. «Вузьке місце» має найнижчу пропускну здатність з усіх частин шляху транзакції.

Тому, розробники систем намагаються уникати «вузьких місць» та спрямовувати зусилля на пошук та видалення існуючих.

Найпоширенішими «вузькими місцями» є наступні [8]:

1. Обмеження процесора. Згідно Microsoft, «вузькі місця процесора виникають, коли процесор настільки зайнятий, що не може відповідати на запити вчасно» [9].

2. Обмеження пам'яті. Це означає, що система не має достатньої кількості або достатньо швидкої оперативної пам'яті. У випадках, коли системі не вистачає пам'яті, комп'ютер почне вивантажувати пам'ять на значно повільніший жорсткий диск або твердотілий накопичувач, щоб продовжувати працювати. Крім того, якщо оперативна пам'ять не може достатньо швидко подавати дані до процесора, пристрій буде відчувати як уповільнення, так і низькі показники використання процесора.

3. Обмеження мережі. «Вузькі місця» в мережі виникають, коли для зв'язку між двома пристроями не вистачає необхідної пропускної здатності або обчислювальної потужності для швидкого виконання завдання.

4. Обмеження диска. Найповільніший компонент всередині комп’ютера або сервера – це, як правило, довгострокове сховище, і часто є неминучим «вузьким місцем» комп’ютера. До цього пункту слід віднести не лише технічні характеристики сервера, на якому розташована база даних, а й запити, що безпосередньо виконуються на цьому сервері. Тривале виконання запитів може створити досить серйозну проблему, особливо коли доступ до бази відбувається асинхронно. В такому випадку один неоптимізований запит може сповільнити роботу всієї системи через довготривале блокування таблиць, які використовуються у оптимізованих запитах [10].

5. Обмеження програмного забезпечення. Іноді падіння продуктивності, пов'язане з «вузькими місцями», походить від самого програмного забезпечення. Найбільше проблем виникає під час виконання асинхронного коду. Навіть, якщо виключити можливість взаємного блокування, може виникнути ситуація, коли багато запитів очікують розблокування певного компоненту, для подальшого виконання програми.

Для пошуку «вузьких місць» у веб-сервісах аналізують такі показники:

– одночасно працюючі користувачів – скільки користувачів одночасно може використовувати продукт без наслідків для його продуктивності;

– запити в секунду – кількість запитів, яке успішно обробляється за 1 секунду;

– транзакції в секунду – кількість успішно завершених за 1 секунду транзакцій, послідовних операцій, які згруповані в логічний ланцюжок.

Для того, щоб оцінити продуктивність роботи веб-сервісу в умовах тестового середовища проводять стрес тестування.

Стрес тестування – це тип тестування програмного забезпечення, яке перевіряє стабільність та надійність програмного забезпечення. Метою тестування на стрес є вимірювання програмного забезпечення щодо його надійності та можливостей обробки помилок в умовах надзвичайно важкого навантаження, а також забезпечення того, щоб програмне забезпечення не виходило з ладу під час кризових ситуацій. Він навіть тестує за межами

нормальних робочих точок і оцінює, як програмне забезпечення працює в екстремальних умовах [11].

Протягом дня, сервіси, зазвичай, навантажені по різному, тому потрібно спершу оцінити проміжок часу, під час якого на сервіс надходить найбільше трафіку. Це допоможе з'ясувати приблизні реальні показники по кількості запитів на сервіс у періоди високого навантаження. Після чого проводять стрес тестування, вимірюючи час затрачений на виконання одного запиту. Якщо, за результатами проведеного стрес тестування, було виявлено аномальну поведінку, під час високих навантажень – потрібно знайти яка саме компонента спричиняє обмеження.

Під час стрес тестування веб-сервісів, проводиться симуляція HTTP-запитів на сервіс. Для оцінки можливих обмежень в мережі потрібно щоб ініціатор запитів та сервер на якому розміщений веб-сервіс не знаходились у одній локальних мережі.

Для оцінки можливих обмежень зі сторони процесора та пам'яті можна використовувати вбудований засіб «Монітор ресурсів». Також за допомогою різних моніторингових засобів, наприклад Prometheus, можна зберігати статистику в окремій базі даних. Що надає можливість порівнювати навантаження на сервер під час різних запусків стрес тестів.

Якщо проблема не була виявлена у апаратній частині – отже слід її шукати у програмній частині. За допомогою профайлерів, наприклад YourKit, можна отримати інформацію про час виконання кожного метода у коді.

Видалення проблемних «вузьких місць» відбувається в залежності від знайденої причини. Для проблем апаратної частини слід розглянути можливість додання нового сервера або покращення існуючого. При проблемах зі сторони бази даних слід провести оптимізацію запитів та аналіз індексів у ключових таблицях. Створювати некластерний індекс потрібно лише за потреби по полю, що забезпечить мінімальну щільність розподілу. В іншому випадку, після кожної операції що змінює дані в таблиці буде відбуватись перебудова всіх індексів

пов’язаних з цією таблицею, що призведе до значної затримки під час виконання даних операцій [12].

Декларативне програмування часто називають описовим. В основі цієї парадигми лежить чітке вираження очікуваного результату, а не чіткий алгоритм для отримання цього результату. Формування алгоритму виконання відбувається «під капотом» виконуючої системи.

Існує декілька видів СКБД, які вирізняються моделями зберігання та доступу до інформації. Відомі ієрархічні, мережні та реляційні моделі даних [13]. Більшість сучасних СКБД використовують реляційну модель, оскільки складаються з набору зв'язаних між собою об'єктів-таблиць.

Для доступу до даних та маніпуляцій з ними в реляційних СКБД використовується декларативна мова написання запитів SQL. З першого погляду може здатись, що відповідність за оптимізацію такого запиту цілком лежить на механізмах СКБД, але на користувачу лежить немала відповідальність за швидкодію запита. Дійсно, у кожної СКБД є власні механізми для оптимізації запитів, але, дуже часто, цього виявляється замало і, в результаті, сервер виконує зайві операції читання.

Для оцінки швидкодії SQL-запита використовують кількість операцій читання. До таких операцій відносять не лише операції отримання даних з таблиць, а й всі операції порівняння даних. Зазвичай час затрачений на операцію залежить від швидкодії та завантаженості сервера, на якому знаходиться база даних. Тому не можна точно виміряти скільки часу буде витрачено на одну таку операцію, однак це дозволяє порівнювати оптимальність запитів виконаних на різних серверах.

У Microsoft SQL Server, для виконання SQL-запитів, використовуються логічні та фізичні оператори. Логічні оператори описують операції реляційної алгебри, що використовуються для обробки інструкцій. Фізичні оператори реалізують самі дії, що описані логічними операторами.

Для того, щоб отримати інформацію про порядок виклику фізичних операторів під час виконання запиту, що були описані логічними операторами, у середовищі MS SQL, використовується інструмент Query Plan.

У результаті проведеного аналізу [14] було виявлено, що найбільш типовими помилками є зайва конвертація даних, використання констант у арифметичних операціях з індексами, некоректне використання функцій в умовах, використання порядкового виконання запиту.

Для прикладу, можна розглянути ситуацію, коли у таблиці Purchasing існує індекс по полю PurchaseOrder, однак при умові «WHERE Purchasing. PurchaseOrder * 2 = 3400» цей індекс не буде використаний, що призводить до сканування всієї таблиці. Дану умову можна просто замінити на подібну «WHERE Purchasing.PurchaseOrder = 3400 / 2» – індекси будуть коректно використовуватись, що дає зменшення операцій читання у декілька разів.

Іншою досить популярною проблемою є використання оператора IN разом із підзапитами [15]. Багато користувачів вважають його дуже зручним для використання не лише зі сталим переліком, а й з вкладеними запитами. Суть проблеми полягає в тому, що такий вкладений підзапит буде виконуватись окремо для кожного запису. Так в умові «WHERE CustomerId NOT IN (SELECT …)», отримуємо не 1 очікуване виконання під запиту, а для кожного запису, що сильно знижує швидкість виконання. Замість некоректного використання оператора IN слід використовувати оператори OUTHER JOIN. Таким чином, змінивши умову на «LEFT JOIN (SELECT …) sub ON main.CustomerId = sub.CustomerId WHERE main.CustomerId IS NULL», отримаємо кількість операцій, що буде рости лінійно залежно від кількості даних, а не експоненціально, як у випадку з використанням оператора IN.

Ще одним цікавим випадком є порядок переліку таблиць в з’єднаннях оператором JOIN. Якщо таблиці сильно відрізняються за кількістю записів, кращою практикою буде розміщення таблиць від менших до більших [16]. Під час виконання для кожного запису з лівої таблиці знаходиться відповідний запис з правої таблиці. Тому, якщо при пошуку ідентифікувати відповідність для

кожного запису з меншої таблиці – це займе набагато менше часу, ніж перевіряти кожен запис з більшої таблиці, навіть у випадку, якщо всі записи більшої таблиці мають зв’язок з записом меншої таблиці.

1.2 Порівняльний аналіз аналогів

Тестування продуктивності – це нефункціональний тип тестування, який проводиться для визначення того, як програма працюватиме під навантаженням. Випробування проводяться за такими показниками, як швидкість, стабільність та масштабованість. Правильні служби тестування продуктивності можуть проводити широкий спектр тестів [17].

Різні види перевірок продуктивності, які проводяться службами тестування продуктивності, включають:

– Тестування навантаження: Це перевіряє робоче навантаження, яке виробляється в нормальних умовах, і вимірює час відгуку програми. Це також допомагає виявити будь – які вузькі місця.

– Стрес-тестування: екстремальне навантаження застосовується, щоб з'ясувати точку завантаження, в якій програма вийде з ладу.

– Спайк-тестування: це передбачає швидку зміну навантаження та моніторинг реакції програми.

– Тестування на витривалість: Це передбачає піддавання програми очікуваному навантаженню протягом тривалого періоду часу, щоб перевірити, наскільки добре вона може з цим впоратися.

– Тестування масштабованості: Цей тест передбачає поступове збільшення робочого навантаження, щоб з'ясувати, наскільки ефективно програмне забезпечення може масштабуватися.

– Об'ємне тестування: Об'ємне тестування передбачає послуги тестування продуктивності, що поповнюють базу програмного забезпечення великими обсягами даних, щоб перевірити її можливості та ефективність обробки даних.

Найпопулярнішими інструментами тестування продуктивності, які широко використовуються в 2021 році є такі:

WebLOAD – це інструмент тестування навантаження у масштабах підприємства, який може створювати реальні та надійні сценарії завантаження навіть у більшості складних систем. Його розумна аналітика дає поглиблену інформацію про продуктивність. Інструмент поставляється з вбудованою підтримкою сотень технологій. Він також інтегрується з кількома інструментами для підтримки більш простого моніторингу. На рисунку 1.1 наведено приклад інтерфейсу головного вікна додатку.

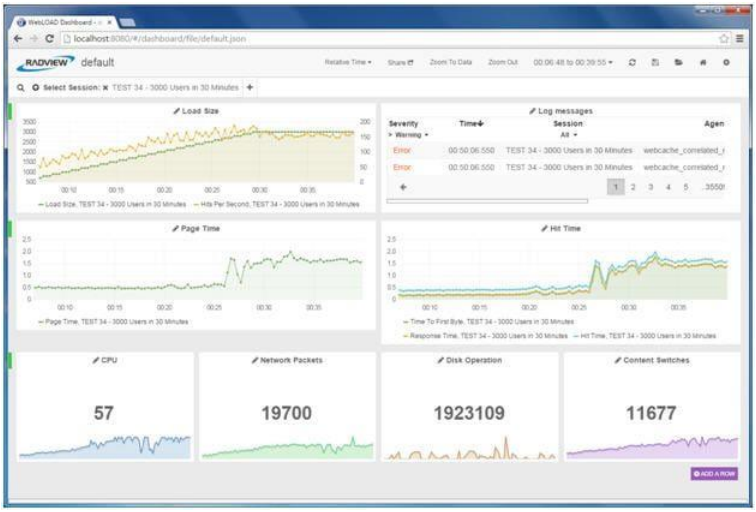


Рисунок 1.1 – Інтерфейс інструмента WebLOAD

Основні особливості цього інструменту тестування продуктивності такі:

– доступний у хмарі або як локально розгорнений;
– легко розширюється і підтримує всі основні веб-технології;
– гнучке тестове середовище;
– автоматично виявляє вузькі місця автоматично.

WebLOAD широко використовується службами тестування продуктивності зі складними та важкими вимогами до навантаження користувачів. Ви можете проводити стрес і навантажувальне тестування на будь-якому веб-програмному забезпеченні, створюючи навантаження з локальних систем та хмари. Він підтримує технології, включаючи корпоративні програми

для веб-протоколів. Він підтримує інтеграцію з такими інструментами, як Selenium, Jenkins та багатьма іншими, щоб забезпечити безперервне навантажувальне тестування для DevOps.

LoadNinja дозволяє створювати без сценаріїв навантажувальні тести. Службам тестування продуктивності він надає перевагу, оскільки допомагає скоротити час тестування вдвічі. Інші переваги включають заміну емуляторів завантаження реальними браузерами та створення дієвих показників. На рисунку 1.2 наведено приклад інтерфейсу головного вікна додатку.

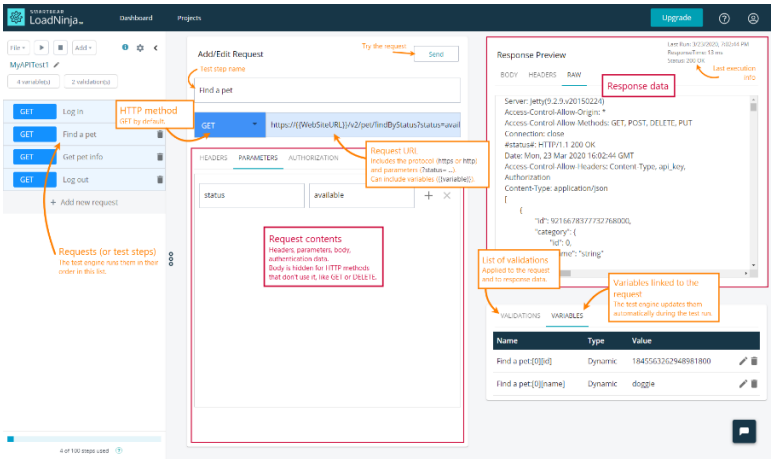


Рисунок 1.2 – Інтерфейс інструмента LoadNinja

Цей інструмент дозволяє виконувати налагодження в режимі реального часу, записувати взаємодії на стороні клієнта, миттєво виявляти проблеми з продуктивністю та робити набагато більше. LoadNinja дозволяє збільшити охоплення тестуванням без шкоди для якості. Він видаляє громіздкі завдання, що повторюються, пов'язані з перекладом сценаріїв, динамічним співвідношенням та очищенням сценаріїв. Це дозволяє витрачати більше часу на розробку масштабованих програм та менше часу на створення сценаріїв навантажувального тестування.

Основні особливості LoadNinja такі:

– справжнє масштабне тестування навантаження браузера;

– управління віртуальною активністю користувачів у режимі реального часу;

– тестова налагодження в режимі реального часу;

– розширені показники на основі браузера з функціями звітування та аналітики;

– хмарний хостинг виключає обслуговування сервера;

JMeter – популярний інструмент тестування продуктивності з відкритим кодом, призначений для перевірки навантаження та продуктивності. Його можна використовувати для аналізу та вимірювання продуктивності широкого спектру програмного забезпечення, що охоплює послуги, включаючи мережі та сервери. В основному він використовується як інструмент тестування навантаження веб-сайту для різних типів програм веб-служб. На рисунку 1.3 наведено приклад інтерфейсу головного вікна додатку.

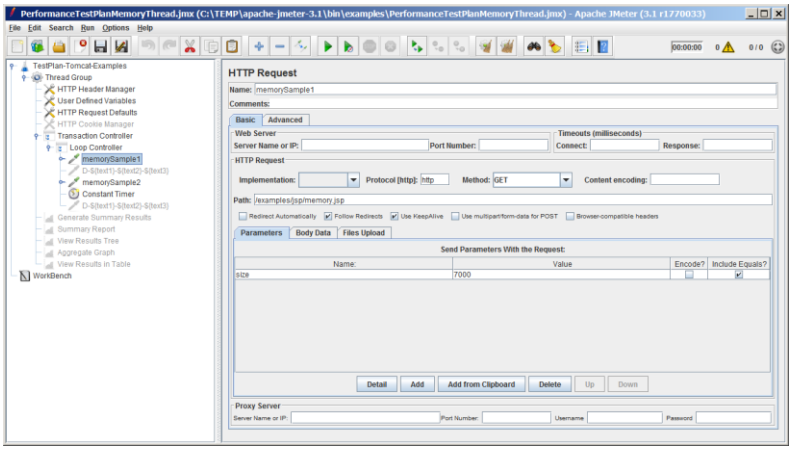


Рисунок 1.3 – Інтерфейс інструмента JMeter

JMeter дозволяє створити план перевірки функцій, крім плану випробування навантаження. Деякі з ключових особливостей такі:

– не вимагає розширеної інфраструктури для тестування навантаження;

– потрібні менші зусилля щодо створення сценаріїв щодо інших інструментів тестування продуктивності API, оскільки він має зручний інтерфейс;

– легкий аналіз ключової статистики завантаження та моніторів використання ресурсів шляхом подання простих графіків та діаграм.

Порівняння власної розробки з аналогами наведено в таблиці 1.1.

Таблиця 1.1 – Порівняльна характеристика програмних продуктів

Критерій	WebLOAD	LoadNinja	JMeter	Власна розробка
Створення користувацьких сценаріїв тестування	-	+	-	+
Розгортання у хмарі	+	-	+	+
Пошук вузьких місць	+	-	-	+
Збереження результатів тестування для аналізу	-	-	+	+
Аналіз результатів тестування	+	-	-	+
Тестування навантаження браузера	-	+	-	-
Сума	3	2	3	5

Таблиця порівняльних характеристик показала, що розробка власної системи є доцільною. В результаті буде отримано продукт, який покриває недоліки існуючих рішень. Користувач отримає доступ до всього необхідного функціоналу такого як: зручний інтерфейс створення користувацьких сценаріїв тестування, можливість розгортання у хмарі, автоматичний пошук вузьких місць у апаратній частині, збереження результатів тестування та автоматизований аналіз результатів тестування.

1.3 Аналіз методів аналізу результатів стрес-тестування

Серед методів реалізації подібних задач найчастіше використовують мультикритеріальні алгоритми тестування веб-додатків. Такі алгоритми можуть допомогти виявити як функціональні проблеми, так і проблеми в продуктивності, а також провести загальну оцінку надійності системи.

Один з таких методів – є алгоритм «Abstract Genetic Algorithm» запропонований Чжень Мін (Джек) Цзян. Цей метод дозволяє провести оцінку результатів стрес-тестування опираючись на попередніх, еталонних результатах. Основною особливістю цього метода є можливість гнучкої зміни еталонних результатів у зв'язку з можливими змінами коду веб-сервісу [3].

В основі даного методу є формування звіту (Рисуннок 1.4) на основі результатів стрес-тестування.

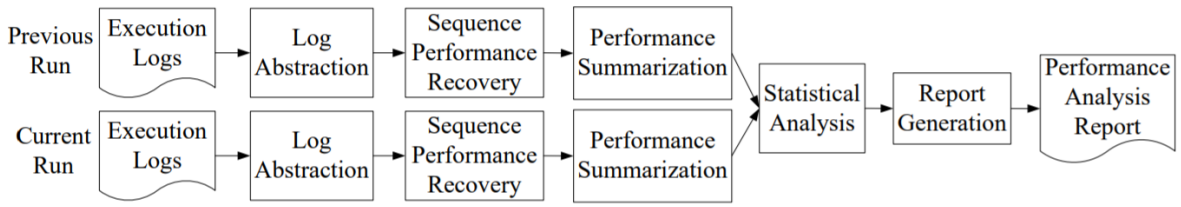


Рисунок 1.4 – Приклад формування звіту

Спочатку необхідно отримати базовий рівень продуктивності системи з попередніх запусків. Потім провести поглиблене порівняння продуктивності з похідним базовим рівнем продуктивності. Дослідження прикладів показують, що такий підхід викликає мало помилкових і розширюється до великих промислових систем.

Даний алгоритм позначає результати стрес-тестування, як сценарії з відхиленням. Оскільки така однозначна оцінка може дати похибку, її слід обчислювати відштовхуючись від загальної кількості запитів на сервер, та кількості запитів, що відповідають очікуваному статус коду з сервера та часу, необхідного на отримання відповіді з сервера на даний запит.

До недоліків даного методу слід віднести ігнорування проблем, пов'язаних з апаратною частиною сервера, на якому запущено веб-сервіс. В результаті чого, це може призвести до вибору некоректних еталонних значень, що створює можливу похибку в оцінці результату.

1.4 Постановка задачі розробки

Після аналізу поточного стану питання та порівняння існуючих рішень за визначеними критеріями визначено завдання, які необхідно виконати, для розробки системи для підвищення продуктивності веб-сервісів під час критичних навантажень на систему:

1. Провести аналіз існуючих методів діагностування продуктивності веб-сервісів.
2. Розробити метод аналізу результатів стрес-тестування.
3. Розробити метод агрегації результатів стрес-тестування.
4. Розробити сервіс для створення користувацьких сценаріїв тестування.
5. Розробити сервіс автоматичного пошуку вузьких місць у апаратній частині.
6. Розробити клієнтську частину.
7. Провести тестування додатку.

Технічне завдання наведено у Додатку А.

1.5 Висновки

У першому розділі було розглянуто актуальний стан питання програмних реалізацій інструментів для проведення тестування продуктивності веб сервісів в умовах критичного навантаження. Досліджено предметну область, а також розглянуто існуючі аналоги та їх недоліки.

Аналіз поточного стану питання показав, що розробка власної системи для підвищення продуктивності веб-сервісів під час критичних навантажень на систему є доцільною, оскільки вона буде містити весь необхідний функціонал, а також не матиме недоліків існуючих реалізацій. В основі методу для аналізу результатів стрес тестування буде удосконалений метод Abstract Genetic Algorithm», оскільки це дозволить забезпечити необхідний рівень швидкодії, а також розрахувати правильно можливі ризики знайдених проблем. Проведено постановку задач розробки.

2 РОЗРОБКА СТРУКТУРИ СИСТЕМИ ТА МЕТОДІВ ПІДВИЩЕННЯ ПРОДУКТИВНОСТІ ВЕБ-СЕРВІСІВ

2.1 Розробка методу агрегації результатів стрес-тестування

Для порівняння результатів з попередніми запусками є необхідним збереження результатів. Існує 2 можливих варіанта збереження даних:

– збереження даних до файлу;

– збереження результатів до БД.

Основною перевагою збереження даних до файлу – є можливість зберегти дані про кожен запит до веб-сервісу. Якщо за один запуск буде генеруватись забагато запитів, при збереженні даних у БД – сховище буде дуже швидко заповнене, тому у БД є сенс у збереженні лише необхідних результатів агрегатних операцій з даними. У файлах не потрібно зберігати результати агрегатних операцій над даними, оскільки потім можна буде виконати будь-які операції з повним набором даних.

Отже найкращим варіантом для збереження даних є змішаний формат, де повний перелік запитів зберігається у окремому файлі, а результати аналізу зберігаються у БД, для зручного поширення результатів між співробітниками.

Необхідно розробити макет БД. Потрібно забезпечити можливість легко розділяти сценарії виконання та групувати однакові. Для цього буде створено таблицю з різними сценаріями. До неї будуть входити наступні атрибути:

– унікальний ідентифікатор сценарію;

– назва сценарію;

– хеш-сума сценарію – за цим атрибутом можна буде швидко розрізняти різні файли сценарію;

– текст сценарію – крім хеш-суми також буде збережено сам сценарій для можливості легко відрізнити сценарії між собою.

Також необхідно розробити таблицю, у якій будуть збережені результати тестування. до неї будуть входити наступні атрибути:

– унікальний ідентифікатор запуску;

– унікальний ідентифікатор сценарію;

– дата та час проведення тестування;

– результати тестування у форматі JSON, для подальшого оброблення та можливості легко розширити функціонал, додавши нові потрібні параметри.

Кінцеву модель БД зображено на рисунку 2.1.

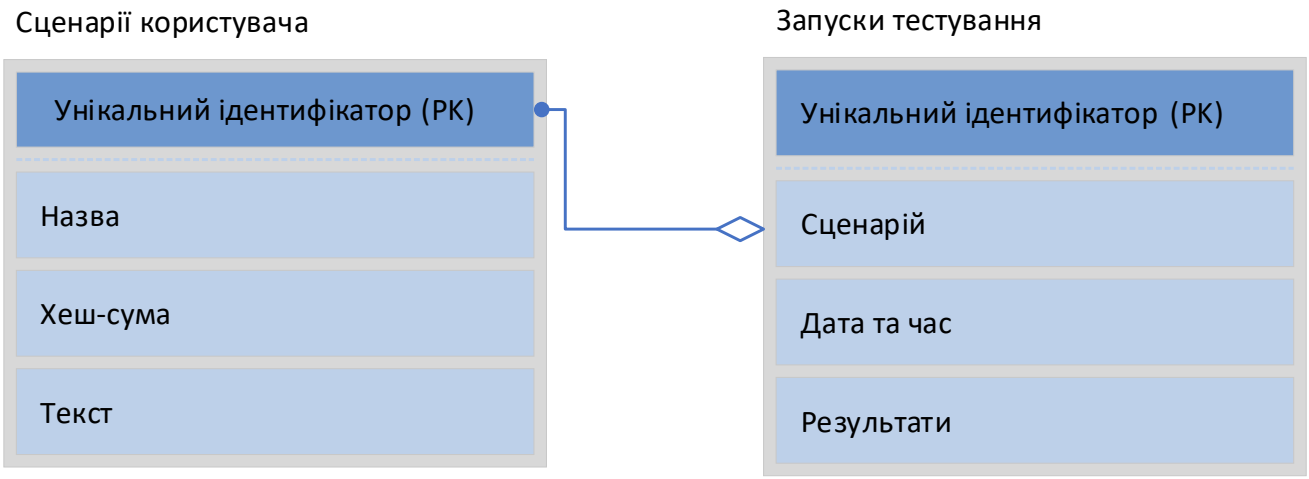


Рисунок 2.1 – Модель БД для збереження результатів тестування

Після проектування бази даних, потрібно обрати, у якому форматі зберігати результати стрес-тестування. Ключовими вимогами є можливість повторного створення звіту на основі даних, а також можливість використання цих даних, для подальшого порівняння з іншими результатами.

Зазвичай, для цього використовуються різні агрегатні операції, такі як середнє значення і медіана вибірки часу виконання запиту, а також максимальне та мінімальне значення. Часто ці значення дуже схожі, тому потрібно розглянути детальніше різницю між ними у даній статистиці.

Арифметичне середнє – сума всіх фіксованих значень набору, поділена на кількість елементів набору. Якщо з контексту зрозуміло, про яке значення йде мова, тоді просто кажуть середнє. Термін середнє арифметичне вживають для того щоб виокремити використовуване значення від інших середніх величин, таких, наприклад, як середнє геометричне або середнє гармонійне.

Медіана – в статистиці це величина ознаки, що розташована посередині ранжованого ряду вибірки, тобто – це величина, що розташована в середині ряду величин, розташованих у зростальному або спадному порядку.

Якщо даних занадто багато, то декілька пікових значень може спотворити результат середнього арифметичного, тому слід використовувати медіану. І навпаки, коли результатів замало, медіана не може дати значну похибку, тому слід формувати обидва значення.

Оскільки у спеціаліста з навантажувального тестування обмежений час, ми оцінюємо потенційно проблемні сценарії, щоб допомогти йому визначити пріоритети свого часу. Ми ранжируємо кожен сценарій за ступенем відхилення продуктивності між попереднім і поточним запуском.

Косинус відстань, який вимірює ступінь подібності між двома розподілами, виводить значення від 0 до 1. Якщо два розподіли дуже подібні, косинусна відстань близька до 1. Якщо вони дуже різні, косинусна відстань близька до 0 Оскільки відхилення є протилежністю подібності, ми використовуємо таку формулу для обчислення відхилення (deviation):

$$deviation(P, C) = 1 - coisine(P, C), \tag{2.1}$$

$$coisine(P, C) = \frac{\sum_x P(x)C(x)}{\sqrt{\sum_x P(x)^2}\ \sqrt{\sum_x C(x)^2}}, \tag{2.2}$$

де P(x) і C(x) відповідають кількості екземплярів у попередньому та поточному запуску, які мають час реакції x для конкретного сценарію відповідно.

Оскільки такий підхід позначає лише сценарії з відхиленнями в продуктивності, його знанням залежить, чи призведе відхилення продуктивності до проблем з продуктивністю.

Тому є необхідним оцінити точність оцінки ефективності даного підходу. Точність (precision) визначається наступним чином:

$$precision(\%) = \left(1 - \frac{\#\ of\ false\ alarm\ scenarios}{Total\ \#\ of\ flagged\ scenarios}\right), \quad (2.3)$$

де «# of false alarm scenarios» – це кількість помилково визначених відхилень, а «Total # of flagged scenarios» – сумарна кількість позначених сценаріїв.

Кілька сценаріїв з відхиленням продуктивності можуть бути викликані однією і тією ж проблемою продуктивності, що може свідчити про вплив цієї проблеми. Якщо позначений сценарій не призводить до проблеми з продуктивністю, цей сценарій розглядається як помилкова тривога [3].

Для того, щоб додати оцінку апаратної частини сервера та використати її у даному методі необхідно зробити заміри завантаженості на сервер безпосередньо перед початком тестування, а також постійно заміряти її під час самого тестування. таким чином можна оцінити, наприклад зміну CPU під час тестування, а також впевнитись в готовності сервера в проведенні стрес-тестування.

Таким чином деякі знайдені витоки у продуктивності можна буде позначити як помилкова тривога, оскільки в даний момент сервер був навантажений ще до початку проведення тестування.

2.2 Розробка методу аналізу результатів стрес-тетсування

Після перевірки правильності функціонування системи під навантаженням наступним кроком є визначення, чи є якісь нефункціональні проблеми (наприклад, проблеми з продуктивністю). Проблеми з продуктивністю відносяться до ситуацій, коли система страждає від несподівано високого часу відгуку або низької пропускної здатності. Важко виявити проблеми продуктивності під час навантажувального тесту через відсутність формально визначених цілей продуктивності та велику кількість даних, які необхідно перевірити.

Багато програмних додатків повинні надавати послуги сотням або тисячам користувачів одночасно. Ці програми повинні бути перевірені навантаженням, щоб переконатися, що вони можуть правильно функціонувати під високим навантаженням. Функціональні проблеми під час навантажувального тестування пов'язані з проблемами середовища навантаження, генераторів навантаження та програми, що тестується. Важливо виявити та вирішити ці проблеми, щоб переконатися, що результати тестування навантажень є правильними, і ці проблеми вирішені. Важко виявити такі проблеми під час навантажувального тесту через велику кількість даних, які необхідно перевірити. Сучасна промислова практика в основному включає в себе трудомісткі ручні перевірки, які, наприклад, збирають журнали програми на наявність повідомлень про помилки.

Тому необхідним є використання підхіду, який автоматично аналізує журнали виконання навантажувального тесту на предмет проблем з продуктивністю.

Оскільки додаткове обладнання або профілювання системи сповільнюють її виконання, ці методи не підходять для аналізу навантажувального тестування. В результаті, підхід «Abstract Genetic Algorithm» аналізує легкодоступні журнали виконання. Для цього використовується метод абстракції журналів, для автоматичного перетворення журналів виконання в події виконання.

Журнали виконання генеруються вихідними операторами, які розробники вставляють у вихідний код. Журнали виконання широко доступні і корисні для моніторингу, віддаленого вирішення проблем і розуміння системи складних корпоративних програм. Існує багато пропозицій щодо стандартизованих форматів журналів, таких як формати W3C і SNMP. Однак більшість програм використовують спеціальні нестандартизовані формати журналу [18]. Автоматизований аналіз таких журналів є складним через нечітко визначену структуру та великий нефіксований словниковий запас слів. Великий обсяг журналів, створених корпоративними програмами, обмежує корисність методів ручного аналізу. Щоб розкрити структуру журналів виконання, необхідні

автоматизовані методи. Використовуючи непокриту конструкцію, можна виконати складний аналіз журналів. Використовуючи відновлену структуру, рядки журналу можна легко узагальнити та класифікувати, щоб допомогти зрозуміти та дослідити складну поведінку великих програмних додатків. Запропонований підхід обробляє рядки журналу довільної форми з мінімальними вимогами до формата рядка журналу.

Оскільки система обробляє одночасні клієнтські запити, рядки журналу з різних сценаріїв змішуються один з одним у журналах виконання. Крім того, деякі рядки журнал, які виводять лише інформацію про стан системи, не пов’язані з жодним сценарієм клієнта. Ці рядки журналу слід відфільтрувати з нашого аналізу ефективності.

Щоб відновити продуктивність усіх сценаріїв замовника, нам потрібно спочатку відновити послідовності подій. Ми відновлюємо послідовності, зв’язуючи відповідні значення параметрів.

Перший етап абстрагує кожен рядок журналу в подію виконання. Тут ми використовуємо динамічні значення, а також імена їх параметрів у події виконання. Пара параметрів ім’я-значення використовуються для зв’язування пов’язаних рядків журналу в послідовності. Продуктивність відновленої послідовності розраховується шляхом різниці в часі між першою та останньою подіями.

Також розраховується тривалість між кожною сусідньою парою подій. Попарна інформація про тривалість використовується пізніше для поетапної діагностики продуктивності.

Для пошуку сценаріїв з відхиленням, порівнюємо ефективність кожного сценарію в попередньому та поточному запусках за допомогою статистичного тесту.

Після того, як сценарії з відхиленою продуктивністю позначені, нам потрібно точно визначити пари подій, які викликають відхилення продуктивності. Ми досягаємо цього, застосовуючи той самий статистичний тест до всіх сусідніх пар подій.

На рисунку 2.2 наведено блок-схему алгоритму для аналізу результатів стрес-тестування.

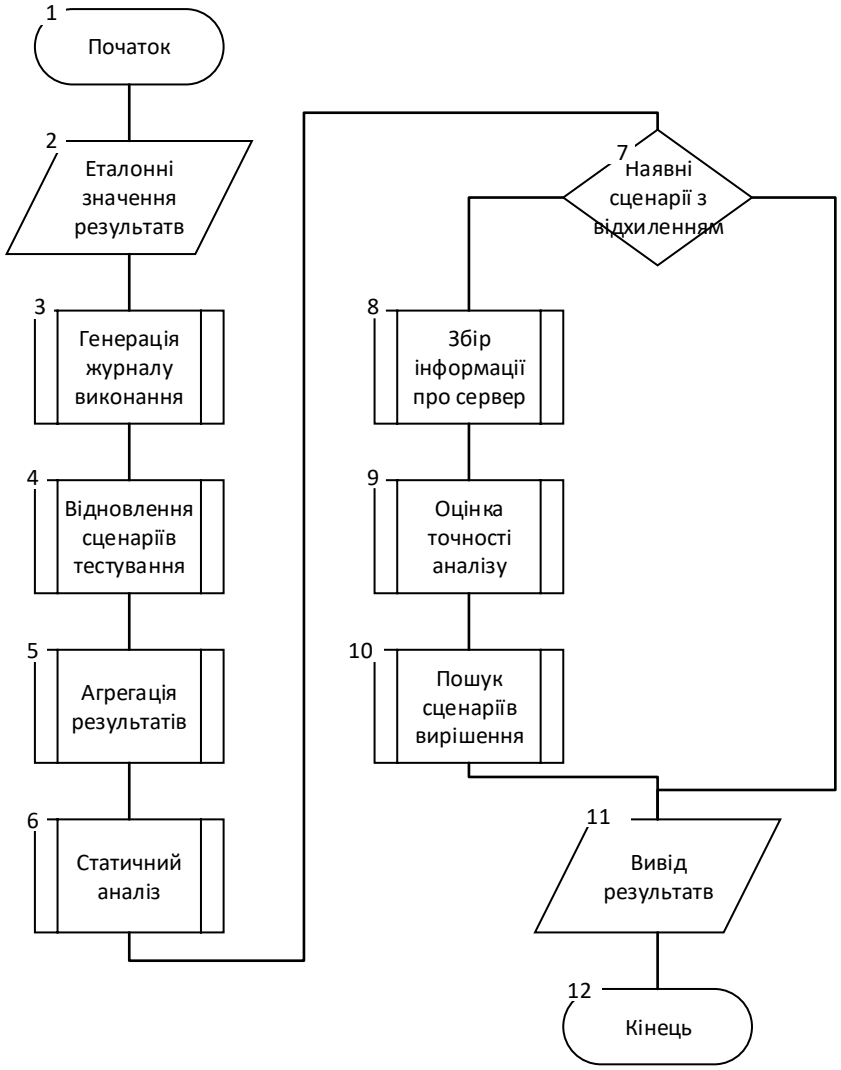


Рисунок 2.2 – Блок-схема алгоритму для аналізу результатів стрес-тестування

Алгоритм роботи програмного модуля наступний:

1. Початок роботи модуля.

2. Зчитування еталонних значень результатів стрес-тестування з бази даних.

3. Генерація журналу виконання.

4. Відновлення сценаріїв тестування на основі користувацького файлу, що використовувався для збирання відповідних даних.

5. Агрегація результатів стрес-тестування.

6. Виконання статичного аналізу, для пошуку сценаріїв з відхиленням.

7. Перевірка, чи наявні сценарії з відхиленням.

8. Збір збереженої інформації про апаратну частину сервера перед та під час запуску стрес-тестування.

9. Оцінка точності вимірювання на основі нових даних про апаратні частину за допомогою формули 2.3.

10. Пошук можливих шаблонів вирішення проблеми сценарію з відхиленням.

11. Формування звіту та поверненн його з підпрограми.

12. Кінець роботи модуля.

Таким чином відбувається аналіз результатів стрес-тестування в додатку.

2.3 Розробка методу пошуку аномалій в SQL запиті та їх вирішення

Для того, щоб провести аналіз запиту, перш за все необхідно обробити вхідний запит. За допомогою інструментів SQL Server, можна обробляти не сам запит, а готовий фізичний план виконання. Однак для визначення місця проблеми необхідно також співставити план виконання та запит.

Додаток повинен працювати як зі з’єднанням до БД, так і без нього, але у вимкненому режимі обробити повноцінно вхідний запит буде неможливо, тому потрібно створити колекцію з підказками користувачу по покращенню запиту. Після аналізу ця колекція буде форматуватись в коментарі формату MS SQL та буде доданою в кінець запиту.

При успішному формуванні плану виконання, слід звернути увагу на попередження від середовища виконання, оскільки це допоможе знайти додаткові помилки, оцінюючи витрати на кожний оператор окремо.

Після повного аналізу, запит потрібно сформувати в зручний для користувача вигляд. Цього легко досягти за допомогою знаків абзацу та табуляції для виокремлення елементів у логічно зв’язані групи.

На рисунку 2.3 зображено блок-схему алгоритму для аналізу та оптимізації SQL-запитів, яка складається з 37 логічних елементів.

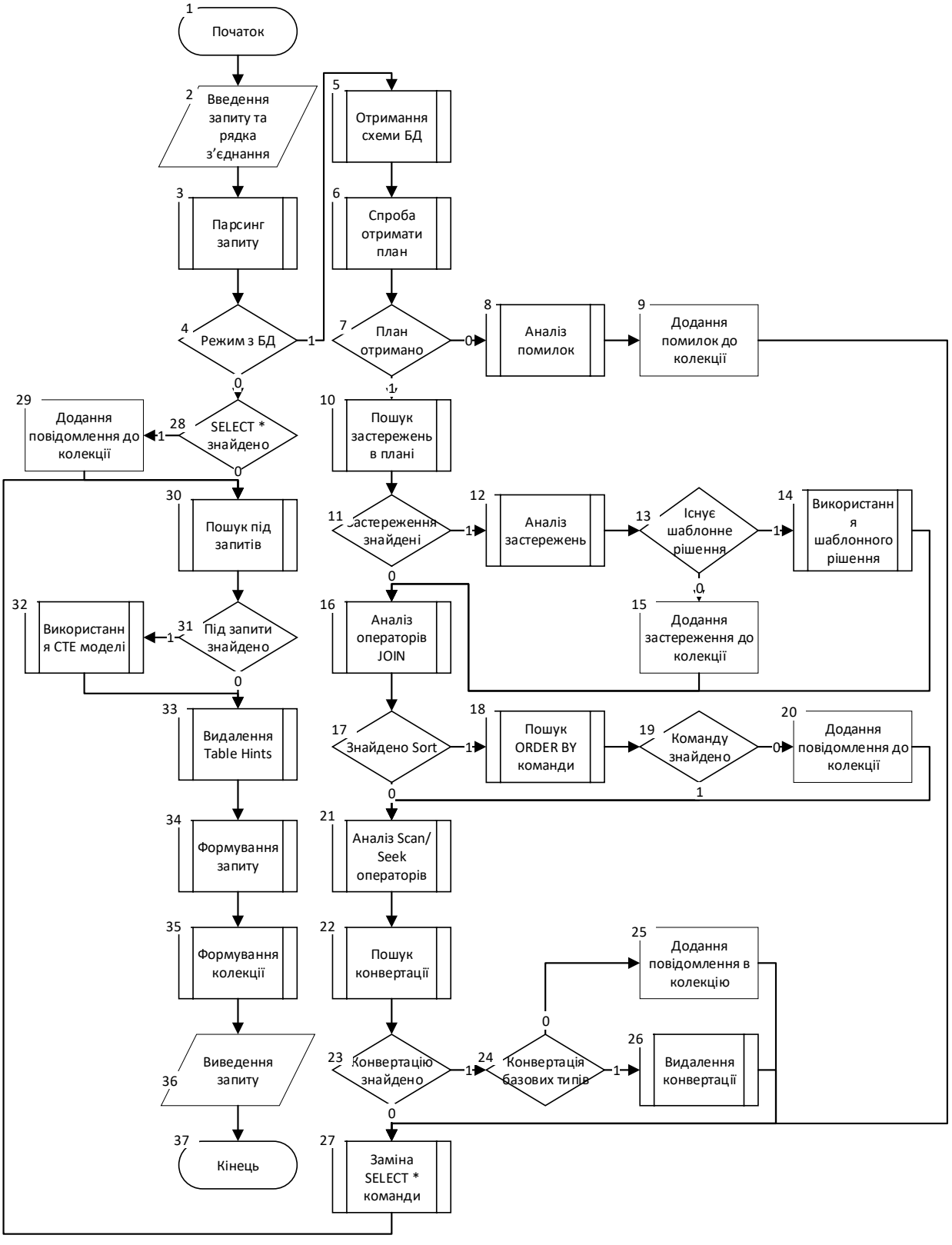


Рисунок 2.3 – Блок-схема алгоритму пошуку аномалій в запиті

Алгоритм роботи програмного модуля наступний:

1. Початок роботи підпрограми.
2. Зчитування запиту та рядку підключення.
3. Парсинг запиту.
4. Перевірка, чи з’єднання з БД активне. Вибір режиму роботи без з’єднання та помилка під час підключення вважається за неактивну БД.
5. Отримання структури таблиць БД з назвами та типами колонок, а також кількість даних в таблицях.
6. Спроба отримати план виконання даного запиту для аналізу.
7. Перевірка результату плану виконання, незадовільний результат зазвичай означає синтаксичну помилку в запиті.
8. Аналіз помилки. Відбувається виділення всіх місць, де могла з’явитись помилка.
9. Повідомлення про помилку створення плану виконання додається до колекції підказок.
10. Аналіз попереджень від SQL Server про наявність аномалій.
11. Перевірка чи були знайдені попередження.
12. Якщо попередження були знайдені, відбувається їх аналіз та співставлення з збереженими шаблонами їх вирішення.
13. Перевірка знайденого шаблону.
14. Накладання шаблону на запит.
15. У випадку, якщо помилка не шаблонна, попередження додається до колекції підказок.
16. Аналіз обраних фізичних операторів з’єднання та співставлення вибору до кількості даних в таблицях.
17. Перевірка чи в плані виконання є оператори сортування.
18. Пошук сортування всередині запиту.
19. Співставлення кількості операторів сортування в запиті з кількістю в плані виконання.

20. Якщо в плані виконання відбувається непередбачене користувачем сортування, повідомлення про це додається до колекції підказок.

21. Аналіз передбачуваної кількості даних, що повертають оператори INDEX SCAN та INDEX SEEK. Якщо кількість даних не відповідає обраному оператору – це означає що проблема в щільності індексу, таку проблему може вирішити лише архітектор БД, тому про це буде попереджений користувач, але як помилка яку користувачу слід виправити відміченою не буде.

22. Пошук конвертації в плані виконання.

23. Перевірка, чи конвертація була знайдена.

24. Перевірка чи конвертація відбувається між простими типами, наприклад числовим та строковим.

25. Якщо конвертація неможливо забрати автоматично, повідомлення з відповідним текстом додається до колекції підказок.

26. Видалення зайвої конвертації, та спростування арифметичних операцій.

27. Заміна конструкції «SELECT *», цю операцію в більшості випадків можна виконати навіть при наявності синтаксичної помилки в запиті. Відбувається заміна «*» на список всіх полів з даної таблиці.

28. Перевірка наявності конструкції «SELECT *» для запиту з неактивним підключенням до БД.

29. Додання тексту помилки до колекції підказок. Оскільки без з’єднання з БД автоматично замінити на список стовпців неможливо.

30. Пошук під запитів з використанням оператора IN.

31. Перевірка, чи під запит було знайдено.

32. Перенесення під запиту за межі головного запиту за допомогою операцій CTE, операторів з’єднань LEFT JOIN, RIGHT JOIN та конструкції умови IS [NOT] NULL.

33. Видалення всіх табличних підказок.

34. Форматування запиту, заміна зайвих пробілів та додання знаків абзацу та табуляції.

35. Конвертація списку підказок у вигляді коментарів вкінці запиту.
36. Повернення з підпрограми сформованого SQL-запиту.
37. Кінець виконання підпрограми.

Таким чином відбувається аналіз та оптимізація SQL-запитів в додатку.

2.4 Розробка архітектури системи

Мікросервіси — це архітектурний та організаційний підхід до розробки програмного забезпечення, де програмне забезпечення складається з невеликих незалежних служб, які спілкуються через чітко визначені API. Ці служби належать невеликим автономним командам [19].

Архітектура мікросервісів полегшує масштабування та швидшу розробку додатків, забезпечуючи інновації та прискорюючи вихід на ринок нових функцій.

Монолітна архітектура вважається традиційним способом побудови додатків. Монолітний додаток будується як єдиний і неподільний блок. Зазвичай таке рішення містить клієнтський інтерфейс користувача, серверну програму та базу даних. Він уніфікований, і всі функції керуються та обслуговуються в одному місці.

Зазвичай монолітні програми мають одну велику кодову базу і не мають модульності. Якщо розробники хочуть щось оновити або змінити, вони мають доступ до тієї самої бази коду. Таким чином, вони вносять зміни у весь стек відразу. У той час як монолітний додаток є єдиним уніфікованим блоком, архітектура мікросервісів розбиває його на набір менших незалежних блоків. Ці підрозділи виконують кожен процес подачі заявок як окрему службу. Тому всі сервіси мають свою логіку та базу даних, а також виконують певні функції [20].

В межах архітектури мікросервісів вся функціональність розділена на модулі, які можна розгортати незалежно один від одного, які спілкуються один з одним за допомогою визначених методів, які називаються API (інтерфейси програмного забезпечення). Кожна служба охоплює свою власну сферу і може оновлюватися, розгортатися та масштабуватися незалежно.

Архітектура системи для підвищення продуктивності веб-сервісів під час критичних навантажень наведена на рисунку 2.4.

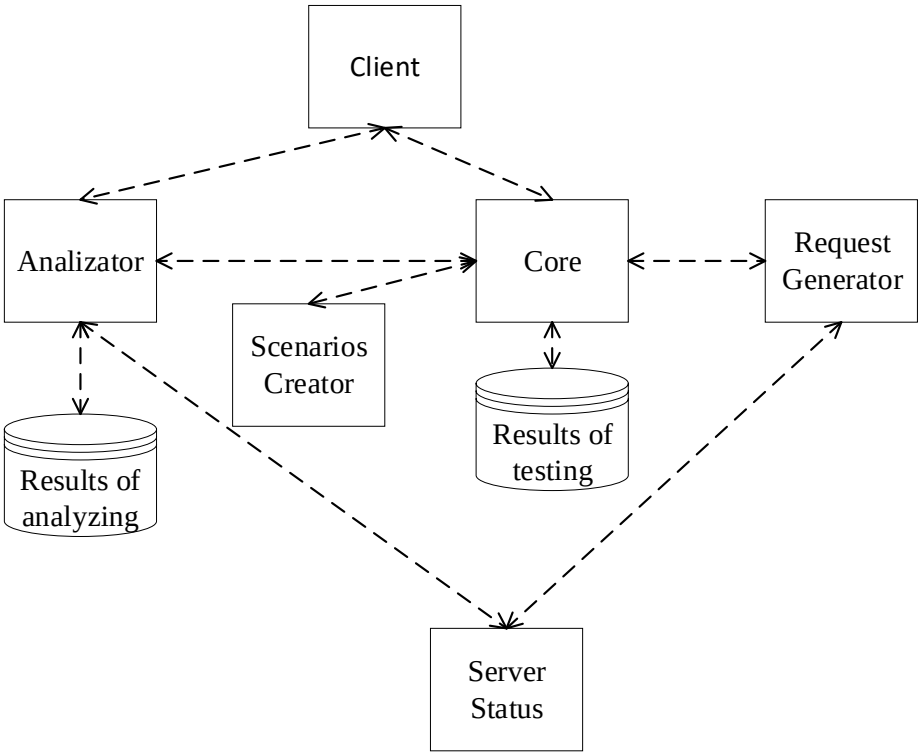


Рисунок 2.4 – Архітектура системи

Отже для розробки програмного продукту було обрано архітектуру мікросервісів. Весь додаток буде складатись з наступних компонентів:

1. Client – модуль, що забезпечує взаємодію з користувачем. Надає графічний інтерфейс для створення нових сценаріїв тестування та їх запуску.

2. ScenariosCreator – модуль, що забезпечує можливість створення нових користувацьких сценаріїв тестування, а також їх серелізацію та валідацію.

3. RequestGenerator – модуль, що відповідає за надсилання трафіку до веб-сервісу. Виконує перевірку відповідності бажаного результату, перевіряючи статус код та тіло відповіді з сервера.

4. Analizator – модуль, що відповідає за аналіз результатів тестування, формування звітів на їх основі.

5. ServerStatus – модуль, що надає інформацію по апаратним даним сервера, на який виступає хостом веб-сервісу, що тестується.

6. Core – основний модуль, який забезпечує взаємодію різних частин системи, виступаючи посередником при відправці повідомлень між ними.

2.5 Розробка діаграми послідовності

Перед побудовою діаграми послідовності необхідно розробити загальну блок-схему роботи додатку. На рисунку 2.5 наведено блок-схему модуля для проведення стрес-тестування.

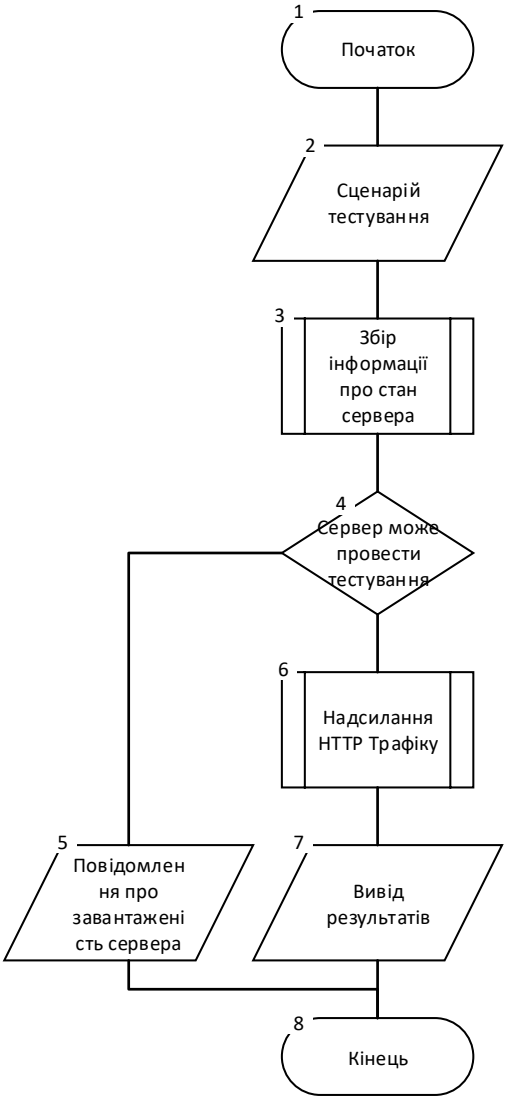


Рисунок 2.5 – Блок-схема модуля для проведення стрес-тестування

Алгоритм роботи програмного модуля наступний:

1. Початок роботи програми

2. Зчитування користувацького сценарію тестування. На даному етапі також проводиться парсинг сценарію.

3. Збір інформації про стан сервера.

4. Перевірка, чи може сервер проводити тестування.

5. Якщо, сервер вже завантажений, то модуль повертає повідомлення про завантаженість сервера.

6. Імітація надсилання HTTP трафіку на веб-сервіс.

7. Вивід результатів стрес-тестування.

8. Кінець роботи модуля.

Пропонується в діаграмі послідовності розглянути основний сценарій використання, а саме проведення стрес-тестування веб-сервісу. Цей сценарій проілюстровано на рисунку 2.6.

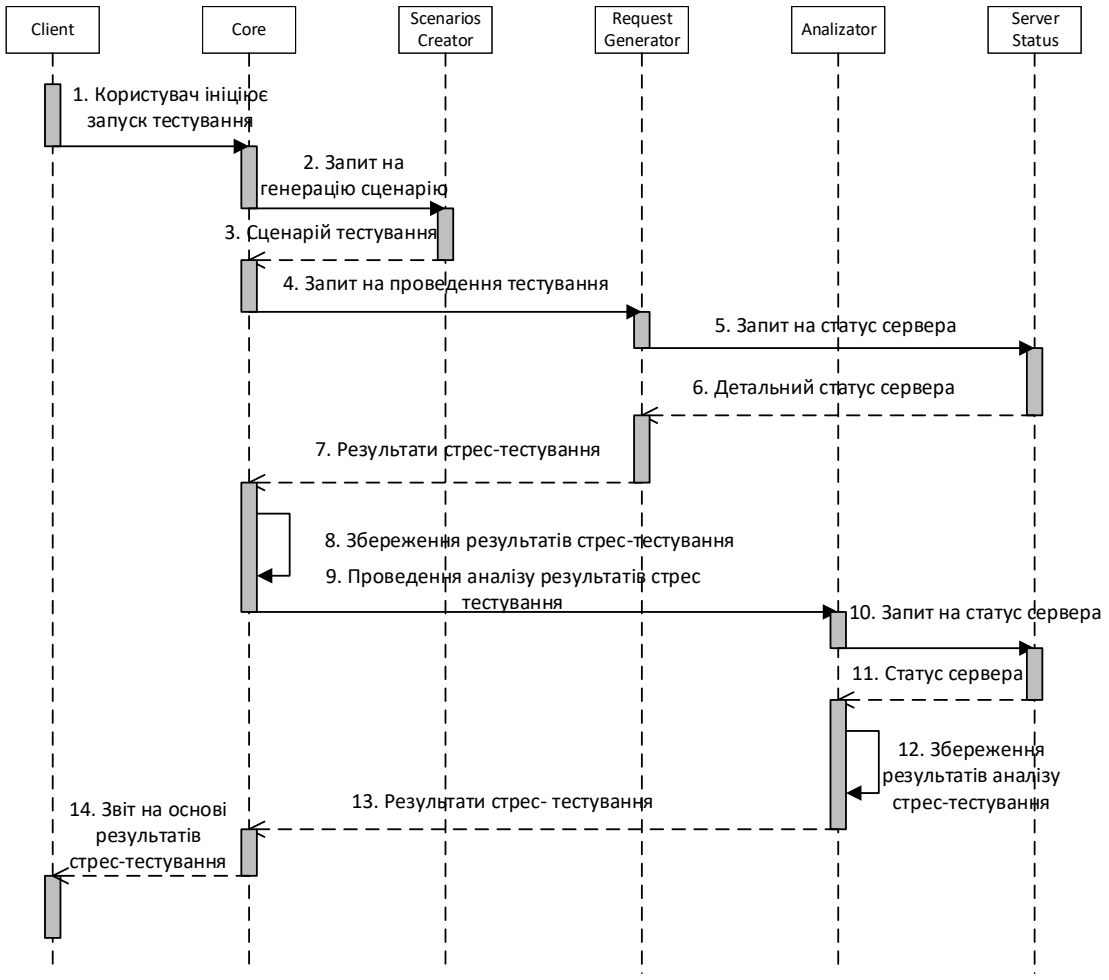


Рисунок 2.6 – Загальна діаграма послідовності

Дана діаграма дозволяє отримати розуміння у роботі окремих елементів системи та спростити конструювання та розробку програмного забезпечення. Також можна зрозуміти як різні сервіси взаємодіють між собою, порядок виклику процедур та життєвий цикл об’єктів.

2.6 Висновки

В другому розділі розроблено метод та алгоритм автоматизованого аналізу результатів стрес-тестування. В результаті чого подальшого розвитку отримав метод автоматизованого аналізу результатів стрес-тестування «Abstract Genetic Algorithm», в якому, на відміну від класичного методу, використано додаткову оцінювальну функцію автоматичного пошуку вузьких місць у апаратній частині сервера, що дало можливість збільшити точність пошуку сценаріїв з низькою продуктивністю. Розроблено метод агрегації результатів стрес-тестування, який, на відміну від існуючих, зберігає окремо результати стрес тестування та їх аналізу, що дало можливість підвищити ефективність проведення аналізу результатів стрес-тестування. А також удосконалено метод оптимізації SQL-запитів у середовищі MS SQL, який, на відміну від відомих алгоритмів, передбачає зменшення кількості операцій читання, що дозволило зменшити час виконання запитів

Розроблено архітектуру системи, яка базується на мікросервісному підході. Розроблено діаграму послідовності для основних сценаріїв використання системи.

3 РОЗРОБКА ПРОГРАМНИХ ЗАСОБІВ

3.1 Варіантний аналіз і обґрунтування вибору засобів реалізації

Фреймворк .NET представляє потужну платформу для створення додатків. Можна виділити такі риси як підтримка декількох мов програмування, кросплатформенність, потужна бібліотека класів та різноманітність технологій.

При компіляції, програмний коду трансформується не на машинний код, як в більшості інших мов програмування, а в CIL (Common Intermediate Language). Це проміжна версія коду, перед перетворенням у машинний, у яку конвертується однаково весь код написаний на платформі .NET незалежно від обраної мови програмування. Це дозволяє розробляти програмне забезпечення, що складається з модулів, написаних на різних мовах програмування, які підтримуються фреймворком .NET [21].

.NET Standart розділяється на дві версії: .NET Framework та .Net Core. Вони відрізняються принципом виконання, тому вважаються несумісними один з одним, однак часто один і той же програмний код може бути виконаний на обох версіях .NET. основною відмінністю є кросплатформенність .NET Core, тому для можливості використання додатку не тільки на операційній системі Windows, а й також MacOS та Linux, буде використовуватись саме ця платформа.

CLR і базова бібліотека класів є основою для цілого стека технологій, які розробники можуть задіяти при побудові тих чи інших додатків. Наприклад, для роботи з базами даних в цьому стеку технологій призначена технологія ADO.NET і Entity Framework Core [22].

Також ще слід відзначити таку особливість фреймворку .NET, як Garbage Collector. А це означає, що нам в більшості випадків не доведеться, на відміну від C++, піклуватися про звільнення пам'яті. Вищезазначена загальномовного середовища CLR сама викличе Garbage Collector і очистить пам'ять.

Garbage Collector запускається лише при необхідності, коли для створення об’єкта відсутня цільна ділянка пам’яті кучі, що задовольнить розмір. Всередині Garbage Collector будує граф, для оцінки всіх об’єктів типу посилань, що

використовуються. Після чого видаляє з пам’яті всі об’єкти, які не були включені в граф, тобто на них більше немає посилань всередині коду. Пам’ять разом з усіма об’єктами, що залишились після виконання, дефрагментується для трансформації всієї вільної пам’яті в одну цільну комірку пам’яті.

Найпопулярнішими мовами програмування в середовищі .NET є C#, F# та VB.NET. Оскільки всі ці мови виконуються в одному середовищі та транслюються в CIL, різниця між ними є досить не велика.

F# – мова програмування, що підтримує одразу декілька парадигм програмування, такі як процедурна, об’єктно орієнтовна та функціональна парадигми.

Visual Basic .NET (VB.NET) – об’єктно орієнтовна мова програмування, що з’явилась в наслідок еволюції мови програмування Visual Basic.

C#, як і VB.NET підтримує лише об’єктно орієнтовну парадигму програмування, але на відміну від попередніх аналогів є стандартизованою в ECMA та ISO

У таблиці 3.1 наведено основні відмінності між мовами програмування середовища .NET.

Таблиця 3.1 – Порівняння мов програмування

Назва критерію	C#	F#	VB.NET
Динамічна типізація	+	-	+
Неявне приведення до типів	+	-	+
Псевдоніми типів	+	+	-
Виведення сигнатури для локальних методів	-	+	-
Багатопоточна компіляція	-	+	+
Створення об’єктів у стеку	+	+	-
Спискові виключень	+	+	-
Значення параметрів по замовчуванню	+	-	+
Цілі числа довільної довжини	+	+	-

Основою платформи є загальномовне середовище виконання Common Language Runtime (CLR), завдяки чому .NET підтримує кілька мов: поряд з C# це також VB.NET, C++, F#, а також різні діалекти інших мов, прив'язані до .NET, наприклад, Delphi. NET.

Динамічна типізація дозволяє створювати динамічні методи та класи, тип яких буде визначений під час виконання роботи програми. А також можливість використовувати один і той самий метод для різних типів даних.

Неявне приведення до типів дозволяє при наслідуванні конвертувати дані з типів наслідуваних класів, до конкретних реалізацій та навпаки. Оскільки в C# та VB.NET всі об’єкти наслідуються від загального класу Object, можна дуже зручно зберігати колекції різнотипних даних у форматі цього класу.

Псевдоніми типів дозволяє перевірити тип даних об’єкта під час виконання роботи програми. Ця функція є дуже зручною разом з динамічною типізацією, оскільки за рахунок перевірок типу можна значно зменшити кількість методів дублікатів.

Виведення сигнатури для локальних методах дозволяє динамічно, під час виконання програми, видаляти та змінювати сигнатуру локальних методів. Подібна поведінка притаманна функціональним мовам програмування, тому вважається неважливим критерієм.

Багатопоточна компіляція коду дозволяє одночасно компілювати різні модулі додатку, що значно скорочує час потрібний для компіляції. Ця функція найкраще себе показує, коли додаток складений з багатьох мало зв’язних модулів. Оскільки в додатку, велика кількість модулів не передбачається, можна цьому критерію поставити нижчий пріоритет за інші.

У .NET передбачена роботи з типами-посилань та типами-значень. Перші зберігаються в купі, що дозволяє їх динамічне видалення за допомогою Garbage Collector. Інші зберігаються у стеку у видаляються по мірі закінчення їх зони видимості. Відсутність можливості зберігання даних в кучі значно збільшує обсяги пам’яті необхідні для виконання.

Спискові виключення допомагають компактно описати чергу подій, необхідних для виконання. Тоді як в F# є спеціальний тип даних для створення цих списків, в C# цю функцію частково виконують делегати та події.

Можливість задати параметри по замовчуванню дозволяє викликати один і той самий метод з різною кількістю параметрів, без необхідності кожен раз при виклику цього метода передавати ці ж самі параметри по замовчуванню.

Оскільки довжина чисел в SQL Server є незалежною від мови програмування, дуже важливо щоб чисельні типи могли завжди вміщатись в змінні обраної мови програмування. В іншому випадку усі арифметичні операції слід обробляти за алгоритмами довгої арифметики, що значно зменшує час виконання та розробки.

Оцінивши основні відмінності між мовами програмування середовища .NET було обрано використовувати мову програмування C#, оскільки вона задовільняє основні важливі аспекти, необхідні для розробки даного додатку, а її недоліки є менш пріоритетними критеріями відносно недоліків аналогів.

Отже, для написаня програмного додатку буде використана мова програмування C#.

Інтегроване середовище розробки – комплексне програмне рішення для розробки програмного забезпечення. Зазвичай складається з редактора коду, інструментів для автоматизації складання проекту та відлагодження [23].

Під час вибору середовища розробки було розглянуто такі додатки: MS Visual Studio, Project Rider, Eclipse.

MS Visual Studio (рисунок 3.1) – це серія продуктів фірми Microsoft. IDE володію великим набором засобів і можливостей: дозволяє розробляти, як консольні додатки так і додатки з графічним інтерфейсом, а також з підтримкою технології Windows Forms, а також веб-сайти, веб-додатки та веб-служби для всіх платформ, що підтримуються. Однак, не дивлячись на велику кількість плюсів в використанні Visual Studio є і мінуси, наприклад неможливість працювати на платформах відмінних від Windows. Остання версія середовища розробки 16.5.3.

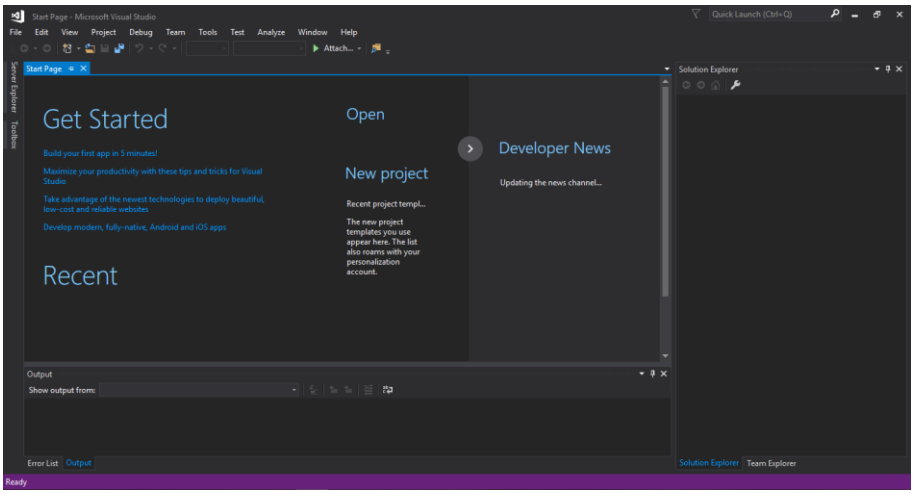


Рисунок 3.1 – Головне вікно додатку «Visual Studio»

Project Rider (рисунок 3.2) – відносно нове середовище розробки, основними перевагами є вбудований плагін ReSharper, кросплатформенність та вбудований контроль версій сумісний з Git, Mercurial і TFS. Також важливим аспектом є підтримка повного циклу, за допомогою якого можна організувати весь цикл розробки ПЗ від проектування до тестування. Також містить багато функціональних переваг, унаслідуваних від платформи IntelliJ і має сумісну підтримку з проектами Visual Studio. До недоліків слід віднести досить велику частину функціональності, яка ще розробляється. Середовище розроблене «JetBrains», остання версія 2019.3.4.

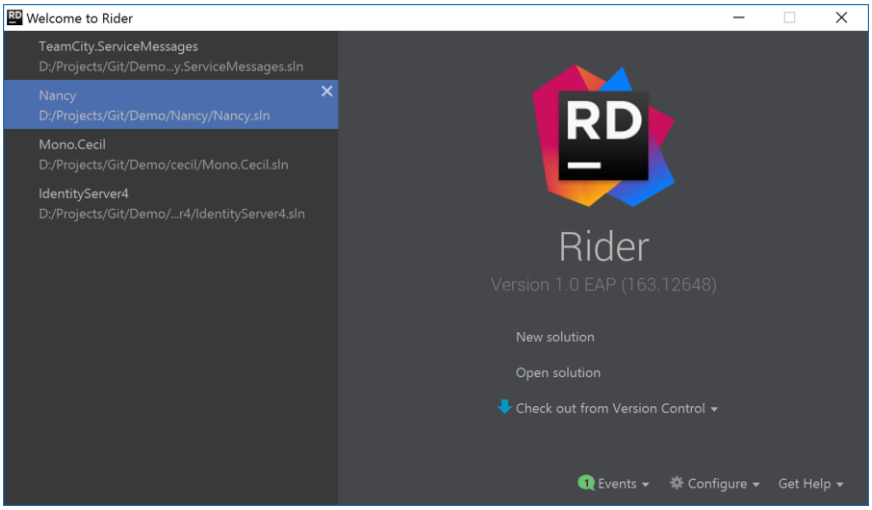


Рисунок 3.2 – Головне вікно додатку «Project Rider»

Eclipse (рисунок 3.3) – одна із найпопулярніших середовищем розробки з підтримкою багатьох мов програмування. Основними перевагами є велика кількість вбудованих плагінів та висока функціональність. До недоліків слід віднести іноді надлишкову функціональність, орієнтацію більше на Java-розробників та ненадійність сторонніх плагінів, яких значно більше ніж офіційних, а через їх велику кількість, часто, витрачає забагато ресурсів пам'яті та збільшується час запуску. Середовище розроблене компанією «Oracle», остання версія 4.11.

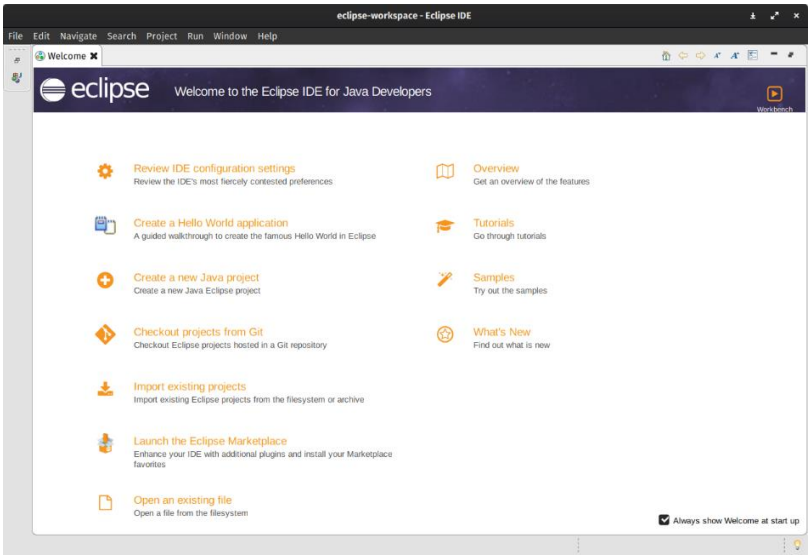


Рисунок 3.3 – Головне вікно додатку «Eclipse»

Результати аналізу приведено у таблиці 3.2.

Таблиця 3.2 – Порівняння середовищ розробки

Параметр	Середовище		
	Eclipse	Project Rider	Visual Studio
Автоформатування коду	-	-	+
Підсвітка функцій	+	-	+
Безкоштовне ліцензування	-	+	-
Карта коду	-	-	+
Статичний аналіз коду	+	+	-

Згідно таблиці 3.2, середовищ розробки «Visual Studio» має перевагу над іншими середовищами розробки, такими як «Project Rider» та «Eclipse». Тому було прийнято рішення обрати середовище Visual Studio для розробки програмного додатку.

3.2 Розробка програмного модуля для створення сценаріїв тестування

У додатку, що розробляється, необхідно забезпечити створення користувацьких сценаріїв тестування. Для цього потрібно розробити можливість користувачу задавати сценарій за допомогою конфігураційного файлу.

Кожна програма, середовище чи система вимагають певного рівня конфігурації. Формати конфігурації, з роками змінюються у популярності. З розвитком потреб у додатках та системі змінюється складність конфігурації та необхідна структура. Сьогодні існує багато форматів конфігурації, усі вони у певних галузях мають більшу популярність. Найпопулярнішими форматами, що підтримують складні конфігурації є наступні [24]:

– XML – це узагальнена мова, яка легко дозволяє реалізувати різні формати зі спільного синтаксису. Основною властивістю є строга структурованість, що надає як рід переваг, наприклад вбудована валідація типів даних, так і ряд недоліків, таких як зайвий синтаксис.

– JSON – це більш проста альтернатива XML. З чистим і простим у використанні синтаксисом, JSON взяв на себе багато налаштувань конфігурації

– YAML – це не мова розмітки (англ. YAML Ain't Markup Language). Основною перевагою YAML є винятковий читабельний людиною синтаксис, оскільки призначена для визначення конфігурацій, а не як мови передачі даних.

– HCL – це поєднання трьох підмов: структурної, виразу та шаблону. Мова розроблялась як легший до читання варіант JSON. Основним недоліком є те, що мова ще значно молода і не має базових бібліотек для підтримки синтаксису мови, що змушує використовувати сторонні бібліотеки.

Оцінивши різні мови, для створення сценаріїв тестування найкраще підходять YAML або HCL. Але оскільки мова HCL ще в активній розробці та не

має вбудованих інтерпретаторів – слід зупинити вибір на мові конфігурації YAML. Оскільки вони є однією з найкращих по розумінню для людини. Основний недолік цієї мови – легко допустити помилку під час створення конфігураційного файлу, отже потрібно передбачити можливість помилки у синтаксисі сценарію, а також розробити детальні інструкцію користувача.

Додаток може використовується у великій різноманітності сценаріїв, які зазвичай включають:

– Запуск спеціальних навантажувальних тестів для окремих API або мікросервісів як частини процесу розробки для вивчення їх характеристик продуктивності та за потреби оптимізації продуктивності (наприклад, запобігання витоку пам’яті або оптимізації коду з великою кількістю процесора).

– Виконання тестів щодо середовищ постановки/функцій як частини конвеєра CI/CD для раннього виявлення регресій продуктивності.

– Масштабні навантажувальні випробування перед виробничими випусками нових сервісів або для підготовки додатків та інфраструктури для інтенсивного трафіку.

– Додавання синтетичного трафіку у виробництво для збереження запасу безпеки від стрибків трафіку

– Запуск синтетичного моніторингу щодо ключових API з кількох географічних місць для перевірки того, що ключові транзакції та потоки працюють належним чином, і попередження, якщо щось зламається.

Отже потрібно надати користувачу можливість конфігурувати різні сценарії. Для цього у конфігураційному файлі передбачений розділ «scenarios». Для симуляції HTTP запиту, розділ може мати такі атрибути:

– url – URL – адреса запиту;

– json – об'єкт JSON для надсилання в тілі запиту;

– body – довільні дані для надсилання в тіло запиту;

– headers – об'єкт JSON, що описує пари ключ – значення заголовка;

– cookie – об'єкт JSON, що описує пари ключ – значення cookie.

На рисунку 3.4 наведено приклад конфігураційного файлу для створення користувацького сценарію.

```yaml
config:
  target: "https://my.app.local"
  phases:
    - duration: 600
      arrivalRate: 10
  variables:
    productIds:
      - ["id1", "id2", "id3"]
      - ["id4", "id5", "id6"]

scenarios:
  - flow:
      - loop:
          - get:
              url: "/products/{{ $loopElement }}"
        over: productIds
```

Рисунок 3.4 – Фрагмент конфігураційного файлу користувацького сценарію

На рисунку 3.5 наведено фрагмент лістингу коду, у якому відбуваєтья запуск навантажувального тестування.

```
async init(_opts) {
  this.state = STATES.initializing;

  this.worker = new Worker(path.join(__dirname, 'worker.js'));
  this.workerId = this.worker.threadId;
  this.worker.on('error', this.onError.bind(this));

  this.worker.on('exit', (exitCode) => {
    this.events.emit('exit', exitCode);
  });
  this.worker.on('messageerror', (err) => {

  });

  await awaitOnEE(this.worker, 'online', 10);
```

Рисунок 3.5 – Фрагмент лістингу коду для запуску сценарію

Спочатку відбувається переведення статусу у статус «ініціалізація», це потрібно для можливої асинхронної роботи програми. Усі можливі статуси наведено на рисунку 3.6. Після чого відбувається асинхронний запуск окремого

потоку. Також для подальшої роботи з даним потоком створюються івенти, які будуть викликані під час помилки (error івент), закриття вікна (exit івент) та помилки під час пересилки повідомлень (messageerror івент). Це необхідно, для коректної підтримки будь яких подій під час роботи додатку.

```
const STATES = {
  initializing: 1,
  online: 2,
  preparing: 3,
  readyWaiting: 4,
  running: 5,
  unknown: 6,
  stoppedError: 7,
  completed: 8,
  stoppedEarly: 9,
  stoppedFailed: 10,
  timedout: 11,
};
```

Рисунок 3.6 – Фрагмент лістингу коду з можливими статусами

Усього, під час роботи, модуль може перейти у наступні статуси:

– initializing – ініціалізація сценарію тестування;

– preparing – підготовка до проведення тестування;

– readyWaiting – модуль готовий до проведення тестування, але очікує на підтвердження;

– running – модуль проводить тестування;

– unknown – перехідний стан від одного до іншого;

– stoppedError – тестування зупинено через помилку;

– completed – тестування успішно завершено;

– stoppedEarly – тестування завершено передчасно за ініціативи користувача;

– stoppedFailed – тестування завершено передчасно за ініціативи сервера, наприклад статус кодом 429 Too Many Requests.

За замовчуванням, після запуску додатку – модуль знаходиться у статусі «online», що свідчить про готовність модуля до запуску сценарію.

3.3 Розробка програмного модуля для автоматичного пошуку вузьких місць у апаратній частині

Високе навантаження на процесор часто є причиною проблем у роботі сервера і, як наслідок, усієї системи загалом. Тому є необхідним пошук вузьких місць у апаратній частині сервера. Одним з інструментів для збору необхідної статистики є сервіс Prometheus.

Prometheus – принципово новий інструмент для моніторингу та збору метрик від розробників SoundCloud, іноді називають системою моніторингу нового покоління [25].

Основною відмінністю від інших подібних систем є те, що даний сервіс працює за pull принципом. Тобто, сервісу, за яким ми спостерігаємо непотрібно кудись надсилати статистику для збереження. За нього це робить спеціальний, окремий компонент, який опитує сервіс та зберігає всю необхідну статистику у окремій базі даних. Після чого, цю інформацію можна візуально подивитись на графіках, наприклад за допомогою додатка Grafana.

Система моніторингу Prometheus складається з наступних компонентів:

– сам сервер, який збирає метрики та зберігає їх у темпоральній базі даних;

– бібліотеки для кількох мов програмування (Go, Java, Python, Ruby, Bash, Node.js, Haskell, .NET/C#);

– Pushgateway – компонент для збору метрик короткочасних процесів (або сервісів, захищених фаєрволлом);

– PromDash – інструмент для створення дашбордів;

– інструменти для експорту даних із Statsd, Ganglia, HAProxy і т.д.;

– AlertManager – менеджер повідомлень;

– prometheus_cli – клієнт командного рядка для виконання запитів до даних.

На рисунку 3.7 наведено схему роботи сервісу Prometheus з офіційного відеокурсу від розробників сервісу [26].

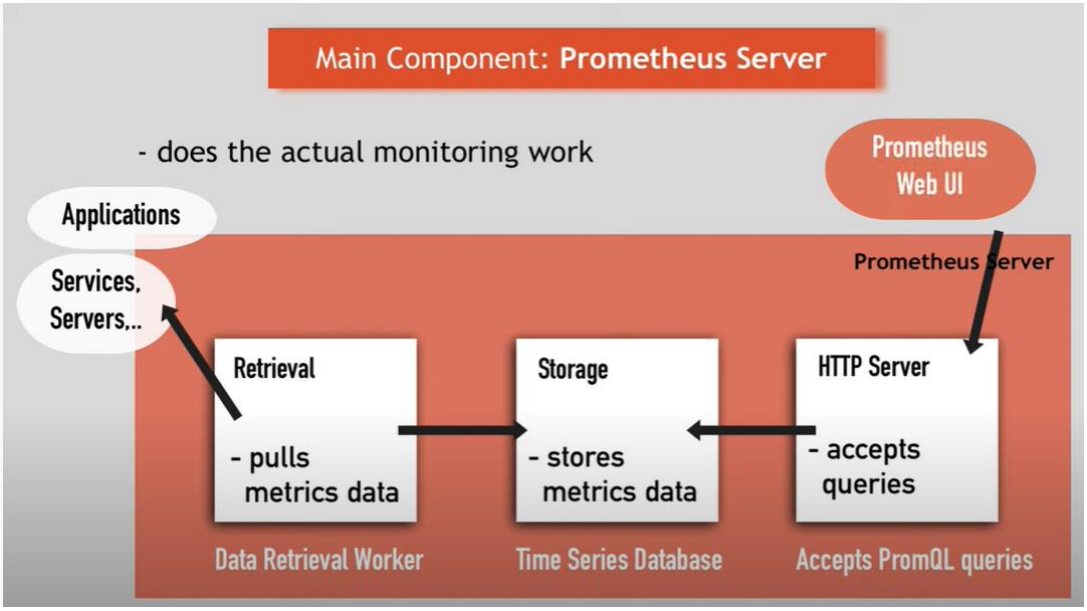


Рисунок 3.7 – Схема роботи Prometheus

На сервері, стан якого необхідно відслідковувати, встановлюється окремий компонент, який відкриває на заданому порту окремий ендпоінт metrics, який надає відповідну інформацію про стан сервера. На рисунку 3.8 наведено приклад відповідної статистики.

```
# HELP app_requests_current Current requests
# TYPE app_requests_current gauge
app_requests_current 33.0
# HELP app_requests_pending Pending requests
# TYPE app_requests_pending gauge
app_requests_pending 7.0
# HELP app_uptime Uptime
# TYPE app_uptime gauge
app_uptime 14383.0
# HELP app_health Health
# TYPE app_health gauge
app_health{app_health="healthy"} 1.0
app_health{app_health="unhealthy"} 0.0
```

Рисунок 3.8 – Фрагмент статистики про стан сервера

Після чого компонент Retrieval опитує даний ендпоінт за заданим інтервалом. Усі зібрані дані надсилаються у компонент Storage, який їх зберігає у відповідн базу даних. Одніє з додаткових переваг є те, що компонент сам видаляє старі записи, що не дає базі займати забагато місця.

Останньою ланкою у даній схемі є HTTP Server Prometheus. Цей компонент зв’язує Storage з іншими компонентами, які працюють з збережиними даними, наприклад графічні середовища такі як Grafana, для побудови графіків. Оскільки модуль що розробляється повинен оцінювати дані про апаратний стан сервера, він буде звертатись саме до цього компонента, для виконання запитів та подальшої оцінки стану.

На рисунку 3.9 зображно фрагмент лістингу коду для оцінки стану сервера.

```
async function loadPlugin(name, config, requirePaths, testScript) {
  let pluginConfigScope = config.scope || testScript.config.pluginsScope;
  let pluginPrefix = pluginConfigScope ? pluginConfigScope : 'artillery-plugin-';
  let requireString = pluginPrefix + name;
  let PluginExport, pluginErr, loadedFrom, version;

  for(const p of requirePaths) {
    debug('Looking for plugin in:', p);
    try {
      loadedFrom = path.join(p, requireString);
      PluginExport = require(loadedFrom);
      if (typeof PluginExport === 'function') {
        version = 1;
      } else if (typeof PluginExport === 'object' && typeof PluginExport.Plugin === 'function') {
        version = 2;
      }
    } catch(err) {
      debug(err);
      pluginErr = err;
    }
```

Рисунок 3.9 – Фрагмент лістингу коду для оцінки стану сервера

Спочатку модуль визначає режим в якому він запущений, після чого з конфігураційного файлу витягуються необхідні дані разом з рядком запиту для доступу до метрик сервера. Наступним етапом є отримання відповідних даних з бази сервера. Список необхідних метрик та запитів для отримання даних з Prometheus зберігається у глобальній конфігурації.

Після отримання даних, модуль проводить оцінку можливості сервера провести зараз стрес-тестування. оскільки можливий варіант, коли сервер вже навантажений напливом реальних клієнтів – проведення стрес-тестування у цей період тільки зашкодить сервісу та кінцевим користувачам. Тому модуль

повинен вернути відповідне попередження, після чого модуль, що відповідає за надсилання HTTP трафіку відмовить у проведенні тестування.

Розроблений модуль працює лише з системою метрик Prometheus, але оскільки була використана мікросервісна архітектура – даний модуль може бути легко замінений іншим з ідентичним інтерфейсом.

3.4 Розробка програмного модуля для оптимізації SQL-запитів

Перед тим як проводити аналіз та подальшу оптимізацію запита, необхідно спочатку провести парсинг цього SQL-запиту. Для можливості повного аналізу разом з підзапитами, найкраще підходить структурний патерн проектування Composite. Згідно цього патерна, створюється деревовидна структура, елементи якої незалежно від типу обробляються однаково. Цей патерн реалізується за допомогою наслідування всіх класів від одного спільного базового абстрактного класу.

Клас TagBase є базовим абстрактним, що наслідується всіма іншими Tag-класами. Цей клас містить в собі поля, що відповідають за тип тегу та саму команду. Для кожної команди у компоненті парсингу розроблено свій клас, який напряму, або через проміжні абстрактні класи наслідує TagBase.

Одним з таких абстрактних класів є клас SimpleOneWordTag, що відповідає за парсинг більшості команд, що складаються з одного слова, таких як SELECT, WHERE або FROM. Для інших команд, що складуються з двох слів, наприклад ORDER BY та GROUP BY, створено інший абстрактний клас SimpleTwoWordTag, що наслідується від SimpleOneWordTag.

Для пошуку співпадінь для кожного Tag-класу було розроблено вкладений MatchTagAttribute-клас. Ці класи відповідають за пошук та точне виокремлення з рядку весь тег разом з всіма параметрами.

Для опрацювання всіх цих класів створено клас SqlParser, який відповідальний за використання всіх класів у правильній послідовності. Серед полів класу присутня колекція базового класу TagBase що складається, з всіх команд, що знайдені в запиті у порядку їх знаходження в запиті. це дозволяє в

подальшому обробляти одночасно як весь запит, так і його частини, а також вставляти додаткові фрагменти запиту у відповідні місця. Оскільки всі SQL-запити структуровані за певним принципом послідовності, для отримання великих фрагментів запиту, таких як блок WHERE, або FROM, розроблено методи для кожної такої команди. На рисунку 3.10 продемонстровано лістинг методу для обробки команди WHERE.

```
/// <summary>
/// Gets or sets the Where clause of the parsed sql.
/// </summary>
public string WhereClause
{
    get
    {
        XmlNode myWhereTagXmlNode = GetWhereTagXmlNode(false);
        if (myWhereTagXmlNode == null)
            return string.Empty;

        StringBuilder myStringBuilder = new StringBuilder();
        XmlNodesToText(myStringBuilder, myWhereTagXmlNode.ChildNodes);
        return myStringBuilder.ToString();
    }
    set
    {
        if (string.IsNullOrEmpty(value))
        {
            XmlNode myWhereXmlNode = GetWhereTagXmlNode(false);
            if (myWhereXmlNode != null)
                myWhereXmlNode.ParentNode.RemoveChild(myWhereXmlNode);
        }
        else
        {
            XmlNode myWhereXmlNode = GetWhereTagXmlNode(true);

            ClearXmlNode(myWhereXmlNode);

            TagBase myWhereTag = TagXmlNodeToTag(myWhereXmlNode);

            ParseBlock(myWhereXmlNode, myWhereTag, value, 0);
        }
    }
}
```

Рисунок 3.10 – Фрагмент лістингу коду для парсингу SQL-запиту

Для роботи програми необхідно провести парсинг фізичного плану виконання та співставити його з запитом мовою SQL. Середовище MS SQL надає для розробників, що користуються SQL CLR відповідну інформацію в форматі XML. XML – запропонований консорціумом World Wide Web Consortium (W3C)

стандарт побудови мов розмітки ієрархічно структурованих даних для обміну між різними застосунками [27].

Оскільки XML є стандартизованою мовою, для оцінки плану виконання потрібно лише дисереалізувати отриманий рядок у форматі XML у об’єкт. Для цієї операції підходять вбудовані класи, що дозволяє виконати цю операцію, не створюючи для цього додаткових класів.

Для отримання даних про структуру БД в Entity Framework передбачений метод SqlConnection.GetSchema(). Цей метод дозволяє отримати всю необхідну інформацію за даним параметром, в випадку «SQL Optimizer», цей параметр «Tables». Що дозволяє витягнути інформацію про назви таблиць, їхні типи та кількість даних. Оскільки після виконання будь якого SQL-запиту, ми отримує колекцію DataRowCollection, необхідно розробити методи розширення для конвертації DataRow в модель DataBase. Лістинг цього метода наведений на рисунку 3.11.

```
/// <summary>
/// Create <see cref="DataBase"/> from <see cref="DataRowCollection"/>
/// </summary>
/// <param name="rows"></param>
/// <returns></returns>
public static DataBase ToDataBase(this DataRowCollection rows)
{
    var database = new DataBase();

    foreach(var row in rows)
    {
        var tableType = ((DataRow)row)["TABLE_TYPE"].ToString();
        if(tableType.ToUpper() != "BASE TABLE")
        {
            continue;
        }

        database.Tables.Add(
            new Table() {
                Name = ((DataRow)row)["TABLE_NAME"].ToString()
            });
    }

    return database;
}
```

Рисунок 3.11 – Фрагмент лістингу коду для отримання даних про БД

З лістингу коду зрозуміло, що опрацьовуються лише таблиці типу «BASE TABLE», тобто всі системні та тимчасові ігноруються. Системні таблиці

виключені з цього списку, оскільки запити до них виконуються у іншому форматі і не потребують оптимізації, оскільки є стандартизованими. Тимчасові таблиці виключені з цього списку разом з табличними змінними, оскільки період життя таких таблиць є обмеженим однією сесію, за якою вони були створені. Оскільки на момент аналізу даних про БД запит користувача не виконувався, цілком вірогідно, що дані тимчасові таблиці створювались стороннім користувачем і їхній період життя є незалежним і може закінчитись в процесі аналізу запиту, що може призвести до помилки виконання оптимізованого запиту в середовищі SQL Server.

Оскільки в SQL Server існує дуже багато різних типів даних, які можуть з легкістю конвертуватись між собою це може призвести до певних проблем під час аналізу запиту, коли порівнюються стовпці з різних таблиць з різними типами даних. Особливо важко буде оцінити конвертацію в плані виконання, що займає 0%, тобто конвертуються, наприклад, два числових типи між собою. Тому для вирішення цієї ситуації створено перелік SqlTypes з групуванням всіх типів по групам. Оскільки врахувати всі типи досить важко через наявність унікальних непопулярних типів, в переліку також додано варіант Other. Для конвертації типів що надає SQL Server в тип переліку створено метод розширення який за допомогою конструкції switch визначає до якої групи віднести даний тип, вкінці цієї конструкції в кейсі default відбувається вибір категорії Other. Фрагмент лістингу даного переліку наведено на рисунку 3.12.

```
public enum SqlTypes
{
    String,
    Date,
    Number,
    Binary,
    Guid,
    Bool,
    Other
}
```

Рисунок 3.12 – Фрагмент лістингу коду переліку SqlTypes

Головною точкою входу в програму вважається UI компонент рішення, що виступає у вигляді Windows Forms проекту. Цей проект складається з головного вікна, та дочірнього модального, яке виконує одразу функції редагування та створення з'єднання з БД. При ініціалізації компонентів головного вікна відбувається зчитування з файлу конфігурацій підключень та створення відповідної кількості кнопок, для обирання з'єднання з БД. Для цих кнопок унікальним ідентифікатором виступає назва з'єднання, тому вона повинна бути унікальним рядком, це також забезпечить точну орієнтацію користувача, коли він спробує створити нове підключення з вже існуючим ідентифікатором.

Для доступу до конфігурацій з будь якого компонента було розроблено окремий статичний клас ConfigurationController в компоненті Common. Оскільки уся робота з конфігураційним файлом відбувається через один екземпляр класу, це дозволяє уникати дедлоків а також тримати в кеші всю необхідну інформацію про з'єднання для зменшення кількості запитів до файлу. На рисунку 3.13 зображено сигнатуру даного класу.

```
public static class ConfigurationController
{
    const string path;
    private static List<DatabaseConfiguration> DatabaseConfigurations;

    /// <summary>
    /// Get connection string from config file
    /// </summary>
    /// <param name="databaseName"></param>
    /// <returns></returns>
    public static string GetConnectionString(string databaseName);

    /// <summary>
    /// Update database name and connection string in file by old database name
    /// </summary>
    /// <param name="databaseName"></param>
    /// <param name="connectionString"></param>
    public static void UpdateConfig(DatabaseConfiguration databaseConfiguration, string key = null);

    /// <summary>
    /// Get list of database name from file
    /// </summary>
    /// <returns></returns>
    public static List<string> GetAllDatabase();

    /// <summary>
    /// Save to file
    /// </summary>
    public static void SaveToFile();

    private static List<DatabaseConfiguration> GetConfig();
}
```

Рисунок 3.13 – Сигнатура інтерфейсу ConfigurationController

Для пошуку оптимального алгоритму потрібно розглянути, що називають критичними запитами. Такі запити можна розділити на дві категорії за частотою та часом виконання:

– Виконуються досить рідко, але їх тривалий час виконання помітно неозброєним поглядом.

– Часто виконуються запити, які самі по собі не займають багато часу, але через те, що виконуються часто, загальний час їх виконання стає критичним.

Для оптимізації запитів при виборці в середовищі MS SQL використовують індекси. Індекс є структурою на диску, яка пов'язана з таблицею і прискорює отримання рядків з таблиці [28]. Індекс містить ключі, побудовані з одного або декількох стовпців в таблиці. Ці ключі зберігаються у вигляді структури збалансованого дерева, яка підтримує швидкий пошук рядків по їх ключовим значенням в SQL Server.

Для індексу існує поняття щільності розподілу. Якщо кожне значення в таблиці унікальне, то щільність буде 1 / число записів. Теорія говорить [29], що чим менша щільність, тим краще – це збільшує вибірковість, а, отже, і цінність побудованого індексу.

Наприклад, якщо колонка містить лише 3 значення, щільність розподілу буде дорівнює 33.3%, що показує марність побудови індексу по даному полю. Індекси займають місце на диску і в оперативній пам'яті і віднімають швидкодію. В ідеалі, найкращий індекс має щільність розподілу дорівнює одиниці, поділеній на кількість записів в таблиці.

При побудові індексів, потрібно звертати увагу на щільність розподілу, якщо вона перевищує 10%, то індекс можна вважати марним. Сканування по таблиці в такому випадку буде більш ефективним. На рисунку 3.14 зображено декілька варіантів створення індексів мовою програмування T-SQL.

В першому випадку формується некластерний індекс по неключовому полі. Такий індекс можна створити для будь якої таблиці або представлення, однак при частих операціях вставлення або видалення він виявиться неефективним.

```
-- Create a nonclustered index on a table or view
CREATE INDEX i1 ON t1 (col1);

-- Create a clustered index on a table and use a 3-part name for the table
CREATE CLUSTERED INDEX i1 ON d1.s1.t1 (col1);

-- Syntax for SQL Server and Azure SQL Database
-- Create a nonclustered index with a unique constraint
-- on 3 columns and specify the sort order for each column
CREATE UNIQUE INDEX i1 ON t1 (col1 DESC, col2 ASC, col3 DESC);
```

Рисунок 3.14 – Приклад створення індексу

У другому випадку створюється кластерний індекс для таблиці. Такий індекс не можна створити для представлення, але можна створити для таблиці на якій базується представлення та отримати всі переваги такого індексу. Кластерний індекс працює значно швидше ніж некластерний, однак його можна створити лише по ключовому полю PRIMARY KEY або FOREIGN KEY. Однак, для зменшення зайвих операцій фрагментації кластерного індексу, дозволяється лише один кластерний індекс на таблицю, це обмеження можна обійти шляхом розділення таблиці на декілька.

У третьому випадку створюється складений некластерний індекс. Оскільки некластерний індекс часто використовується для стовбців з неунікальними значеннями, досить корисно провести сортування одразу по декількох стовпцях в заданому порядку.

Фільтрований індекс – це оптимізований некластерний індекс, особливо підходить для запитів, які здійснюють вибірку з певної підмножини даних [29]. Він використовує предикат фільтра для індексування частини рядків в таблиці. Добре спроектований фільтрований індекс дозволяє підвищити продуктивність запитів, а також знизити витрати на обслуговування і зберігання індексів у порівнянні з повно табличними індексами.

Індекс з неключових стовпцями може значно підвищити продуктивність запиту, коли всі стовпці запиту включені в індекс як ключові або неключових. Продуктивність підвищується завдяки тому, що SQL Server може знайти все

значення стовпців в цьому індексі. При цьому немає доступу до даних таблиць або кластерних індексів, що призводить до меншої кількості дискових операцій вводу-виводу.

Вибір типу з'єднання таблиць в запиті (INNER / LEFT / RIGHT) залежить від завдання. Вважається, що якщо в запиті, який має на увазі однозначне з'єднання однієї таблиці з іншого, немає великої різниці, використовувати INNER або LEFT JOIN. SQL Server генерує для таких запитів однаковий план.

Однак в залежності від наявності індексів та кількості даних в таблицях буде обрано один з фізичних операторів з’єднання: HASH JOIN, MERGE JOIN або NESTED LOOPS.

Оператор NESTED LOOPS фактично працює як вкладений цикл, порівнюючи кожен елемент з лівої таблиці, з кожним елементом з правої таблиці. Найкраще проявляє себе, коли одна з таблиць значно менша за іншу. Як показує практика, вважається оператором, що використовується найчастіше.

Інший оператор – MERGE JOIN. Для цього використання цього оператора є певні обмеження, наприклад, необхідні індекси в обох таблицях по стовпцю, за яким відбувається з’єднання. Вважається найшвидшим оператором з’єднання, оскільки працює однаково швидко як для малих таблиць, так і для великих.

Останнім і найповільнішим оператором з’єднання є HASH JOIN. Всередині цього оператора відбувається підрахунок хеш-сум та створення тимчасової таблиці для збереження цієї хеш таблиці. Вважається, що якщо середовище вирішило використати цей оператор з’єднання, отже запит написаний невірно.

Отже найоптимальніше – використати оператор з’єднання MERGE JOIN, для цього необхідною умовою є наявність індексів в обох таблицях. А також важливою умовою є уникнення оператора HASH JOIN. Для корекції вибору операторів з’єднання було розроблено клас JoinOptimizer, який буде аналізувати можливі оператори з’єднань.

Також цей клас буде збирати інформацію про виконання запитів для всіх таблиць без індексів, якщо в даній таблиці відбувається значно більше вибірок,

ніж операцій зміни даних, таких як UPDATE, DELETE, INSERT користувачу буде запропоновано створити індекс для даної таблиці, відповідно до частоти використаних полів.

На рисунку 3.15 зображено внутрішню деревовидну будову індексів. Оскільки в індексах усі елементи відсортовані, рухаючись по дереву можна досить швидко знайти необхідний елемент.

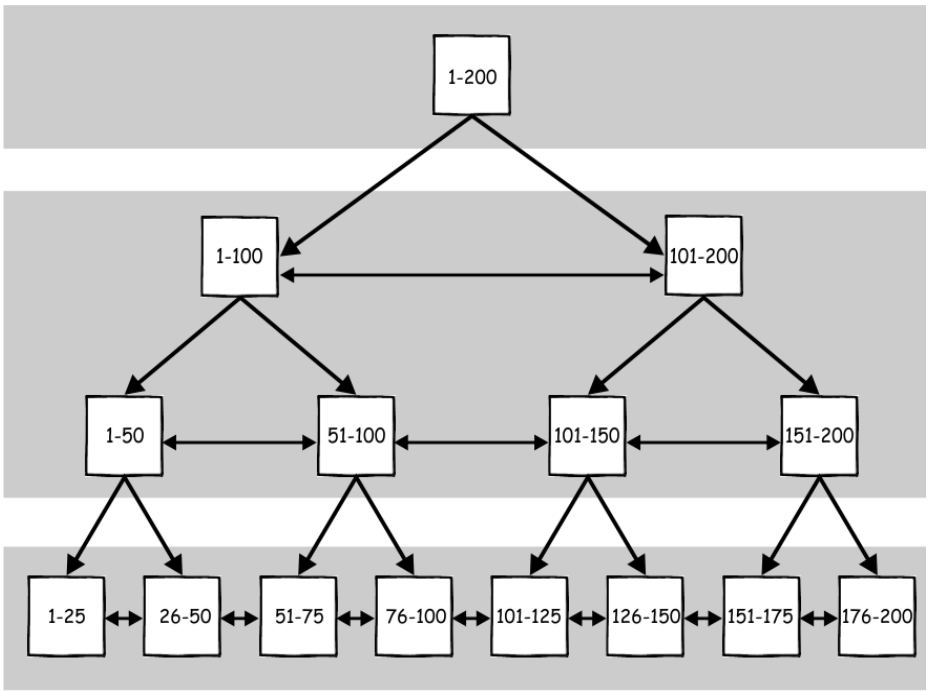


Рисунок 3.15 – Внутрішня структура індексу

Основною відмінністю від звичайної деревовидної структури є наявність зв’язків між сусідніми гілками та листками на одному рівні. За допомогою цих зв’язків, середовище MS SQL може рухатись у будь яку сторону від знайденого індексу, перевіряючи всі наступні.

При використанні команди WHERE з індексами використовуються фізичні оператори INDEX SEEK та INDEX SCAN. Середовище завдяки внутрішній деревовидній будові індексу дозволяє досить швидко віднайти потрібний елемент з всієї множини.

Оператор INDEX SEEK використовується коли прогнозується повернути лише декілька відсотків записів. Оператор INDEX SCAN використовується коли

потрібно повернути значну частину записів з таблиці. При цьому оператори можуть оптимально використовуватись разом, наприклад при використанні операторів порівння >, >=, <, <= спочатку знаходиться початок послідовності, а потім перевіряються усі подальші елементи на інші умови.

Однак якщо при використанні оператора SCAN, буде повернено менше половини значень, отже запит виконаний неефективно. Така поведінка зустрічається при конвертації індексу до іншого типу, або при арифметичних операціях з індексом.

Тому, для запобіганню зайвого сканування всієї таблиці було розроблено алгоритм, який при наявності фізичного оператора SCAN перевіряє можливі варіанти використати оператор SEEK. Перш за все перевіряється чи використаний разом з скануванням, оператор точкового пошуку, якщо він використовується, отже запит вже працює оптимально. Далі перевіряється наявність арифметичних операцій, або конвертації даних, якщо такі є, користувача попереджає про проблемну частину. Якщо проблем не знайдено, запит кешується і після його виконання середовищем, відбувається асинхронний колбек, який в фоновому режимі перевіряє відсоток даних, що повернулись. Якщо повернулось даних значно менше очікуваного, наступного разу, при виконанню такого запита, користувача буде попереджено про не оптимальність, без можливих вирішень задачі.

Одним з найбільш важливих сценаріїв помилок є використання під запиту в операторі IN. Проблема полягає в тому, що такий під запит буде виконуватись на кожній ітерації зовнішнього запиту. Для вирішення такої проблеми розроблено алгоритм по перебудові такого запиту в CTE тимчасову таблицю, що ініціалізується перед запитом, термін життя якої є час виконання одного запиту, перед яким вона ініціалізована. Такій тимчасовій таблиці надається ім’я «sub_{Number}». Далі для використання цієї таблиці її необхідно з’єднати з запитом, для цього використовується оператор з’єднань LEFT JOIN що дозволяє при з’єднані з таблицею не втратити жодного елемента з першої послідовності. А для вибірки потрібних даних у блоці WHERE попередня умова буде замінена на

конструкцію IS [NOT] NULL в залежності від того чи ми відбирали елементи які є в під запиті, чи яких не було у під запиті.

3.5 Висновки

У третьому розділі було проведено варіантний аналіз різноманітних технологій та засобів для розробки системи. У результаті якого було прийнято рішення: для серверної частини використати фреймворк .net core, у якості бази даних використати реляційну БД MS SQL. Розробка проводиться за допомогою мови програмування C#.

Визначено функціонал сервісу для створення сценаріїв тестування та проведено аналіз процесу його розробки.

Визначено функціонал сервісу для проведення стрес-тестування та проведено аналіз процесу його розробки.

Визначено функціонал сервісу для збереження результатів стрес-тестування та проведено аналіз процесу його розробки.

4 ТЕСТУВАННЯ СИСТЕМИ

4.1 Тестування методу аналізу результатів стрес-тестування

Тестування програмного забезпечення – процес дослідження, випробування програмного продукту, який має на меті перевірку відповідності між реальною поведінкою програми і її очікуваним поведінкою на кінцевому наборі тестів, обраних певним чином [30].

За знанням внутрішньої будови системи виділяють такі види тестування:

– тестування білого ящика;
– тестування чорного ящика;
– тестування сірого ящика.

Тестування білого ящика – тип тестування, який враховує внутрішні механізми системи або компонента. Зазвичай виконується розробником відповідної компоненти, або іншим учасником, що приймав активну участь у розробці компоненти, що тестується.

Тестування чорного ящика передбачає відсутність знань про внутрішні механізми системи. Такий тип тестування більше схожий на реальні сценарії користувача, оскільки потенційний користувач не буде знати нічого про внутрішні зв’язки та деталі реалізації. Виконується сторонньою від розробки системи особою, що не приймала жодної участі у процесі написання коду.

Тестування сірого ящика – це поєднання методів чорного та білого ящиків. Передбачається що особа, що здійснює тестування має загальне уявлення про роботу системи, але не знає точних подробиць реалізації. Такий тип тестування виконується, зазвичай, одним з розробників, що приймав участь у розробці системи, за умови що він розробляв лише якусь частину продукту. Також таке тестування може здійснюватись іншою особою, яка ознайомлена з загальними алгоритмами роботи продукту.

За об’єктом тестування можна розділити на:

– функціональне тестування;
– нефункціональне тестування.

Функціональне тестування – це тестування ПО з метою перевірки можливості бути реалізованим функціональних вимог, тобто здатності ПО в певних умовах вирішувати завдання, потрібні користувачам. Функціональні вимоги визначають, що саме робить ПЗ, які завдання воно вирішує.

До нефункціонального тестування відносять інших вимог до ПЗ, таких як продуктивність, безпека, масштабованість, зручність використання, надійність тощо. Для кількісної оцінки результату такого тестування, воно повинне бути виконане вузько направленим спеціалістом, оскільки не в компетенції розробників та осіб, що виконують функціональне тестування оцінювати такі параметри, як, наприклад, зручність у використанні.

За ступенем автоматизації тестуванні поділяють на:

– автоматизоване тестування;
– ручне тестування;
– напівавтоматизоване тестування.

До автоматизованого тестування відносять зазвичай Unit тести, які зазвичай створює один з розробників системи для перевірки різних нюансів продукту.

Ручне тестування виконують тоді, коли автоматизоване тестування створити неможливо, або це займає набагато більше часу ніж ручне тестування, що невигідно для бізнесу. Таке тестування може використовувати методології будь-якого ящика.

До напівавтоматизованого відносять випадки, коли частина тест кейсу виконується автоматизовано, а інша частина в ручному режимі. До такого типу відносять Unit тести, які покривають лише певні модулі додатку, а тестування інтеграції відбувається вручну.

Для тестування обрано метод білої скриньки, що передбачає врахування всіх внутрішніх механізмів. Такий метод найкраще підходить, коли тестування відбувається розробником системи, а не сторонньою людиною, що не вмішувалась в написання коду.

Оскільки було обрано ручне тестування білою скринькою, необхідно буде перевірити всі модулі окремо. Для тестування було розроблено окремий сервіс за допомогою фреймворка ASP.Net Core WebApi з пустою логікою, який було розгорнуто на окремій віртуальні машині. Це необхідно для усунення будь-яких сторонніх чинників, наприклад інших сервісів, що працюють у фоновому режимі, під час проведення функціонального тетсування додатку.

Першим протестуємо модуль валідації запиту. Для цього необхідно створити конфігураційний файл без розділу «scenarios». Очікувана поведінка – програма зупинить виконання з відповідною помилкою. На рисунку 4.1 зображено приклад роботи додатка з невалідними даними.

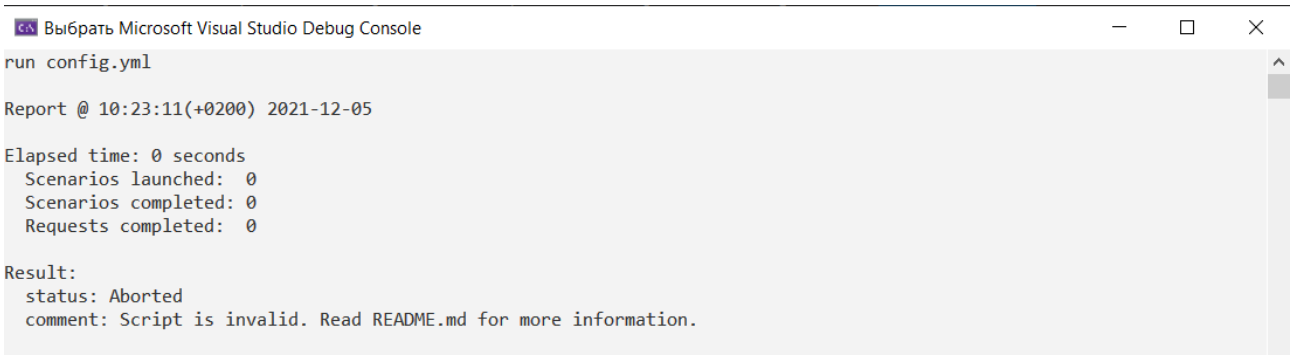


Рисунок 4.1 – Приклад тестування з невалідними даними

Оскільки відбулась помилка і користувач отримав відповідне повідомлення – можна стверджувати, що модуль валідації працює справно.

Для подальшого тестування необхідно створити валідний конфігураційний файл, який буде надсилати на наш веб-сервіс 100 запитів. В подальшому ми не будемо його змінювати, усі маніпуляції повинні відбуватись на стороні сервера, для перевірки, чи визначить наш додаток проблему, та чи запропонує варіанти її вирішення.

Спочатку перевыримо випадок, коли у нас немає жодних проблем з сервісом. Оскільки він розгорнутий на окремій віртуальній машині, та практично не має внутрішньої логіки – можна вважати даний запуск еталонним. На рисунку 4.2 наведено приклад тестування повністю робочого веб-сервіса.

Рисунок 4.2 – Приклад тетсування без відхилень

Згідно з результату, в середньому, запит обробляється 173 мілісекунди. Та всі запити пройшли успішно без жодних аномалій.

Далі нам необхідно перевірити декілька випадків, коли існують помітні проблеми в сервісі або сервері. Для першого варіанту змінимо код сервісу, щоб він запускав в окремому потоці важкообчислювальні операції при кожному запиті. Результат такого тестування наведено на рисунку 4.3.

Рисунок 4.3 – Приклад тестування з зростаючим використання ЦП

В результаті ми отримали очікуваний статус «Failed». Та коментар, що повідомляє, про збільшення використання ЦП під час проведення тестування на 63%. Точність помилки в даному кейсі – 63%. Це зумовлено тим, що перші

запити оброблялись швидко, але в процесі проведення тестування було задіяно забагато ресурсів ЦП, що призвело до значної затримки на наступних запитах.

Наступним ми перевіримо випадок, коли сервер вже навантажений і проведення тестування може тільки зашкодити сервісу. Для цього одразу спробуємо запустити тестування знову, поки сервіс ще обробляє в окремих потоках складні обчислювальні операції. Результат тестування наведено на рисунку 4.4.

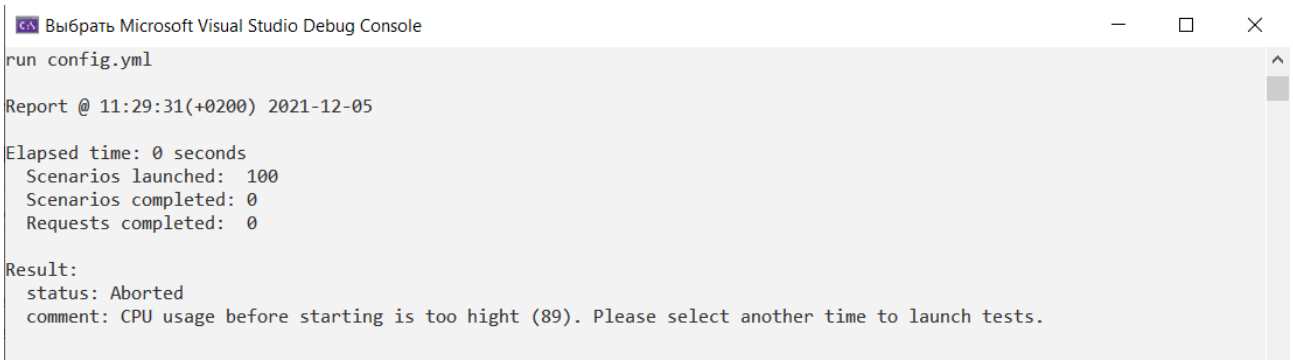


Рисунок 4.4 – Приклад тестування з високим використанням ЦП

Результат відповідає очікуваному – тестування було відмінено, оскільки використання ЦП на сервері вже досягло 89% перед початком тестування.

Наступним слід перевірити можливі обмедення в мережі. На рисунку 4.5 наведено приклад, після налаштування обмежень в черзі на IIS.

Рисунок 4.5 – Приклад тестування з обмеженням у мережі

Згідно результату, 65 запитів було обірвано зі статус кодом 429, що означає, що IIS одразу вернув запитам помилку, як тільки черга запитів заповнилась. У цьому випадку точність 65% відповідає кількості результатів, що отримали у відповіді помилку. У даному сценарії, усі заити виконались швидко. Але через значну кількість помилок компонент, що проводить аналіз результатів позначив запуск як «Failed».

Для перевірки можливих проблем у коді додамо до веб-сервіса затримку у 5 секунд при кожному запиті. Результат виконання такого сценарію наведенно на рисунку 4.6.

Рисунок 4.6 – Приклад тестування з відхиленням у коді

Згідно результату – запуск ввжається з аномалією, оскільки значення часу виконання запитів значно перевищує очікуваних, наведених на рисунку 4.2. Аналіз результатів не зміг виявити проблеми в апаратній частині, оскільки їх не було, тому у коментарі повідомлено, що проблема не розпізнана. Точність вимірювання становить 97%, оскільки 3 запити повернулись з помилкою 408 Request Timeout.

Також необхідно протестувати інші можливі апаратні обмеження. На рисунку 4.7 наведено приклад тестування при високому використанні ОЗУ.

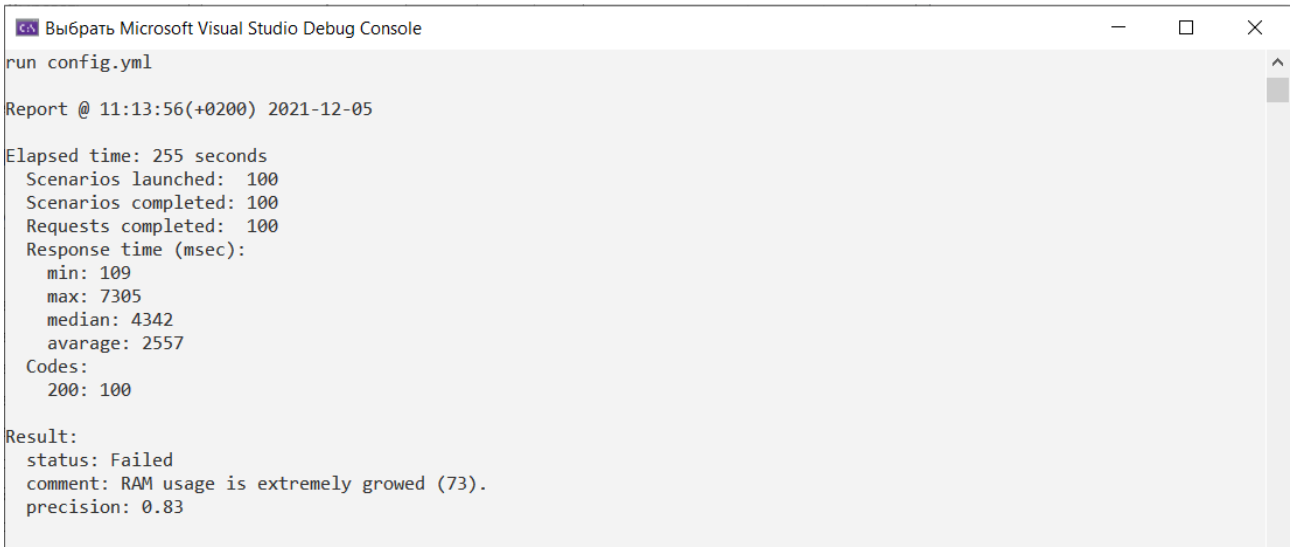


Рисунок 4.7 – Приклад тестування з зростаючим використанням ОЗУ

Згідно коментаря до реультату – під час виконання запитів використання ОЗУ збільшилось до 73%, що спричинило затримки на деяких запитах. Точність резльтату 83%, що становить кількість запитів, що перевищили очікуваний поріг по часу.

Отже, розроблений програмний додаток працює вірно. Тестування програмного продукту показало повну відповідність поставленому технічному завданню.

4.2 Тестування методу оптимізації SQL-запитів

Для тестування даного програмного модуля було обрано ручне тестування білою скринькою. Оскільки даний модуль буде важко покрити юніт тестами та тестування буде проводитись безпосередньо розробником модуля.

Для тестування методу оптимізації SQL-запитів, потрібно виконати наступний сценарій: спершу потрібно протестувати декілька сценаріїв запиту з очікуваним результатом, а потім перевірити степінь збільшення швидкості.

Протестуємо одну з найпоширеніших проблем, коли користувач, замість точного вказання необхідних полів для повернення, використовує конструкцію «SELECT *». На рисунку 4.8 зображено результат аналізу даного запиту в режимі

з підключенням до БД. Очікуваним результатом є заміни «*» на список всіх колонок таблиці.

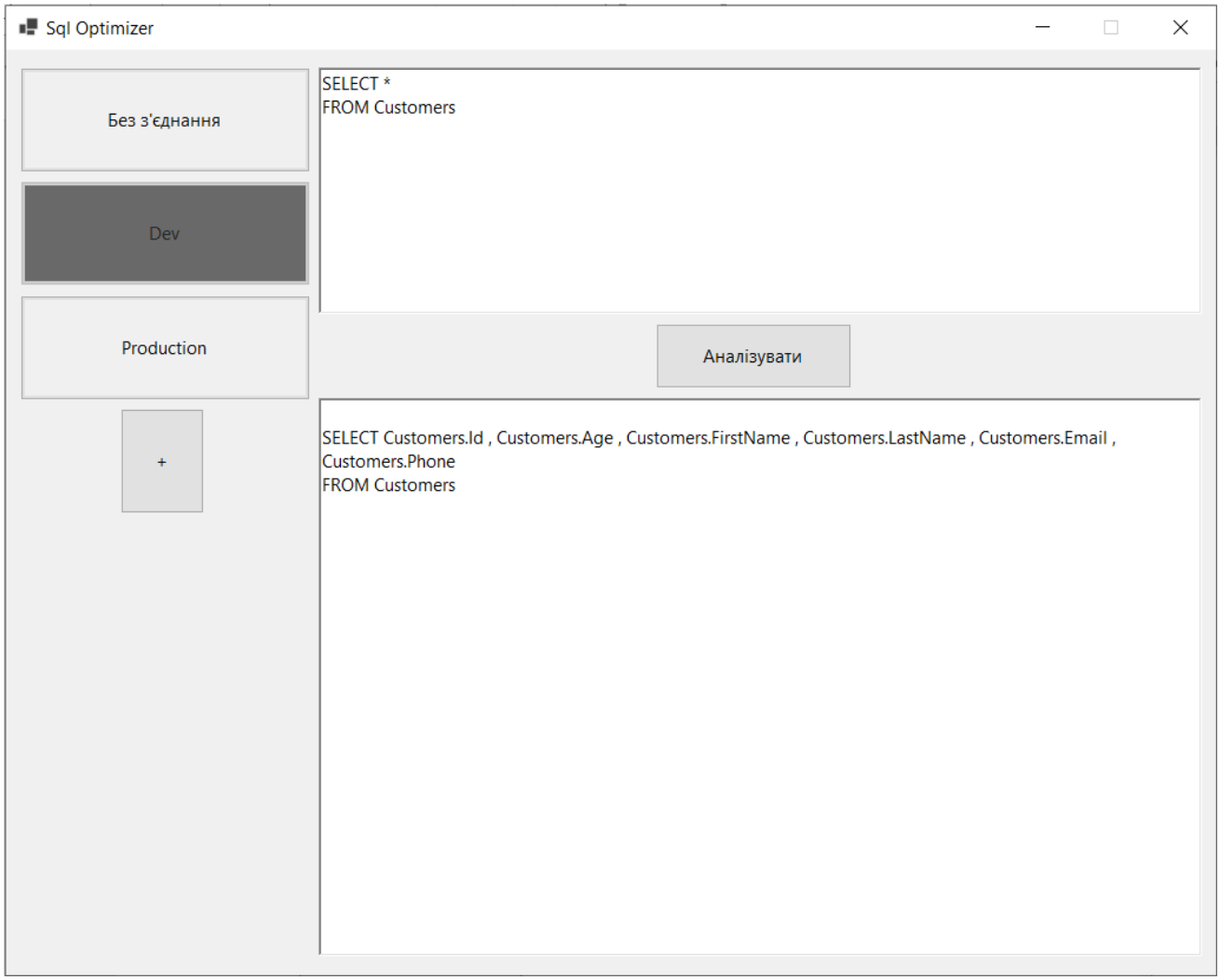


Рисунок 4.8 – Результат виконання аналізу «SELECT *»

Для проблеми конвертації даних спробуємо виконати вибірку по полю з типом UNIQUEIDENTIFIER, та конвертувати його в рядковий тип. Помилка була знайдена та оскільки не може бути вирішена автоматично, позначена червоним кольором як небезпечна ділянка запиту. Як видно з рисунка 4.9, помилка виділена чітко, без зайвих символів, а також знизу знаходиться підказка про суть проблеми з вказанням точних типів, між якими відбувається конвертація.

Якщо на лівій панелі обрати режим роботи без з’єднання до БД, натиснувши кнопку «Without connection», додаток не знаходить жодної аномалії. Це пов’язане з тим, що аномалію конвертації можна знайти лише оцінюючи план виконання.

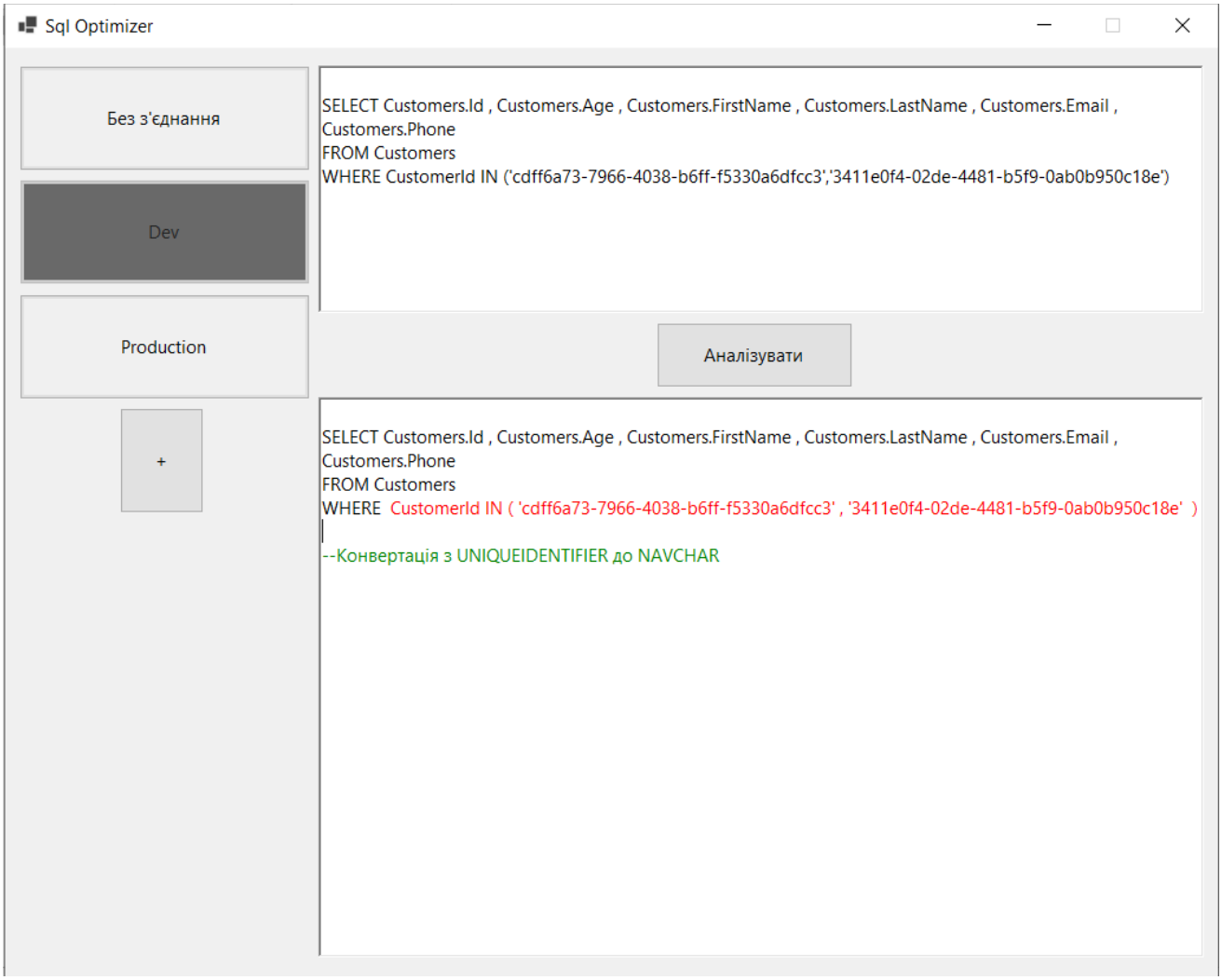


Рисунок 4.9 – Результат виконання для аномалії конвертації

Наступним тестовим випадком є використання табличних підказок. Неможливо наперед правильно оцінити потрібність табличних підказок у кожному запиті. Оскільки за критеріями середовища, план виконання є максимально оптимальним, використання табличних підказок є дуже ризикованим ходом, оскільки може значно пришвидшити виконання, або значно сповільнити час виконання, що трапляється частіше. Тому для оцінки

необхідності використання кожної табличної підказки лежить на користувачі у вигляді запуску одного й того ж запиту з підказками та без них. Оскільки у функціоналі додатку «SQL Optimizer», не передбачено запуск запиту, табличні підказки вважаються аномалією яку потрібно ліквідувати незалежно від режиму роботи додатку. На рисунку 4.10 зображено результат виконання запиту з табличними підказками. Таблична підказка була видалена WITH(SNAPSHOT), та додано відповідний коментар.

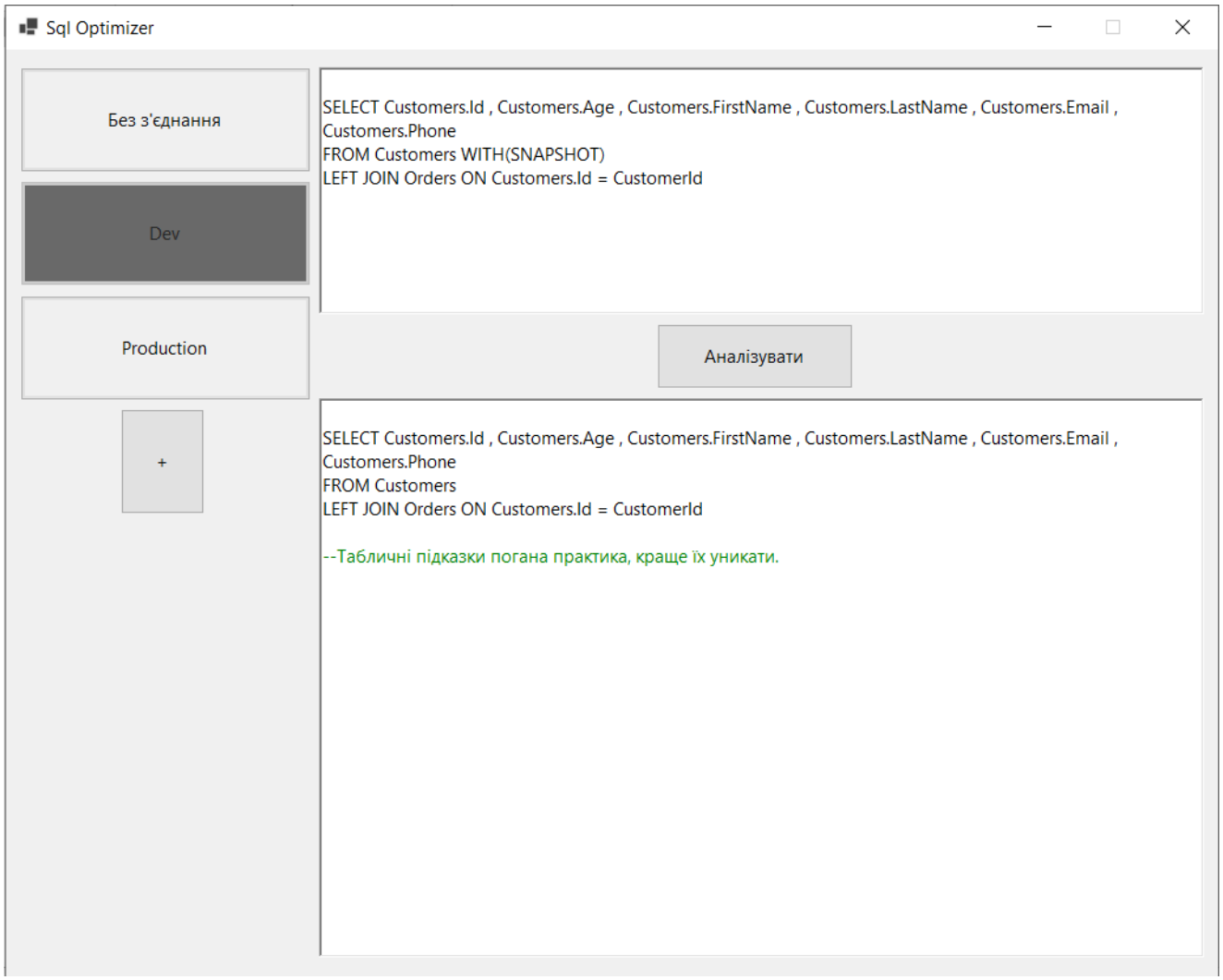


Рисунок 4.10 – Результат виконання для запиту з табличними підказками

Далі протестуємо правильне використання операцій з індексами. Для цього створимо некластерний індекс по полю Phone для таблиці Customers та виконаємо прості арифметичні операції зі змінними, наприклад операції

додавання. На рисунку 4.11 зображено результат оптимізації даного запиту. Згідно арифметичних законів перестановки було отримано той самий результат, але тепер буде виконуватись запит з правильним використанням індексів.

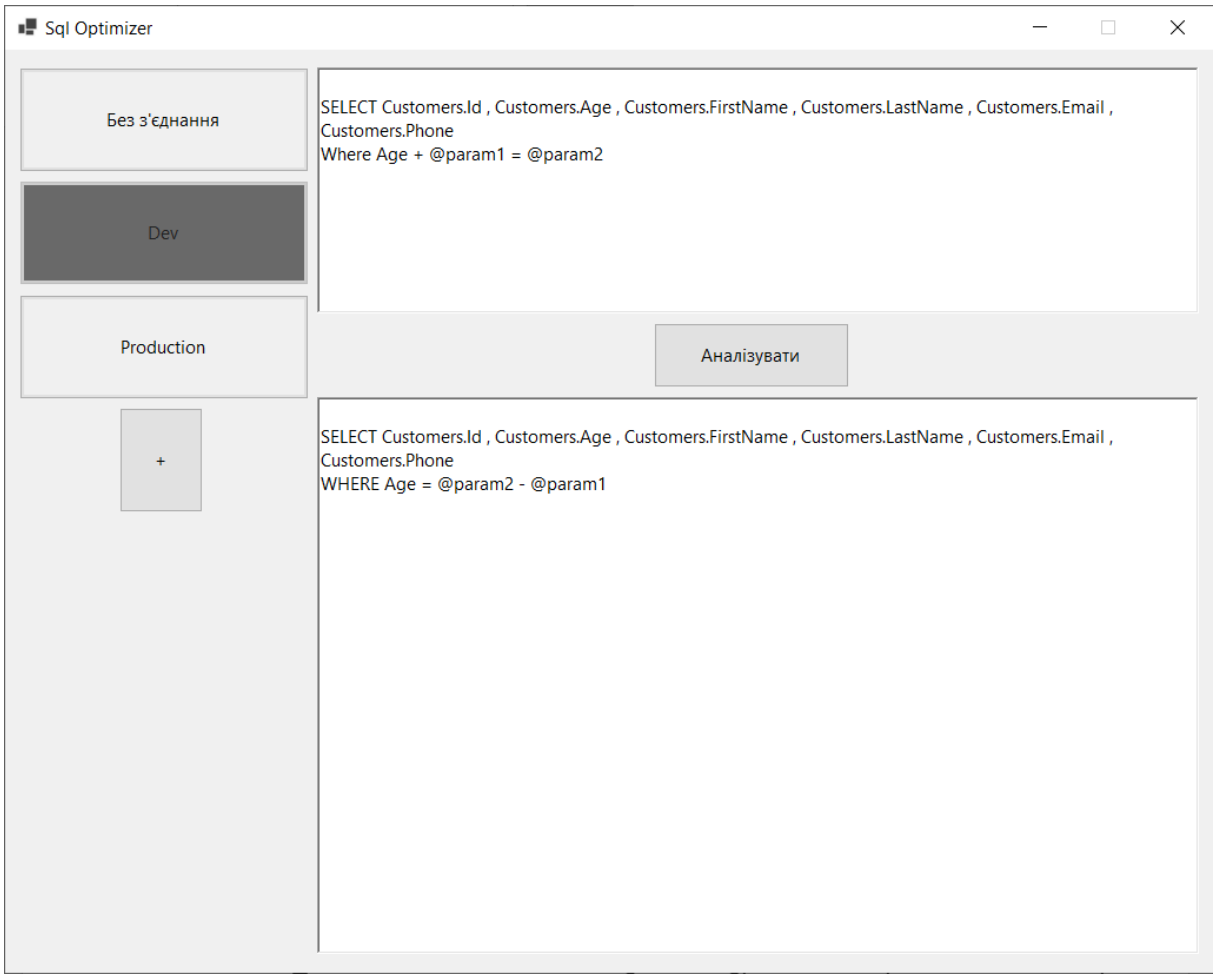


Рисунок 4.11 – Результат оптимізації запиту з арифметичними операціями

Отже можна вважати, що додаток правильно обробляє прості запити, в яких було знайдено не більше 1 помилки. Далі необхідно перевірити пошук помилок у більш складних запитах, де може бути знайдено одразу декілька помилок. Для прикладу можна зробити вибірку по всім покупцям, які здійснювали покупки в певних часових межах. На рисунку 4.12 зображено результат виконання такого запиту. Додаток замінив конструкцію «SELECT *» на перелік відповідних полів, а також створив на початку запиту під запит у форматі CTE, який виконується лише один раз та з’єднав його за допомогою оператора LEFT JOIN. У умовному блоці WHERE на місці попередньою умови

з’явилась інша конструкція IS NOT NULL, що перевіряє чи поле з з’єднання було правильно співставлено.

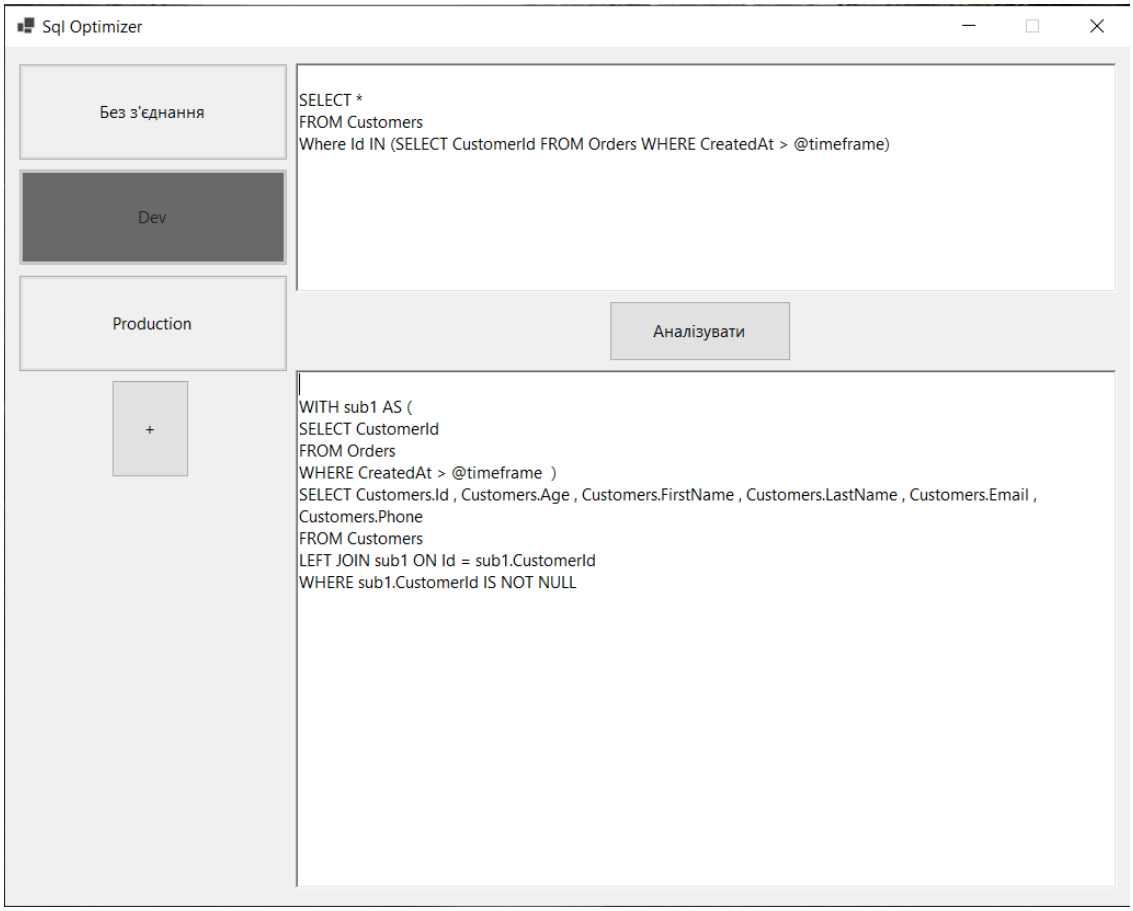


Рисунок 4.12 – Результат оптимізації запиту з під запитом

Одразу видно, що даний запит збільшився в декілька разів, але згідно проведеного дослідження, він повинен виконуватись значно менше часу. Для оцінки збільшення продуктивності необхідно ввімкнути статистику в середовищі MS SQL за допомогою функції «Query Analyzer» та виконати два рази наш запит. Перший раз оптимізований, а другий раз неоптимізований та порівняти їх результат виконання. На рисунку 4.13 зображено порівняння статистики виконання: зверху зображено статистику виконання неоптимізованого запиту, а знизу статистику виконання запиту після обробки додатком «SQL Optimizer». Зі статистики зрозуміло, що оптимізований запит виконує на 2,5 секунди швидше, що свідчить про підвищення продуктивності в конкретному випадку на 20%.

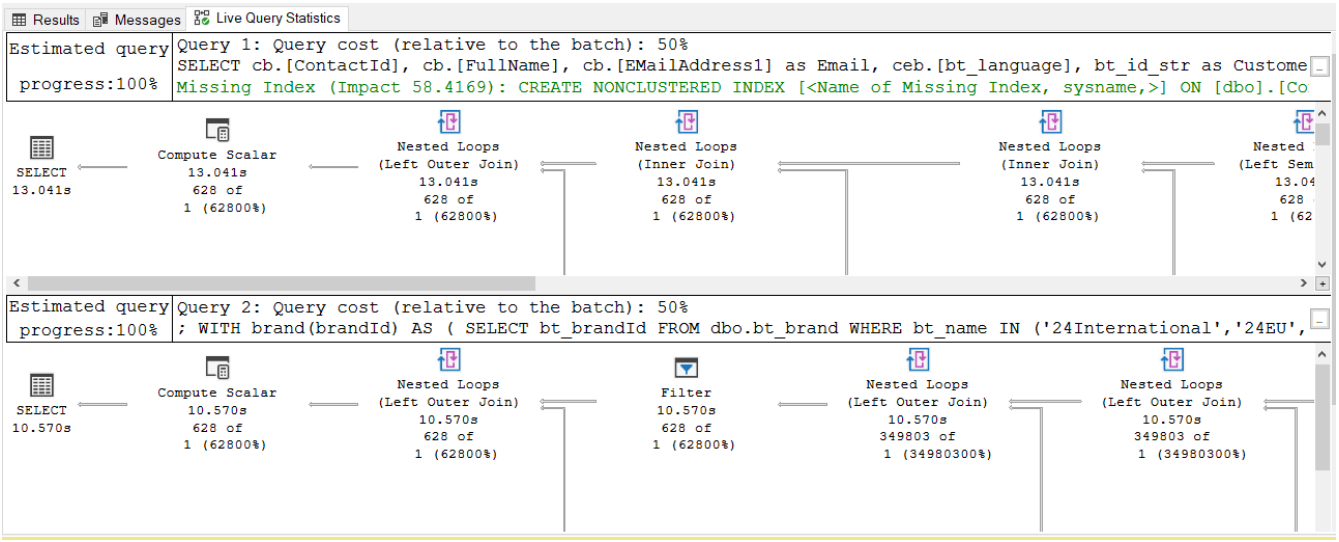


Рисунок 4.13 – Порівняння статистики виконання

Ще одним важливим аспектом, який потрібно протестувати це форматування запиту. Оскільки в T-SQL, є незалежним від регістру букв, а також від зайвих відступів, або їх відсутність. Для перевірки, спробуємо проаналізувати звичайну вибірку по всій таблиці, без умов, але вказавши зайві пропуски після кожного оператора та перевірити правильність сформованого запиту. Результат форматування наведено на рисунку 4.14.

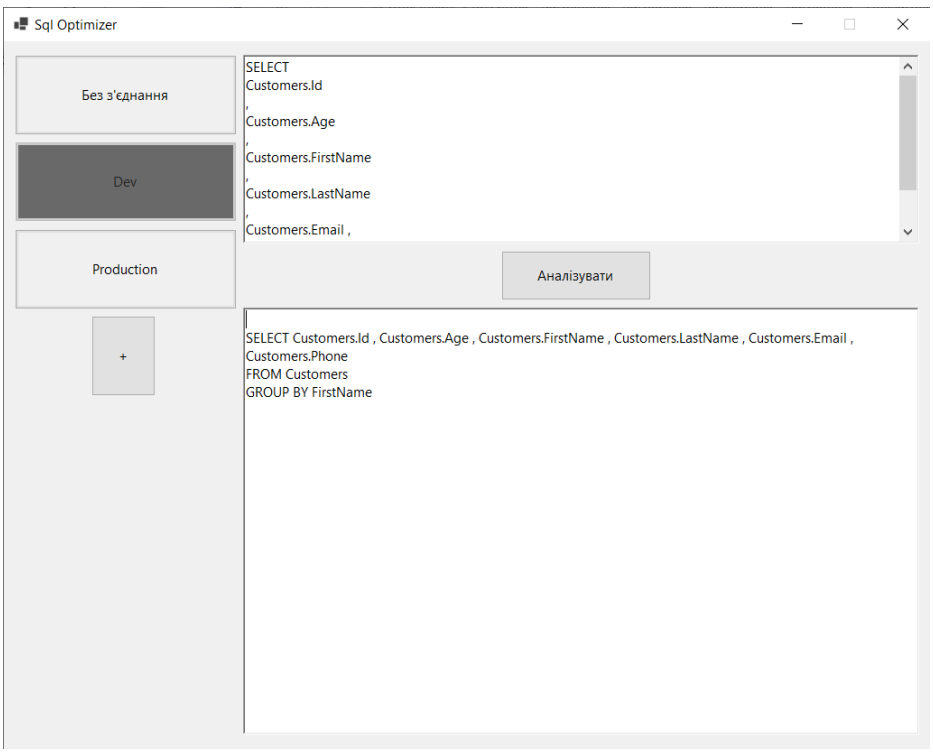


Рисунок 4.14 – Тестування форматування вихідного запиту

Також потрібно протестувати поведінку додатку при синтаксичній помилці виконання запиту. Для цього спробуємо використати неіснуючу таблицю SpecialCustomers та виконаємо вибірку по даній таблиці. На рисунку 4.15 зображено результат виконання при синтаксичній помилці. Додаток вказує помилку отриману від середовища виконання, а також виділяє червоним кольором можливі місця виникнення проблеми.

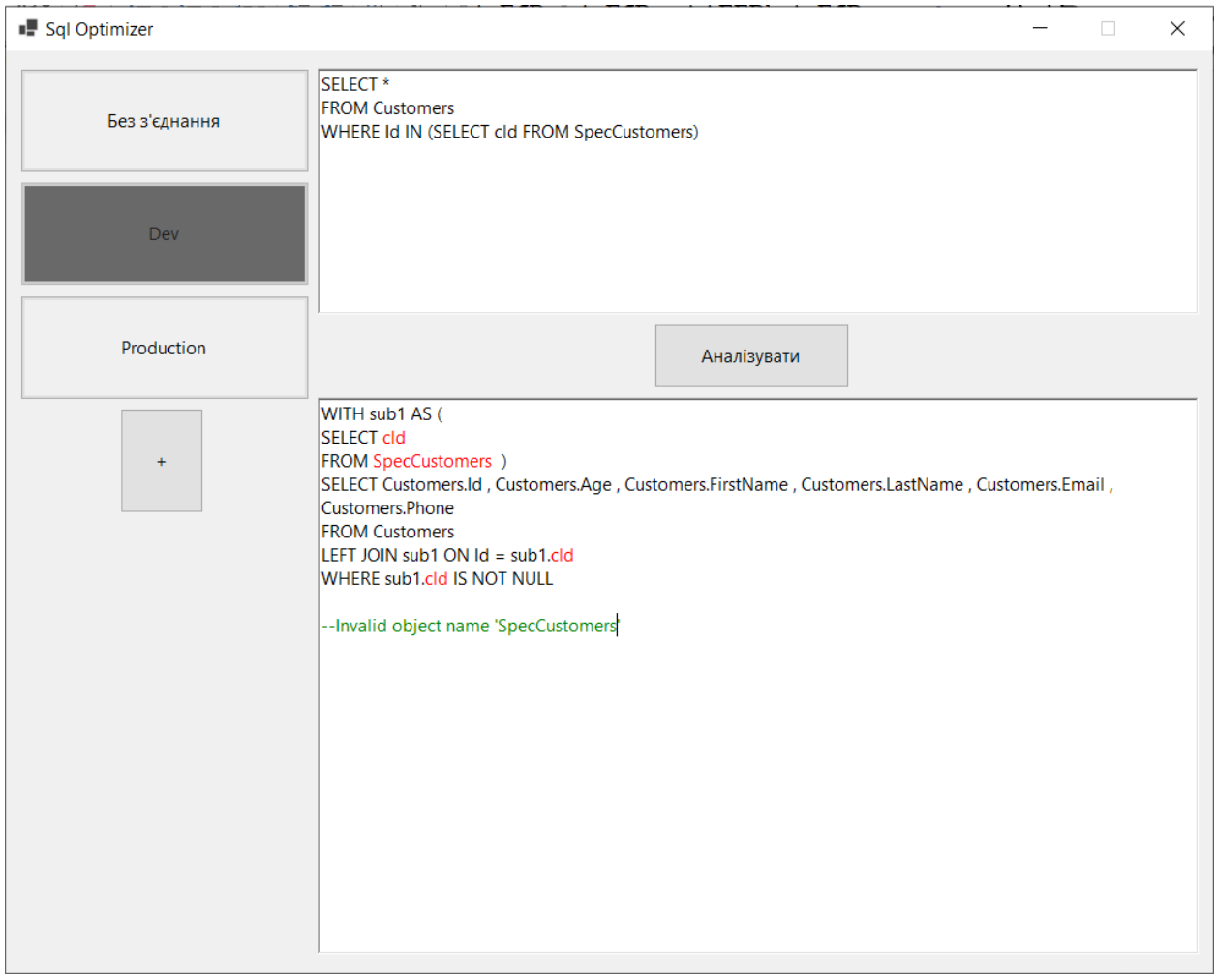


Рисунок 4.15 – Результат роботи додатку з синтаксичною помилкою

Останнім пунктом тестування є перевірка правильності функціонування редагування з'єднань. Для цього спробує додати нове з'єднання з неунікальним ідентифікаторам. В результаті отримуємо повідомлення про помилку зв'язану з не унікальністю даного імені (рисунок 4.16).

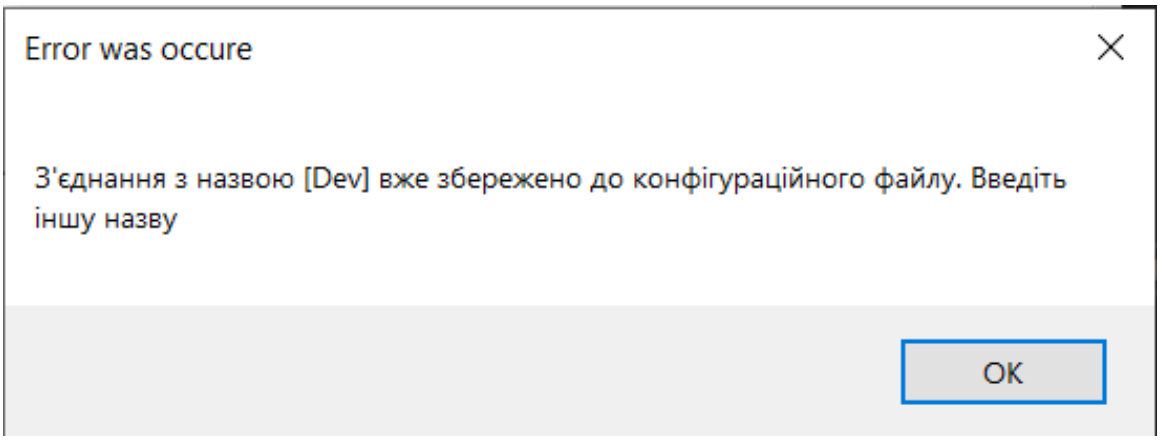


Рисунок 4.16 – Повідомлення про помилку

При доданні з’єднання з унікальним ідентифікатором, операція відбувається успішно, а на панелі зліва, одразу з’являється нова кнопка що відповідає створеному з’єднанню.

При спробі редагувати вже існуюче з’єднання та вказати новий вже існуючий ідентифікатор, відображається повідомлення про помилку, таке саме, як при створенні.

Тепер потрібно перевірити зберігання у файл, яке може відбуватись у фоні у випадку помилки. Для виконання цього сценарію необхідно відкрити вікно для створення нового з’єднання та заповнити всі поля коректними даними. Після чого вручну відкрити конфігураційний файл. Далі необхідно натиснути на кнопку «OK» для підтвердження додання нового з’єднання з БД. Оскільки файл зараз відкритий іншим додатком, «SQL Optimizer» не зможе зберегти у файл зміни, але збереже їх у кеші. Після чого необхідно закрити конфігураційний файл, та відкрити знову через деякий час. У конфігураційному файлі було додано зміни, що свідчить про правильність роботи компоненти ConfigurationController.

Отже, розроблений програмний додаток працює вірно. Тестування програмного продукту показало повну відповідність поставленому технічному завданню.

4.3 Розробка інструкції користувача

Оскільки для розробки додатку було обрано мікросервісну архітектуру – потрібно приділити окрему увагу розгортанню сервісів у інструкції користувача. Отже, для розгортання системи необхідно виконати наступні дії:

1. Запустити файл T_Core.exe та слідувати інструкціям з встановлення, для встановлення основних компонентів додатку.
2. Запустити файл T_Status_Prometheus.exe та слідувати інструкціям з встановлення для встановлення компонента з аналізом метрик Prometheus.
3. Розгорнути Windows Server Monitoring компонент Prometheus згідно офіційної інструкції [31].
4. Відкрити порт обраний для метрик Prometheus для доступу. Для перевірки, потрібно з сервера на якому встановлено T_Status_Prometheus отримати метрики сервісу на якому встановлено Windows Server Monitoring система.

На рисунку 4.17 зображено приклад виконання сценарію тетування. У результаті присутня наступна інформація:

Рисунок 4.17 – Приклад результату виконання.

1. Дата та час проведення тестування.
2. Основна статистика: скільки тестування зайняло часу, скільки успішно виконаних сценаріїв, скільки сценаріїв не було виконано.

3. Час відповіді: основні метрики, такі як мінімум, максиммедіана та середнє арефметичне.

4. Коди відповіді: кількість та типи статус коів, що були отримані в результаті.

5. Результат аналізку: статус, коментар та точність визначення даного результату.

Для створення інструкцій користувача використовувати файл README.md, у якому наведено всі можливі ключові слова для створення сценаріїв. Рекомендується створювати файли-сценарії з розширенням. yml, для уникнення можливих проблем.

4.4 Висновки

Для тестування системи було обрано ручне тестування білої скриньки.

Проведено тестування методу аналізу результатів стрес-тестування, що показало повну працездатність програмного продукту.

Проведено тестування методу оптимізації SQL-запитів, що показало повну працездатність програмного продукту.

Розроблено інструкцію користувача.

5 ЕКОНОМІЧНА ЧАСТИНА

5.1 Оцінювання комерційного потенціалу розробки

Метою проведення комерційного та технологічного аудиту є оцінювання комерційного потенціалу методів і програмних засобів для підвищення продуктивності веб-сервісів під час критичних навантажень на систему.

Для проведення технологічного аудиту було залучено 3-х незалежних експертів Вінницького національного технічного університету з кафедри програмного забезпечення к.т.н., доцента Романюк О.В., к.т.н., доцента Войтко В.В., к.т.н.. доцента Ракитянська Г.Б. Для технологічного аудиту було використано таблицю 5.1 [32] де за п’ятибальною шкалою використовуючи 12 критеріїв оцінено комерційний потенціал.

Таблиця 5.1 – Рекомендовані критерії оцінювання комерційного потенціалу розробки та їх можлива бальна оцінка

Критерії оцінювання та бали (за 5-ти бальною шкалою)					
Кри-терій	0	1	2	3	4
Технічна здійсненність концепції:					
1	Достовірність концепції не підтверджена	Концепція підтверджена експертними висновками	Концепція підтверджена розрахунками	Концепція перевірена на практиці	Перевірено роботоздатність продукту в реальних умовах
Ринкові переваги (недоліки):					
2	Багато аналогів на малому ринку	Мало аналогів на малому ринку	Кілька аналогів на великому ринку	Один аналог на великому ринку	Продукт не має аналогів на великому ринку
3	Ціна продукту значно вища за ціни аналогів	Ціна продукту дещо вища за ціни аналогів	Ціна продукту приблизно дорівнює цінам аналогів	Ціна продукту дещо нижче за ціни аналогів	Ціна продукту значно нижче за ціни аналогів
4	Технічні та споживчі властивості продукту значно гірші, ніж в аналогів	Технічні та споживчі властивості продукту трохи гірші, ніж в аналогів	Технічні та споживчі властивості продукту на рівні аналогів	Технічні та споживчі властивості продукту трохи кращі, ніж в аналогів	Технічні та споживчі властивості продукту значно кращі, ніж в аналогів

Продовження табл. 5.1

5	Експлуатаційні витрати значно вищі, ніж в аналогів	Експлуатаційні витрати дещо вищі, ніж в аналогів	Експлуатаційні витрати на рівні експлуатаційних витрат аналогів	Експлуатаційні витрати трохи нижчі, ніж в аналогів	Експлуатаційні витрати значно нижчі, ніж в аналогів
Ринкові перспективи					
6	Ринок малий і не має позитивної динаміки	Ринок малий, але має позитивну динаміку	Середній ринок з позитивною динамікою	Великий стабільний ринок	Великий ринок з позитивною динамікою
7	Активна конкуренція великих компаній на ринку	Активна конкуренція	Помірна конкуренція	Незначна конкуренція	Конкурентів немає
Практична здійсненність					
8	Відсутні фахівці як з технічної, так і з комерційної реалізації ідеї	Необхідно наймати фахівців або витрачати значні кошти та час на навчання наявних фахівців	Необхідне незначне навчання фахівців та збільшення їх штату	Необхідне незначне навчання фахівців	Є фахівці з питань як з технічної, так і з комерційної реалізації ідеї
9	Потрібні значні фінансові ресурси, які відсутні. Джерела фінансування ідеї відсутні	Потрібні незначні фінансові ресурси. Джерела фінансування відсутні	Потрібні значні фінансові ресурси. Джерела фінансування є	Потрібні незначні фінансові ресурси. Джерела фінансування є	Не потребує додаткового фінансування
10	Необхідна розробка нових матеріалів	Потрібні матеріали, що використовуються у військово-промисловому комплексі	Потрібні дорогі матеріали	Потрібні досяжні та дешеві матеріали	Всі матеріали для реалізації ідеї відомі та давно використовуються у виробництві
11	Термін реалізації ідеї більший за 10 років	Термін реалізації ідеї більший за 5 років. Термін окупності інвестицій більше 10-ти років	Термін реалізації ідеї від 3-х до 5-ти років. Термін окупності інвестицій більше 5-ти років	Термін реалізації ідеї менше 3-х років. Термін окупності інвестицій від 3-х до 5-ти років	Термін реалізації ідеї менше 3-х років. Термін окупності інвестицій менше 3-х років
12	Необхідна розробка регламентних документів та отримання великої кількості дозвільних документів на виробництво та реалізацію продукту	Необхідно отримання великої кількості дозвільних документів на виробництво та реалізацію продукту, що вимагає значних коштів та часу	Процедура отримання дозвільних документів для виробництва та реалізації продукту вимагає незначних коштів та часу	Необхідно тільки пові-домлення відповідним органам про виробництво та реалізацію продукту	Відсутні будь-які регламентні обмеження на виробництво та реалізацію продукту

Таблиця 5.2 – Рівні комерційного потенціалу розробки

Середньоарифметична сума балів СБ, розрахована на основі висновків експертів	Рівень комерційного потенціалу розробки
0-10	Низький
11-20	Нижче середнього
21-30	Середній
31-40	Вище середнього
41-48	Високий

В таблиці 5.3 наведено результати оцінювання експертами комерційного потенціалу розробки.

Таблиця 5.3 – Результати оцінювання комерційного потенціалу розробки

Критерії	Прізвище, ініціали, посада експерта		
	Експерт 1	Експерт 2	Експерт 3
	Бали, виставлені експертами:		
1	4	3	3
2	3	3	2
3	3	3	4
4	3	4	3
5	4	4	4
6	3	3	4
7	3	3	3
8	4	4	4
9	4	4	4
10	4	4	4
11	4	4	3
12	4	4	4
Сума балів	$СБ_1$=43	$СБ_2$=43	$СБ_3$=42
Середньоарифметична сума балів $\overline{СБ}$	$\overline{СБ} = \frac{\sum_1^3 СБ_i}{3} = \frac{43+43+42}{3} = 42{,}6$		

Середньоарифметична сума балів, розрахована на основі висновків експертів склала 43 бали, що згідно таблиці 4.2 вважається, що рівень комерційного потенціалу проведених досліджень є високим.

Програмний засіб для підвищення продуктивності веб-сервісів під час критичних навантажень на систему буде цікава розробникам, яким необхідно розробити стійкий до навантажень веб-сервіс.

Порівняємо нову розробку з аналогами існуючими на ринку. В якості аналога для розробки було обрано LoadNinja. Основними недоліками аналога є неможливість збереження результатів стрес тестування та відсутність автоматичного аналізу результатів. Також до недоліків можна віднести неможливість розгортання у хмарі.

У розробці дана проблема вирішується доданням нового функціоналу, який забезпечує необхідні можливості. Також система випереджає аналог за такими параметрами як час, витрачений в середньому на створення користувацьких сценаріїв тестування.

Проведемо оцінку якості і конкурентоспроможності нової розробки порівняно з аналогом. В таблиці 5.4 наведені основні техніко-економічні показники аналога і нової розробки.

Таблиця 5.4 – Основні параметри нової розробки та товару-конкурента

Показник	Варіанти		Відносний показник якості	Коефіцієнт вагомості параметра
	Базовий (товар-конкурент)	Новий (інноваційне рішення)		
1	2	3	4	5
Використання ОЗУ, ГБ	1	0,8	1,25	25%
Час на розгортання, хв	20	17	1,17	25%
Розмір розгорненої програми, ГБ	1,2	1,0	1,2	10%
Максимальна кількість запитів у секунду	20	30	1,5	20%
Розмір файлу з результатами одного запиту, КБ	5	3	1,7	20%

Як бачимо з таблиці 5.4 у аналога відсутній достатній функціонал. Нова розробка більш продуктивно використовує пам’ять комп’ютера, що критично

при постійному використанні додатку. Також доступна більша кількість запитів у секунду, що збільшує можливість навантаження сервісу, що тестується.

Проведемо оцінку якості продукції, яка є найефективнішим засобом забезпечення вимог споживачів та порівняємо її з аналогом.

Визначимо відносні одиничні показники якості по кожному параметру за формулами (5.1) та (5.2) і занесемо їх у відповідну колонку табл. 5.5.

$$q_i = \frac{P_{Hi}}{P_{Бi}} \tag{5.1}$$

або

$$q_i = \frac{P_{Бi}}{P_{Hi}} \tag{5.2}$$

де P_{Hi}, $P_{Бi}$ – числові значення i-го параметру відповідно нового і базового виробів.

$$q_1 = \frac{1}{0{,}8} = 1{,}25;$$

$$q_2 = \frac{17}{20} = 1{,}17;$$

$$q_3 = \frac{1{,}2}{1} = 1{,}2;$$

$$q_4 = \frac{30}{20} = 1{,}5;$$

$$q_5 = \frac{5}{3} = 1{,}7.$$

Відносний рівень якості нової розробки визначаємо за формулою:

$$К_{я.в.} = \sum_{i=1}^{n} q_i \cdot \alpha_i, \tag{5.3}$$

$$К_{я.в.} = 1{,}25 \cdot 0{,}25 + 1{,}17 \cdot 0{,}25 + 1{,}2 \cdot 0{,}1 + 1{,}5 \cdot 0{,}2 + 1{,}7 \cdot 0{,}2 = 1{,}37.$$

Відносний коефіцієнт показника якості нової розробки більший одиниці, отже нова розробка якісніший базового товару-конкурента.

Наступним кроком є визначення конкурентоспроможності товару. Конкурентоспроможність товару є головною умовою конкурентоспроможності підприємства на ринку і важливою основою прибутковості його діяльності.

Однією із умов вибору товару споживачем є збіг основних ринкових характеристик виробу з умовними характеристиками конкретної потреби покупця. Такими характеристиками найчастіше вважають нормативні та технічні параметри, а також ціну придбання та вартість споживання товару.

В табл. 5.5 наведено технічні та економічні показники для розрахунку конкурентоспроможності нової розробки відносно товару-аналога, технічні дані взяті з попередніх розрахунків.

Таблиця 5.5 – Нормативні, технічні та економічні параметри нової розробки і товару-виробника

Показники	Варіанти	
	Базовий (товар-конкурент)	Новий (інноваційне рішення)
1	2	3
1. Нормативно-технічні показники		
Використання ОЗУ, ГБ	1	0,8
Час на розгортання, хв	20	17
Розмір розгорненої програми, ГБ	1,2	1,0
Максимальна кількість запитів у секунду	20	30
Розмір файлу з результатами одного запиту, КБ	5	3
2. Економічні показники		
Ціна придбання, грн.	450	300

Загальний показник конкурентоспроможності інноваційного рішення (К) з урахуванням вищезазначених груп показників можна визначити за формулою:

$$K = \frac{I_{m.n.}}{I_{e.n.}}, \tag{5.4}$$

де $I_{m.n.}$ – індекс технічних параметрів; $I_{e.n.}$ – індекс економічних параметрів.

Індекс технічних параметрів є відносним рівнем якості інноваційного рішення. Індекс економічних параметрів визначається за формулою (4.5)

$$I_{e.n.} = \frac{\sum_{i=1}^{n} P_{Hei}}{\sum_{i=1}^{n} P_{Бei}}, \tag{5.5}$$

де P_{Hei}, $P_{Бei}$ – економічні параметри (ціна придбання та споживання товару) відповідно нового та базового товарів.

$$\text{I}_{\text{е.п.}} = \frac{300}{450} = 0{,}67;$$

$$\text{К} = \frac{1{,}37}{0{,}67} = 2{,}04.$$

Зважаючи на розрахунки, можна зробити висновок, що нова розробка буде конкурентоспроможніше, ніж конкурентний товар.

5.2 Прогнозування витрат на виконання науково-дослідної роботи

Витрати, пов’язані з проведенням науково-дослідної роботи групуються за такими статтями: витрати на оплату праці, витрати на соціальні заходи, матеріали, паливо та енергія для науково-виробничих цілей, витрати на службові відрядження, програмне забезпечення для наукових робіт, інші витрати, накладні витрати.

1. Основна заробітна плата кожного із дослідників $З_О$, якщо вони працюють в наукових установах бюджетної сфери визначається за формулою:

$$\text{З}_\text{О} = \frac{\text{М}}{\text{Т}_\text{р}} * t \text{ (грн)}, \tag{5.6}$$

де M – місячний посадовий оклад конкретного розробника (інженера, дослідника, науковця тощо), грн.;

T_P – число робочих днів в місяці; приблизно $T_P \approx 21...23$ дні;

t – число робочих днів роботи дослідника.

Для розробки програмних засобів для підвищення продуктивності веб-сервісів під час критичних навантажень на систему необхідно залучити програміста з посадовим окладом 9000 грн. Кількість робочих днів у місяці складає 22, а кількість робочих днів програміста складає 20. Зведемо сумарні розрахунки до таблиця 5.6.

Таблиця 5.6 – Заробітна плата дослідника в науковій установі бюджетної сфери

Найменування посади	Місячний посадовий оклад, грн.	Оплата за робочий день, грн.	Число днів роботи	Витрати на заробітну плату, грн.
Керівник	15000	681,8	5	3409
Програміст	9000	409,1	22	9000
Всього				12409

2. Розрахунок додаткової заробітної плати робітників

Додаткова заробітна плата $З_д$ всіх розробників та робітників, які приймали устають в розробці нового технічного рішення розраховується як 10 - 12 % від основної заробітної плати робітників.

На даному підприємстві додаткова заробітна плата начисляється в розмірі 10% від основної заробітної плати.

$$З_д = (З_о + З_р) * \frac{Н_{дод}}{100\%}, \quad (5.7)$$

$$З_д = 0{,}11 * 12409 = 1365 \text{ (грн)}.$$

3. Нарахування на заробітну плату $Н_{ЗП}$ дослідників та робітників, які брали участь у виконанні даного етапу роботи, розраховуються за формулою (5.3):

$$Н_{ЗП} = (З_о + З_д) * \frac{\beta}{100} \text{ (грн)}, \quad (5.8)$$

де $З_о$ – основна заробітна плата розробників, грн.;

$З_д$ – додаткова заробітна плата всіх розробників та робітників, грн.;

β – ставка єдиного внеску на загальнообов'язкове державне соціальне страхування, % .

Дана діяльність відноситься до бюджетної сфери, тому ставка єдиного внеску на загальнообов'язкове державне соціальне страхування буде складати 22%, тоді**:**

$$Н_{ЗП} = (12409 + 1365) * \frac{22}{100} = 3030{,}3 \text{ (грн)}.$$

4. Витрати на матеріали М та комлектуючі К, що були використані під час виконання даного етапу роботи, розраховуються по кожному виду матеріалів за формулою:

$$М = \sum_{1}^{n} Н_i \cdot Ц_i \cdot К_i - \sum_{1}^{n} В_i \cdot Ц_в \text{ грн.}, \quad (5.9)$$

де $Н_i$ – витрати матеріалу *і*-го найменування, кг;

$Ц_i$ – вартість матеріалу *і*-го найменування, грн./кг.;

$К_i$ – коефіцієнт транспортних витрат, $К_i$ = (1,1…1,15);

$В_i$ – маса відходів матеріалу *і*-го найменування, кг;

$Ц_в$ – ціна відходів матеріалу *і*-го найменування, грн/кг;

n – кількість видів матеріалів.

Інформацію про використані матеріали подамо у вигляді табл. 5.7.

Таблиця 5.7 – Матеріали, що використані на розробку

Найменування матеріалу	Ціна за одиницю, грн.	Витрачено	Вартість витраченого матеріалу, грн.
Папір	140	1	140
Ручка	30	1	30
CD-диск	10	1	10
Флешка	145	1	145
Всього			325
З врахуванням коефіцієнта транспортування			357,5

5. Програмне забезпечення для наукової роботи включає витрати на розробку та придбання спеціальних програмних засобів і програмного забезпечення необхідного для проведення дослідження.

Балансову вартість програмного забезпечення розраховують за формулою:

$$В_{прг} = \sum_{i=1}^{k} Ц_{іпрг} \cdot С_{пргі} \cdot К_і, \quad (5.10)$$

де $Ц_{іпрг}$ – ціна придбання одиниці програмного засобу цього виду, грн;

$С_{пргі}$ – кількість одиниць програмного забезпечення відповідного найменування, які придбані для проведення досліджень, шт.;

$К_і$ – коефіцієнт, що враховує інсталяцію, налагодження програмного засобу тощо ($К_і$ = 1,10…1,12).

k – кількість найменувань програмних засобів.

Отримані результати занесемо в таблицю 5.8.

Таблиця 5.8 – Витрати на придбання програмних засобів по кожному виду

Найменування устаткування	Кількість, шт	Ціна за одиницю, грн	Вартість
Microsoft Visual Studio Enterprise 2019	1	850	850
Всього			850

6. Амортизація обладнання, комп'ютерів та приміщень, які використовувались під час виконання даного етапу роботи

Дані відрахування розраховують по кожному виду обладнання, приміщенням тощо.

$$A = \frac{Ц \cdot T}{Ткор \cdot 12} \quad [грн], \tag{5.11}$$

де Ц – балансова вартість даного виду обладнання (приміщень), грн.;

$Т_{кор}$ – час користування;

Т – термін використання обладнання (приміщень), цілі місяці.

Згідно пункта 137.3.3 Податкового кодекса амортизація нараховується на основні засоби вартістю понад 2500 грн. В нашому випадку для написання магістерської роботи використовувався персональний комп'ютер вартістю 15000 грн.

$$А = \frac{15000 \cdot 1}{2 \cdot 12} = 625.$$

7. До статті «Паливо та енергія для науково-виробничих цілей» відносяться витрати на всі види палива й енергії, що безпосередньо використовуються з технологічною метою на проведення досліджень.

$$В_е = \sum_{i=1}^{n} \frac{W_{yt} \cdot t_i \cdot Ц_е \cdot К_{впі}}{\eta_i}, \tag{5.12}$$

де W_{yt} – встановлена потужність обладнання на певному етапі розробки, кВт;

t_i – тривалість роботи обладнання на етапі дослідження, год;

$Ц_е$ – вартість 1 кВт-години електроенергії, грн;

$К_{впі}$ – коефіцієнт, що враховує використання потужності, $К_{впі} < 1$;

η_i – коефіцієнт корисної дії обладнання, $\eta_i < 1$.

Для написання магістерської роботи використовується персональний комп'ютер для якого розрахуємо витрати на електроенергію.

$$В_е = \frac{0{,}3 \cdot 175 \cdot 4{,}1 \cdot 0{,}5}{0{,}8} = 134{,}53\ .$$

Накладні (загальновиробничі) витрати Внзв охоплюють: витрати на управління організацією, оплата службових відряджень, витрати на утримання, ремонт та експлуатацію основних засобів, витрати на опалення, освітлення, водопостачання, охорону праці тощо. Накладні (загальновиробничі) витрати Внзв можна прийняти як (100…150)% від суми основної заробітної плати розробників та робітників, які виконували дану МКНР, тобто:

$$В_{нзв} = (З_о + З_р) \cdot \frac{Н_{нзв}}{100\%}, \quad (5.13)$$

де $Н_{нзв}$ – норма нарахування за статтею «Інші витрати».

$$В_{нзв} = 12409 \cdot \frac{100}{100\%} = 12409 \text{ грн.}$$

Сума всіх попередніх статей витрат дає витрати, які безпосередньо стосуються даного розділу МКНР

В = 12409+1365+3030,3+357,5+850+625+135,53+12409=31180,5 грн.

Прогнозування загальних втрат ЗВ на виконання та впровадження результатів виконаної МКНР здійснюється за формулою:

$$ЗВ = \frac{В}{\eta}, \quad (5.14)$$

де η – коефіцієнт, який характеризує стадію виконання даної НДР.

Оскільки, робота знаходиться на стадії науково-дослідних робіт, то коефіцієнт β = 0,9. Звідси:

$$ЗВ = \frac{31180,5}{0,9} = 34645,01 \text{ грн.}$$

5.3 Розрахунок економічної ефективності науково-технічної розробки

У даному підрозділі кількісно спрогнозуємо, яку вигоду, зиск можна отримати у майбутньому від впровадження результатів виконаної наукової

роботи. Розрахуємо збільшення чистого прибутку підприємства $\Delta П_i$, для кожного із років, протягом яких очікується отримання позитивних результатів від впровадження розробки, за формулою

$$\Delta П_i = \sum_{1}^{n} \left(\Delta Ц_o \cdot N + Ц_o \cdot \Delta N \right)_i \cdot \lambda \cdot \rho \cdot \left(1 - \frac{\upsilon}{100} \right), \quad (5.15)$$

де $\Delta Ц_0$ – покращення основного оціночного показника від впровадження результатів розробки у даному році.

N – основний кількісний показник, який визначає діяльність підприємства у даному році до впровадження результатів наукової розробки;

ΔN – покращення основного кількісного показника діяльності підприємства від впровадження результатів розробки:

Цо – основний оціночний показник, який визначає діяльність підприємства у даному році після впровадження результатів наукової розробки;

n – кількість років, протягом яких очікується отримання позитивних результатів від впровадження розробки:

л – коефіцієнт, який враховує сплату податку на додану вартість. Ставка податку на додану вартість дорівнює 20%, а коефіцієнт *л* = 0,8333.

р – коефіцієнт, який враховує рентабельність продукту. р = 0,25;

х – ставка податку на прибуток. У 2021 році – 18%.

Припустимо, що при впроваджені результатів наукової розробки покращується якість, що дозволяє підвищити ціну його реалізації на 80 грн. Кількість одиниць реалізованої продукції також збільшиться: протягом першого року на 900 шт., протягом другого року – на 1000 шт., протягом третього року на 1200 шт. Реалізація продукції до впровадження розробки складала 1 шт., а її ціна 300 грн. Розрахуємо прибуток, яке отримає підприємство протягом трьох років.

$$\Delta П_1 = [80 \cdot 1 + (300 + 80) \cdot 900] \cdot 0{,}833 \cdot 0{,}25 \cdot \left(1 + \frac{18}{100}\right) = 58436{,}33 \text{ грн.}$$

$$\Delta П_2 = [80 \cdot 1 + (300 + 80) \cdot (900 + 1000)] \cdot 0{,}833 \cdot 0{,}25 \cdot \left(1 + \frac{18}{100}\right)$$
$$= 123416{,}73 \text{ грн.}$$

$$\Delta П_3 = [80 \cdot 1 + (300 + 80) \cdot (900 + 1000 + 1200)] \cdot 0{,}833 \cdot 0{,}25 \cdot \left(1 + \frac{18}{100}\right)$$
$$= 201313{,}65 \text{ грн.}$$

5.4 Розрахунок ефективності вкладених інвестицій та періоду їх окупності

Розрахуємо основні показники, які визначають доцільність фінансування наукової розробки певним інвестором, є абсолютна і відносна ефективність вкладених інвестицій та термін їх окупності.

Розрахуємо величину початкових інвестицій PV, які потенційний інвестор має вкласти для впровадження і комерціалізації науково-технічної розробки.

$$PV = k_{\text{інв}} \cdot \text{ЗВ}, \qquad (5.16)$$

$k_{\text{інв}}$ – коефіцієнт, що враховує витрати інвестора на впровадження науково-технічної розробки та її комерціалізацію. Це можуть бути витрати на підготовку приміщень, розробку технологій, навчання персоналу, маркетингові заходи тощо ($k_{\text{інв}} = 2…5$).

$$PV = 2 \cdot 34645{,}01 = 69290{,}03.$$

Розрахуємо абсолютну ефективність вкладених інвестицій $E_{абс}$ згідно наступної формули:

$$E_{абс} = (ПП - PV), \qquad (5.17)$$

де ПП – приведена вартість всіх чистих прибутків, що їх отримає підприємство від реалізації результатів наукової розробки, грн.;

$$ПП = \sum_{1}^{T} \frac{\Delta П_i}{(1+\tau)^t}, \qquad (5.18)$$

де $\Delta П_i$ – збільшення чистого прибутку у кожному із років, протягом яких виявляються результати виконаної та впровадженої НДЦКР, грн.;

Т – період часу, протягом якою виявляються результати впровадженої НДДКР, роки;

τ – ставка дисконтування, за яку можна взяти щорічний прогнозований рівень інфляції в країні; для України цей показник знаходиться на рівні 0,2; t – період часу (в роках).

$$ПП = \frac{58436,33}{(1+0,2)^1} + \frac{123416,73}{(1+0,2)^2} + \frac{201313,62}{(1+0,2)^3} = 251445,8 \text{ грн.}$$

$$Е_{абс} = (251445,8 - 69290,03\ ) = 182155,78 \text{грн.}$$

Оскільки $E_{абс} > 0$ то вкладання коштів на виконання та впровадження результатів НДДКР може бути доцільним.

Розрахуємо відносну (щорічну) ефективність вкладених в наукову розробку інвестицій $E_в$. Для цього користуються формулою:

$$E_в = \sqrt[T_ж]{1 + \frac{E_{абс}}{PV}} - 1, \qquad (5.19)$$

$T_ж$ – життєвий цикл наукової розробки, роки.

$$Е_в = \sqrt[3]{1 + \frac{182155,78}{69290,03}} - 1 = 0,84 = 84\%.$$

Визначимо мінімальну ставку дисконтування, яка у загальному вигляді визначається за формулою:

$$\tau = d + f, \qquad (5.20)$$

де d – середньозважена ставка за депозитними операціями в комерційних банках; в 2018 році в Україні d= (0,14…0,2);

f – показник, що характеризує ризикованість вкладень; зазвичай, величина f = (0,05...0,1).

$$\tau_{\min} = 0,18 + 0,05 = 0,23.$$

Так як $E_в > \tau_{\min}$ то інвестор може бути зацікавлений у фінансуванні даної наукової розробки.

Розрахуємо термін окупності вкладених у реалізацію наукового проекту інвестицій за формулою:

$$T_{ок} = \frac{1}{E_в} \tag{5.21}$$

$$\mathrm{T_{ок}} = \frac{1}{0,84} = 1,2 \text{ роки.}$$

Так як $T_{ок} \leq$ 3...5-ти років, то фінансування даної наукової розробки в принципі є доцільним.

5.5 Висновки

Було проведено оцінку комерційного потенціалу розробки методів і програмних засобів для підвищення продуктивності веб-сервісів під час критичних навантажень на систему, яка є на високому рівні. При порівнянні нової розробки з аналогом виявлено, що вона є якіснішою і конкурентоспроможнішою відносно аналога, а також краще по технічним і економічним показникам.

Прогнозування витрат на виконання науково-дослідної роботи по кожній з статей витрат складе 31180,5 грн. Загальна ж величина витрат на виконання та впровадження результатів даної НДР буде складати 34645,01грн.

Вкладені інвестиції в даний проект окупляться через 1,2 роки при прогнозованому прибутку 251445,8 грн. за три роки.

ВИСНОВКИ

У магістерській кваліфікаційній роботі було розроблено методи підвищення продуктивності під час критичних навантажень на систему, а саме метод автоматизованого аналізу результатів стрес-тестування, метод агрегації результатів стрес-тестування та метод оптимізації SQL-запитів у середовищі MS SQL.

Проведено аналіз предметної області та основних аналогів, досліджено їх переваги та недоліки. Проведено дослідження існуючих методів для проведення стрес-тестування, а також існуючих програмних реалізацій. На основі цього аналізу було прийнято рішення про розробку власної системи, яка покриває недоліки існуючих рішень.

Подальшого розвитку отримав метод автоматизованого аналізу результатів стрес-тестування «Abstract Genetic Algorithm», в якому, на відміну від класичного методу, використано додаткову оцінювальну функцію автоматичного пошуку вузьких місць у апаратній частині сервера, що дало можливість збільшити точність пошуку сценаріїв з з низькою продуктивністю.

Подальшого розвитку отримав метод агрегації результатів стрес-тестування, який, на відміну від існуючих, зберігає окремо результати стрес тестування та їх аналізу, що дало можливість підвищити ефективність проведення аналізу результатів стрес-тестування.

Удосконалено метод оптимізації SQL-запитів у середовищі MS SQL, який, на відміну від відомих алгоритмів, передбачає зменшення кількості операцій читання, що дозволило зменшити час виконання запитів в середньому на 20%.Розроблено архітектуру програмної системи, яка базується на мікросервісному підході. Це дозволило забезпечити простоту для підтримки та тестування, слабку зв’язаність між компонентами системи та високу маштабованість.

Проведено варіантний аналіз і обґрунтування вибору засобів реалізації. У результаті аналізу прийнято рішення застосувати мову програмування C# та фреймворк ASP.Net Core.

Виконано тестування програмної системи, а також методу аналізу результатів стрес тестування за допомогою ручного тестування білої скриньки. В результаті тестування зроблено висновок про відповідність програмного продукту критеріям якості. Розроблено інструкцію користувача, яка містить необхідну інформацію для користування системою.

Отримані в магістерській кваліфікаційній роботі наукові та практичні положення застосовано в системі для підвищення продуктивності веб-сервісів під час критичних навантажень.

Результати роботи доповідались на науково-технічних конференціях та опубліковані в наукових публікаціях.

Проведено аналіз комерційного потенціалу розробки, який довів, що програмний продукт за своїми характеристиками випереджає аналогічні програмні продукти і є перспективною розробкою. Він має кращі функціональні показники, а тому є конкурентоспроможним товаром на ринку.

Задачу магістерської кваліфікаційної роботи виконано в повному обсязі: проаналізовано предметну область, розроблено методи та описано алгоритми підвищення продуктивності веб-сервісів в умовах високого навантаження на систему, розроблено програмну реалізацію системи, проведено тестування та розроблено інструкцію користувача.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Блог a1qa О тестировании и качестве ПО. Как измерить эффективность вашего веб-сервиса? [Електронний ресурс]. URL: https://www.a1qa.ru/blog/kak-izmerit-effektivnost-vashego-veb-servisa/

2. What is STRESS Testing in Software Testing? Tools, Types, Examples [Електронний ресурс]. URL: https://www.guru99.com/stress-testing-tutorial.html

3. Automated Analysis of Load Testing Results / Zhen Ming Jiang. – Кінгстон : Queen's University, 2013. – 222 с.

4. Свіжак В.В. Шляхи підвищення продуктивності роботи веб-сервісів під час критичних навантажень / В.В.Свіжак, О.В.Романюк // Осінні наукові зібрання – 2021, LXXII Міжнародна науково-практична інтернет-конференція. – м. Київ, 27 вересня 2021 року. – С.173-178.

5. Свіжак В.В. Програмний додаток для аналізу продуктивності веб-сервесів в умовах критичних навантаженнь / В.В.Свіжак, О.В.Романюк // Збірник матеріалів Міжнародної науково-практичної Інтернет конференції 9-10 листопада 2021 р. – Суми/Вінниця: НІКО/ВНТУ, 2021. – 224 с

6. Свіжак В.В. Підвищення продуктивності виконання SQL-запитів / В.В.Свіжак, О.В.Романюк // Матеріали молодіжної науково-практичної інтернет-конференції студентів аспірантів та молодих науковців «Молодь в науці: дослідження, проблеми, перспективи (МН-2020)» : збірник матеріалів. – Вінниця: ВНТУ, 2020. – С.594-595. – ISBN 978-966-641-805-3

7. Свіжак В.В. Програмний додаток для дослідження продуктивності SQL-запитів / В.В.Свіжак, О.В.Романюк // Матеріали конференції «XLIX Науково-технічна конференція підрозділів Вінницького національного технічного університету 2020)»: збірник доповідей. – Вінниця : ВНТУ, 2020. – С.1103-1104.

8. Understanding Performance Bottlenecks [Електронний ресурс]. URL: https://core.vmware.com/blog/understanding-performance-bottlenecks

9. The 5 Most Common Performance Bottlenecks [Електронний ресурс]. URL: https://www.apica.io/blog/5-common-performance-bottlenecks/

10. Recognizing a Processor Bottleneck [Електронний ресурс]. URL: https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-2000-server/cc938609(v=technet.10)?redirectedfrom=MSDN

11. What is STRESS Testing in Software Testing? Tools, Types, Examples [Електронний ресурс]. URL: https://www.guru99.com/stress-testing-tutorial.html

12. Unblocking Bottlenecks. Fixing Unbalanced Processes [Електронний ресурс]. URL: https://www.mindtools.com/pages/article/newTMC_76.htm

13. DB-Engines [Електронний ресурс]. URL: https://db-engines.com/

14. Романюк О. Н. Організація баз даних і знань : навчальний посібник / О. Н. Романюк, Т. О. Савчук. - Вінниця : УНІВЕРСУМ-Вінниця, 2003. – 217 с.

15. 10 Common SQL Programming Mistakes and How to Avoid Them [Електронний ресурс]. URL: https://www.upwork.com/hiring/data/common-sql-programming-mistakes/

16. Understanding SQL Server Physical Joins [Електронний ресурс]. URL: https://www.mssqltips.com/sqlservertip/2115/understanding-sql-server-physical-joins/

17. Автоматизоване тестування. Все про Perfomance testing або тес [Електронний ресурс]. URL: https://www.quality-assurance-group.com/performance-testing-what-is-it/

18. An industrial case study of customizing operational profiles using log compression. In Proceedings of the 30th international conference on Software engineering / A. E. Hassan, D. J. Martin, P. Flora, P. Mansfield, and D. Dietz. – Нью Йорк : ICSE, 2008 – 713 с.

19. What are microservices? [Електронний ресурс]. URL: https://microservices.io/index.html

20. Microservices vs Monolithic Architecture? [Електронний ресурс]. URL: https://www.mulesoft.com/resources/api/microservices-vs-monolithic

21. What is .NET? [Електронний ресурс]. URL: https://dotnet.microsoft.com/learn/dotnet/what-is-dotnet

22. CLR Integration [Електронний ресурс]. URL: https://docs.microsoft.com/en-us/sql/relational-databases/clr-integration/clr-integration-overview?view=sql-server-ver15

23. 5 лучших IDE [Електронний ресурс]. URL: https://timeweb.com/ru/community/articles/5-luchshih-ide-1

24. The 3 Best Config File Formats [Електронний ресурс]. URL: https://jhall.io/posts/best-config-file-formats/

25. Monitoring Windows servers using Prometheus — wmi_exporter [Електронний ресурс]. URL: https://rakeshjain-devops.medium.com/monitoring-windows-servers-using-prometheus-wmi-exporter-eb082fcbaffb

26. How Prometheus Monitoring works | Prometheus Architecture explained [Електронний ресурс]. URL: https://www.youtube.com/watch?v=h4Sl21AKiDg

27. XML [Електронний ресурс]. URL: https://uk.wikipedia.org/wiki/XML

28. Strate J. Expert Performance Indexing in SQL Server, 2015 p. – 432 c.

29. Index Element (DTA) [Електронний ресурс]. URL: https://docs.microsoft.com/en-us/sql/tools/dta/index-element-dta?view=sql-server-ver15

30. Тестирование. Фундаментальная теория. [Електронний ресурс]. URL: https://habr.com/ru/post/279535/

31. Prometheus Docs [Електронний ресурс]. URL: https://prometheus.io/docs/introduction/overview/

32. Методичні вказівки до виконання економічної частини магістерських кваліфікаційних робіт / Уклад. : В. О. Козловський, О. Й. Лесько, В. В. Кавецький. – Вінниця : ВНТУ, 2021. – 42 с.

Додаток А Технічне завдання

Міністерство освіти і науки України

Вінницький національний технічний університет

Факультет інформаційних технологій та комп'ютерної інженерії

ЗАТВЕРДЖУЮ

д.т.н., проф. О. Н. Романюк

"____"_____2021 р.

Технічне завдання

на магістерську кваліфікаційну роботу «Розробка методів і програмних засобів для підвищення продуктивності веб-сервісів під час критичних навантажень на систему» за спеціальністю

121 – Інженерія програмного забезпечення

Керівник магістерської кваліфікаційної роботи:

_____ к.т.н., доцент О.В.Романюк

"____"_____2021 р.

Виконав:

_____студент гр.1ПІ-20м В.В. Свіжак

"____"_____2021 р.

Вінниця – 2021 року

1. Найменування та галузь застосування

Магістерська кваліфікаційна робота: «Розробка методів і програмних засобів для підвищення продуктивності веб-сервісів під час критичних навантажень на систему».

Галузь застосування – веб-сервіси стійкі до високого навантаження.

2. Підстава для розробки.

Підставою для виконання магістерської кваліфікаційної роботи (МКР) є індивідуальне завдання на МКР та наказ № 277 ректора по ВНТУ про закріплення тем МКР.

3. Мета та призначення розробки.

Метою роботи є підвищення продуктивності веб-сервісів під час високих навантажень на систему.

Призначення роботи – розробка методів і засобів для підвищення продуктивності веб-сервісів під час критичних навантажень на систему.

3 Вихідні дані для проведення НДР

Перелік основних літературних джерел, на основі яких буде виконуватись МКР.

1. Романюк О. Н. Організація баз даних і знань : навчальний посібник / О. Н. Романюк, Т. О. Савчук. - Вінниця : УНІВЕРСУМ-Вінниця, 2003. – 217 с.

2. Пєтух А. М. Бази даних. Мови запитів, управління транзакціями, розподілена обробка даних [Електронний ресурс] / А. М. Пєтух, О. В. Романюк, О. Н. Романюк // ВНТУ. – 2016. – Режим доступу до ресурсу: https://web.posibnyky.vntu.edu.ua/fitki/11petuh_bazdanyh_movy_zalitiv/.

3. Automated Analysis of Load Testing Results / Zhen Ming Jiang. – Кінгстон : Queen’s University, 2013. – 222 с.

4. Strate J. Expert Performance Indexing in SQL Server, 2015 р. – 432 с.

4. Технічні вимоги

Вихідні дані до роботи: базові методи підвищення продуктивності веб-сервісів – методи автоматизованого аналізу результатів стрес-тестування «Abstract Genetic Algorithm»; система керування базами даних – Microsoft SQL Server; мова запитів – T-SQL; вхідні дані для оптимізації – SQL-запит; режими роботи – з підключенням до БД та без підключення; спосіб підключення до БД – рядок підключень ADO.NET; вихідні дані – аналіз результата стрес-тестування.

5. Конструктивні вимоги.

Конструкція пристрою повинна відповідати естетичним та ергономічним вимогам, повинна бути зручною в обслуговуванні та керуванні.

Графічна та текстова документація повинна відповідати діючим стандартам України.

6. Перелік технічної документації, що пред'являється по закінченню робіт:

– пояснювальна записка до МКР;

– технічне завдання;

– лістинги програми.

7. Вимоги до рівня уніфікації та стандартизації

При розробці програмних засобів слід дотримуватися уніфікації і ДСТУ.

8. Стадії та етапи розробки:

№ з/п	Назва етапів магістерської кваліфікаційної Роботи	Строк виконання етапів роботи
1	Аналіз стану питання та постановка задач дослідження	15.09. 2021 – 29.09.2021
2	Розробка структури системи та методів підвищення продуктивності веб-сервісів	30.09.2021 – 14.10.2021

3	Розробка програмних засобів	15.10.2021 – 29.10.2021
4	Тестування системи	30.10.2021 – 13.11.2021
5	Економічна частина	14.11.2021 – 30.11.2021

9. Порядок контролю та прийняття.

Виконання етапів магістерської кваліфікаційної роботи контролюється керівником згідно з графіком виконання роботи. Прийняття магістерської кваліфікаційної роботи здійснюється ДЕК, затвердженою зав. кафедрою згідно з графіком

Додаток Б

ПРОТОКОЛ ПЕРЕВІРКИ НАВЧАЛЬНОЇ (КВАЛІФІКАЦІЙНОЇ) РОБОТИ

Назва роботи: **Розробка методів і програмних засобів для підвищення продуктивності веб-сервісів під час критичних навантажень на систему.**

Тип роботи: кваліфікаційна робота

Підрозділ : кафедра програмного забезпечення, ФІТКІ, 1ПІ – 20м

Науковий керівник: к.т.н. доц. Романюк О. В.

Unicheck	
Оригінальність	**94,2%**
Схожість	5,8%

Аналіз звіту подібності

■ **Запозичення, виявлені у роботі, оформлені коректно і не містять ознак плагіату.**

□ Виявлені у роботі запозичення не мають ознак плагіату, але їх надмірна кількість викликає сумніви щодо цінності роботи і відсутності самостійності її автора. Роботу направити на доопрацювання.

□ Виявлені у роботі запозичення є недобросовісними і мають ознаки плагіату та/або в ній містяться навмисні спотворення тексту, що вказують на спроби приховування недобросовісних запозичень.

Заявляю, що ознайомлена з повним звітом подібності, який був згенерований Системою щодо роботи «Розробка методів і програмних засобів для підвищення продуктивності веб-сервісів під час критичних навантажень на систему».

Автор _____ Свіжак Віктор Вячеславович

Опис прийнятого рішення: **допустити до захисту**

Особа, відповідальна за перевірку _____ Черноволик Г. О.
(підпис) (прізвище, ініціали)

Експерт _____ _____
(за потреби) (підпис) (прізвище, ініціали, посада)

Додаток В Лістинг коду

MainForm.Designer.cs

```
using System.Collections.Generic;
using Common;

namespace SqlOptimizer
{
    partial class MainForm
    {
        /// <summary>
        ///  Required designer variable.
        /// </summary>
        private System.ComponentModel.IContainer components = null;

        /// <summary>
        ///  Clean up any resources being used.
        /// </summary>
        /// <param name="disposing">true if managed resources should be disposed; otherwise,
false.</param>
        protected override void Dispose(bool disposing)
        {
            if (disposing && (components != null))
            {
                components.Dispose();
            }
            base.Dispose(disposing);
        }

        #region Windows Form Designer generated code

        /// <summary>
        ///  Required method for Designer support - do not modify
        ///  the contents of this method with the code editor.
        /// </summary>
        private void InitializeComponent()
        {
            this.QueryInput = new System.Windows.Forms.RichTextBox();
            this.QueryOutput = new System.Windows.Forms.RichTextBox();
            this.AnalyzeButton = new System.Windows.Forms.Button();
            this.AddButton = new System.Windows.Forms.Button();
            this.Buttons = new List<System.Windows.Forms.Button>();
            this.SuspendLayout();
            //
            // QueryInput
            //
            this.QueryInput.Location = new System.Drawing.Point(265, 12);
            this.QueryInput.Name = "QueryInput";
            this.QueryInput.Size = new System.Drawing.Size(749, 168);
            this.QueryInput.TabIndex = 0;
            this.QueryInput.Text = "";
            //
            // QueryOutput
            //
            this.QueryOutput.Location = new System.Drawing.Point(265, 236);
            this.QueryOutput.Name = "QueryOutput";
            this.QueryOutput.Size = new System.Drawing.Size(749, 379);
            this.QueryOutput.TabIndex = 1;
            this.QueryOutput.Text = "";
            //
            // AnalyzeButton
            //
            this.AnalyzeButton.Location = new System.Drawing.Point(551, 186);
            this.AnalyzeButton.Name = "AnalyzeButton";
```

```
            this.AnalyzeButton.Size = new System.Drawing.Size(166, 44);
            this.AnalyzeButton.TabIndex = 3;
            this.AnalyzeButton.Text = "Аналізувати";
            this.AnalyzeButton.UseVisualStyleBackColor = true;
            this.AnalyzeButton.Click += new System.EventHandler(this.AnalyzeButton_Click);
            //
            // DB_0
            //
            var DB_0 = new System.Windows.Forms.Button();
            DB_0.Location = new System.Drawing.Point(12, 12);
            DB_0.Name = "DB_0";
            DB_0.Size = new System.Drawing.Size(246, 71);
            DB_0.TabIndex = 5;
            DB_0.Text = "Без з'єднання";
            DB_0.UseVisualStyleBackColor = true;
            DB_0.MouseClick += new System.Windows.Forms.MouseEventHandler(this.DB_Click);
            this.Buttons.Add(DB_0);
            //
            // DB_N
            //
            var databases = ConfigurationController.GetAllDatabase();
            for (int i = 0; i < databases.Count; i++)
            {
                var DB_N = new System.Windows.Forms.Button();
                DB_N.Location = new System.Drawing.Point(12, 12 + 77 * (i + 1));
                DB_N.Name = $"DB_{i + 1}";
                DB_N.Size = new System.Drawing.Size(246, 71);
                DB_N.TabIndex = 6 + i;
                DB_N.Text = databases[i];
                DB_N.UseVisualStyleBackColor = true;
                DB_N.MouseClick += new
System.Windows.Forms.MouseEventHandler(this.DB_Click);
                this.Buttons.Add(DB_N);
            }
            //
            // AddButton
            //
            this.AddButton.Location = new System.Drawing.Point(97, 12 + 77 *
(this.Buttons.Count));
            this.AddButton.Name = "AddButton";
            this.AddButton.Size = new System.Drawing.Size(71, 71);
            this.AddButton.TabIndex = 4;
            this.AddButton.Text = "+";
            this.AddButton.UseVisualStyleBackColor = true;
            this.AddButton.Click += new System.EventHandler(this.AddButton_Click);
            //
            // MainForm
            //
            this.AutoScaleDimensions = new System.Drawing.SizeF(8F, 16F);
            this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
            this.ClientSize = new System.Drawing.Size(1026, 627);
            this.Controls.Add(this.AddButton);
            foreach(var button in this.Buttons)
            {
                this.Controls.Add(button);
            }
            this.Controls.Add(this.AnalyzeButton);
            this.Controls.Add(this.QueryOutput);
            this.Controls.Add(this.QueryInput);
            this.FormBorderStyle = System.Windows.Forms.FormBorderStyle.FixedSingle;
            this.MaximizeBox = false;
            this.Name = "MainForm";
            this.Text = "Sql Optimizer";
            this.FormClosing += new
System.Windows.Forms.FormClosingEventHandler(MainForm_Closing);
```

```
            this.ResumeLayout(false);

            this.DefaultColorButton = DB_0.BackColor;
            ShowButton(DB_0);
        }

        #endregion

        private System.Windows.Forms.RichTextBox QueryInput;
        private System.Windows.Forms.RichTextBox QueryOutput;
        private System.Windows.Forms.Button AnalyzeButton;
        private System.Windows.Forms.Button AddButton;
        private List<System.Windows.Forms.Button> Buttons;
    }
}
```

MainForm.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Data.SqlClient;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;
using Common;
using Models;
using Parser;

namespace SqlOptimizer
{
    public partial class MainForm : Form
    {
        private Color DefaultColorButton;

        private string CurrentConnectionString;

        private DataBase DataBase;

        public MainForm()
        {
            InitializeComponent();
        }

        #region Events

        private void AnalyzeButton_Click(object sender, EventArgs e)
        {
            if (string.IsNullOrEmpty(QueryInput.Text.Trim()))
            {
                QueryInput.Text = string.Empty;
                return;
            }

            try
            {
                var result = SqlAnalyze.Analyze(QueryInput.Text, DataBase);

                QueryOutput.Text = "";

                var currentColor = 0;
                var selctedColors = new List<Color>()
```

```
                {
                    Color.Black,
                    Color.Red
                };

                QueryOutput.SelectionColor = selctedColors[currentColor];
                for(int i = 0; i < result.Length; i++)
                {
                    var letter = result[i];

                    if(letter=='%')
                    {
                        currentColor = (currentColor + 1) % 2;
                    }
                    else
                    {
                        QueryOutput.SelectionColor = selctedColors[currentColor];
                        QueryOutput.SelectedText += letter;
                    }
                }

                QueryOutput.SelectionColor = Color.Green;
                QueryOutput.SelectedText += "\n\n--";
            }
            catch (Exception myEx)
            {
                MessageBox.Show(myEx.Message);
            }
        }

        private void MainForm_Closing(object sender, FormClosingEventArgs e)
        {
            ConfigurationController.SaveToFile();
        }

        private void DB_Click(object sender, MouseEventArgs e)
        {
            var button = ((Button)sender);

            if (e.X > button.Location.X + button.Size.Width / 2)
            {
                DB_RightClick(sender, e);
                return;
            }

            var databaseName = button.Text;
            var needConnection = databaseName != "Без з'єднання";

            DataBase db = null;

            if (needConnection)
            {
                CurrentConnectionString =
ConfigurationController.GetConnectionString(databaseName);

                using (var conn = new SqlConnection(CurrentConnectionString))
                {
                    conn.Open();
                    DataTable allTablesSchemaTable = conn.GetSchema("Tables");
                    db = conn.GetSchema("Tables").Rows.ToDataBase();
                    Console.WriteLine();
                }

                for (int i = 0; i < db.Tables.Count; i++)
                {
```

```
                    var table = new Table();

                    using (var conn = new SqlConnection(CurrentConnectionString))
                    {
                        var command = new
SqlCommand(SqlQueries.GetTableSchema(db.Tables[i].Name), conn);
                        command.Connection.Open();
                        var reader = command.ExecuteReader();
                        table = reader.ToTable();
                    }

                    db.Tables[i] = table;
                }
            }

            DataBase = db;

            ShowButton(button);
        }

        private void AddButton_Click(object sender, EventArgs e)
        {
            //MessageBox.Show("З'єднання з назвою [Dev] вже збережено до конфігураційного
файлу. Введіть іншу назву", "Error was occure");
            //return;
            var databaseConfiguration = new DatabaseConfiguration();
            var names = ConfigurationController.GetAllDatabase();
            var addForm = new AddDB(databaseConfiguration);
            addForm.ShowDialog();

            if(addForm.DialogResult == DialogResult.OK &&
                !names.Contains(databaseConfiguration.DatabaseName) &&
                !string.IsNullOrEmpty(databaseConfiguration.ConnectionString) &&
                !string.IsNullOrEmpty(databaseConfiguration.DatabaseName))
            {
                ConfigurationController.UpdateConfig(databaseConfiguration);

                var DB_N = new System.Windows.Forms.Button();
                DB_N.Location = new System.Drawing.Point(12, 12 + 77 *
(this.Buttons.Count));
                DB_N.Name = $"DB_{this.Buttons.Count}";
                DB_N.Size = new System.Drawing.Size(246, 71);
                DB_N.TabIndex = 6 + this.Buttons.Count;
                DB_N.Text = databaseConfiguration.DatabaseName;
                DB_N.UseVisualStyleBackColor = true;
                DB_N.MouseClick += new
System.Windows.Forms.MouseEventHandler(this.DB_Click);
                this.Buttons.Add(DB_N);
                this.Controls.Add(DB_N);
                this.AddButton.Location = new System.Drawing.Point(97, 12 + 77 *
(this.Buttons.Count));
                return;
            }
            MessageBox.Show("Incorrect data");
        }

        private void DB_RightClick(object sender, MouseEventArgs e)
        {
            var button = ((Button)sender);
            var databaseName = button.Text;
            var needConnection = databaseName != "Without connection";

            if(!needConnection)
            {
                return;
```

```
            }

            var connString = ConfigurationController.GetConnectionString(databaseName);
            var dbConf = new DatabaseConfiguration
            {
                DatabaseName = databaseName,
                ConnectionString = connString
            };

            var form = new AddDB(dbConf);
            form.ShowDialog();

            if(form.DialogResult == DialogResult.Cancel)
            {
                return;
            }

            ConfigurationController.UpdateConfig(dbConf, databaseName);
            if(dbConf.DatabaseName != databaseName)
            {
                button.Text = dbConf.DatabaseName;
            }
        }

        #endregion

        private void ShowButton(Button button)
        {
            this.Buttons.ForEach(b =>
            {
                b.Enabled = true;
                b.BackColor = DefaultColorButton;
            });

            button.Enabled = false;
            button.BackColor = Color.DimGray;
        }
    }
}
```

AddDB.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using Common;
using Models;

namespace SqlOptimizer
{
    public partial class AddDB : Form
    {
        private DatabaseConfiguration DatabaseConfiguration;

        public AddDB() : this(new DatabaseConfiguration())
        { }

        public AddDB(DatabaseConfiguration databaseConfiguration)
        {
            DatabaseConfiguration = databaseConfiguration;
```

```
            InitializeComponent();
        }

        private void OkButton_Click(object sender, EventArgs e)
        {
            this.DatabaseConfiguration.ConnectionString = this.ConnString.Text;
            this.DatabaseConfiguration.DatabaseName = this.DbName.Text;

            this.DialogResult = DialogResult.OK;
            this.Close();
        }

        private void CancelButton_Click(object sender, EventArgs e)
        {
            this.DialogResult = DialogResult.Cancel;
            this.Close();
        }
    }
}
```

ConfigurationController.cs

```
using Newtonsoft.Json;
using System.Collections.Generic;
using System.ComponentModel;
using System.IO;
using System.Linq;

namespace Common
{
    public static class ConfigurationController
    {
        //TODO
        const string path = @"D:\Projects\SQL Optimizer\Configuration.json";
        private static List<DatabaseConfiguration> DatabaseConfigurations;

        /// <summary>
        /// Get connection string from config file
        /// </summary>
        /// <param name="databaseName"></param>
        /// <returns></returns>
        public static string GetConnectionString(string databaseName)
        {
            var items = GetConfig();

            return items.Single(db => db.DatabaseName == databaseName).ConnectionString;
        }

        /// <summary>
        /// Update database name and connection string in file by old database name
        /// </summary>
        /// <param name="databaseName"></param>
        /// <param name="connectionString"></param>
        public static void UpdateConfig(DatabaseConfiguration databaseConfiguration, string
key = null)
        {
            var items = GetConfig();

            for (int i = 0; i < items.Count; i++)
            {
                if(items[i].DatabaseName == key)
                {
                    items[i].DatabaseName = databaseConfiguration.DatabaseName;
                    items[i].ConnectionString = databaseConfiguration.ConnectionString;
```

```
                    return;
                }
            }

            items.Add(databaseConfiguration);
        }

        /// <summary>
        /// Get list of database name from file
        /// </summary>
        /// <returns></returns>
        public static List<string> GetAllDatabase()
        {
            var items = GetConfig();

            return items.Select(db => db.DatabaseName).ToList();
        }

        /// <summary>
        /// Save to file
        /// </summary>
        public static void SaveToFile()
        {
            //TODO
            var items = GetConfig();
            using (StreamWriter file = File.CreateText(path))
            {
                JsonSerializer serializer = new JsonSerializer();
                serializer.Serialize(file, items);
            }
        }

        private static List<DatabaseConfiguration> GetConfig()
        {
            if(DatabaseConfigurations != null)
            {
                return DatabaseConfigurations;
            }

            using (var r = new StreamReader(path))
            {
                string json = r.ReadToEnd();
                DatabaseConfigurations =
JsonConvert.DeserializeObject<List<DatabaseConfiguration>>(json);
            }

            return DatabaseConfigurations;
        }
    }
}
```

SqlAnalyze.cs

```
using Models;
using Parser;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Reflection.PortableExecutable;
using System.Text;
using System.Text.Json;
using System.Text.RegularExpressions;

namespace Common
{
```

```
    public static class SqlAnalyze
    {
        public static string Analyze(string query, DataBase dataBase = null)
        {
            var parse = new SqlParser();
            parse.Parse(query);
            query = parse.ToText();
            var xml = parse.ToXml();

            query = Regex.Replace(query, @"\@ ", @"@");

            if (dataBase != null)
            {
                query = Regex.Replace(query, SqlQueries.PatternSelectAll, m =>
                {
                    var from = Regex.Match(query[m.Index..], @"FROM .* (WHERE|GROUP BY|$)");
                    var tables = dataBase.Tables.Where(t => from.Value.Contains(t.Name));
                    var columns = "";
                    tables.ToList().ForEach(t => columns += $", {string.Join(", ",
t.Columns.Select(c => $"{t.Name}.{c.Key}"))}");
                    return $"SELECT {columns[1..]}";
                }, RegexOptions.IgnoreCase);


                query = Regex.Replace(query, SqlQueries.PatternGuidIn, m =>
                {
                    foreach (var table in dataBase.Tables)
                    {
                        foreach (var column in table.Columns.Where(x => x.Value ==
SqlTypes.Guid))
                        {
                            if(m.Value.Contains(column.Key))
                            {
                                return string.Format(SqlQueries.DangerPart, m.Value);
                            }
                        }
                    }
                    return m.Value;
                }, RegexOptions.IgnoreCase);
            }
            else
            {
                query = Regex.Replace(query, SqlQueries.PatternSelectAll, m =>
                    string.Format(SqlQueries.DangerPart, m.Value), RegexOptions.IgnoreCase);
            }

            query = Regex.Replace(query, SqlQueries.PatternUnion, m =>
                $"{m.Value} ALL", RegexOptions.IgnoreCase);

            query = Regex.Replace(query, SqlQueries.PatternHints, m =>
                "", RegexOptions.IgnoreCase);

            query = Regex.Replace(query, SqlQueries.PatternCursor, m =>
                string.Format(SqlQueries.DangerPart, m.Value), RegexOptions.IgnoreCase);

            query = Regex.Replace(query, SqlQueries.PatternArifmet, m =>
            {
                var end = "";
                var start = Regex.Replace(m.Value, SqlQueries.PatternArifmetPart, inM =>
                {
                    switch (inM.Value[0])
                    {
                        case '+':
                            end += '-';
                            break;
```

```
                        case '-':
                            end += '+';
                            break;
                        case '*':
                            end += '/';
                            break;
                        case '/':
                            end += '*';
                            break;
                    }
                    end += inM.Value[1..^1];
                    return " =";
                }, RegexOptions.IgnoreCase);
                return start + end;
            }, RegexOptions.IgnoreCase);

            var isInSub = Regex.IsMatch(query, SqlQueries.PatternInSub,
RegexOptions.IgnoreCase);
            if(isInSub)
            {
                var match = Regex.Match(query, SqlQueries.PatternInSub,
RegexOptions.IgnoreCase);
                var sub = Regex.Match(match.Value, @"\(.*\)");
                var subFiled = Regex.Match(sub.Value[9..], @"[^ ]*");
                var tableFiled = Regex.Match(match.Value, @"[^ ]*");

                var cte = SqlQueries.CreateCte("sub1", sub.Value[1..^1]);
                query = cte + query;

                var notPart = Regex.IsMatch(match.Value, "NOT", RegexOptions.IgnoreCase) ?
"" : "NOT";
                var where = $"sub1.{subFiled} IS {notPart} NULL";
                query = query.Replace(match.Value, where);

                var join = $"LEFT JOIN sub1 ON {tableFiled} = sub1.{subFiled}";
                query = Regex.Replace(query, "WHERE", m => $"{join} WHERE");
            }

            parse.Parse(query);
            query = parse.ToText();


            if (dataBase != null)
            {
                var result = "";

                var allowfields = new List<string>();
                dataBase.Tables.ForEach(t =>
                {
                    allowfields.Add(t.Name.ToUpper());
                    foreach (var col in t.Columns)
                    {
                        allowfields.Add(col.Key.ToUpper());
                    }
                });
                allowfields.AddRange(SqlQueries.FormatOperators);
                allowfields.AddRange(SqlQueries.NotFormatOperators);

                var fields = query.Split(' ');
                foreach(var field in fields)
                {
                    result += ' ';

                    if(Regex.IsMatch(field, "^[a-zA-Z]*$"))
                    {
```

```
                        if(!allowfields.Contains(field.ToUpper()) || field.StartsWith("@"))
                        {
                            result += string.Format(SqlQueries.DangerPart, field);
                            continue;
                        }
                    }

                    result += field;
                }
                query = result[1..];
            }

            query = Regex.Replace(query, @"( |)\.( |)", ".");

            SqlQueries.FormatOperators.ForEach(x =>
            {
                query = Regex.Replace(query, x, $"\n{x}", RegexOptions.IgnoreCase);
            });

            return query;
        }
    }
}
```

Extensions.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using Models;
using System.Data;
using System.Data.SqlClient;

namespace Common
{
    public static class Extensions
    {
        /// <summary>
        /// Convert string sql type to <see cref="SqlTypes"/>
        /// </summary>
        /// <param name="type"></param>
        /// <returns></returns>
        public static SqlTypes ToSqlType(this string type)
        {
            switch(type.ToUpper())
            {
                case ("TINYINT"):
                case ("SMALLINT"):
                case ("INT"):
                case ("BIGINT"):
                case ("DECIMAL"):
                case ("NUMERIC"):
                case ("SMALLMONEY"):
                case ("MONEY"):
                case ("FLOAT"):
                case ("REAL"):
                    return SqlTypes.Number;
                case ("DATE"):
                case ("TIME"):
                case ("DATETIME"):
                case ("DATETIME2"):
                case ("SMALLDATETIME"):
                case ("DATETIMEOFFSET"):
                    return SqlTypes.Date;
                case ("CHAR"):
```

```
            case ("VARCHAR"):
            case ("NCHAR"):
            case ("NVARCHAR"):
            case ("TEXT"):
            case ("NTEXT"):
                return SqlTypes.String;
            case ("BINARY"):
            case ("VARBINARY"):
                return SqlTypes.Binary;
            case ("BIT"):
                return SqlTypes.Bool;
            case ("UNIQUEIDENTIFIER"):
                return SqlTypes.Guid;
            default:
                return SqlTypes.Other;
        }
    }

    /// <summary>
    /// Create <see cref="Table"/> from <see cref="SqlDataReader"/>
    /// </summary>
    /// <param name="reader"></param>
    /// <returns></returns>
    public static Table ToTable(this SqlDataReader reader)
    {
        var table = new Table();

        while (reader.Read())
        {
            table.Columns.Add(
                reader["COLUMN_NAME"].ToString(),
                reader["TYPE_NAME"].ToString().ToSqlType());
            if(string.IsNullOrEmpty(table.Name))
            {
                table.Name = reader["TABLE_NAME"].ToString();
            }
        }

        return table;
    }

    /// <summary>
    /// Create <see cref="DataBase"/> from <see cref="DataRowCollection"/>
    /// </summary>
    /// <param name="rows"></param>
    /// <returns></returns>
    public static DataBase ToDataBase(this DataRowCollection rows)
    {
        var database = new DataBase();

        foreach(var row in rows)
        {
            var tableType = ((DataRow)row)["TABLE_TYPE"].ToString();
            if(tableType.ToUpper() != "BASE TABLE")
            {
                continue;
            }

            database.Tables.Add(
                new Table() {
                    Name = ((DataRow)row)["TABLE_NAME"].ToString()
                });
        }

        return database;
```

```
        }
    }
}
```

Models.cs

```
using System;
using System.Collections.Generic;
using System.Text;

namespace Models
{
    public class DataBase
    {
        public List<Table> Tables;

        public DataBase()
        {
            Tables = new List<Table>();
        }
    }

    public class Table
    {
        public string Name { get; set; }
        public Dictionary<string, SqlTypes> Columns { get; set; }

        public Table()
        {
            Columns = new Dictionary<string, SqlTypes>();
        }

        public override string ToString()
        {
            return $"Table: {Name}, {Columns.Count} columns";
        }
    }
}

using System;

namespace Common
{
    public class DatabaseConfiguration
    {
        public string DatabaseName { get; set; }

        public string ConnectionString { get; set; }
    }
}

using System;
using System.Collections.Generic;
using System.Text;

namespace Models
{
    public enum SqlTypes
    {
        String,
        Date,
        Number,
        Binary,
        Guid,
        Bool,
```

```
        Other
    }
}
```

SqlParser.cs

```
using System;
using System.Collections.Generic;
using System.Text;

namespace Parser
{
	#region BracesTag

	[TagType(BracesTag.cTagName)]
	[MatchBracesTag]
	internal class BracesTag : TagBase
	{
		#region Consts

		/// <summary>
		/// The name of the tag (its identifier).
		/// </summary>
		public const string cTagName = "BRACES";

		/// <summary>
		/// The start of the tag.
		/// </summary>
		public const string cStartTag = "(";

		/// <summary>
		/// The end of the tag.
		/// </summary>
		public const string cEndTag = ")";

		#endregion

		#region Methods

		#region Common

		/// <summary>
		/// Reads the tag at the specified position in the specified word and
separator array.
		/// </summary>
		/// <returns>
		/// The position after the tag (at which to continue reading).
		/// </returns>
		protected override int InitializeCoreFromText(ParserBase parser, string sql,
int position, TagBase parentTag)
		{
			#region Check the arguments

			ParserBase.CheckTextAndPositionArguments(sql, position);

			#endregion

			int myResult = MatchStartStatic(sql, position);

			if (myResult < 0)
				throw new Exception("Cannot read the Braces tag.");
```

```
                Parser = parser;

                HasContents = true;

                return myResult;
            }

            /// <summary>
            /// Returns a value indicating whether there is the tag ending at the
specified position.        ///
            /// </summary>
            /// <returns>
            /// If this value is less than zero, then there is no ending; otherwise the
            /// position after ending is returned.
            /// </returns>
            public override int MatchEnd(string sql, int position)
            {
                CheckInitialized();

                #region Check the arguments

                ParserBase.CheckTextAndPositionArguments(sql, position);

                #endregion

                return MatchEndStatic(sql, position);
            }

            /// <summary>
            /// Writes the start of the tag.
            /// </summary>
            public override void WriteStart(StringBuilder output)
            {
                CheckInitialized();

                #region Check the parameters

                if (output == null)
                    throw new ArgumentNullException();

                #endregion

                output.Append(cStartTag);
            }

            /// <summary>
            /// Writes the end of the tag.
            /// </summary>
            public override void WriteEnd(StringBuilder output)
            {
                CheckInitialized();

                #region Check the parameters

                if (output == null)
                    throw new ArgumentNullException();

                #endregion

                output.Append(cEndTag);
            }

            #endregion

            #region Static
```

```
		/// <summary>
		/// Checks whether there is the tag start at the specified position
		/// in the specified sql.
		/// </summary>
		/// <returns>
		/// The position after the tag or -1 there is no tag start at the position.
		/// </returns>
		public static int MatchStartStatic(string sql, int position)
		{
			#region Check the arguments

			ParserBase.CheckTextAndPositionArguments(sql, position);

			#endregion

			if (string.Compare(sql, position, cStartTag, 0, cStartTag.Length, true)
!= 0)
				return -1;

			return position + cStartTag.Length;
		}

		/// <summary>
		/// Checks whether there is the tag end at the specified position
		/// in the specified sql.
		/// </summary>
		/// <returns>
		/// The position after the tag or -1 there is no tag end at the position.
		/// </returns>
		public static int MatchEndStatic(string sql, int position)
		{
			#region Check the arguments

			ParserBase.CheckTextAndPositionArguments(sql, position);

			#endregion

			if (string.Compare(sql, position, cEndTag, 0, cEndTag.Length, true) !=
0)
				return -1;

			return position + cEndTag.Length;
		}

		#endregion

		#endregion
	}

	#endregion

	#region MatchBracesTagAttribute

	internal class MatchBracesTagAttribute : MatchTagAttributeBase
	{
		#region Methods

		public override bool Match(string sql, int position)
		{
			return BracesTag.MatchStartStatic(sql, position) >= 0;
		}

		#endregion
	}
```

```
        #endregion
}
using System;
using System.Collections.Generic;
using System.Text;

namespace Parser
{
        #region ForUpdateTag

        [TagType(ForUpdateTag.cTagName)]
        [MatchForUpdateTag]
        internal class ForUpdateTag : SimpleTwoWordTag
        {
                #region Consts

                /// <summary>
                /// The name of the tag (its identifier).
                /// </summary>
                public const string cTagName = "FOR_UPDATE";

                /// <summary>
                /// The first part of tag.
                /// </summary>
                public const string cTagFirstPart = "FOR";

                /// <summary>
                /// The second part of tag.
                /// </summary>
                public const string cTagSecondPart = "UPDATE";

                #endregion

                #region Properties

                /// <summary>
                /// Gets the name of the tag.
                /// </summary>
                protected override string Name
                {
                        get
                        {
                                return cTagName;
                        }
                }

                /// <summary>
                /// Gets the first word of the tag.
                /// </summary>
                protected override string FirstWord
                {
                        get
                        {
                                return cTagFirstPart;
                        }
                }

                /// <summary>
                /// Gets the second word of the tag.
                /// </summary>
                protected override string SecondWord
                {
                        get
                        {
```

```
                        return cTagSecondPart;
                }
        }

        #endregion
}

#endregion

#region MatchForUpdateTagAttribute

internal class MatchForUpdateTagAttribute : MatchSimpleTwoWordTagAttribute
{
        #region Properties

        /// <summary>
        /// Gets the name of the tag (its identifier and sql text)
        /// </summary>
        protected override string Name
        {
                get
                {
                        return ForUpdateTag.cTagName;
                }
        }

        /// <summary>
        /// Gets the first word of the tag.
        /// </summary>
        protected override string FirstWord
        {
                get
                {
                        return ForUpdateTag.cTagFirstPart;
                }
        }

        /// <summary>
        /// Gets the second word of the tag.
        /// </summary>
        protected override string SecondWord
        {
                get
                {
                        return ForUpdateTag.cTagSecondPart;
                }
        }

        #endregion
}

#endregion
}
```

Додаток Г Ілюстративна частина

РОЗРОБКА МЕТОДІВ І ПРОГРАМНИХ ЗАСОБІВ ДЛЯ ПІДВИЩЕННЯ ПРОДУКТИВНОСТІ ВЕБ-СЕРВІСІВ ПІД ЧАС КРИТИЧНИХ НАВАНТАЖЕНЬ НА СИСТЕМУ

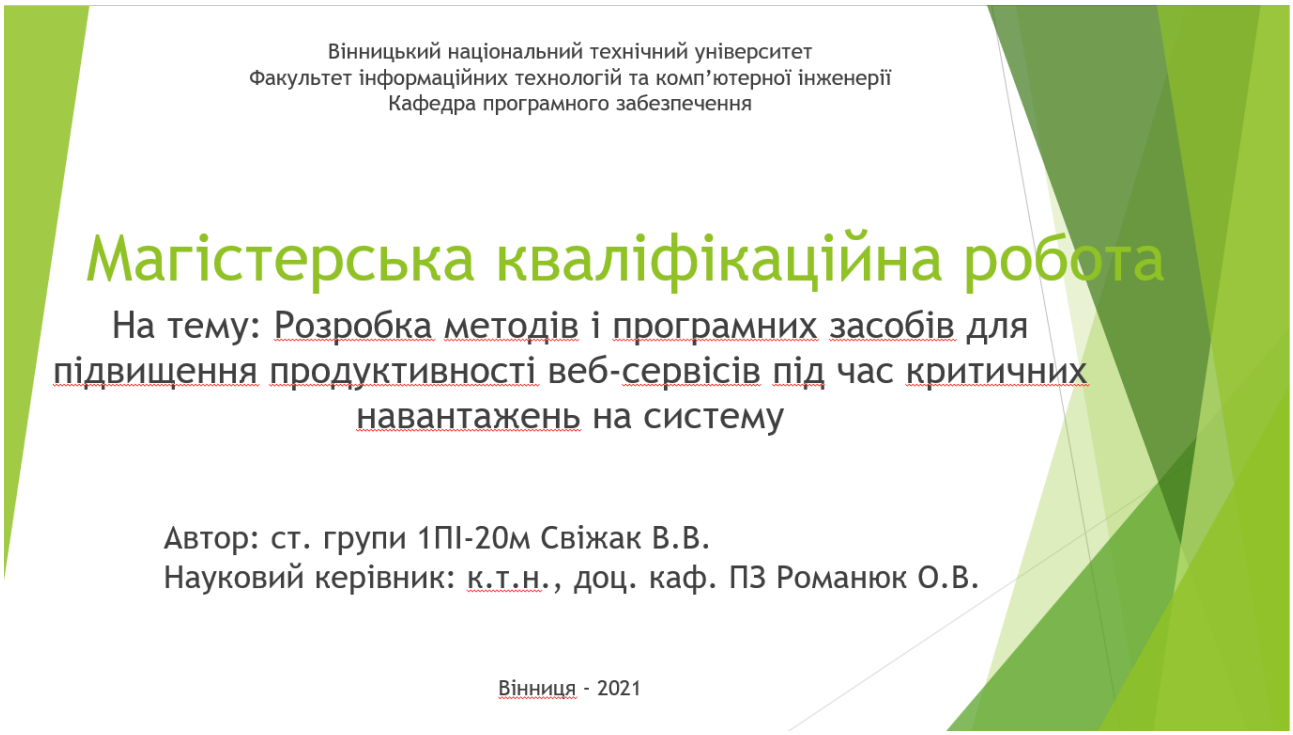


Рисунок Г.1 – Титульний слайд

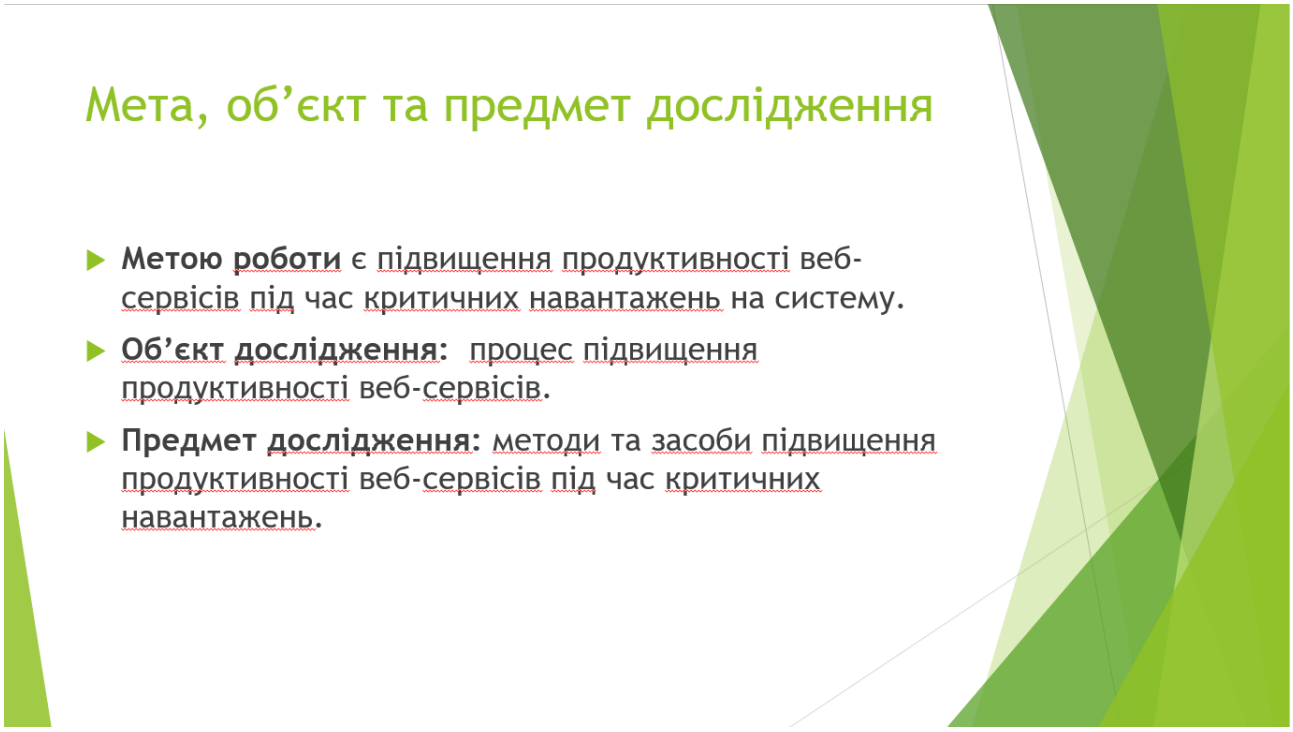


Рисунок Г.2 – Мета, об'єкт та предмет дослідження

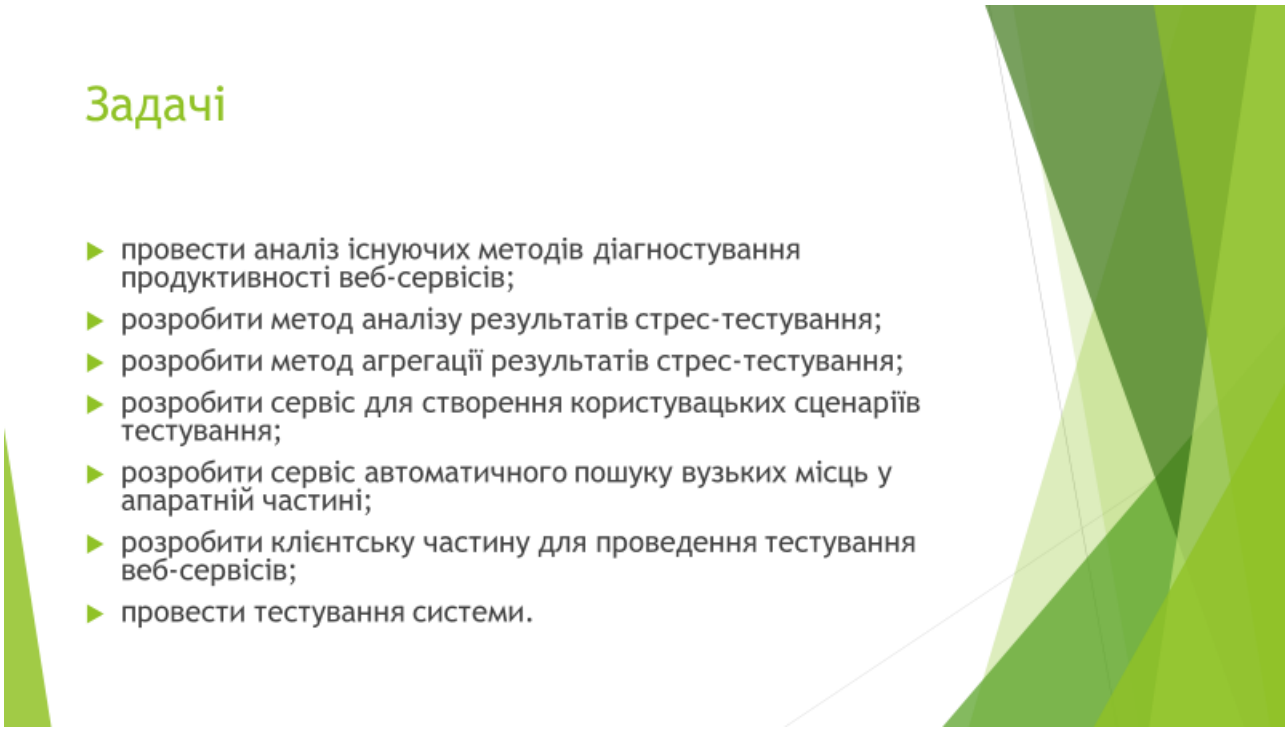


Рисунок Г.3 – Завдання дослідження

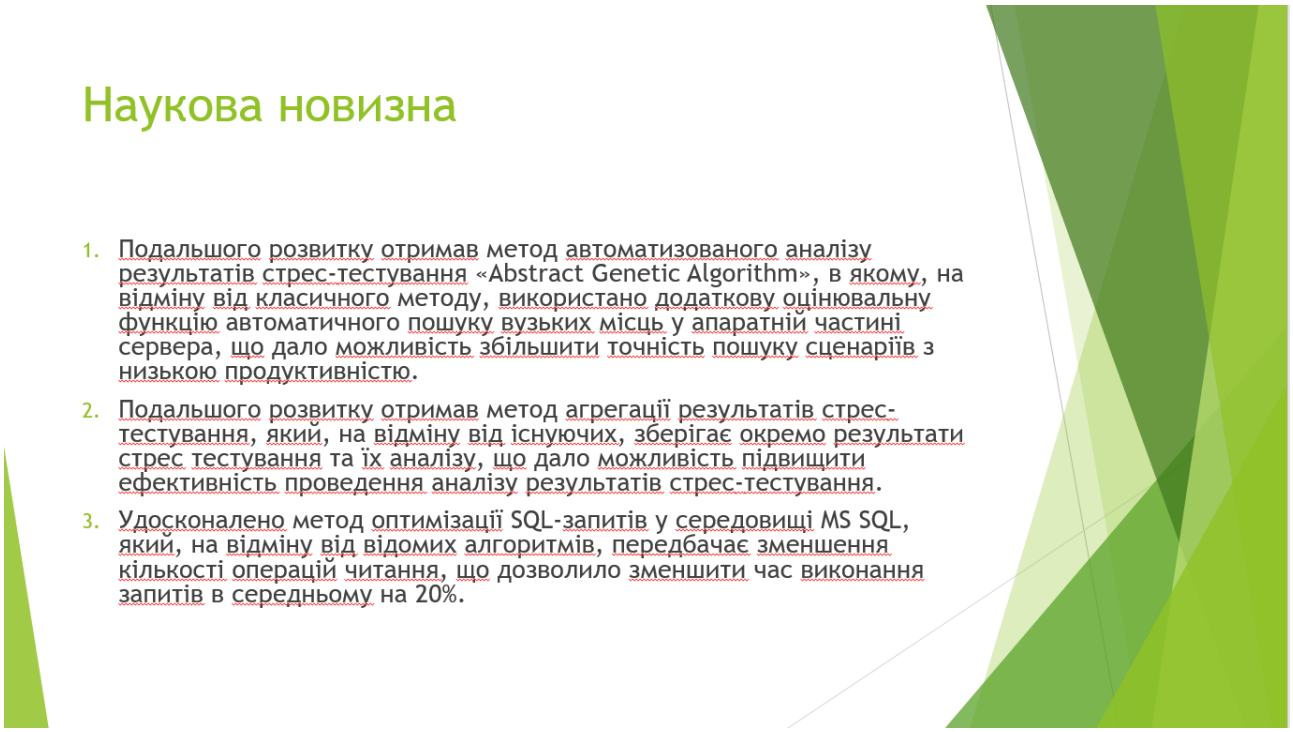


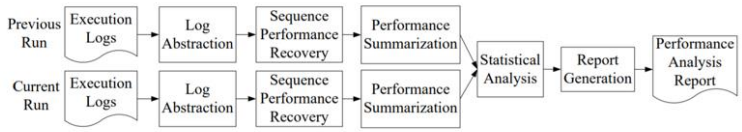
Рисунок Г.4 – Наукова новизна одержаних результатів

Порівняльний аналіз аналогів

Критерій	WebLOAD	LoadNinja	JMeter	Власна розробка
Створення користувацьких сценаріїв тестування	-	+	-	+
Розгортання у хмарі	+	-	+	+
Пошук вузьких місць	+	-	-	+
Збереження результатів тестування для аналізу	-	-	+	+
Аналіз результатів тестування	+	-	-	+
Тестування навантаження браузера	-	+	-	-
Сума	3	2	3	5

Рисунок Г.5 – Порівняння з аналогами

Метод агрегації результатів тестування



- Оцінка подібності результатів стрес-тестування визначається за наступними формулами:
- $deviation(P, C) = 1 - coisine(P, C)$,
- $coisine(P, C) = \frac{\sum_x P(x)C(x)}{\sqrt{\sum_x P(x)^2}\sqrt{\sum_x C(x)^2}}$,
- де P(x) і C(x) відповідають кількості екземплярів у попередньому та поточному запуску, які мають час реакції x для конкретного сценарію відповідно.

Рисунок Г.6 – Метод агрегації результатів стрес-тестування

Рисунок Г.8 – Схема алгоритму алгоритму аналізу результатів стрес-тестування

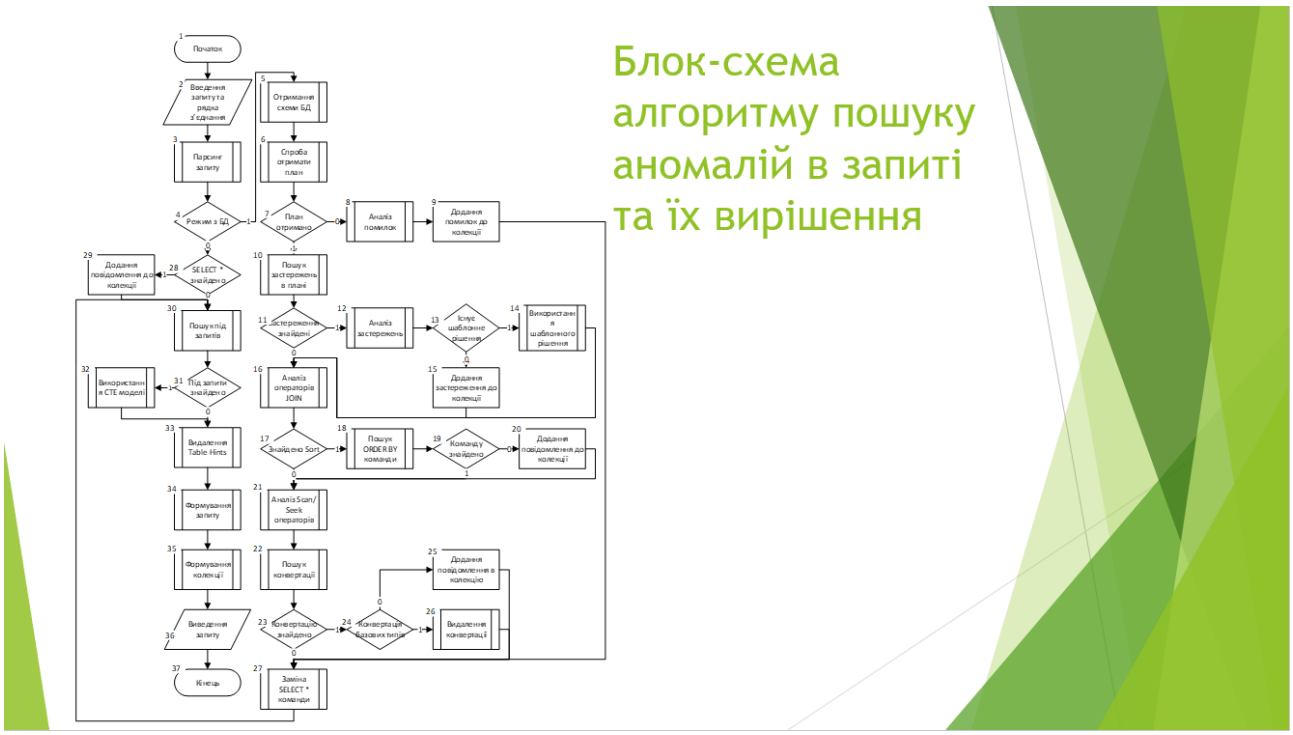


Рисунок Г.9 – Блок-схема алгоритму пошуку аномалій в запиті та їх вирішення

Рисунок Г.10 – Архітектура системи

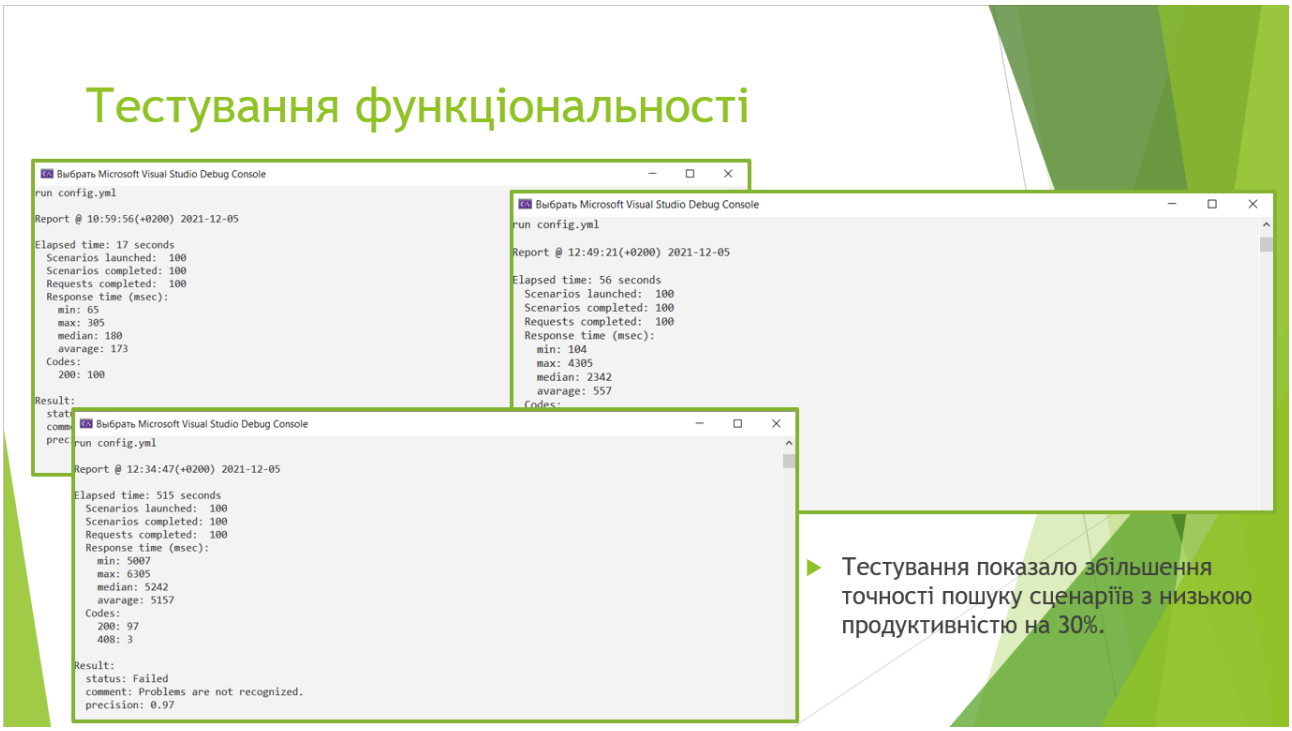


Рисунок Г.11 – Тестування системи

Економічна частина

- Загальний показник конкурентоспроможності інноваційного рішення:
- $К = \frac{1,37}{0,67} = 2,04$
- Прогнозування загальних втрат ЗВ на виконання та впровадження результатів виконаної МКНР:
- $ЗВ = \frac{31180,5}{0,9} = 34645,01$ грн
- Термін окупності вкладених у реалізацію наукового проекту інвестицій:
- $Т_{ок} = \frac{1}{0,84} = 1,2$ роки.

Рисунок Г.12 – Економічна частина

Апробація та публікація матеріалів бакалаврської дипломної роботи

- Основні результати досліджень опубліковано в 2 наукових працях.Основні положення бакалаврської дипломної роботи доповідалися та обговорювалися на Міжнародних і Всеукраїнських конференціях:
  - Молодь в науці: дослідження, проблеми, перспективи (Вінниця, 2020),
  - XLIX Науково-технічна конференція факультету інформаційних технологій та комп'ютерної інженерії (Вінниця, 2020).

Рисунок Г.13 – Апробація та публікації

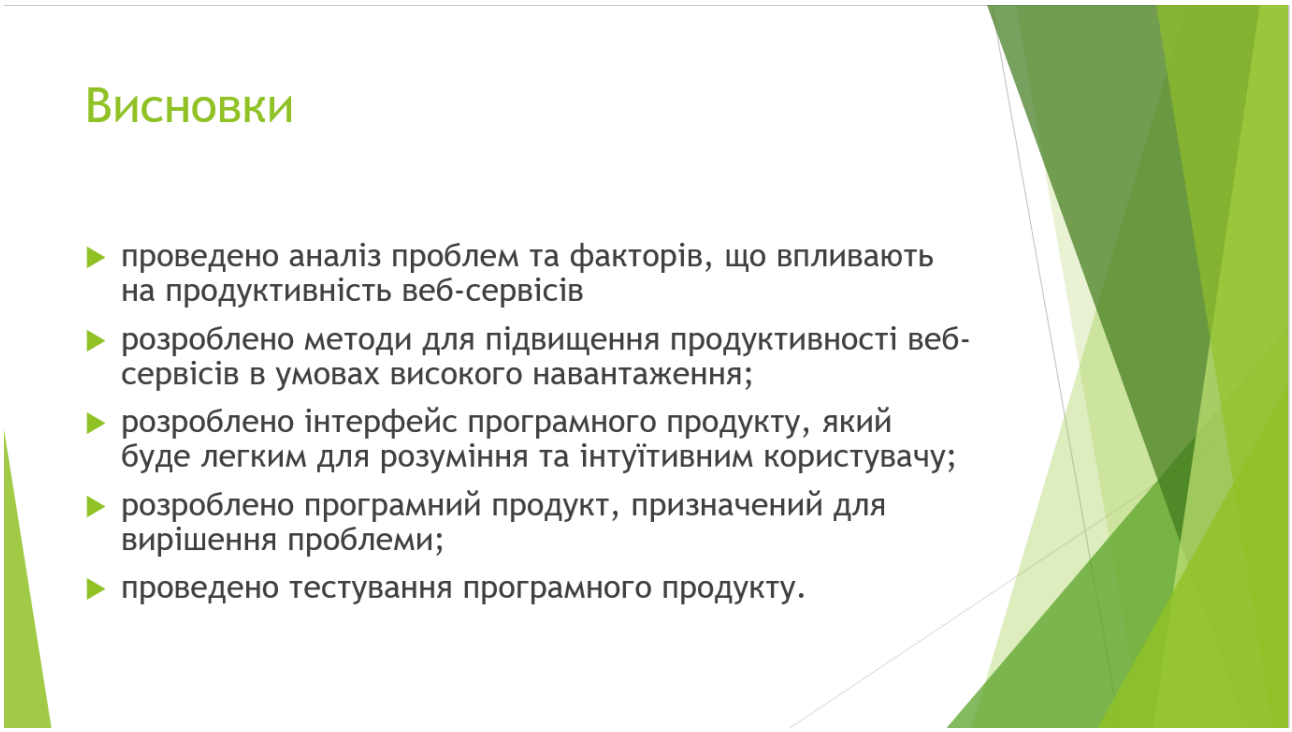


Рисунок Г.14 – Висновки.