НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Факультет прикладної математики

Кафедра програмного забезпечення комп'ютерних систем

«На правах рукопису»
УДК 004.051

«До захисту допущено»
Завідувач кафедри
_____ Євгенія СУЛЕМА
«___»_____2024 р.

Магістерська дисертація

на здобуття ступеня магістра
за освітньо-професійною програмою
«Інженерія програмного забезпечення мультимедійних та інформаційно-пошукових систем»
зі спеціальності 121 Інженерія програмного забезпечення
на тему: «Метод та програмне забезпечення для збільшення швидкодії системи управління персоналом підприємства»

Виконав:
Студент ІІ курсу, групи КП-31мп
Поваров Євгеній Валерійович _____

Керівник:
Старший викладач кафедри ПЗКС, к.т.н. ,
Хіцко Яна Володимирівна _____

Консультант з нормоконтролю:
Доцент кафедри ПЗКС, к.т.н., доцент,
Онай Микола Володимирович _____

Рецензент:
Доцент кафедри СП і СКС ФПМ, к.т.н., доцент,
Боярінова Юлія Євгеніївна _____

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.
Студент _____

Київ – 2024 року

Національний технічний університет України

«Київський політехнічний інститут імені Ігоря Сікорського»

Факультет прикладної математики

Кафедра програмного забезпечення комп'ютерних систем

Рівень вищої освіти – другий (магістерський)

Спеціальність (спеціалізація) – 121 «Інженерія програмного забезпечення»

Освітньо-професійна програма «Інженерія програмного забезпечення мультимедійних та інформаційно-пошукових систем»

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____ Євгенія СУЛЕМА

«___»_____2023 р.

ЗАВДАННЯ
на магістерську дисертацію студенту

Поварову Євгенію Валерійовичу

1. Тема дисертації «Метод та програмне забезпечення для збільшення швидкодії системи управління персоналом підприємства», науковий керівник дисертації Хіцко Яна Володимирівна, к.т.н., ст. викладач, затверджені наказом по університету від «08» листопада 2024 р. №5007-с.

2. Термін подання студентом дисертації «17» грудня 2024 р.

3. Об’єкт дослідження: Процес обробки даних серверною частиною застосунку та базою даних.

4. Предмет дослідження: Методи, способи та програмні засоби збільшення швидкості обробки запитів системою управління персоналом підприємства.

5. Перелік завдань, які потрібно розробити:
   - аналіз існуючих методів та програмних рішень для управління персоналом, виявлення їхніх переваг та недоліків;
   - розробка методу для збільшення швидкодії системи управління персоналом підприємства з урахуванням сучасних потреб підприємств;
   - реалізація програмного забезпечення для методу для збільшення швидкодії системи управління персоналом підприємства;
   - проведення аналізу результатів, отриманих за допомогою розробленого програмного забезпечення;
   - розробка бізнес-моделі.

6. Орієнтовний перелік графічного (ілюстративного) матеріалу:
   - схема архітектури програмного забезпечення;
   - схема бази даних;
   - схема роботи пула підключень;

- діаграма класів розробленої бібліотеки;
- графік порівняння швидкості розробленого підходу та аналогів;
- дерево проблем.

7. Орієнтовний перелік публікацій:
   - Тези доповіді “Метод для збільшення швидкодії обробки запитів серверною частиною застосунку”

8. Консультанти розділів дисертації

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Нормоконтроль	Онай М.В., доцент кафедри ПЗКС		

9. Дата видачі завдання «04» жовтня 2023 р.

Календарний план

№ з/п	Назва етапів виконання магістерської дисертації	Термін виконання етапів магістерської дисертації	Примітка
1.	Ґрунтовне ознайомлення з предметною галуззю	17.10.2023	
2.	Визначення структури магістерської дисертації; вивчення літератури, пошук додаткової літератури, патентний пошук	04.12.2023	
3.	Робота над першим розділом магістерської дисертації; проведення наукового дослідження	15.02.2024	
4.	Проведення наукового дослідження; робота над другим розділом магістерської дисертації; розроблення програмного забезпечення	05.04.2024	
5.	Проведення наукового дослідження; робота над статтею за результатами наукового дослідження	15.05.2024	
6.	Проведення наукового дослідження; робота над третім розділом магістерської дисертації	14.06.2024	
7.	Завершення роботи над основною частиною магістерської дисертації; підготовка ілюстративного матеріалу; підготовка матеріалів доповіді на конференції ПМК-2024	05.11.2024	
8.	Оформлення текстової і графічної частини магістерської дисертації	04.12.2024	

Студент Євгеній ПОВАРОВ

Науковий керівник Яна ХІЦКО

РЕФЕРАТ

Актуальність теми: актуальність теми дослідження обумовлена сучасними тенденціями розвитку інформаційних технологій і необхідністю підвищення ефективності управління персоналом на підприємствах. В умовах зростаючої конкуренції, підприємства стикаються з вимогою швидкої адаптації до змін на ринку, що потребує ефективного управління ресурсами, зокрема людськими. Розробка та впровадження методів і програмного забезпечення, спрямованих на збільшення швидкодії системи управління персоналом, є критично важливими для забезпечення конкурентоспроможності підприємств.

Зростаюча складність бізнес-процесів і великий обсяг даних вимагають використання сучасних інформаційних технологій, здатних оптимізувати процеси прийняття рішень, автоматизувати рутинні завдання та забезпечити високу продуктивність роботи всієї системи управління персоналом. Таким чином, дослідження в цьому напрямку є актуальним і відповідає нагальним потребам сучасного бізнесу.

Об’єкт дослідження: процес обробки даних серверною частиною застосунку та базою даних.

Предмет дослідження: методи, способи та програмні засоби збільшення швидкості обробки запитів системою управління персоналом підприємства.

Мета роботи: підвищити швидкість обробки запитів серверною стороною застосунку за рахунок нового підходу до розробки ORM фреймворку, з поєднанням високорівневих та низькорівненвих можливостей мови програмування C#.

Методи досліджень: теоретичний та емпіричний аналіз існуючих методів і сервісів для управління персоналом підприємства.

Наукова новизна: вперше запропоновано модифікований спосіб написання ORM фреймворку, характерними рисами якого є використання

низькорівневих конструкцій мови C# та апаратного прискорення системи, що дозволяє пришвидшити обробку даних сервером в середньому до 27%.

Практична цінність: роботи полягає в імплементації нового методу обробки запитів у програмному забезпеченні, який використовує ORM-фреймворк на мові C#. Цей метод поєднує низькорівневі конструкції мови програмування з апаратним прискоренням системи, що забезпечує значне підвищення продуктивності серверної частини додатка.

Апробація роботи: Основні положення і результати роботи представлені та обговорені на XVII науковій конференції магістрантів та аспірантів «Прикладна математика та комп'ютинг» ПМК-2024.

Структура та обсяг роботи: Магістерська дисертація складається з вступу, п'яти основних розділів, висновків та додатків.

У вступі надано огляд дослідження, обґрунтовано актуальність теми, сформульовано мету і задачі, підкреслено наукову новизну та практичну значущість отриманих результатів, а також наведено інформацію про апробацію та впровадження результатів дослідження.

У першому розділі розглянуто теоретичні аспекти та сучасні тенденції у сфері управління персоналом, а також проведено аналіз існуючих підходів до оптимізації систем управління персоналом на підприємствах.

У другому розділі описано метод для підвищення швидкості обробки даних серверною частиною системи управління персоналом, зокрема, детально розглянуто підходи до оптимізації ресурсів і поліпшення продуктивності через впровадження сучасних технологій.

Третій розділ охоплює технічні аспекти розробки фреймворку, включаючи вибір інструментів та технологій.

Четвертий розділ містить аналіз результатів, отриманих від реалізації ORM фреймворку, із проведенням випробувань на різних наборах даних.

П'ятий розділ зосереджується на розробці бізнес-моделі для стартапу, включаючи аналіз ринку, виявлення зацікавлених сторін, формулювання

унікальної ціннісної пропозиції, а також розрахунок та систематизацію доходів і витрат.

У висновках підбито підсумки проведеної роботи, висвітлено ключові здобутки та перспективи подальшого розвитку проєкту.

Робота виконана на 80 аркушах, містить 3 додатки та посилання на список використаних літературних джерел з 24 найменувань. У роботі наведено 10 рисунків та 18 таблиць.

Ключові слова: програмне забезпечення, ORM фреймворк, апаратне прискорення, оптимізація.

ABSTRACT

Relevance of the topic: the relevance of this research is driven by modern trends in the development of information technology and the need to enhance the efficiency of personnel management in enterprises. In the context of increasing competition, businesses face the demand for rapid adaptation to market changes, which requires effective resource management, particularly human resources. The development and implementation of methods and software aimed at increasing the speed of personnel management systems are critical to ensuring the competitiveness of enterprises.

The growing complexity of business processes and the large volume of data require the use of modern information technologies capable of optimizing decision-making processes, automating routine tasks, and ensuring high performance of the entire personnel management system. Therefore, research in this area is relevant and addresses the pressing needs of modern business.

Research object: the process of data processing by the server-side application and the database.

Research subject: methods, approaches, and software tools for increasing the speed of request processing in the personnel management system of an enterprise.

Purpose of the work: increase the speed of request processing on the server side of the application through a new approach to ORM framework development, combining high-level and low-level capabilities of the C# programming language.

Research methods: theoretical and empirical analysis of existing methods and services for enterprise personnel management.

Scientific novelty: for the first time, a modified method for writing an ORM framework is proposed, characterized by the use of low-level C# constructs and hardware acceleration, which allows the server to speed up data processing by an average of 27%.

Practical value of the work: the practical value lies in the implementation of a new request processing method within the software, utilizing an ORM framework in C#. This method integrates low-level programming constructs with system hardware acceleration, providing a significant performance boost to the server-side of the application.

Approbation: the main provisions and results of the work were presented and discussed at the XVII Scientific Conference of Master's and PhD Students "Applied Mathematics and Computing" PMC-2024.

Structure and scope of the work: the master's thesis consists of an introduction, five main chapters, conclusions, and appendices.

The introduction provides an overview of the research, justifies the relevance of the topic, formulates the purpose and objectives, highlights the scientific novelty and practical significance of the results obtained, and provides information on the approval and implementation of the research findings.

The first chapter examines the theoretical aspects and current trends in the field of personnel management, as well as analyzes existing approaches to optimizing personnel management systems in enterprises.

The second chapter describes a method for increasing the speed of data processing by the server-side of the personnel management system, detailing approaches to resource optimization and performance improvement through the implementation of modern technologies.

The third chapter covers the technical aspects of framework development, including the selection of tools and technologies.

The fourth chapter analyzes the results obtained from the implementation of the ORM framework, with tests conducted on various data sets.

The fifth chapter focuses on developing a startup's business model, including market analysis, identification of stakeholders, formulation of a unique value proposition, and calculation and systematization of revenues and expenses.

The conclusions summarize the work carried out, highlight key achievements, and outline the prospects for further project development.

The work comprises 80 pages, contains 3 appendices, and references 24 sources of literature. It includes 10 figures and 18 tables.

Keywords: software, ORM framework, hardware acceleration, optimization.

ЗМІСТ

СПИСОК ТЕРМІНІВ, СКОРОЧЕНЬ ТА ПОЗНАЧЕНЬ

API (Application Programming Interface) – інтерфейс прикладного програмування; набір функцій та протоколів для взаємодії між програмними компонентами.

ERP (Enterprise Resource Planning) – планування ресурсів підприємства; інтегрована система для управління всіма основними бізнес-процесами, такими як фінанси, виробництво, постачання, облік персоналу та інше в межах підприємства.

GPU (Graphics Processing Unit) – графічний процесор; апаратний компонент, оптимізований для паралельних обчислень.

HRM (Human Resource Management) – управління людськими ресурсами; система та процеси, пов'язані з управлінням персоналом організації.

HTML (HyperText Markup Language) – мова гіпертекстової розмітки, стандартна мова розмітки для створення веб-сторінок.

HTTP (Hypertext Transfer Protocol) – протокол передачі гіпертексту; використовується для обміну даними між веб-клієнтами та серверами.

JSON (JavaScript Object Notation) – текстовий формат обміну даними, що часто використовується у веб-додатках.

ORM (Object-Relational Mapping) – об'єктно-реляційне відображення; методологія для сполучення об'єктів програмування з реляційними базами даних.

SQL (Structured Query Language) – мова структурованих запитів; стандартна мова для роботи з реляційними базами даних.

ПЗ (Програмне Забезпечення) – загальний термін, що відноситься до програм і даних, які забезпечують функціональність комп'ютерам.

СУБД (Система управління базами даних) – програмне забезпечення для створення, управління та використання баз даних.

ВСТУП

У сучасному світі, де цифрові технології розвиваються швидкими темпами, проблема підвищення ефективності обробки даних є надзвичайно актуальною. Одним із ключових аспектів оптимізації інформаційних систем є вдосконалення процесів управління персоналом, оскільки ефективне використання людських ресурсів є основою конкурентоспроможності підприємств. Сучасні організації стикаються з необхідністю швидкої адаптації до змін на ринку, що передбачає потребу в інтегрованих і високопродуктивних системах управління. Це завдання ускладнюється величезними обсягами даних, що потребують обробки, а також зростаючою складністю бізнес-процесів. В умовах таких викликів традиційні підходи до обробки даних не завжди здатні забезпечити необхідну швидкість і ефективність. Тому розробка методів, що дозволяють підвищити продуктивність обробки запитів у системах управління персоналом, стає важливою складовою для підвищення ефективності бізнес-процесів.

Метою цієї роботи є розробка нових підходів до оптимізації обробки запитів у системах управління персоналом за допомогою модифікації ORM-фреймворку. Запропонований підхід поєднує використання низькорівневих конструкцій мови програмування C# та технологій апаратного прискорення для значного підвищення швидкості обробки даних. Цей метод дозволяє скоротити час, необхідний для обробки великих обсягів інформації, що є важливим фактором для ефективного управління ресурсами підприємства.

Об'єктом дослідження є процес обробки даних серверною частиною застосунку та базою даних, а предметом дослідження — методи, способи та програмні засоби підвищення швидкості обробки запитів в системах управління персоналом. На основі аналізу сучасних підходів до розробки ORM-фреймворків, запропоновано новий метод, який дозволяє оптимізувати цей процес і підвищити продуктивність серверної частини.

1. АНАЛІЗ ТЕМИ ТА ПОШУК РІШЕННЯ ПОСТАВЛЕНОЇ ЗАДАЧІ

1.1. Огляд наявних підходів до управління персоналом підприємства

Сучасні підходи до управління персоналом підприємств значною мірою змінилися завдяки впровадженню інформаційних технологій. Зі зростанням обсягів бізнесу і збільшенням чисельності персоналу, традиційні методи управління стають менш ефективними. Сьогодні багато великих компаній активно використовують електронні системи управління персоналом, які дозволяють автоматизувати значну частину адміністративних процесів.

Електронні системи управління персоналом (HRM-системи) забезпечують ефективний облік, аналіз та управління даними про працівників. Вони інтегрують в собі функції, які раніше виконувалися окремо, такі як облік робочого часу, розрахунок заробітної плати, управління відпустками, навчання персоналу тощо. Це дозволяє знизити навантаження на HR-відділи та покращити точність і своєчасність обробки інформації. З огляду на великі обсяги даних, які генеруються в процесі управління великими колективами, з'являється необхідність у спеціалізованих програмних рішеннях.

Сучасні HRM-системи використовують бази даних, хмарні технології та аналітичні інструменти для зберігання, обробки та аналізу інформації про працівників. Це забезпечує можливість створення детальних звітів та прогнозів, які допомагають керівництву приймати обґрунтовані рішення. Застосування електронних систем управління персоналом дозволяє не лише оптимізувати внутрішні процеси підприємства, але й підвищити рівень задоволеності працівників. Завдяки автоматизації рутинних задач працівники HR-відділів можуть зосередитися на стратегічних аспектах управління, таких як розвиток талантів, корпоративна культура, мотивація персоналу та інших можливих векторах розвитку.

Таким чином, сучасні підходи до управління персоналом підприємств базуються на використанні електронних систем, що дозволяють ефективно обробляти великі обсяги даних, автоматизувати адміністративні процеси та підвищити загальну ефективність управління.

1.2. Аналіз існуючих систем

Управління персоналом є критичним елементом успішного функціонування сучасних підприємств. З ростом конкуренції на ринку, здатність компанії ефективно керувати своїм кадровим потенціалом стає важливим чинником для досягнення стратегічних цілей та забезпечення конкурентних переваг. Традиційні методи управління персоналом, які включають паперові документообіги та ручну обробку даних, вже не відповідають вимогам сучасного бізнесу, особливо у великих організаціях з численним штатом працівників.

Розвиток інформаційних технологій суттєво вплинув на управління персоналом, відкривши нові можливості для автоматизації процесів та підвищення ефективності роботи HR-відділів. Впровадження електронних систем управління персоналом (HRM-систем) дозволяє значно спростити облік, аналіз та управління даними про працівників, що в свою чергу підвищує загальну продуктивність підприємства. Сьогодні підприємства різного масштабу активно впроваджують HRM-системи, які забезпечують інтеграцію різних функцій управління персоналом, таких як облік робочого часу, розрахунок заробітної плати, управління відпустками та навчання персоналу. Ці системи дозволяють знизити адміністративні витрати, мінімізувати людський фактор у процесі обробки даних та приймати обґрунтовані управлінські рішення на основі аналізу великих обсягів інформації. Таким чином, впровадження інформаційних технологій в управління персоналом є невід'ємною складовою успішного розвитку сучасних підприємств. Автоматизація HR-процесів та використання електронних систем управління персоналом дозволяють не лише підвищити

ефективність роботи HR-відділів, але й створити сприятливі умови для розвитку кадрового потенціалу компанії.

1.2.1. Типи систем управління персоналом

Існує два типи систем управління персоналом: традиційні та електронні [1]. Традиційні методи управління персоналом включають ручне ведення обліку, паперову документацію та використання простих програм, таких як електронні таблиці. Ці методи зазвичай характеризуються високою трудомісткістю, низькою ефективністю та схильністю до помилок. Вони підходять для малих організацій з обмеженою кількістю персоналу, але стають недієвими в міру зростання компанії. Електронні системи управління персоналом, або HRM-системи, являють собою програмні рішення, призначені для автоматизації та оптимізації процесів управління персоналом. Вони дозволяють централізовано зберігати дані про співробітників, автоматизувати обробку кадрової документації, покращувати комунікацію між співробітниками та керівництвом, а також сприяти підвищенню продуктивності праці.

Окрім базових функцій, електронні системи управління персоналом також надають можливості для аналізу даних та прогнозування. За допомогою аналітичних інструментів HRM-системи дозволяють відстежувати ключові показники ефективності (KPI) [2], проводити оцінку продуктивності співробітників та аналізувати тренди в управлінні персоналом. Це дає можливість керівникам приймати обґрунтовані рішення щодо розвитку співробітників, планування навчання та кар'єрного зростання. Сучасні HRM-системи часто інтегруються з іншими бізнес-додатками, такими як системи фінансового управління, ERP-системи та платформи для управління проєктами. Це забезпечує узгодженість даних між різними департаментами компанії, знижує кількість дублювань інформації та покращує загальну ефективність бізнес-процесів. Інтеграція також дозволяє автоматизувати складні завдання, такі як розрахунок заробітної плати, управління відпустками та оцінка відповідності

співробітників нормативним вимогам. Крім того, HRM-системи часто включають функціональність для підвищення залученості та мотивації співробітників. Це можуть бути інструменти для управління винагородами, системи зворотного зв'язку та опитування задоволеності працівників. Такі функції допомагають створити більш позитивне та продуктивне робоче середовище, де співробітники відчувають свою цінність та можуть активно брати участь у житті компанії.

1.2.2. Огляд основних HRM-систем

На сучасному ринку існує велика кількість HRM-систем, кожна з яких має свої особливості та орієнтована на різні сегменти бізнесу. Серед найвідоміших HRM-систем можна виділити SAP SuccessFactors, Workday, Oracle HCM Cloud, BambooHR та ADP Workforce Now [3], кожна з яких пропонує унікальний набір функцій та інструментів для управління людськими ресурсами.

SAP SuccessFactors є потужною платформою, яка охоплює широкий спектр HRM-функцій. Ця система надає інструменти для управління талантами, навчання, рекрутингу та управління продуктивністю. Вона орієнтована на великі корпорації, де потрібна інтеграція різних бізнес-процесів. Завдяки своєму широкому функціоналу, SAP SuccessFactors дозволяє компаніям ефективно управляти персоналом, забезпечуючи можливості для розвитку та оптимізації кадрових ресурсів. Проте, через складність налаштувань і високу вартість, ця система може бути недоступною для менших компаній.

Workday є хмарною платформою, яка не тільки охоплює HRM-функції, але й включає в себе фінансове управління та управління талантами. Основний акцент Workday робить на інтеграції та аналізі даних, що дозволяє підприємствам отримувати вичерпну інформацію для прийняття стратегічних рішень. Ця система також орієнтована на великі підприємства, які потребують комплексного управління не лише

персоналом, але й фінансовими потоками. Завдяки високій зручності використання та потужним аналітичним можливостям, Workday стає важливим інструментом для компаній, що прагнуть досягти максимальної ефективності.

Oracle HCM Cloud також є потужним інструментом для управління персоналом, пропонуючи широкий спектр функцій, включаючи рекрутинг, навчання, управління компенсаціями та аналітику. Oracle HCM Cloud, подібно до SAP SuccessFactors, орієнтована на великі корпорації, які потребують інтеграції HRM-функцій з іншими бізнес-системами, такими як ERP. Ця платформа забезпечує високу продуктивність і гнучкість, дозволяючи компаніям адаптувати HRM-процеси до своїх специфічних потреб, хоча, як і інші великі платформи, вона може бути досить складною у впровадженні та дороговартісною.

Для малих та середніх компаній, які шукають прості та зручні у використанні HRM-рішення, відмінним вибором може стати BambooHR. Ця система є популярною завдяки своїй простоті у використанні та широкому набору базових функцій, таких як управління персоналом, рекрутинг, навчання та розвиток співробітників. BambooHR орієнтована на малі та середні підприємства, які не потребують складних інтеграцій або розширених функцій, але цінують легкість налаштування та використання системи. Крім того, ця платформа є більш доступною за ціною порівняно з іншими великими HRM-системами.

ADP Workforce Now є комплексним рішенням для управління персоналом, яке включає модулі для управління зарплатою, рекрутингом, навчанням та розвитком співробітників. Ця система також орієнтована на малі та середні компанії, пропонуючи високу зручність використання та швидку адаптацію без значних витрат на впровадження. ADP Workforce Now є особливо корисною для компаній, які потребують ефективного управління великим обсягом даних, але не готові інвестувати в складні та дорогі HRM-системи. Вона забезпечує базові функції управління

персоналом, що дозволяє компаніям зосередитися на своєму основному бізнесі.

Розглядаючи функціональні можливості цих HRM-систем, можна відзначити, що SAP SuccessFactors, Workday та Oracle HCM Cloud пропонують повний спектр функцій для управління талантами, навчання та розвитку, рекрутингу, управління компенсаціями та аналітики. Вони також мають інтеграцію з ERP-системами, що робить їх ідеальними для великих корпорацій з комплексними потребами. Водночас, BambooHR та ADP Workforce Now надають лише основні функції управління талантами та компенсаціями, що робить їх більш підходящими для невеликих компаній.

Вибір HRM-системи залежить від потреб конкретної компанії. Великі корпорації, які потребують потужної інтеграції та широкого функціоналу, ймовірно, віддадуть перевагу SAP SuccessFactors, Workday або Oracle HCM Cloud. Малі та середні підприємства, які цінують зручність використання та швидке впровадження, можуть обрати BambooHR або ADP Workforce Now, що забезпечують необхідний набір функцій за більш доступною ціною.

1.2.3. Інтеграція HRM-систем з ERP-системами

Інтеграція HRM-систем з ERP-системами є важливим кроком для забезпечення комплексного управління бізнес-процесами в організації. ERP-системи (Enterprise Resource Planning) включають в себе широкі функціональні можливості для управління різними аспектами діяльності підприємства, такими як фінанси, виробництво, постачання та інші. Об'єднання HRM-систем з ERP дозволяє централізувати управління кадровими процесами, знизити кількість дублювань даних та забезпечити ефективний обмін інформацією між різними департаментами компанії. Така інтеграція не тільки покращує координацію та продуктивність, але й сприяє прийняттю більш обґрунтованих управлінських рішень на основі інтегрованих даних.

Інтеграція HRM-систем з ERP-системами має свої переваги та виклики. Зокрема, вона дозволяє автоматизувати рутинні завдання, що значно покращує ефективність роботи HR-відділу. Завдяки цьому, також покращується точність даних і підвищується прозорість та комунікація між різними відділами. Однак, незважаючи на ці переваги, процес впровадження інтеграції може бути досить складним і дорогим. Висока вартість впровадження, складність інтеграції з іншими системами, а також необхідність навчання персоналу є основними викликами, з якими стикаються компанії. Крім того, можуть виникати проблеми з адаптацією до нових підходів, що потребують часу та зусиль для їх подолання.

Отже, проаналізувавши таблицю можемо сказати наступне про інтеграцію HRM-систем з ERP-системами. Інтеграція HRM-систем з ERP-системами забезпечує централізоване управління даними, дозволяючи зберігати всю інформацію про співробітників в одній системі. Це значно полегшує доступ до даних, підвищує точність інформації та сприяє кращій координації між різними відділами компанії. Крім того, така інтеграція підвищує ефективність роботи за рахунок автоматизації рутинних процесів, таких як нарахування зарплат, обробка заявок на відпустку та управління навчанням. Це дозволяє знизити навантаження на HR-відділ і зменшити ймовірність помилок. Об'єднання даних з HRM та ERP-систем також покращує аналітику та звітність, дозволяючи отримувати більш точні та детальні звіти для прийняття обґрунтованих управлінських рішень. Підвищення задоволеності співробітників є ще однією важливою перевагою, оскільки системи дозволяють працівникам самостійно управляти своїми даними, що сприяє прозорості процесів та підвищенню довіри до компанії.

Основні переваги для підприємства включають автоматизацію рутинних завдань, що дозволяє знизити трудовитрати на виконання повторюваних процесів, таких як обробка заявок на відпустку та управління записами співробітників. Це також допомагає покращити точність даних

завдяки автоматизованим процесам та централізованому зберіганню інформації. Підвищення ефективності роботи HR-відділу стає можливим завдяки тому, що спеціалісти можуть зосередитися на стратегічних завданнях, таких як розвиток талантів та управління змінами. Крім того, покращується комунікація та прозорість, оскільки співробітники отримують доступ до своїх даних та можливість самостійно ними управляти, що підвищує довіру до системи управління.

Впровадження HRM-систем не обходиться без викликів та проблем. Висока вартість впровадження може бути значним бар'єром, особливо для малих та середніх компаній, які не завжди мають достатні ресурси. Складність інтеграції HRM-систем з існуючими ERP-системами або іншими програмними продуктами може вимагати значних ресурсів та часу як зі сторони розробників та і зі сторони кінцевого користувача, що ускладнює процес впровадження. Подальша підтримка продуктів такого масштабу може виявитись викликом, оскільки теж потребуватиме ресурсів. Необхідність навчання персоналу є ще однією проблемою, оскільки нова система потребує часу для освоєння, що може призвести до тимчасового зниження продуктивності. Проблеми з адаптацією також можуть виникати, оскільки не всі співробітники швидко адаптуються до нових технологій та процесів, що може викликати опір змінам.

1.3. Обґрунтування необхідності нового підходу

Як було зазначено в попередніх пунктах, найбільша очевидна вигода від впровадження HRM-систем та їх інтеграції ERP-системами для великих компаній з великим штатом співробітників. В цьому випадку витрати ресурсів та часу на впровадження та підтримку повністю компенсуються покращенням процесів та екеномічною доцільністю. Проте, як вже було зазначено, чим більша компанія, тим більше штат співробітників, а отже і більші об’єми даних які потрібно обробляти в рамках виконання рутинних операцій. З часом об’єм даних не зменшується, оскільки відбувається

розширення компанії, або структурні зміни в рамках яких додаються нові елементи, які потрібно буде враховувати. Це призводить до того що в певний момент часу виконання задач займає більше часу ніж бізнес вважає прийнятним. Для вирішення даної проблеми необхідно детальніше розглянути її першоджерела.

Розробка систем подібних за масштабами до ERP потребує використання мов програмування, що мають відповідну швикодію та набір інструментів, наприклад C#, JAVA та інші. Найчастіше в такому випадку для взаємодії з базою даних використовуються ORM фреймворки, які цінуються за їх зручність для розробників, проте при цьому швидкодія системи знижується через особливості написання ядра та їх роботи.

З розвитком технологій та появою нових інструментів мови програмування отримують більше як високорівневих, для збільшення комфорту та швидкості розробки, так і низькорівневих можливостей для покращення швидкодії виконання написаного коду. Впровадження цих нових елементів та підходів при розробці ORM фреймворків для використання в ERP та HRM системах і дозволить підвищити їх швидкодію при роботі з великими об’ємами даних.

1.4. Висновки до розділу

У цьому розділі проведено аналіз сучасних підходів до управління персоналом підприємств, зокрема вивчено роль інформаційних технологій у цій сфері. Розглянуто ключові типи HRM-систем, їх функціональні можливості та інтеграцію з ERP-системами. Зроблено висновок, що традиційні методи управління персоналом є недостатньо ефективними для великих організацій, оскільки вони не можуть забезпечити оперативну обробку великих обсягів даних, необхідних для прийняття стратегічних рішень.

Аналіз наявних рішень показав, що хоча наявні системи не завжди відповідають потребам сучасних компаній у швидкості обробки даних.

Зокрема, проблеми з продуктивністю систем під час обробки великих обсягів інформації, а також виклики, пов’язані з високими витратами на впровадження та інтеграцію таких рішень, потребують удосконалення технологічного підходу.

Виявлено, що розробка систем на основі сучасних мов програмування та застосування оптимізованих ORM-фреймворків є перспективним напрямком для підвищення продуктивності. Це дозволяє не тільки вирішити проблему повільної обробки великих масивів даних, а й забезпечити ефективну інтеграцію HRM-систем з іншими бізнес-додатками. Нові підходи до проєктування ORM-фреймворків, що враховують можливості сучасних мов програмування, мають стати основою для створення більш продуктивних та гнучких рішень у сфері управління персоналом.

Таким чином, проведений аналіз підтвердив необхідність створення нового підходу до розробки HRM-систем, який буде орієнтований на подолання існуючих обмежень та забезпечення високої швидкодії, ефективності та адаптивності у роботі з великими обсягами даних.

2. ОПИС МЕТОДУ ДЛЯ ПІДВИЩЕННЯ ШВИДКОСТІ ОБРОБКИ ДАНИХ СЕРВЕРНОЮ ЧАСТИНОЮ ЗАСТОСУНКУ

2.1. Аргументація обраного підходу до розробки

У розробці ERP та HRM систем для великих підприємств ключовим завданням є забезпечення високої продуктивності та масштабованості без значного ускладнення архітектури. Обраний підхід, що базується на використанні ORM фреймворків у поєднанні з мовою програмування C#, є оптимальним для досягнення цих цілей. Це рішення ґрунтується на кількох важливих аспектах, які відрізняють його від традиційних підходів.

ORM (Object-Relational Mapping) фреймворки, такі як Entity Framework Core (EF Core), забезпечують високий рівень абстракції над базами даних. Вони дозволяють розробникам працювати з даними як з об'єктами, автоматично перетворюючи їх у SQL-запити [7]. Це значно спрощує розробку, дозволяючи зосередитися на бізнес-логіці замість технічних деталей роботи з базою даних. EF Core пропонує ряд зручностей, таких як автоматичне відображення об'єктів на таблиці баз даних, відстеження змін у об'єктах, а також підтримку асинхронних операцій, що дозволяє ефективніше використовувати ресурси системи. Однак, попри ці переваги, традиційні ORM рішення, включаючи EF Core, мають кілька суттєвих недоліків, особливо при роботі з великими обсягами даних і високими вимогами до швидкодії.

У сучасних ORM фреймворків для мови C#, хоча це стосується і інших мов програмування існують певні недоліки та обмеження які варто враховувати при їх використанні. Більшість ORM фреймворків, включаючи EF Core, активно використовують пам'ять для відстеження стану об'єктів, створення і знищення об'єктів під час виконання операцій. Це створює додаткове навантаження на збирач сміття (Garbage Collector) і може призводити до затримок під час обробки великих наборів даних. EF Core автоматично генерує SQL-запити на основі LINQ-запитів, написаних розробниками. Однак ці автоматично згенеровані запити не завжди є

оптимальними, особливо у випадках складних або специфічних запитів. Це може призводити до підвищеного навантаження на базу даних і зниження продуктивності системи. У системах із великим навантаженням стандартні підходи ORM до обробки даних можуть бути недостатньо ефективними. Це особливо актуально для ERP та HRM систем, де обробка великих обсягів даних та підтримка високої швидкодії є критично важливими.

EF Core потребує певних змін для підвищення його ефективності в умовах високих навантажень та великих обсягів даних. Основні напрямки в яких EF Core потребує оптимізації:

Накладні витрати на відстеження змін: Хоча відстеження змін спрощує роботу з об'єктами, воно також створює додаткове навантаження на систему, особливо при роботі з великими наборами даних. Відстеження змін вимагає додаткових алокацій пам'яті і може знижувати продуктивність у масштабних системах.

Неоптимальні SQL-запити: Автоматично генеровані EF Core SQL-запити не завжди оптимальні. У випадках складних або дуже специфічних запитів фреймворк може створювати зайві, або менш ефективні запити, що призводить до зниження продуктивності бази даних.

Використання пам'яті: EF Core активно використовує пам'ять для відстеження стану об'єктів і керування ними, що може призводити до надмірного навантаження на систему, особливо у випадках, коли обробляються великі обсяги даних або є високий рівень конкуренції за ресурси.

Обмежені можливості для низькорівневої оптимізації: Хоча EF Core і надає високий рівень абстракції, він обмежує можливість виконання низькорівневих оптимізацій, які могли б значно підвищити продуктивність. Це особливо актуально у випадках, коли йдеться про обробку критично важливих для бізнесу даних або необхідність досягнення максимальної продуктивності.

Таким чином, обраний підхід до розробки, буде поєднувати переваги ORM фреймворків та сучасні можливості C#, забезпечує високу продуктивність, масштабованість та стабільність роботи систем. Це дозволяє ефективно вирішувати проблеми, пов'язані з обробкою великих обсягів даних, зберігаючи при цьому зручність та ефективність розробки.

2.2. Запропоновані підходи до роботи з ресурсами системи

Для подолання обмежень сучасних ORM-фреймворків та досягнення максимальної продуктивності системи запропоновано кілька важливих змін. Ці підходи спрямовані на оптимізацію використання пам'яті, покращення ефективності обробки даних та вдосконалення генерації SQL-запитів. У цьому розділі докладно розглянуто кожен із запропонованих механізмів.

2.2.1. Оптимізація роботи з пам'яттю

Ефективне управління пам'яттю є ключовим фактором для забезпечення високої продуктивності сучасних систем, особливо під час роботи з великими обсягами даних. Основною проблемою багатьох ORM-фреймворків залишається надмірне споживання пам'яті, що призводить до втрат швидкості та стабільності роботи. Для вирішення цього виклику пропонується впровадження сучасних інструментів, таких як Span<T>, Memory<T> і ArrayPool<T>, які дозволяють мінімізувати зайві алокації пам'яті, скоротити кількість викликів збирача сміття та оптимізувати використання системних ресурсів [8].

Span<T> і Memory<T> забезпечують значне підвищення ефективності роботи з буферами та блоками пам'яті. Span<T> дозволяє працювати з ділянками пам'яті без створення додаткових об'єктів, що скорочує витрати на виділення нових ресурсів і зменшує фрагментацію пам'яті. Цей підхід особливо корисний у ситуаціях, де потрібно опрацьовувати дані у фіксованих межах. У свою чергу, Memory<T> доповнює функціональність Span<T>, додаючи можливість виконувати асинхронні операції, що робить

його ідеальним для багатопотокового середовища, де обробка даних здійснюється паралельно. ArrayPool<T> – орієнтований на повторне використання масивів, що значно зменшує витрати на створення і видалення об’єктів пам’яті. Це особливо ефективно в умовах високих навантажень, коли програма працює з великою кількістю одночасних запитів. Масиви виділяються з пулу, використовуються для обробки завдань і повертаються назад, що дозволяє уникнути перевантаження системи частими операціями виділення пам’яті. Таким чином, знижується навантаження на збирач сміття, а програма працює більш стабільно навіть під час пікових навантажень.

Практична інтеграція цих інструментів включає оптимізацію ключових ділянок бізнес-логіки, де обробляються найбільші обсяги даних. Використання Span<T> у методах обробки буферів або застосування ArrayPool<T> для роботи з тимчасовими масивами дозволяє досягти максимального ефекту за мінімальних змін у коді.

Оптимізація роботи з пам’яттю таким чином дозволяє забезпечити баланс між продуктивністю та ефективністю використання ресурсів. Завдяки сучасним підходам до управління пам’яттю система стає більш надійною та готовою до роботи навіть у складних умовах із великими обсягами даних і високими навантаженнями.

2.2.2. Поліпшення генерації SQL-запитів

Автоматична генерація SQL-запитів, яку забезпечують сучасні ORM-фреймворки, часто спрощує розробку, проте нерідко призводить до створення неефективних запитів [4]. Це може викликати надмірне навантаження на базу даних, збільшувати час виконання операцій і знижувати загальну продуктивність системи. Для вирішення цієї проблеми пропонується кілька підходів, спрямованих на вдосконалення процесу генерації SQL-запитів.

Одним із ключових кроків є надання розробникам інструментів для гнучкого втручання у процес створення запитів. Це може бути реалізовано

шляхом інтеграції механізмів кастомізації, тобто можливості змінювати структуру запитів залежно від специфіки завдань, забезпечуючи оптимальне використання ресурсів бази даних. Наприклад, розробник може налаштувати вибір лише необхідних полів у складних запитах, або виконати об'єднання декількох операцій в один пакетний запит дозволяє мінімізувати кількість звернень до бази даних і скоротити мережеві затримки.

У результаті, оптимізація генерації SQL-запитів дозволяє скоротити час їх виконання, зменшити навантаження на базу даних і забезпечити стабільну роботу системи навіть за умов високих обсягів запитів.

2.2.3. Інтеграція апаратного прискорення

Впровадження апаратного прискорення, зокрема використання графічних процесорів (GPU), є ефективним методом підвищення продуктивності систем під час обробки великих обсягів даних і виконання складних обчислень. GPU, завдяки своїй архітектурі з великою кількістю обчислювальних ядер, забезпечує можливість паралельного виконання багатьох операцій одночасно. Це робить їх незамінними для задач, що потребують високої продуктивності, таких як обробка SQL-запитів, складний статистичний аналіз і генерація звітів.

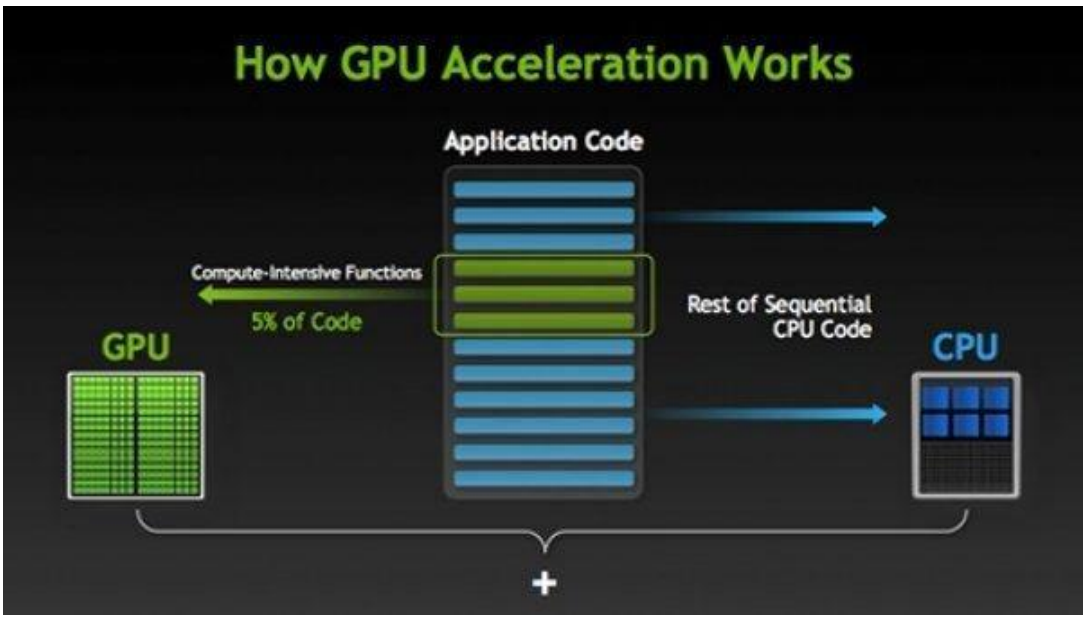


Рис 2.1. Принцип роботи апаратного прискорення [9]

Однією з основних переваг використання GPU є здатність обробляти великі масиви даних у режимі паралельних операцій. Наприклад, типові SQL-запити, що включають сортування, агрегацію або фільтрацію, значно прискорюються завдяки розподілу цих операцій на блоки. Кожен блок може виконуватись окремо на різних ядрах GPU, що суттєво скорочує загальний час виконання запиту. У випадках, коли стандартна обробка на центральному процесорі (CPU) потребує значного часу, використання GPU дозволяє досягти в десятки разів кращих показників швидкості. Окрім оптимізації обробки SQL-запитів, GPU є надзвичайно корисними для задач, що включають математичні розрахунки чи складний статистичний аналіз. У багатьох випадках ці обчислення можна розділити на частини, кожна з яких буде оброблена окремо за допомогою потоків GPU. Наприклад, під час генерації звітів або виконання фінансових моделей паралельна обробка дозволяє отримати результати значно швидше, ніж за використання лише CPU.

Інтеграція GPU також передбачає адаптацію архітектури програмного забезпечення для забезпечення максимальної продуктивності. Це включає написання коду з урахуванням специфіки обчислень на GPU, оптимізацію алгоритмів для ефективного використання ресурсів і налаштування обробки даних так, щоб вони швидко передавались між CPU і GPU без затримок. Завдяки поєднанню сучасних технологій GPU із можливостями ORM-фреймворків стає можливим створення систем, здатних обробляти надзвичайно великі обсяги даних із мінімальними витратами часу. Такий підхід дозволяє системам залишатись стабільними та продуктивними навіть за умов пікових навантажень, забезпечуючи високу конкурентоспроможність і ефективність роботи.

2.2.4. Адаптивне управління пулами з'єднань

Ефективне управління з'єднаннями з базою даних є важливим аспектом для забезпечення високої продуктивності систем, особливо у

випадках, коли кількість одночасних користувачів або запитів до бази значно зростає. Запропонований підхід базується на адаптивному управлінні пулами з’єднань, що дозволяє оптимізувати використання ресурсів і уникнути вузьких місць у роботі системи. Кожне нове з’єднання з базою даних потребує певних ресурсів і часу для ініціалізації [3]. У випадку високих навантажень ці витрати можуть значно знижувати продуктивність системи. Використання пулу з’єднань дозволяє уникнути створення нових підключень при кожному запиті, повторно використовуючи існуючі з’єднання.

Адаптивний підхід полягає в тому, щоб зберігати вже встановлені підключення у пулі для повторного використання, динамічно масштабувати розмір пулу відповідно до поточного навантаження, тобто збільшувати кількість доступних з’єднань у пікові періоди та скорочувати їх у разі зменшення запитів та оптимізувати ресурси сервера, виділяючи саме ту кількість з’єднань, яка необхідна для поточного навантаження. І таким чином забезпечує рівномірний розподіл з’єднань між активними запитами, уникаючи ситуацій, коли деякі підключення простоюють, а інші перевантажені. Наприклад обмеження максимальної кількості підключень до бази даних для захисту від її перевантаження. Якщо запитів більше, ніж може обробити поточний пул, система додає запити в чергу, а не створює нові підключення, що могло б перевищити ресурси сервера.

Для забезпечення адаптивного управління пулами з’єднань можуть використовувати спеціалізовані бібліотеки, як Connection Pooling у .NET, або інші вбудовані механізми для роботи з пулом з’єднань, які забезпечують ефективне управління підключеннями до SQL Server.

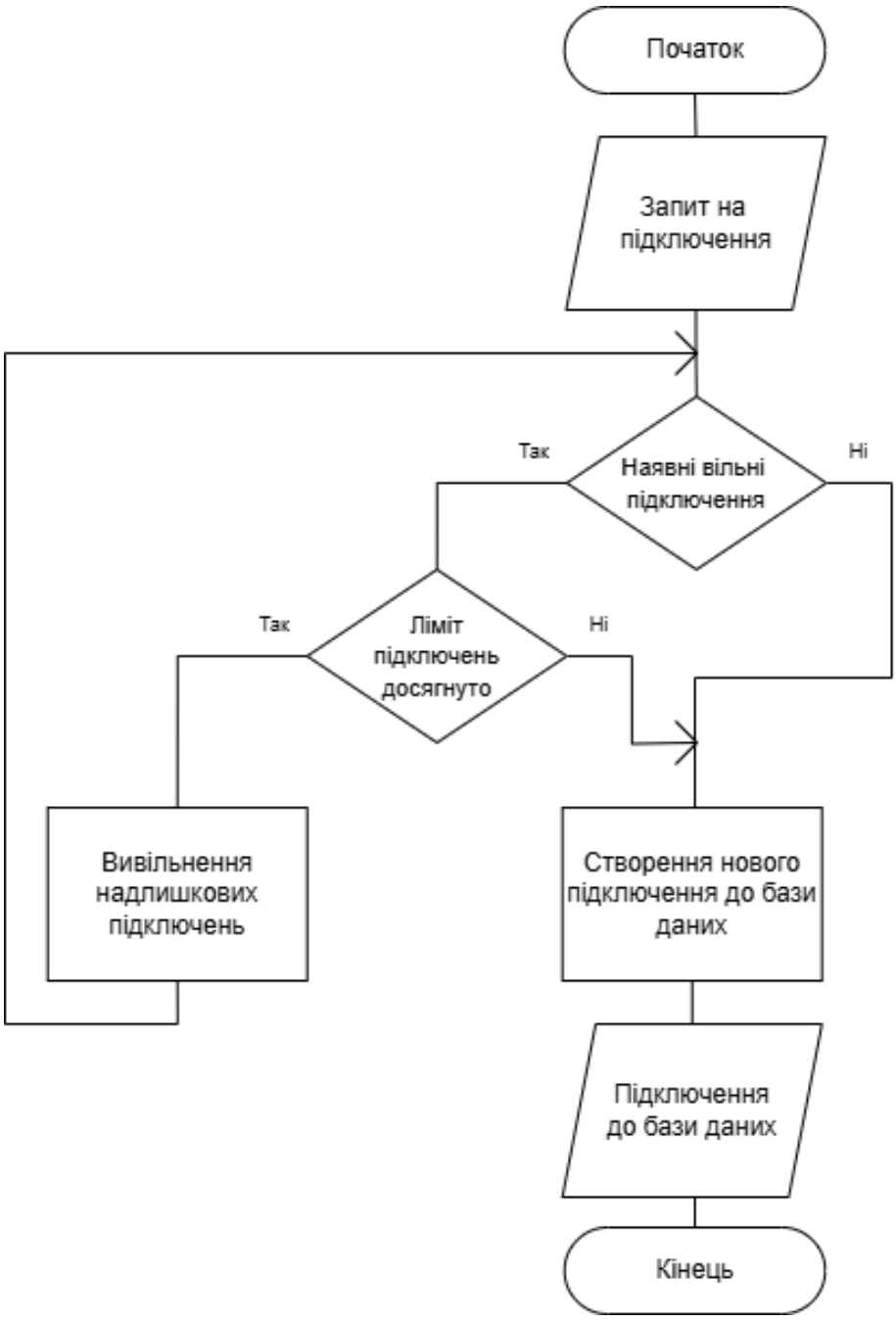


Рис. 2.2. Схема роботи пула підключень

2.3. Особливості розробленого методу

Запропонований метод покликаний збільшити швидкодію обробки запитів між серверною частиною застосунку та базою даних шляхом розробки нової ORM, де вперше запропоновано використати поєднання апаратного прискорення та низькорівневих конструкцій мови C#.

Розроблений метод для підвищення швидкості обробки даних серверною частиною ERP і HRM систем базується на комплексному підході

до ефективного використання ресурсів, оптимізації обробки даних та інтеграції сучасних технологій. Він поєднує кілька взаємодоповнюваних стратегій, що створюють синергію для досягнення максимальної продуктивності. Однією з головних особливостей цього методу є мінімізація використання пам’яті, яка досягається через впровадження таких інструментів, як Span<T>, Memory<T> та ArrayPool<T>. Ці інструменти дозволяють значно знизити витрати на пам’ять, що є критично важливим при обробці великих обсягів даних. Зменшення кількості алокацій пам’яті сприяє зниженню навантаження на збирач сміття (Garbage Collector), запобігаючи затримкам під час роботи. Крім того, метод забезпечує оптимізацію SQL-запитів, що генеруються ORM фреймворком, дозволяючи розробникам вручну налаштовувати та оптимізувати запити під специфічні потреби. Такий підхід знижує навантаження на базу даних і дозволяє скоротити час виконання запитів, що безпосередньо впливає на загальну продуктивність системи.

Ще однією ключовою особливістю методу є інтеграція апаратного прискорення, зокрема, використання GPU для паралельної обробки даних. Паралельна обробка на GPU надає можливість одночасного виконання великої кількості обчислень, що дозволяє значно прискорити обробку складних завдань і великих наборів даних. Це особливо важливо для задач, які вимагають високої обчислювальної потужності, таких як обробка статистичних даних і генерація звітів. Метод також передбачає оптимізацію алгоритмів під паралельне виконання на GPU, що дозволяє досягти максимальної продуктивності. Це включає як вибір найбільш підходящих алгоритмів, так і їх адаптацію для ефективної роботи в умовах паралельної обробки.

Крім того, метод забезпечує можливість легкої інтеграції з існуючими ORM фреймворками, такими як Entity Framework Core, що дозволяє зберегти всі переваги ORM, включаючи високий рівень абстракції та зручність розробки, одночасно забезпечуючи високу продуктивність

системи. Незважаючи на необхідність низькорівневих оптимізацій, цей метод дозволяє зберегти більшість переваг ORM, таких як автоматичне відображення об'єктів на таблиці баз даних і підтримка асинхронних операцій, що значно спрощує розробку та забезпечує стабільну й ефективну роботу системи. У підсумку, розроблений метод поєднує в собі інноваційні підходи до управління ресурсами, оптимізації процесів і ефективного використання апаратних засобів, що дозволяє забезпечити високу продуктивність і надійність системи навіть у найскладніших умовах експлуатації.

2.4. Висновки до розділу

У цьому розділі розглянуто методи підвищення швидкості обробки даних серверною частиною системи управління персоналом, акцентуючи увагу на подоланні обмежень традиційних ORM фреймворків, таких як Entity Framework Core, та забезпеченні більшої продуктивності при обробці великих обсягів даних. Запропоновані рішення включають загальний підхід до оптимізації ресурсів системи, поліпшення роботи з пам’яттю, вдосконалення генерації SQL-запитів і інтеграцію апаратного прискорення. Впровадження нових технологій управління пам’яттю дозволяє ефективніше використовувати ресурси системи, знижуючи вплив навантаження на продуктивність.

Окрема увага приділяється оптимізації SQL-запитів, що дозволяє зменшити навантаження на базу даних і скоротити час їх виконання, тоді як інтеграція апаратного прискорення, зокрема використання GPU, забезпечує значне прискорення обробки складних завдань і великих наборів даних, що є критично важливим для задач з високими вимогами до обчислювальної потужності. Таким чином, розроблений метод поєднує інноваційні підходи до управління ресурсами та інтеграції сучасних технологій, що дозволяє досягти значного підвищення продуктивності системи, покращуючи

швидкість обробки даних і забезпечуючи стабільну і ефективну роботу систем у складних умовах експлуатації.

3. РОЗРОБКА ТА ОСОБЛИВОСТІ РЕАЛІЗАЦІЇ ПРОГРАМНОЇ ЧАСТИНИ

3.1. Опис системи

Розроблена система є бібліотекою, яка базується на спеціально створеному ORM (Object-Relational Mapping) для мови програмування C#. Основною метою цієї бібліотеки є забезпечення ефективного управління базами даних із зменшеним споживанням пам’яті та підвищеною продуктивністю. Для досягнення цього були використані сучасні інструменти управління пам’яттю, що дозволяє значно зменшити накладні витрати на пам’ять при роботі з великими наборами даних. Окрім цього, в бібліотеці реалізовано підтримку апаратного прискорення, що ще більше покращує продуктивність системи, особливо в умовах високих навантажень. Ця бібліотека може бути використана іншими розробниками для створення систем управління персоналом або інших бізнес-додатків, які потребують ефективного і продуктивного ORM. Вона спроєктована таким чином, щоб бути легко інтегрованою в існуючі проєкти та забезпечувати високий рівень налаштовуваності під конкретні вимоги користувачів.

3.2. Аналіз інструментів та засобів розробки

Для розробки системи управління персоналом була обрана мова програмування C#, яка надає розробникам потужний інструментарій для створення ефективних та продуктивних додатків.

У рамках проєкту особлива увага приділялася використанню сучасних конструкцій та засобів оптимізації пам’яті, які пропонує .NET платформа.

3.2.1. Мова програмування C#

C# є однією з найпотужніших і найпопулярніших мов програмування, яка широко використовується для розробки корпоративних додатків різної складності та масштабів. Ця мова поєднує в собі простоту використання, що робить її зручною та доступною навіть для початківців, і водночас

забезпечує високий рівень продуктивності, подібний до мов програмування, таких як C і C++. Прості й зрозумілі синтаксичні конструкції C#, схожі на ті, що використовуються в Java, дозволяють розробникам швидко і легко писати, читати та підтримувати код. Це значно знижує поріг входження для нових програмістів і прискорює процес розробки.

Крім того, C# має розвинену систему типізації, яка забезпечує безпечність коду і знижує ймовірність помилок, що можуть виникнути через невірно оброблені дані. Завдяки строгій типізації, компілятор C# здатен виявляти і виправляти помилки ще на етапі компіляції, що значно підвищує надійність програмного забезпечення. Мова також підтримує сучасні парадигми програмування, включаючи об'єктно-орієнтоване, функціональне і асинхронне програмування, що дозволяє створювати гнучкі, масштабовані та ефективні додатки, які легко адаптуються до змінних потреб бізнесу.

C# тісно інтегрована з потужною .NET платформою [10], яка пропонує великий набір бібліотек і фреймворків, що значно спрощують розробку складних додатків. Ці бібліотеки надають інструменти для роботи з базами даних, обробки тексту, роботи з мережею, графічного інтерфейсу користувача, обробки XML і JSON, управління пам'яттю, та багато інших. Інтеграція з .NET дозволяє розробникам використовувати всі ці інструменти, що значно скорочує час на розробку та впровадження нових функціональностей. Завдяки цьому, C# є ідеальним вибором для створення масштабованих, високопродуктивних додатків, які можуть працювати на різних платформах, включаючи Windows, Linux та macOS.

Крім того, C# постійно оновлюється та вдосконалюється, що дозволяє розробникам використовувати новітні технології та підходи, забезпечуючи актуальність та конкурентоспроможність створюваного програмного забезпечення. Це робить C# однією з найбільш перспективних мов програмування, яка поєднує зручність, продуктивність та

багатофункціональність, необхідні для сучасного корпоративного програмування.

Отже можемо виділити наступні переваги використання мови C# в рамках розробки:

- Сучасні парадигми програмування. Завдяки підтримці різноманітних парадигм, включаючи об’єктно-орієнтоване, функціональне та асинхронне програмування, C# дозволяє створювати додатки, що легко масштабуються та адаптуються до змін у бізнес-процесах. Наприклад, можливість використовувати лямбда-вирази та LINQ (Language Integrated Query) значно полегшує роботу з даними та підвищує продуктивність коду.
- Підтримка паралельного програмування. C# також забезпечує ефективне використання багатопотоковості через бібліотеки Task Parallel Library (TPL) і асинхронні методи (async/await), що дозволяє значно підвищити швидкодію додатків, особливо в багатозадачних середовищах.
- Багатофункціональні бібліотеки. .NET надає багатий набір бібліотек, які включають засоби для роботи з базами даних, обробки XML, роботи з мережевими протоколами та багато іншого. Це дозволяє швидко розробляти комплексні рішення, не витрачаючи час на реалізацію базової функціональності.
- Кросплатформенність .NET Core. C# у поєднанні з .NET Core дозволяє створювати кросплатформені додатки, які можуть працювати на різних операційних системах, включаючи Windows, Linux і macOS. Це особливо важливо для підприємств, які використовують різні платформи в своїх IT-інфраструктурах.

3.2.2. Переваги використання Span<T>

- Зменшення накладних витрат: Span<T> дозволяє працювати з підмасивами великих масивів без необхідності їх копіювання, що значно знижує накладні витрати на управління пам'яттю. Це

особливо корисно у випадках, коли потрібно обробляти великі масиви даних, наприклад, при роботі з великими файлами або потоками даних.

- Безпека та продуктивність: Span<T> надає безпечний доступ до пам’яті, оскільки забезпечує перевірку виходу за межі масиву під час виконання. Крім того, ця структура підтримує стекову пам’ять, що дозволяє уникнути роботи з динамічною пам'яттю, прискорюючи роботу програми.
- Підтримка різних типів даних: Span<T> підтримує роботу не лише з масивами, але й з іншими типами даних, такими як строки, байтові буфери та структури, що дозволяє використовувати його в різних сценаріях. Це спрощує обробку даних і робить код більш універсальним.
- Сумісність з асинхронними операціями: Span<T> може бути ефективно використаний в асинхронних операціях для передачі даних без зайвих копіювань, що забезпечує швидку та ефективну обробку даних у багатопотокових додатках.

Span<T> є надзвичайно ефективною та гнучкою структурою для роботи з пам’яттю в .NET, яка надає розробникам потужний інструмент для оптимізації використання пам'яті в додатках. Ця структура дозволяє маніпулювати континуальними ділянками пам'яті, надаючи можливість працювати з ними без створення зайвих копій і, відповідно, без додаткових витрат ресурсів. Завдяки цьому, Span<T> забезпечує підвищену продуктивність додатка, знижуючи накладні витрати на обробку даних та зменшуючи кількість операцій з виділення пам'яті. Це особливо важливо для високопродуктивних додатків, які обробляють великі обсяги даних або працюють в реальному часі, де кожна мілісекунда має значення.

Однією з ключових переваг Span<T> є його здатність працювати з різними типами даних, включаючи примітивні типи, такі як int, double, byte, та складніші типи, такі як структури та класи. Ця універсальність робить

Span<T> ідеальним інструментом для багатьох завдань, від обробки великих масивів числових даних до маніпуляцій з текстовими рядками або бінарними даними. Наприклад, при обробці великих масивів чисел, Span<T> дозволяє виконувати операції над частинами масиву без необхідності створювати підмасиви, тобто копії, що значно економить пам'ять і час виконання. Це також зручно при роботі з потоком даних, коли необхідно ефективно обробляти підмножини даних, не створюючи нові масиви.

Крім того, Span<T> підтримує різні варіанти ініціалізації, включаючи ініціалізацію з масивів, стекових ділянок пам'яті, а також небезпечних (unsafe) кодів. Це дає розробникам більше контролю над тим, як і де розміщуються дані, що забезпечує додаткові можливості для оптимізації і налаштування продуктивності додатків під конкретні завдання. Наприклад, при обробці даних в реальному часі можна використовувати Span<T> для роботи з ділянками пам'яті, які розташовані безпосередньо в стеку, що забезпечує мінімальні затримки при доступі до цих даних, чим значно скоротити час виконання програмного коду.

Таким чином, Span<T> є не просто інструментом для ефективної роботи з пам'яттю, а й універсальним засобом, який дозволяє розробникам гнучко підходити до управління даними та оптимізувати додатки для різних сценаріїв використання. Завдяки своїй гнучкості, продуктивності та універсальності, Span<T> стає незамінним компонентом сучасних додатків на платформі .NET, які перша за все потребують високої ефективності та надійності.

3.2.3. Переваги використання Memory<T>

- Розширені можливості управління ресурсами: Memory<T> дозволяє працювати з об’єктами, що знаходяться в керованій та некерованій пам’яті, що надає більшу свободу у виборі підходу до оптимізації роботи з пам’яттю. Це дозволяє ефективно працювати з

великими масивами даних та зменшувати кількість алокацій пам'яті.

- Підтримка синхронного та асинхронного коду: Memory<T> добре інтегрований з асинхронними операціями, що робить його ідеальним для використання у високонавантажених системах, де потрібно мінімізувати блокування потоків і забезпечувати швидке оброблення даних.
- Підтримка довготривалих операцій: Memory<T> може бути використаний для реалізації довготривалих операцій з обробки даних, таких як аналіз великих обсягів інформації або робота з потоками даних у реальному часі. Це дозволяє оптимізувати використання пам’яті та забезпечити безперервність обробки.
- Сумісність із різними типами пам’яті: Memory<T> може працювати з різними типами пам’яті, включаючи стекову, динамічну та фіксовану пам’ять, що робить його універсальним інструментом для оптимізації роботи додатків на низькому рівні.

Memory<T> надає значно більш гнучкий та розширений підхід до управління пам’яттю в порівнянні з Span<T>, відкриваючи нові можливості для розробників .NET додатків. Однією з головних переваг цієї структури є її здатність працювати як з керованими, так і з некерованими ресурсами, що дозволяє ефективно управляти пам’яттю в різноманітних сценаріях, включаючи роботу з великими масивами даних або даними, що зберігаються в небезпечних областях пам'яті. Завдяки цьому Memory<T> може використовуватися для управління складними структурами даних, які можуть змінюватися під час виконання програми, забезпечуючи при цьому високу продуктивність і зниження витрат на алокацію пам'яті. Іншою важливою особливістю Memory<T> є її здатність зберігати стан обробки даних між викликами методів. Це особливо корисно в контексті потокової обробки або роботи з великими наборами даних, де необхідно підтримувати безперервність обробки даних. Наприклад, при обробці великих файлів або

потоків даних, Memory<T> дозволяє зберігати поточний стан обробки і відновлювати його при наступних викликах, що дозволяє уникнути зайвих копіювань і повторних обчислень. Це забезпечує не тільки ефективність обробки, але й підвищену надійність роботи додатка, оскільки зменшує ризик помилок при повторній обробці тих самих даних.

Крім того, Memory<T> надає можливість створювати "слайси" або підмножини даних без необхідності створювати нові об'єкти, що знову ж таки знижує витрати на пам'ять і підвищує продуктивність. Це особливо важливо при роботі з великими масивами або списками, де необхідно часто маніпулювати підмножинами даних. Наприклад, при обробці масивів зображень або звукових даних, Memory<T> дозволяє легко створювати підмножини даних для окремої обробки, не витрачаючи додаткові ресурси на створення нових масивів. Ще однією перевагою Memory<T> є його інтеграція з іншими структурами та методами .NET, що дозволяє безперешкодно використовувати його в комбінації з іншими інструментами для оптимізації пам’яті та підвищення продуктивності. Наприклад, Memory<T> може використовуватися разом з Span<T> для роботи з певними ділянками пам'яті, а потім ці дані можуть передаватися іншим методам або зберігатися для подальшої обробки.

Таким чином, Memory<T> забезпечує не просто гнучкий підхід до управління пам’яттю, але й значно розширює можливості розробників у створенні ефективних та продуктивних додатків, які можуть працювати з великими наборами даних, підтримуючи високу швидкість обробки і оптимізацію ресурсів. Завдяки своїй універсальності та продуктивності, Memory<T> стає незамінним інструментом для розробки сучасних корпоративних додатків, де критично важливими є швидкість і надійність обробки даних.

3.2.4. Переваги використання ArrayPool<T>

- Повторне використання масивів: ArrayPool<T> дозволяє багаторазово використовувати один і той самий масив, що значно

знижує витрати на його створення та знищення, особливо в додатках, де інтенсивно обробляються великі обсяги даних. Це забезпечує зменшення накладних витрат на управління пам’яттю та покращує продуктивність системи.

- Зменшення фрагментації пам'яті: використання ArrayPool<T> допомагає знизити фрагментацію пам'яті, оскільки масиви не видаляються, а повертаються в пул для повторного використання. Це дозволяє підтримувати стабільну роботу програми навіть при тривалому виконанні.
- Зменшення накладних витрат на створення масивів: у випадках, коли програма часто створює і видаляє великі масиви, ArrayPool<T> дозволяє значно знизити накладні витрати на ці операції, забезпечуючи швидкий доступ до готових масивів з пулу.
- Підвищення стабільності роботи: зменшення кількості алокацій та звільнень пам’яті сприяє стабільності роботи програми, знижуючи ризик виникнення помилок, пов’язаних з управлінням пам’яттю, таких як витоки пам'яті або перевантаження збирача сміття.

ArrayPool<T> є надзвичайно потужним інструментом для управління пам'яттю у програмних додатках, який дозволяє значно підвищити ефективність використання масивів і знизити витрати на їх створення та видалення. Ця структура дозволяє повторно використовувати вже виділені масиви, що є особливо корисним у сценаріях з інтенсивним використанням пам’яті, таких як обробка великих обсягів даних або системи обробки потоків. Завдяки цьому підходу зменшується необхідність в частих операціях алокації і деалокації пам'яті, що в свою чергу знижує накладні витрати на управління пам'яттю і підвищує загальну продуктивність системи. Однією з ключових особливостей ArrayPool<T> є можливість повторного використання масивів без необхідності їх створення та видалення щоразу, коли масив стає непотрібним. Це особливо корисно в ситуаціях, коли масиви часто створюються і знищуються, що може

призвести до значних витрат на управління пам'яттю. Наприклад, у системах обробки потоків або великих базах даних, де потрібно швидко обробляти великі обсяги даних, ArrayPool<T> дозволяє зберігати і повторно використовувати масиви без додаткових витрат на їх створення і видалення. Це не тільки зменшує навантаження на систему пам'яті, але й знижує час виконання програми, оскільки повторне використання масивів є значно швидшим, ніж їх повторна алокація.

ArrayPool<T> також дозволяє контролювати розмір масивів, які використовуються, що дає змогу більш ефективно управляти пам’яттю. Розробники можуть налаштовувати розмір пулу і визначати, скільки масивів певного розміру повинно бути доступно в пулі. Це дозволяє адаптувати використання пам'яті до конкретних потреб програми і оптимізувати її продуктивність. Наприклад, в системах з великими обсягами даних, де важливо підтримувати швидкий доступ до даних, ArrayPool<T> може бути налаштований для використання масивів оптимального розміру, що дозволяє зменшити витрати на обробку і підвищити швидкість роботи додатка.

Таким чином, ArrayPool<T> забезпечує не тільки ефективне управління пам’яттю, але й допомагає знижувати накладні витрати на управління масивами, підвищуючи загальну продуктивність системи. Його використання є критично важливим для розробки додатків, які обробляють великі обсяги даних або виконують інтенсивні обчислення, де оптимізація пам’яті і швидкість виконання є ключовими для досягнення високої продуктивності.

3.2.5. Переваги використання SIMD

- Прискорення числових обчислень: SIMD дозволяє значно підвищити швидкість виконання числових операцій, що є критичним у задачах, пов’язаних з обробкою великих масивів даних, таких як обробка зображень, звуку або наукові розрахунки.

- Зменшення часу обробки даних: використання SIMD інструкцій дозволяє виконувати кілька операцій одночасно, що знижує загальний час виконання алгоритмів та підвищує ефективність використання апаратних ресурсів.
- Оптимізація роботи процесора: SIMD дозволяє оптимізувати роботу процесора, виконуючи кілька операцій над різними елементами даних одночасно, що знижує загальне навантаження на процесор і підвищує швидкодію програми.
- Забезпечення масштабованості: використання SIMD дозволяє досягти високої масштабованості додатків, оскільки інструкції можуть бути ефективно використані на багатоядерних процесорах, що забезпечує кращу продуктивність у багатозадачних середовищах.

Векторні інструкції SIMD (Single Instruction, Multiple Data) є потужним інструментом для оптимізації обробки даних, що дозволяє значно прискорити виконання чисельних обчислень. Основна перевага SIMD інструкцій полягає в їх здатності виконувати одну й ту ж операцію одночасно над кількома одиницями даних, що забезпечує суттєве зменшення часу обробки порівняно з традиційними скалярними інструкціями. SIMD інструкції працюють з векторними реєстрами, які можуть зберігати кілька елементів даних, таких як числа з плаваючою комою або цілі числа, у одному регістрі процесора. Це дозволяє виконувати обчислення на великій кількості даних паралельно, без необхідності виконувати одне і те ж обчислення для кожного елемента даних окремо. Наприклад, при обробці зображень або відео, де кожен піксель може бути оброблений за допомогою однієї і тієї ж операції, SIMD інструкції дозволяють обробляти кілька пікселів одночасно, що значно підвищує швидкість обробки зображень та відео. Ще однією важливою перевагою SIMD є можливість значної економії обчислювальних ресурсів. Завдяки виконанню одночасних обчислень над кількома даними, SIMD інструкції

дозволяють зменшити загальне число циклів процесора, необхідних для виконання програми. Це означає, що процесор може виконувати більше обчислень за одиницю часу, що підвищує загальну продуктивність додатка і знижує енергоспоживання.

SIMD інструкції також забезпечують зменшення накладних витрат на передачу даних між пам’яттю та процесором. Оскільки обробка кількох одиниць даних одночасно дозволяє зменшити кількість операцій зчитування та запису в пам’ят, це призводить до зниження часу доступу до даних і зменшення навантаження на системну шину. Використання SIMD є особливо корисним у випадках, коли необхідно виконувати чисельні розрахунки з великими обсягами даних або обробляти дані в реальному часі, наприклад, у наукових розрахунках, фінансових моделюваннях, ігрових додатках та системах штучного інтелекту. Завдяки SIMD, додатки можуть досягти значних поліпшень у продуктивності, забезпечуючи швидше виконання складних обчислювальних завдань та підвищення загальної ефективності.

3.3. Архітектура програмного забезпечення

Архітектура цього застосунку системи управління персоналом підприємства представляє собою типову багатошарову архітектуру, яка включає в себе frontend, backend та базу даних. Кожен шар виконує свої специфічні завдання та спілкується з іншими шарами, забезпечуючи роботу всієї системи. Важливу роль в цьому відіграє розроблена бібліотека, яка інтегрується в шар доступу до даних та забезпечує ефективне виконання запитів до бази даних. Вона дозволяє оптимізувати процеси взаємодії з базою, використовуючи апаратне прискорення для обробки великих обсягів даних. Шар бізнес-логіки покладається на розроблену бібліотеку для швидкого виконання обчислень та передачі результатів у шар представлення. Це дозволяє зменшити час очікування користувачів на отримання результатів і підвищує загальну продуктивність системи.

Бібліотека забезпечує модульність та повторне використання коду, що сприяє легкому масштабуванню та підтримці програмного забезпечення.

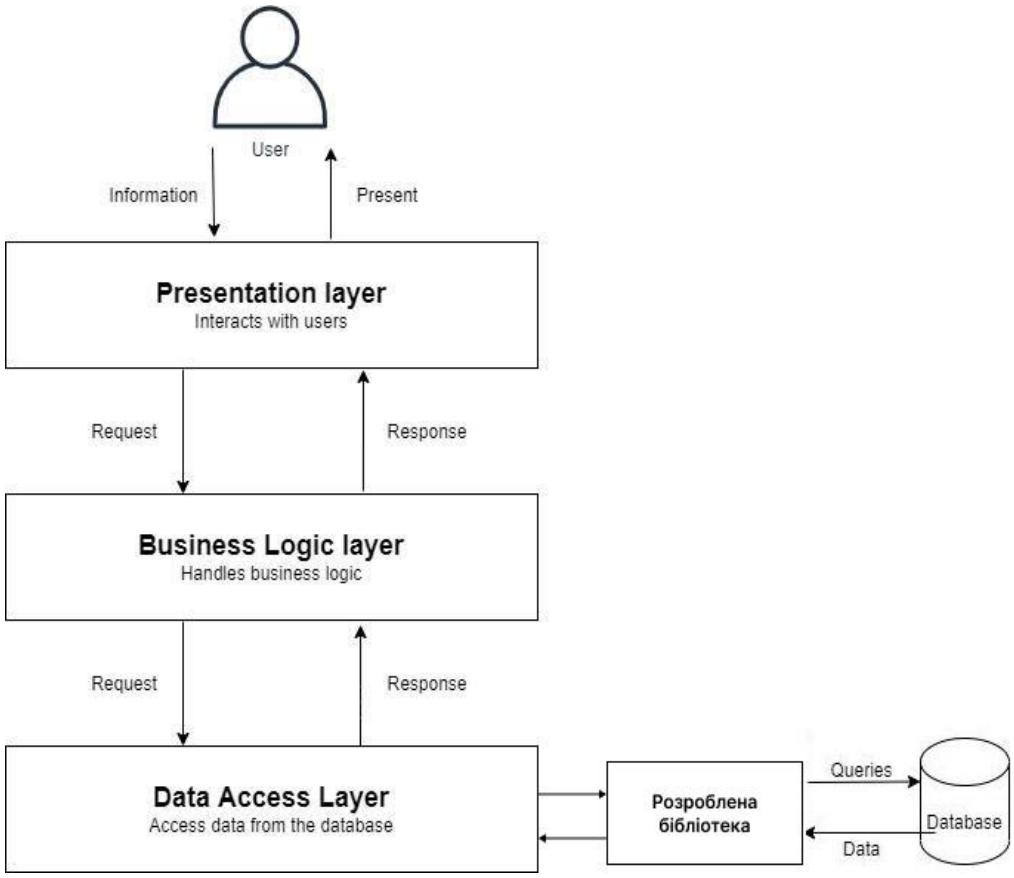


Рис. 3.1. Архітектура додатку

Діаграма класів розробленої бібліотеки представлена на рис. 3.2. Зображені основні класи та методи.

Клас MyORM є основним компонентом, який взаємодіє з пулом з'єднань DbConnectionPool для виконання запитів до бази даних. Основні методи MyORM забезпечують виконання параметризованих SQL-запитів та роботу з об'єктами. Унікальною особливістю бібліотеки є використання методів, які дозволяють обробляти великі обсяги даних із залученням апаратного прискорення. DbConnectionPool відповідає за управління пулом підключень, забезпечуючи отримання та повернення об'єктів з'єднань. Завдяки використанню пулу з'єднань досягається ефективне управління

ресурсами для підключення до бази даних, що значно підвищує продуктивність роботи бібліотеки.

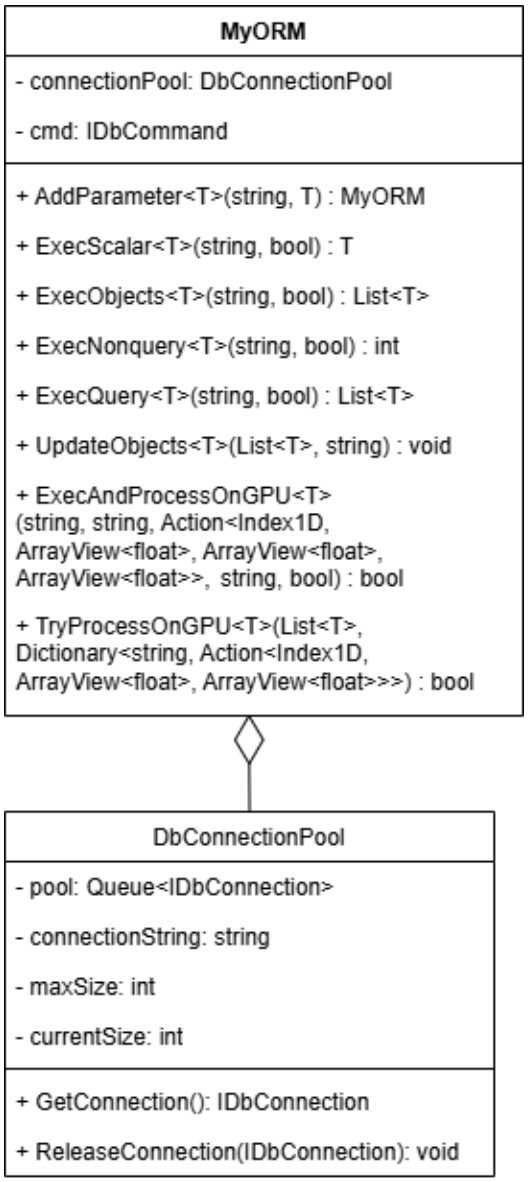


Рис. 3.2. Діаграма класів розробленої ORM

Опишемо детально розподіл відповідальностей для кожного шару.

3.2.1. Frontend (React)

Визначимо відповідальності для frontend частини вебзастосунку:

- Графічний Інтерфейс Користувача (GUI). Відображення форм для реєстрації та авторизації користувачів, інтерфейс для перегляду та коригування інформації про працівників.

- Взаємодія з Сервером. Відправка запитів до сервера для передачі даних користувача, вибору методу завантаження даних, та отримання результатів.

3.2.2. Backend (C#)

Описуємо відповідальності для backend частини системи:

- Логіка Бізнес-процесів. Реалізація логіки, пов'язаної з управлінням персоналом, включаючи додавання, видалення та редагування інформації про працівників, обробку запитів на зміни статусу працівників та інші бізнес-правила.
- Забезпечення Безпеки. Реалізація механізмів аутентифікації та авторизації користувачів, контроль доступу до ресурсів системи та захист даних.
- Взаємодія з Базою Даних. Виконання запитів до бази даних для отримання та збереження інформації про працівників, зберігання сесійних даних та інші операції взаємодії з даними.

Та розроблена бібліотека, яка відповідає на наступне:

- Обробка написаного на мові C# коду. Використання низькорівневих можливостей мови для збільшення швидкості обробки отриманих даних з бази.
- Пришвидшене формування тексту SQL запиту. За рахунок шаблонів збільшити швидкість генерації запитів.

3.2.3. База Даних та її опис

Описуємо функції та характеристики бази даних:

- Зберігання Даних. Забезпечення надійного та ефективного зберігання інформації про працівників, їхні дані та історію змін.
- Забезпечення Цілісності та Консистентності. Здійснення контролю за цілісністю даних та підтримка їхньої консистентності у всіх операціях бази даних.

- Підтримка Запитів. Виконання запитів на вибірку, вставку, оновлення та видалення даних з бази даних з метою забезпечення доступу до інформації про працівників та їхніх дій.

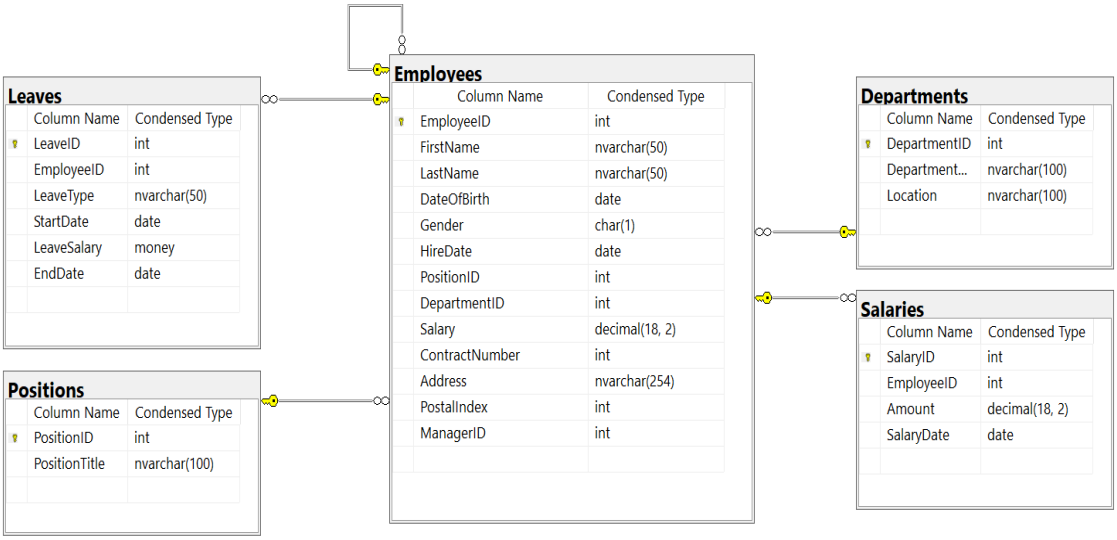


Рис. 3.3. Схема бази даних

Employees (Співробітники)

Ця таблиця є основною для управління даними про співробітників підприємства і містить всю необхідну інформацію, яка використовується для адміністрування персоналу. Вона зберігає такі персональні дані, як ім'я, прізвище, дата народження, що дає змогу ідентифікувати кожного співробітника. Окрім цього, таблиця включає дані про займану посаду, до якого департаменту належить співробітник, а також його заробітну плату.

У таблиці також відображається інформація про контракт співробітника, включаючи дату укладення та закінчення договору, а також умови працевлаштування. Крім того, тут містяться дані про адресу проживання співробітника, що можуть використовуватися для логістики або офіційного листування. У разі внесення змін до будь-якої інформації, таблиця автоматично оновлюється, щоб забезпечити актуальність даних. Це гарантує точність і надійність обліку персоналу навіть у випадку частих змін у штатному розкладі.

Особливістю цієї таблиці є наявність ідентифікатора менеджера, який дозволяє визначити, хто з співробітників є безпосереднім керівником для кожного з підлеглих. Це полегшує побудову організаційної структури підприємства і сприяє ефективнішій взаємодії між різними рівнями працівників. Така структура також допомагає у формуванні звітів і аналізі кадрових ресурсів, що є важливим для стратегічного управління підприємством.

Таблиця 3.1

Поля таблиці Employees

Назва колонки	*Тип даних*	*Є первинним ключ*	*Nullable*
EmployeeID	int	+	–
FirstName	nvarchar(50)	–	–
LastName	nvarchar(50)	–	–
DateOfBirth	Date	–	+
Gender	char(1)	–	+
HireDate	Date	–	-
PositionID	int	–	+
DepartmentID	int	–	+
Salary	decimal(18, 2)	–	+
ContractNumber	int	–	–
Address	nvarchar(254)	–	–
PostalIndex	int	–	–
ManagerID	int	–	+

Departments (Департаменти)

Таблиця відповідає за збереження інформації про департаменти (відділи) підприємства. Кожен запис містить унікальний ідентифікатор

департаменту, його назву та розташування. Ця інформація використовується для розподілення співробітників за департаментами.

Таблиця 3.2

Поля таблиці Departments

Назва колонки	*Тип даних*	*Є первинним ключ*	*Nullable*
DepartmentID	int	+	–
DepartmentName	nvarchar(100)	–	–
Location	nvarchar(100)	–	+

Salaries (Зарплати)

Таблиця використовується для збереження інформації про нарахування зарплати співробітникам. Вона містить записи про суми виплат, дати нарахувань, а також ідентифікатори співробітників, яким були здійснені виплати. Ця таблиця дозволяє відстежувати історію виплат для кожного співробітника.

Таблиця 3.3

Поля таблиці Salaries

Назва колонки	*Тип даних*	*Є первинним ключ*	*Nullable*
SalaryID	int	+	–
EmployeeID	int	–	+
Amount	money	–	–
SalaryDate	date	–	–

Positions (Посади)

Таблиця зберігає дані про всі посади на підприємстві. Кожен запис містить унікальний ідентифікатор посади та її назву. Ця таблиця допомагає

визначити, яку посаду займає той чи інший співробітник. Таблиця також використовується для аналізу організаційної структури, що полегшує управління ресурсами та розподіл обов'язків між департаментами.

Таблиця 3.4

Поля таблиці Positions

Назва колонки	*Тип даних*	*Є первинним ключ*	*Nullable*
PositionID	int	+	–
PositionTitle	nvarchar(100)	–	–

Leaves (Відпустки)

Таблиця відповідає за збереження даних про відпустки співробітників. У ній міститься інформація про тип відпустки (оплачувана чи неоплачувана), дати початку та закінчення відпустки, а також суми виплат за час відпустки. Таблиця дозволяє відстежувати, коли і на який період співробітники йшли у відпустку, яку при цьому отримував виплату та використати ці дані при необхідності перевірки в майбутньому.

Таблиця 3.5

Поля таблиці Leaves

Назва колонки	*Тип даних*	*Є первинним ключ*	*Nullable*
LeaveID	int	+	–
EmployeeID	int	–	+
LeaveType	nvarchar(50)	–	–
StartDate	date	–	–
EndDate	money	–	–
LeaveSalary	date	–	–

3.2.4. Використання шаблону “Model-View-Controller”

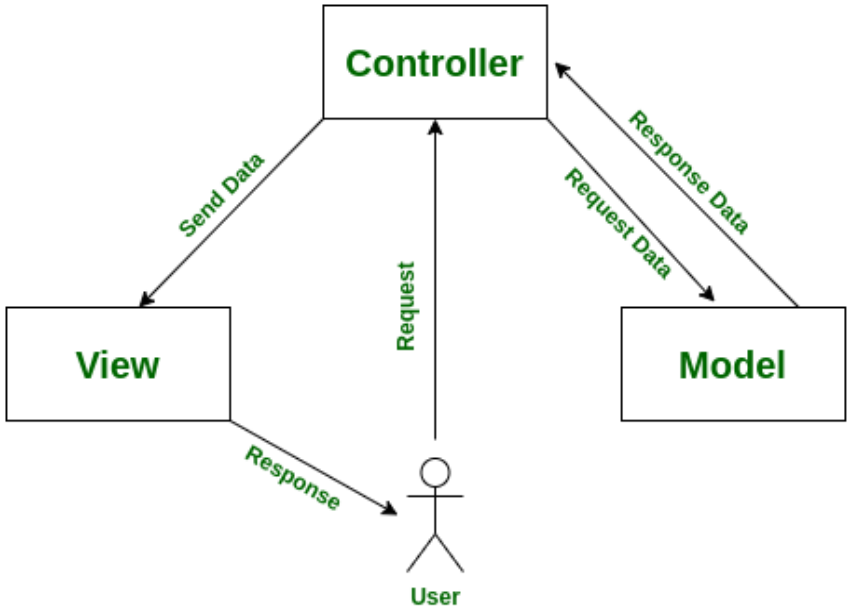


Рис. 3.4. Схема взаємодії користувача з системою

Введення шаблону "Model-View-Controller" доповнює архітектурний дизайн програмного забезпечення, забезпечуючи чітке розподілення відповідальностей між компонентами системи:

- Модель (Model). Відповідає за представлення та обробку даних, включаючи логіку бізнес-процесів та взаємодію з базою даних. Модель відділена від користувацького інтерфейсу, забезпечуючи його незалежність від способу представлення даних.
- Вид (View). Відповідає за відображення даних користувачеві та обробку користувацьких подій. Вид отримує дані з Моделі та відображає їх у відповідному форматі для користувача.
- Контролер (Controller). Відповідає за обробку запитів користувача, взаємодію з Моделлю та оновлення Виду. Контролер реагує на дії користувача та ініціює відповідні зміни в Моделі або Виді, забезпечуючи взаємодію між ними.

Шаблон MVC дозволяє розділити логіку програми на компоненти з різними відповідальностями, що спрощує розробку, тестування та підтримку системи управління персоналом підприємства.

3.4. Висновки до розділу

У ході розробки системи для управління персоналом підприємства було виконано аналіз потреб та сформульовано конкретні вимоги до додатку. Для реалізації проєкту обрано мову програмування C# у поєднанні з MSSQL Server, що забезпечує гнучкість та широкі можливості для масштабування. Розроблену систему характеризує використання власної бібліотеки-ORM з попередньою компіляцією запитаів для підвищення швидкості їх виконання та виконання обробки даних на стороні сервера з використанням апаратного прискорення. Особлива увага приділена розробці архітектури додатку, яка включає модулі обробки даних, передачі даних на рівень бази даних та відображення даних. Кожен модуль системи розроблений з урахуванням потреби в подальшому покращенні, розширенні можливостей та масштабуванні, що робить додаток гнучким та адаптивним до змінних вимог.

4. АНАЛІЗ РОЗРОБЛЕНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

У цьому розділі описано процес тестування та перевірки функціонування системи на різних наборах даних. Виявлено деякі обмеження, проведено аналіз різних варіантів роботи системи та здійснено порівняння з іншими існуючими програмними рішеннями.

4.1. Критерії оцінювання розробленого методу

У ході розробки та оптимізації програмного продукту важливу роль відіграє тестування продуктивності, яке здійснюється за допомогою бенчмарків (далі – Benchmarking). Методологія Benchmarking передбачає вимірювання швидкості виконання окремих методів і обчислень з метою виявлення найефективніших варіантів реалізації. Цей підхід не лише дозволяє оптимізувати роботу алгоритмів, але й забезпечує високу продуктивність системи навіть при значному навантаженні.

Рис. 4.1. Структура роботи під час використання бенчмарків

На рис. 4.1 показана структура розробки з використанням Benchmarking. Застосування бенчмарків є доцільним у цій магістерській роботі, особливо з огляду на необхідність оптимізації обчислень. Робота з оптимізованими модулями та методами відповідає вимогам цього підходу. Бенчмарки особливо корисні у сценаріях, де важливо проводити тести на продуктивність, що дозволяє детально оцінити час виконання кожного компонента алгоритму, забезпечуючи високу ефективність та надійність всієї системи.

4.2. Тестування розробленого ПЗ

Для оцінки та порівняння з існуючими аналогами було використано бенчмарки. Цей підхід забезпечив детальне вимірювання продуктивності на різних наборах даних і сприяв виявленню вузьких місць системи. Тестування продуктивності проводилося на заздалегідь підготовлених наборах даних, що дало можливість порівняти швидкість виконання в різних сценаріях у розробленому програмному забезпеченні. Порівняння відбуваються між використанням традиційного підходу з використанням Entity Framework та реалізованого в рамках роботи програмного забезпечення.

Лістинг 4.1. Приклад бенчмарку

```
public class EmployeeDataBenchmark
{
    private readonly string _connectionString = "Database";
    private readonly DateTime _startDate = new DateTime(2020, 01, 01);
    private readonly DateTime _endDate = DateTime.Now;


    [Benchmark]
    public List<EmployeeInfo> GetEmployeeData()
    {
        var command = CustomSqlClient.CreateCommand();
        command.CommandText = @"
            SELECT
                e.EmployeeID,
                e.FirstName,
                e.LastName,
                e.HireDate,
```

Продовження лістингу 4.1

```
                e.Salary,
                d.DepartmentName,
                p.PositionTitle
            FROM Employees e
            LEFT JOIN Departments d ON e.DepartmentID = d.DepartmentID
            LEFT JOIN Positions p ON e.PositionID = p.PositionID
            LEFT JOIN Analytics a ON e.AnalyticID = a.AnalyticID
            LEFT JOIN Salaries s ON e.SalaryID = s.SalaryID
        WHERE e.HireDate BETWEEN @startDate AND @endDate AND
        a.SuccessRate > (select AVG(SuccessRate) from Analytics a) AND
        s.Vacations > (select AVG(Vacations) from Salaries s)";


          command.SqlParameters.AddValue("startDate", _startDate);
          command.SqlParameters.AddValue("endDate", _endDate);


          var result = command.ExecObject<EmployeeInfo>();
          return result;
      }

    [Benchmark]
      public List<EmployeeInfo> GetEmployeeDataEF()
      {
          using (var context = new MyDbContext())
          {
              var result = context.Employees
                  .Where(e => e.HireDate >= _startDate && e.HireDate <= _endDate)
                  .Select(e => new EmployeeInfo
                  {
                      EmployeeID = e.EmployeeID,
                      FirstName = e.FirstName,
                      LastName = e.LastName,
                      HireDate = e.HireDate,
                      Salary = e.Salary,
                      DepartmentName = e.Department.DepartmentName,
                      PositionTitle = e.Position.PositionTitle
                  })
                  .ToList();

              return result;
          }
      }
    }
```

4.3. Оцінка результатів роботи

Для оцінки результатів роботи розробленого програмного забезпечення будуть розглянуті такі дані з бенчмарків як швидкість

виконання, кількість виділеної пам’яті (allocated memory) та кількість викликів garbage collector для очищення пам’яті.

У рамках тестування продуктивності було проведено порівняння між розробленим програмним забезпеченням для роботи з базою даних та підходом, що використовує Entity Framework (EF) або Dapper. Власна бібліотека була розроблена для забезпечення більш оптимізованого доступу до даних з мінімальними накладними витратами. Це стало можливим завдяки використанню низькорівневих оптимізацій, ефективного управління пам'яттю та урахуванню особливостей апаратного забезпечення.

Тестування виконувалось на заздалегідь підготовлених наборах даних та показало, що власна бібліотека демонструє швидший час виконання запитів у порівнянні з існуючими аналогами, завдяки прямішому доступу до бази даних і мінімальним обчислювальним витратам. Основними метриками для аналізу були час виконання (Mean), обсяг виділеної пам'яті (Allocated Memory), а також кількість викликів garbage collector (GC). Отримані результати підтвердили перевагу підходу, орієнтованого на апаратне прискорення, для оптимізації серверної обробки даних.

Усі експерименти виконувалися в однакових умовах, із використанням фіксованих параметрів середовища та ідентичних сценаріїв запитів для забезпечення коректності та відтворюваності результатів.

Таблиця 4.1

Результати роботи власної бібліотеки

Кількість оброблених записів	Час виконання (мс)	Виділена пам’ять (КБ)	Кількість викликів GC		
			(Gen 0)	(Gen 1)	(Gen 2)
1000	47.82	584.35	2	0	0
10000	179.34	1123.87	2	0	0
100000	509.12	2538.93	3	1	0

Таблиця 4.2

Результати роботи Entity Framework

Кількість оброблених записів	Час виконання (мс)	Виділена пам’ять (КБ)	Кількість викликів GC		
			(Gen 0)	(Gen 1)	(Gen 2)
1000	92.56	1087.24	3	1	0
10000	286.12	2431.76	4	2	0
100000	772.49	4976.55	5	3	1

Таблиця 4.3

Результати роботи Dapper

Кількість оброблених записів	Час виконання (мс)	Виділена пам’ять (КБ)	Кількість викликів GC		
			(Gen 0)	(Gen 1)	(Gen 2)
1000	48.26	583.41	2	0	0
10000	185.56	1162.33	2	0	0
100000	560.91	2788.14	3	1	0

Як можна побачити, різниця в необхідному часі для виконання та виділеній при цьому пам’яті на користь розробленої бібліотеки. Далі пропоную розглянути результати бенчмарків стосовно створення нових записів в базі.

Таблиця 4.4

Результати роботи власної бібліотеки

Кількість оброблених записів	Час виконання (мс)	Виділена пам’ять (КБ)	Кількість викликів GC		
			(Gen 0)	(Gen 1)	(Gen 2)
1000	62.14	428.42	2	0	0

Продовження табл. 4.4

10000	213.87	897.62	2	0	0
100000	663.78	1794.12	3	1	0

Таблиця 4.5

Результати роботи Entity Framework

Кількість оброблених записів	Час виконання (мс)	Виділена пам’ять (КБ)	Кількість викликів GC		
			(Gen 0)	(Gen 1)	(Gen 2)
1000	89.56	934.13	4	1	0
10000	273.25	1863.45	4	2	0
100000	846.34	3647.89	5	3	1

Таблиця 4.6

Результати роботи Dapper

Кількість оброблених записів	Час виконання (мс)	Виділена пам’ять (КБ)	Кількість викликів GC		
			(Gen 0)	(Gen 1)	(Gen 2)
1000	68.12	435.45	2	0	0
10000	233.54	923.92	2	0	0
100000	702.21	1856.47	3	1	0

Як бачимо, кількість виділеної пам’яті все ще менша порівняно з аналогами, проте різниця в часі виконання не така велика як в попередньому прикладі, оскільки внесення даних виконується на стороні бази даних.

Також пропоную розглянути результати при виконанні оновлення даних в базі даних. Отримані результати також демонструють перевагу розробленого методу, проте оскільки оновлення потребує виконання на стороні бази даних, різниця в часі виконання менша, якщо порівнювати з попередніми бенчмарками. Це пов’язано з тим, що операції оновлення

значною мірою залежать від архітектури і продуктивності самої бази даних, а також від швидкості обміну даними між сервером і клієнтом. Однак навіть у цьому випадку спостерігається стабільне покращення швидкості виконання.

Таблиця 4.7

Результати роботи власної бібліотеки

Кількість оброблених записів	Час виконання (мс)	Виділена пам’ять (КБ)	Кількість викликів GC		
			(Gen 0)	(Gen 1)	(Gen 2)
1000	58.96	499.32	2	0	0
10000	194.85	1029.24	2	0	0
100000	596.24	2187.93	3	1	0

Таблиця 4.8

Результати роботи Entity Framework

Кількість оброблених записів	Час виконання (мс)	Виділена пам’ять (КБ)	Кількість викликів GC		
			(Gen 0)	(Gen 1)	(Gen 2)
1000	117.38	928.57	4	1	0
10000	319.76	2016.78	4	2	0
100000	912.56	4365.12	5	3	1

Таблиця 4.9

Результати роботи Dapper

Кількість оброблених записів	Час виконання (мс)	Виділена пам’ять (КБ)	Кількість викликів GC		
			(Gen 0)	(Gen 1)	(Gen 2)
1000	62.48	512.23	2	0	0

Продовження табл. 4.9

10000	209.14	1101.33	2	0	0
100000	656.78	2316.12	3	1	0

Наглядно порівняти узагальнену швидкість виконання трьох розглянутих підходів можна на рис. 4.2. Як можна побачити, відчутна різниця в швидкодії спостерігається між EF Core та Dapper, та EF Core та розробленою ORM. Хоч різниця між Dapper та розробленою ORM не є такою ж великою, проте все ще спостерігається.

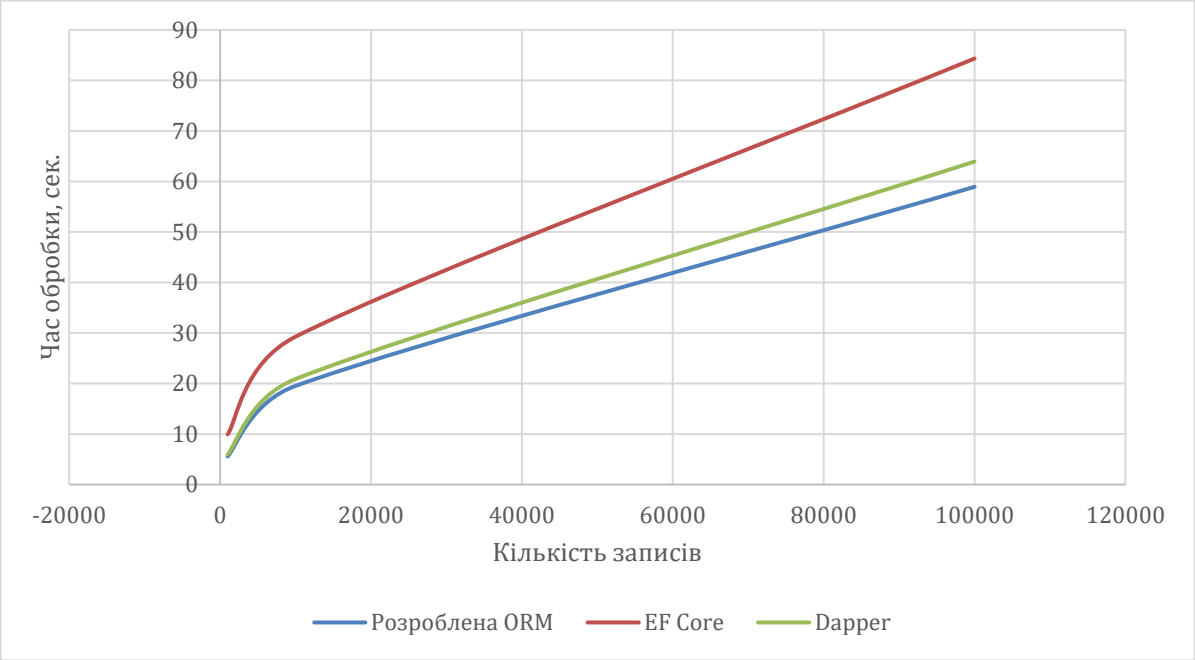


Рис. 4.2. Порівняння швидкості розробленого підходу та аналогів

Останнім пропоную порівняти швидкість обробки при роботі з великими обсягами даних з та без використання апаратного прискорення. Усі експерименти виконувалися в однакових умовах, із використанням фіксованих параметрів середовища та ідентичних сценаріїв запитів для забезпечення коректності та відтворюваності результатів.

Таблиця 4.10

Порівняння швидкості обробки з та без використання апаратного прискорення

Кількість оброблених записів	Швидкість виконання, сек.			
	Розроблена ORM (No GPU)	Розроблена ORM (GPU)	EF Core	Dapper
1000	5,9	10,2	6,5	6,0
10000	23,9	39,5	35,3	26,3
100000	108,1	68,7	166,4	112,6

Наглядно порівняти швидкість обробки з та без використання апаратного прискорення можна на рис. 4.3.

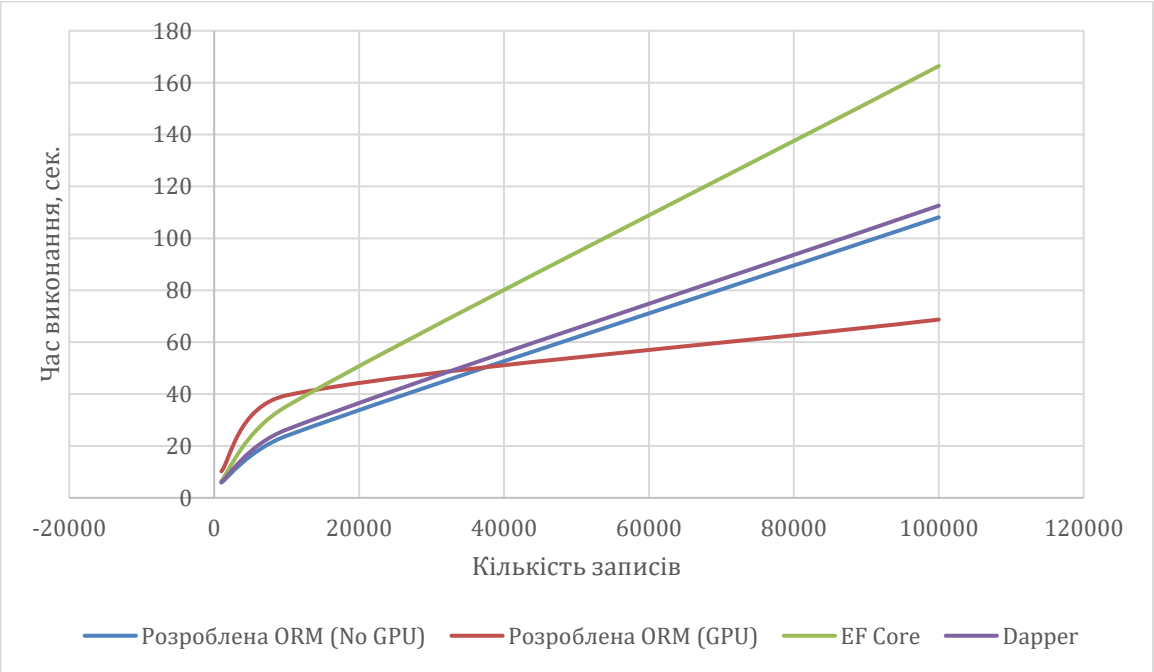


Рис. 4.3. Порівняння швидкість обробки з та без використання апаратного прискорення

Отже з отриманих даних можна зробити наступні висновки. Реалізоване програмне забезпечення дозволяє виконувати запити до 50%

швидше ніж аналогічна логіка написана з використанням Entity Framework та на 5% швидше ніж з використанням Dapper, а при роботі з великими обсягами даних (від 50000 записів в базі даних) та виконанні складних операцій над ними використання апаратного прискорення дозволяє зменшити час виконання до 97%.

4.4. Можливості для покращення розробленого ПЗ

Нижче представлено кілька пропозицій щодо майбутніх оновлень:

1. Вдосконалення алгоритмів для обробки даних. Оптимізація та вдосконалення алгоритмів дозволить збільшити швидкість роботи системи, що є критично важливим для забезпечення ефективної обробки великих обсягів даних. Можливе застосування нових підходів до паралельної обробки або використання більш ефективних структур даних.
2. Покращення інтеграції обчислень на відеокарті. Оскільки відеокарти не підтримують виконання коду, написаного на мові C#, напряму, цей функціонал реалізовано через unsafe код. Використання відеокарт (GPU) для обчислень може значно підвищити продуктивність при виконанні ресурсномістких операцій, однак цей підхід вимагає від програміста контролю за використанням ресурсів системи.
3. Рефакторинг коду бібліотеки для налаштування запитів. Проведення рефакторингу дозволить створити більш гнучку архітектуру, що дасть користувачам можливість налаштовувати виконання запитів до бази даних під їхні потреби. Це може включати параметризацію запитів, підтримку складніших сценаріїв фільтрації та сортування даних, а також підвищення продуктивності за рахунок кастомізації коду під конкретні завдання.

4. Розширення можливостей кешування. Впровадження покращених стратегій кешування може знизити навантаження на базу даних та скоротити час доступу до часто запитуваної інформації. Це особливо корисно в системах з великим об'ємом запитів, де повторне отримання однакових даних може значно впливати на продуктивність.
5. Підтримка асинхронних операцій. Додавання повної підтримки асинхронних методів дозволить підвищити масштабованість та продуктивність системи, оскільки виконання кількох операцій одночасно зменшить час простою під час очікування відповіді від бази даних. Це також підвищить чутливість системи та покращить досвід користувачів при роботі з великими наборами даних.

Наведені пропозиції дадуть змогу покращити систему та зробити її більш адаптивною та гнучкою.

4.5. Висновки до розділу

У результаті проведеного аналізу було продемонстровано ефективність розробленого програмного забезпечення, зокрема його переваги у продуктивності та використанні ресурсів порівняно з Entity Framework. Завдяки тестуванню різних операцій, таких як вибірка, вставка, видалення та оновлення даних, на різних наборах записів, вдалося підтвердити швидкість та оптимальність реалізованої бібліотеки.

Основним досягненням стало зменшення часу виконання запитів до 50% порівняно з Entity Framework та на 5% порівняно з Dapper, а при роботі з великими обсягами даних та виконанні складних операцій над ними використання апаратного прискорення дозволяє зменшити час виконання в середньому на 97%. Використання оперативної пам’яті зменшилось в середньому на 86% порівняно з Entity Framework та на 2% порівняно з Dapper, що дозволило зменшити кількість викликів GC.

Крім того, було визначено кілька можливих напрямків для подальшого розвитку, включаючи вдосконалення алгоритмів, покращення інтеграції обчислень на GPU, а також рефакторинг і оптимізацію кодової бази для більш гнучкого налаштування під потреби користувачів. Ці покращення можуть забезпечити ще більшу продуктивність, масштабованість та адаптивність системи в майбутньому.

5. ПОБУДОВА БІЗНЕС МОДЕЛІ

5.1. Опис проблеми

У розробці програмного забезпечення особливе місце займають ORM (Object-Relational Mapping) фреймворки, що дозволяють перетворювати реляційні дані в об'єктно-орієнтовані структури. ORM значно полегшують роботу з базами даних, абстрагуючи розробників від необхідності написання складних SQL-запитів та взаємодії з низькорівневими інтерфейсами баз даних. Однак стандартні ORM фреймворки мають ряд недоліків, які обмежують їх продуктивність у високонавантажених системах.

Основною проблемою є недостатньо ефективне використання апаратних ресурсів сервера при обробці великих обсягів даних. Стандартні ORM реалізації часто створюють надмірне навантаження на центральний процесор та оперативну пам'ять, використовуючи послідовні обчислення і високорівневі абстракції. Це призводить до зниження продуктивності при роботі з великими наборами даних, особливо в умовах багатокористувацького доступу до системи в режимі реального часу.

Іншою проблемою є обмеження на гнучкість налаштувань взаємодії з базою даних. Стандартні ORM фреймворки орієнтовані на загальні випадки використання і часто не дозволяють розробникам точно контролювати оптимізацію запитів або поведінку системи на низькому рівні. Це може призводити до втрати продуктивності через неоптимальні SQL-запити та непотрібні операції, які виконуються під час перетворення даних.

Таким чином, постає завдання розробки модифікованого ORM фреймворку, який би використовував низькорівневі конструкції мови C# та апаратне прискорення системи для оптимізації процесів обробки даних на сервері.

Всі вище описані проблеми ми можемо узагальнити за допомогою дерева проблем на рис. 5.1.

Рис. 5.1. Дерево проблем

5.2. Зацікавлені сторони

Вирішення проблем, пов’язаних із низькою продуктивністю традиційних ORM фреймворків та необхідністю підвищення ефективності обробки даних, цікавить кілька ключових сторін. Визначимо та опишемо їх:

1. Розробники програмного забезпечення. Основною метою розробників є створення продуктивних та масштабованих систем, здатних обробляти великі обсяги даних у реальному часі. Їх цікавить можливість використання низькорівневих конструкцій мови C# та апаратного прискорення для оптимізації роботи баз даних і зменшення затримок у обробці запитів. Розробники прагнуть створювати інноваційні рішення, які допоможуть покращити продуктивність бізнес-процесів, зберігаючи зручність розробки за допомогою ORM фреймворків.
2. Великі підприємства та корпорації. Підприємства, що використовують ERP та HRM системи для управління своїми ресурсами, зацікавлені у підвищенні швидкодії своїх серверів та можливості масштабування без втрати продуктивності. Вони прагнуть зменшити час простою систем, забезпечити стабільну

роботу баз даних під час великих навантажень і підвищити ефективність процесів прийняття рішень завдяки швидкій обробці інформації.

3. Інтегратори IT-рішень. Компанії, що займаються впровадженням IT-рішень для підприємств, зацікавлені в оптимізованих ORM фреймворках, які можуть забезпечити швидке й стабільне функціонування систем. Вони прагнуть отримати гнучкі та продуктивні інструменти для інтеграції, що дозволять легко адаптувати рішення під специфічні потреби клієнтів, мінімізуючи витрати на налаштування та оптимізацію.
4. Системні архітектори та інженери. Ця група зацікавлена у використанні нових технологій для підвищення продуктивності серверних систем. Системні архітектори прагнуть максимально ефективно використовувати апаратні ресурси (процесор, оперативну пам'ять), а також впроваджувати рішення, які дозволять знизити навантаження на збирач сміття та оптимізувати SQL-запити на низькому рівні. Їх цікавить можливість більшого контролю за процесами оптимізації даних.
5. Бізнес-аналітики та керівники. Керівництво підприємств зацікавлене у стабільній і швидкій обробці бізнес-даних для прийняття своєчасних і обґрунтованих рішень. Для них важливо мати системи, які не просто ефективно працюють, але й адаптуються до зростаючих потреб компанії, дозволяючи обробляти великі обсяги даних без затримок і збоїв.
6. Хмарні провайдери. Постачальники хмарних рішень зацікавлені у впровадженні високоефективних і продуктивних ORM фреймворків, які можуть використовувати апаратне прискорення для роботи з великими обсягами даних. Це дозволить їм надавати клієнтам більш стабільні та ефективні хмарні рішення з меншою вартістю володіння через оптимізацію ресурсів.

Ці зацікавлені сторони відображають широке коло інтересів, що стосуються підвищення продуктивності серверних рішень та оптимізації обробки даних, і підкреслюють важливість розробки сучасного та ефективного ORM фреймворку.

Узагальнимо список зацікавлених сторін, їх інтересів щодо використання розроблюваного програмного забезпечення, вплив на процес розробки та стратегії їх приваблення у таблицях 5.2 та 5.3.

Таблиця 5.2

Зацікавлені сторони проєкту та їхні інтереси

№	Зацікавлена сторона	Сфера зацікавленості
1	Розробники	Основною метою розробників є створення продуктивних та масштабованих систем, здатних обробляти великі обсяги даних у реальному часі. Їх цікавить можливість використання низькорівневих конструкцій мови C# та апаратного прискорення для оптимізації роботи баз даних і зменшення затримок у обробці запитів. Розробники прагнуть створювати інноваційні рішення, які допоможуть покращити продуктивність бізнес-процесів, зберігаючи зручність розробки за допомогою ORM фреймворків.
2	Великі підприємства та корпорації	Підприємства, що використовують ERP та HRM системи для управління своїми ресурсами, зацікавлені у підвищенні швидкодії своїх серверів та можливості масштабування без втрати продуктивності. Вони прагнуть зменшити час простою систем, забезпечити стабільну роботу баз даних під час великих навантажень і підвищити ефективність процесів прийняття рішень завдяки швидкій обробці інформації.
3	Інтегратори IT-рішень	Компанії, що займаються впровадженням IT-рішень для підприємств, зацікавлені в оптимізованих ORM фреймворках, які можуть забезпечити швидке й стабільне функціонування систем. Вони прагнуть отримати гнучкі та продуктивні інструменти для інтеграції, що дозволять легко адаптувати рішення під специфічні потреби клієнтів, мінімізуючи витрати на налаштування та оптимізацію.

Продовження табл. 5.2

4	Системні архітектори та інженери	Ця група зацікавлена у використанні нових технологій для підвищення продуктивності серверних систем. Системні архітектори прагнуть максимально ефективно використовувати апаратні ресурси (процесор, оперативну пам'ять), а також впроваджувати рішення, які дозволять знизити навантаження на збирач сміття та оптимізувати SQL-запити на низькому рівні. Їх цікавить можливість більшого контролю за процесами оптимізації даних.
5	Бізнес-аналітики та керівники	Керівництво підприємств зацікавлене у стабільній і швидкій обробці бізнес-даних для прийняття своєчасних і обґрунтованих рішень. Для них важливо мати системи, які не просто ефективно працюють, але й адаптуються до зростаючих потреб компанії, дозволяючи обробляти великі обсяги даних без затримок і збоїв.
6	Хмарні провайдери	Постачальники хмарних рішень зацікавлені у впровадженні високоефективних і продуктивних ORM фреймворків, які можуть використовувати апаратне прискорення для роботи з великими обсягами даних. Це дозволить їм надавати клієнтам більш стабільні та ефективні хмарні рішення з меншою вартістю володіння через оптимізацію ресурсів.

Таблиця 5.3

Аналіз зацікавлених сторін

№	Зацікавлена сторона	Сфера зацікавленості	В	З	Р=В*З
1	Розробники	Розробники прагнуть створювати інноваційні рішення, які допоможуть покращити продуктивність бізнес-процесів, зберігаючи зручність розробки за допомогою ORM фреймворків.	8	10	80
2	Великі підприємства та корпорації	Прагнуть зменшити час простою систем, забезпечити стабільну роботу баз даних під час великих навантажень і підвищити ефективність процесів прийняття рішень завдяки швидкій обробці інформації.	9	8	72

Продовження табл. 5.3

3	Інтегратори IT-рішень	Отримати гнучкі та продуктивні інструменти для інтеграції, що дозволять легко адаптувати рішення під специфічні потреби клієнтів, мінімізуючи витрати на налаштування та оптимізацію.	4	8	32
4	Системні архітектори та інженери	Системні архітектори прагнуть максимально ефективно використовувати апаратні ресурси системи.	6	8	48
5	Бізнес-аналітики та керівники	Керівництво підприємств зацікавлене у стабільній і швидкій обробці бізнес-даних для прийняття своєчасних і обґрунтованих рішень.	4	6	24
6	Хмарні провайдери	Надавати клієнтам більш стабільні та ефективні хмарні рішення з меншою вартістю володіння через оптимізацію ресурсів.	8	8	64

Отже, найбільшу силу впливу мають розробники даного застосунку та великі підприємства та корпорації.

Після проведених досліджень та аналізу проблеми необхідно розробити стратегію взаємодії з зацікавленими сторонами. Для цього використовуватимемо наявний матеріал та принципи, що стосуються розробки високопродуктивного ORM фреймворку з низькорівневими оптимізаціями.

Розробники програмного забезпечення мають найбільшу силу впливу, оскільки вони безпосередньо впливають на створення та впровадження оптимізованого ORM фреймворку. З ними будуть проводитися регулярні та тісні комунікації для координації процесу розробки, а також для забезпечення інтеграції нових технологій в інфраструктуру існуючих систем.

Великі підприємства та корпорації, які використовують ERP та HRM системи, мають силу впливу в діапазоні 50-75. Вони є ключовими

споживачами рішень, тому ми будемо активно залучати їх до обговорення вимог і зворотного зв'язку щодо продуктивності та масштабованості нової системи. Їхні інтереси будуть враховуватися під час прийняття рішень про функціональні зміни та впровадження нових оптимізацій.

Інтегратори ІТ-рішень, системні архітектори та інженери мають середню силу впливу (25-50), тому взаємодія з ними буде менш інтенсивною, але необхідною для врахування їхніх рекомендацій щодо впровадження апаратних прискорень і інтеграцій з іншими корпоративними системами. Комунікація з цією групою буде носити більш формальний характер і зосереджуватиметься на забезпеченні підтримки та відповідності вимогам до інфраструктури.

Інші зацікавлені сторони, такі як бізнес-аналітики та керівники, мають незначний вплив на технічні рішення, але їхня зацікавленість у стабільності та продуктивності систем робить їх важливими для інформування про ключові етапи впровадження та підвищення продуктивності. Комунікація з цією групою буде формальною та орієнтованою на представлення результатів роботи.

5.3. Інноваційне програмне рішення. Основні характеристики

Розроблене програмне рішення є інноваційним завдяки використанню низькорівневих конструкцій мови програмування C# та апаратного прискорення, що дозволяє значно підвищити продуктивність обробки даних на сервері. Його основні характеристики полягають у поєднанні традиційних ORM-фреймворків з сучасними методами оптимізації обчислювальних ресурсів і зниженні затримок при виконанні SQL-запитів.

- Використання низькорівневих конструкцій C#. Це дозволяє обходити обмеження високорівневих ORM-рішень, таких як Entity Framework Core. Доступ до апаратних можливостей процесора і пам'яті дає змогу досягти більш ефективної роботи зі складними

запитами, а також знизити накладні витрати на відстеження станів об'єктів і управління пам'яттю.

- Апаратне прискорення. Одна з ключових інновацій полягає у використанні апаратного прискорення на рівні серверів, що дозволяє суттєво збільшити швидкість обробки запитів до баз даних. Використання таких технологій, як SIMD (Single Instruction, Multiple Data) та багатоядерна обробка даних, робить можливим зниження часу виконання завдань, що є критично важливим для великих підприємств із великими обсягами інформації.
- Оптимізація SQL-запитів. Завдяки новому підходу до генерації SQL-запитів фреймворк генерує більш ефективні запити, мінімізуючи надмірне навантаження на базу даних. Це знижує час виконання складних запитів і забезпечує високу продуктивність при роботі з великими наборами даних.
- Масштабованість. Рішення здатне ефективно працювати з великими обсягами даних, забезпечуючи високу масштабованість без втрати продуктивності. Це особливо важливо для ERP та HRM систем, де щоденно обробляються великі масиви інформації, що зростають разом із розвитком підприємства.
- Ефективне управління пам'яттю. Знижене навантаження на збирач сміття (Garbage Collector) за рахунок оптимізації роботи з пам'яттю та об'єктами дозволяє підвищити стабільність роботи системи і запобігти затримкам при обробці великих обсягів даних.

Таким чином, це рішення не лише забезпечує високу продуктивність і масштабованість, але й значно оптимізує використання апаратних ресурсів системи, що робить його ідеальним для застосування в корпоративних системах, де стабільність і ефективність є ключовими вимогами.

5.4. Комерційний програмний продукт. Конкурентні переваги програмного продукту

Розроблене програмне рішення має значні конкурентні переваги, які роблять його ефективним і привабливим для великих підприємств і компаній, що працюють з ERP та HRM системами. Поєднання інноваційних технологій і оптимізованих підходів до обробки даних дозволяє цьому програмному продукту вирізнятися на ринку.

1. Висока продуктивність і ефективність обробки даних. Завдяки використанню низькорівневих конструкцій C# та апаратного прискорення, наш продукт забезпечує значно вищу швидкість обробки запитів і даних порівняно з традиційними ORM-фреймворками. Це дозволяє компаніям швидше отримувати результати обчислень і знижувати навантаження на сервери, що важливо для бізнесу, який потребує швидкодії в реальному часі.
2. Масштабованість і стабільність. Програмний продукт спеціально розроблений для великих підприємств з великими обсягами даних. Завдяки своїй здатності ефективно працювати в умовах масштабованості, продукт легко адаптується до зростання бізнесу та зростаючих вимог до обробки інформації, зберігаючи при цьому стабільну продуктивність.
3. Оптимізація використання ресурсів. Використання апаратного прискорення і оптимізації SQL-запитів дозволяє значно знизити навантаження на сервери та ресурси системи. Це робить наш продукт більш економічним у використанні, знижуючи потребу в дорогому серверному обладнанні або постійній оптимізації на рівні бази даних.
4. Гнучкість і налаштованість. Рішення пропонує можливість гнучкої адаптації до конкретних потреб підприємства, включаючи налаштування параметрів продуктивності, оптимізацію запитів під конкретні задачі та підтримку різних конфігурацій серверного

обладнання. Це робить його універсальним інструментом для різних бізнес-моделей.

5. Зниження витрат на підтримку та оновлення. Завдяки підвищеній стабільності і оптимізованій роботі з пам'яттю, наш продукт потребує менше втручання розробників для технічної підтримки та оновлень. Це знижує загальні витрати на його експлуатацію, дозволяючи компаніям концентруватися на своїй основній діяльності.
6. Безперервна інтеграція та підтримка сучасних стандартів. Програмне рішення легко інтегрується з існуючими корпоративними системами та відповідає сучасним стандартам безпеки й продуктивності. Це дозволяє швидко впровадити продукт у вже існуючу інфраструктуру без тривалих процесів налаштування.

Таким чином, комерційний програмний продукт вирізняється на ринку завдяки високій продуктивності, масштабованості, економії ресурсів та можливості адаптації до різних потреб бізнесу. Ці конкурентні переваги роблять його ідеальним рішенням для великих підприємств, що шукають ефективний інструмент для управління своїми даними та підвищення продуктивності серверних систем.

5.5. Ринок аналогічних програмних рішень

На сучасному ринку програмних рішень існує велика кількість ORM-фреймворків, які використовуються для спрощення роботи з базами даних у корпоративних системах. Найпопулярнішими серед них є Entity Framework Core (EF Core), NHibernate та Dapper, кожен із яких має свої сильні та слабкі сторони. EF Core забезпечує високий рівень абстракції, що дозволяє швидко розробляти рішення без глибокого занурення у SQL, але при цьому його продуктивність знижується при роботі з великими обсягами даних. NHibernate пропонує багатий функціонал для складних систем, однак, як і

EF Core, може страждати від проблем із продуктивністю. Dapper, на відміну від попередніх рішень, є "мікро ORM" і надає більш контрольовану й швидку роботу з SQL, але вимагає більше зусиль від розробника через менший рівень автоматизації.

Крім традиційних ORM-фреймворків, з'являються нові гравці на ринку, які пропонують інноваційні підходи до обробки даних, особливо в умовах великих навантажень. Такі рішення, як GraphQL та NoSQL бази даних, дозволяють обробляти дані за іншими принципами, часто уникаючи звичайних обмежень реляційних баз даних. Наприклад, GraphQL дає змогу запитувати саме ті дані, які потрібні користувачу, що знижує навантаження на базу даних. Проте, більшість компаній все ще продовжують використовувати реляційні бази даних через їхню надійність, перевіреність часом та простоту інтеграції в існуючі системи.

Однак, попри широкий вибір рішень, більшість існуючих ORM-фреймворків стикаються з тими ж проблемами: низька продуктивність при обробці великих наборів даних, надмірне використання пам'яті та обмежені можливості для низькорівневих оптимізацій. Це створює вікно можливостей для інноваційних продуктів, які можуть запропонувати ефективніше управління даними, кращу інтеграцію з сучасними апаратними можливостями та здатність масштабуватися без втрати продуктивності. Наш продукт з апаратним прискоренням та низькорівневими конструкціями C# має всі шанси зайняти цю нішу, забезпечуючи підприємствам новий рівень продуктивності й ефективності.

5.6. Унікальна ціннісна пропозиція

Унікальна ціннісна пропозиція нашого програмного продукту полягає в поєднанні високої продуктивності, масштабованості та ефективності обробки великих обсягів даних за допомогою інноваційного підходу, що використовує низькорівневі конструкції C# та апаратне прискорення. Це рішення виходить за рамки традиційних ORM-фреймворків, пропонуючи

значно вищу швидкість виконання операцій, що критично важливо для сучасних підприємств, які працюють з великими ERP та HRM системами. Завдяки цій особливості наш продукт може забезпечити стабільну роботу навіть за високих навантажень.

Ключовою особливістю є можливість тонкої оптимізації роботи з базами даних і використання апаратних ресурсів. Традиційні ORM рішення не завжди дозволяють ефективно керувати ресурсами, особливо при роботі з великими наборами даних, що призводить до перевантаження систем і зниження продуктивності. Наш продукт вирішує цю проблему, пропонуючи гнучкі налаштування та оптимізоване використання пам’яті та процесорних ресурсів. Це дозволяє компаніям не лише підвищити продуктивність своїх систем, але й знизити витрати на серверне обладнання та технічну підтримку.

Окрім цього, продукт забезпечує простоту інтеграції та гнучкість у налаштуванні, що робить його універсальним для використання у різних галузях. Це означає, що підприємства можуть легко адаптувати рішення під свої специфічні бізнес-процеси, зберігаючи при цьому високу продуктивність і стабільність роботи систем. Унікальна комбінація інноваційних технологій і зручності використання створює суттєву перевагу над конкурентами, роблячи наш продукт ідеальним вибором для компаній, що прагнуть максимально ефективно використовувати свої ресурси і забезпечити конкурентоздатність на ринку.

5.7. Потоки доходів. Структура витрат

Потоки доходів є важливою складовою будь-якої бізнес моделі. Потоки доходів нашого програмного продукту базуватимуться на декількох ключових джерелах.

1. Ліцензування програмного забезпечення. Основним джерелом доходу стане ліцензування нашого програмного продукту для

підприємств, які використовують ERP і HRM системи. Ми пропонуємо:

- Одноразову покупку з підтримкою оновлень;
- Модель підписки з доступом до регулярних оновлень і технічної підтримки.

2. Індивідуальні рішення. Для великих клієнтів, які потребують специфічної адаптації продукту під їхні унікальні бізнес-процеси, можливе впровадження індивідуальних рішень, що також стане додатковим джерелом доходів.
3. Консалтингові послуги. Додатковими джерелами доходу можуть бути консалтингові послуги з оптимізації роботи з базами даних і технічного налаштування програмного забезпечення, враховуючи його низькорівневі можливості.

Що стосується структури витрат, то основними статтями будуть наступні.

1. Витрати на розробку та підтримку програмного забезпечення. Основними статтями витрат будуть:
   - Зарплата команди розробників і тестувальників;
   - Витрати на серверні ресурси для забезпечення інфраструктури.
2. Маркетинг і просування продукту. Важливим компонентом витрат буде:
   - Проведення рекламних кампаній;
   - Презентації на технологічних конференціях;
   - Навчання потенційних клієнтів щодо переваг нашого продукту.
3. Технічна підтримка. Значна частина ресурсів буде спрямована на надання технічної підтримки, що забезпечить клієнтів своєчасними оновленнями, консультаціями та усуненням технічних проблем.
4. Дослідження та розробки. Відповідно до наших стратегічних цілей, ми інвестуватимемо в дослідження та розробки для

подальшого вдосконалення технологій апаратного прискорення та оптимізації низькорівневих алгоритмів. Ці інноваційні інвестиції допоможуть зберігати конкурентну перевагу на ринку та підвищувати ефективність програмного продукту, забезпечуючи його адаптацію до нових викликів і потреб користувачів.

5.8. Канва бізнес-моделі стартапу

Для того щоб узагальнити всі вище згадані пункти сформуємо бізнес–модель у вигляді lean canvas.

Споживачі. Великі корпорації, компанії та окремі розробники.

Проблема. Довготривала обробка великих обсягів даних.

Рішення. Програмне забезпечення у вигляді веб-додатку, яке використовує розроблену бібліотеку для швидкої та ефективної обробки даних.

Унікальна ціннісна пропозиція. Програмний продукт пропонує високу продуктивність, масштабованість та ефективність обробки даних для корпоративних клієнтів.

Структура витрат. Включає розробку продукту, зарплату команди, серверні ресурси, хостинг, маркетинг та підтримку клієнтів. Основними статтями витрат є розробка продукту, включаючи оплату праці розробників та тестувальників, витрати на серверні ресурси та хостинг. Значні ресурси також будуть вкладені у маркетинг, просування та технічну підтримку клієнтів. Окрім цього, частина коштів буде спрямована на дослідження та розвиток для подальшого вдосконалення технологій.

Канали. Прямі продажі, співпраця із системними інтеграторами для впровадження рішень у великих корпораціях.

Ключові метрики. Кількість оформлених підписок, їх термін дії та прибуток користувачів.

Прихована перевага. Можливість використання бібліотеки для власних розробок.

Таблиця 5.4

Канва бізнес–моделі

<table>
<tr>
<td rowspan="2">Проблема
Довгий час обробки великих об’ємів даних.</td>
<td>Рішення
Програмне забезпечення у вигляді веб-додатку, який використовує розроблену бібліотеку.</td>
<td rowspan="2">Унікальна ціннісна пропозиція
Програмний продукт пропонує унікальну цінність для корпоративних клієнтів, яка полягає в поєднанні високої продуктивності, масштабованості та ефективності обробки великих обсягів даних.</td>
<td>Прихована перевага
Можливість використання бібліотеки для власних розробок.</td>
<td rowspan="2">Споживачі
Великі корпорації, компанії, окремі розробники.</td>
</tr>
<tr>
<td>Ключові метрики
Кількість проданих підписок та їх термін дії, прибуток користувачів</td>
<td>Канали
Прямі продажі ліцензій на програмне забезпечення підприємствам, співпраця з системними інтеграторами для впровадження рішень у великих корпораціях.</td>
</tr>
<tr>
<td colspan="3">Структура витрат
Основними статтями витрат є розробка продукту, включаючи оплату праці розробників та тестувальників, витрати на серверні ресурси та хостинг. Значні ресурси також будуть вкладені у маркетинг, просування та технічну підтримку клієнтів. Окрім цього, частина коштів буде спрямована на дослідження та розвиток для подальшого вдосконалення технологій.</td>
<td colspan="2">Потоки доходів
Основні потоки доходів надходитимуть від ліцензування продукту, включаючи модель підписки або одноразові покупки. Крім того, додаткові доходи планується отримувати від консалтингу та надання послуг із впровадження продукту на підприємствах, а також від індивідуальної адаптації рішень для великих клієнтів.</td>
</tr>
</table>

5.9. Висновки до розділу

У п'ятому розділі було детально проаналізовано бізнес-модель стартапу, що базується на впровадженні інноваційного програмного продукту для оптимізації обробки великих обсягів даних у корпоративних ERP та HRM системах. Завдяки використанню низькорівневих конструкцій

мови C# та апаратного прискорення, продукт має суттєві конкурентні переваги, такі як висока продуктивність і масштабованість. Це дозволяє забезпечити ефективне управління бізнес-процесами та суттєво підвищити швидкість обробки даних.

Було визначено ключові джерела доходів і структуру витрат, що підкреслює важливість грамотного розподілу ресурсів та стратегії ліцензування. Ми також дослідили ринок аналогічних рішень, виявивши унікальні переваги нашого продукту, які дозволяють йому виділитися серед конкурентів. Особлива увага була приділена формуванню унікальної ціннісної пропозиції, що відповідає потребам великих підприємств і надає їм можливість оптимізувати свої бізнес-процеси.

Таким чином, запропонована бізнес-модель є стійкою та має потенціал до масштабування, що дозволить стартапу успішно виходити на ринок і задовольняти потреби клієнтів у сучасному високопродуктивному програмному забезпеченні.

ВИСНОВКИ

У цій магістерській роботі було проведено аналіз сучасних підходів до управління персоналом та визначено недоліки традиційних HRM-систем. Зокрема, встановлено, що вони не здатні забезпечити оперативну обробку великих обсягів даних у великих організаціях. Запропонована методологія розробки, заснована на використанні низькорівневих конструкцій мови програмування C# та апаратного прискорення, вирішує ці проблеми, забезпечуючи високу продуктивність, масштабованість та інтеграцію з іншими корпоративними рішеннями.

Робота фокусується на оптимізації серверної частини системи управління персоналом, враховуючи недоліки традиційних ORM-фреймворків, таких як Entity Framework Core. Розроблена бібліотека забезпечує зменшення часу виконання за рахунок використання апаратного прискорення системи, ефективного управління пам’яттю та використання пула з’єднань з базою даних. Використання GPU для апаратного прискорення дозволило досягти значного зниження часу виконання складних обчислювальних задач.

Важливою складовою роботи стало впровадження розробленого рішення на практиці. Тестування показало зменшення часу виконання операцій у середньому на 27% порівняно з Entity Framework Core і на 5% порівняно з Dapper. Апаратне прискорення дало змогу скоротити час обробки складних операцій над 500000 записами в базі даних до 97%, що робить його ідеальним для роботи з великими обсягами даних. Тестування продуктивності також показало значні переваги у використанні ресурсів. Зокрема, споживання оперативної пам’яті зменшилося на 86% порівняно з Entity Framework Core та на 2% порівняно з Dapper. Це дозволило суттєво знизити частоту викликів збору сміття (Garbage Collection), що позитивно впливає на стабільність роботи системи.

Додатково було розроблено бізнес-модель для комерціалізації програмного продукту, яка орієнтована на середні та великі підприємства. Розроблений фреймворк виділяється серед конкурентів завдяки високій продуктивності, адаптивності та ефективності використання апаратних ресурсів. Унікальна ціннісна пропозиція та стратегії монетизації, включаючи ліцензування, підтримку та консалтинг, забезпечують стійке джерело доходів для стартапу.

Таким чином, результати дослідження підтверджують, що запропонований підхід є інноваційним рішенням для корпоративних систем. Він дозволяє суттєво підвищити ефективність обробки даних, оптимізувати бізнес-процеси та забезпечити масштабованість. Запропонований ORM-фреймворк має потенціал стати важливим елементом сучасних ERP та HRM-систем, задовольняючи високі вимоги ринку та клієнтів.

СПИСОК ВИКОРИСТАНИХ ЛІТЕРАТУРНИХ ДЖЕРЕЛ

1. Smith, John. "Optimizing Data Exchange Between Server and Database" [Text] / Journal of Database Management, vol. 25, no. 3, 2021, pp. 45-60.
2. Johnson, Emily. "Improving System Performance with Asynchronous Processing" [Text] / International Conference on Software Engineering, 2022, pp. 112-125.
3. Організаційна структура системи управління персоналом підприємства [Електронний ресурс]. – Режим доступу: http://lib-net.com/content/9491_Organizaciina_stryktyra_sistemi_ypravlinnya_personalom.html.
4. Brown, David. "Effective Use of Connection Pooling in Database Applications" [Text] / ACM Transactions on Database Systems, vol. 18, no. 2, 2022, pp. 89-104.
5. Vickler, Andy. "Advanced SQL Query Optimization Techniques" [Text] / IEEE Transactions on Knowledge and Data Engineering, 2021, pp. 201-215.
6. Jean-Louis. "Understanding the Anatomy of GPUs Using Pokémon" [Text] / Published 2019.
7. Object–relational Mapping [Електронний ресурс]. – Режим доступу: https://en.wikipedia.org/wiki/Object%E2%80%93relational_mapping.
8. Microsoft Docs. "Memory and Spans in .NET" [Електронний ресурс]. – Режим доступу: https://learn.microsoft.com/uk-ua/dotnet/standard/memory-and-spans/.
9. Апаратне прискорення [Електронний ресурс]. – Режим доступу: https://uk.wikipedia.org/wiki/%D0%90%D0%BF%D0%B0%D1%80%D0%B0%D1%82%D0%BD%D0%B5_%D0%BF%D1%80%D0%B8%D1%81%D0%BA%D0%BE%D1%80%D0%B5%D0%BD%D0%BD%D1%8F.

10. Tanenbaum, Andrew S., and Bos, Herbert. "Modern Operating Systems" [Text] / Pearson Education, 2014, pp. 232-245, ISBN 978-0-13-359162-0.
11. Bass, Len, et al. "Software Architecture in Practice" [Text] / Addison-Wesley, 2012, pp. 312-328, ISBN 978-0-321-81573-6.
12. Hennessy, John L., and Patterson, David A. "Computer Organization and Design: The Hardware/Software Interface" [Text] / Morgan Kaufmann, 2020, pp. 110-125, ISBN 978-0-12-822173-4.
13. Cormen, Thomas H., et al. "Introduction to Algorithms" [Text] / MIT Press, 2009, pp. 865-887, ISBN 978-0-262-03384-8.
14. Fowler, Martin. "Refactoring: Improving the Design of Existing Code" [Text] / Addison-Wesley, 2018, pp. 98-120, ISBN 978-0-13-475759-9.
15. Gamma, Erich, et al. "Design Patterns: Elements of Reusable Object-Oriented Software" [Text] / Addison-Wesley, 1994, pp. 245-267, ISBN 978-0-201-63361-0.
16. Hecht, Thomas. "Parallel Computing for Optimization Algorithms" [Text] / Springer, 2019, pp. 76-89, ISBN 978-3-030-15532-2.
17. Li, Kai, and Schaefer, R. "High-Performance Computing for Large-Scale Data Analysis" [Text] / Wiley, 2021, pp. 123-145, ISBN 978-1-119-65190-0.
18. Klein, Mark. "Human Resource Management Systems: Challenges and Solutions" [Text] / Routledge, 2018, pp. 34-56, ISBN 978-1-138-54631-2.
19. Goodfellow, Ian, et al. "Deep Learning" [Text] / MIT Press, 2016, pp. 289-301, ISBN 978-0-262-03561-3.
20. Tan, Pang-Ning, et al. "Introduction to Data Mining" [Text] / Pearson, 2018, pp. 98-110, ISBN 978-0-13-312890-1.
21. Sedgewick, Robert, and Wayne, Kevin. "Algorithms" [Text] / Addison-Wesley, 2011, pp. 305-325, ISBN 978-0-321-57351-3.

22. Zuboff, Shoshana. "In the Age of the Smart Machine: The Future of Work and Power" [Text] / Basic Books, 1988, pp. 245-265, ISBN 978-0-465-09051-8.
23. Krohn, Philipp. "Advanced Data Structures for High-Performance Computing" [Text] / Springer, 2020, pp. 215-230, ISBN 978-3-030-23401-0.
24. Хіцко Я.В., Поваров Є.В. Метод для збільшення швидкодії обробки запитів серверною частиною застосунку // Збірник тез XVII конференції молодих вчених «Прикладна математика та комп'ютинг», Київ. – 2024. - с. 273-277.

ДОДАТКИ

Додаток 1

Копії графічних матеріалів

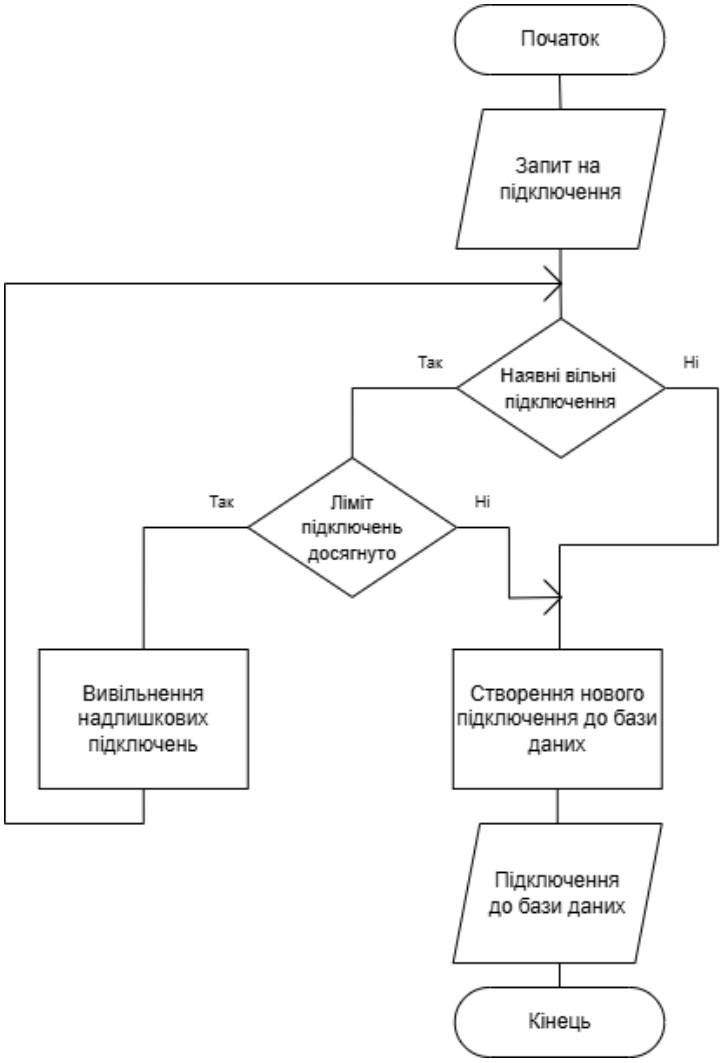
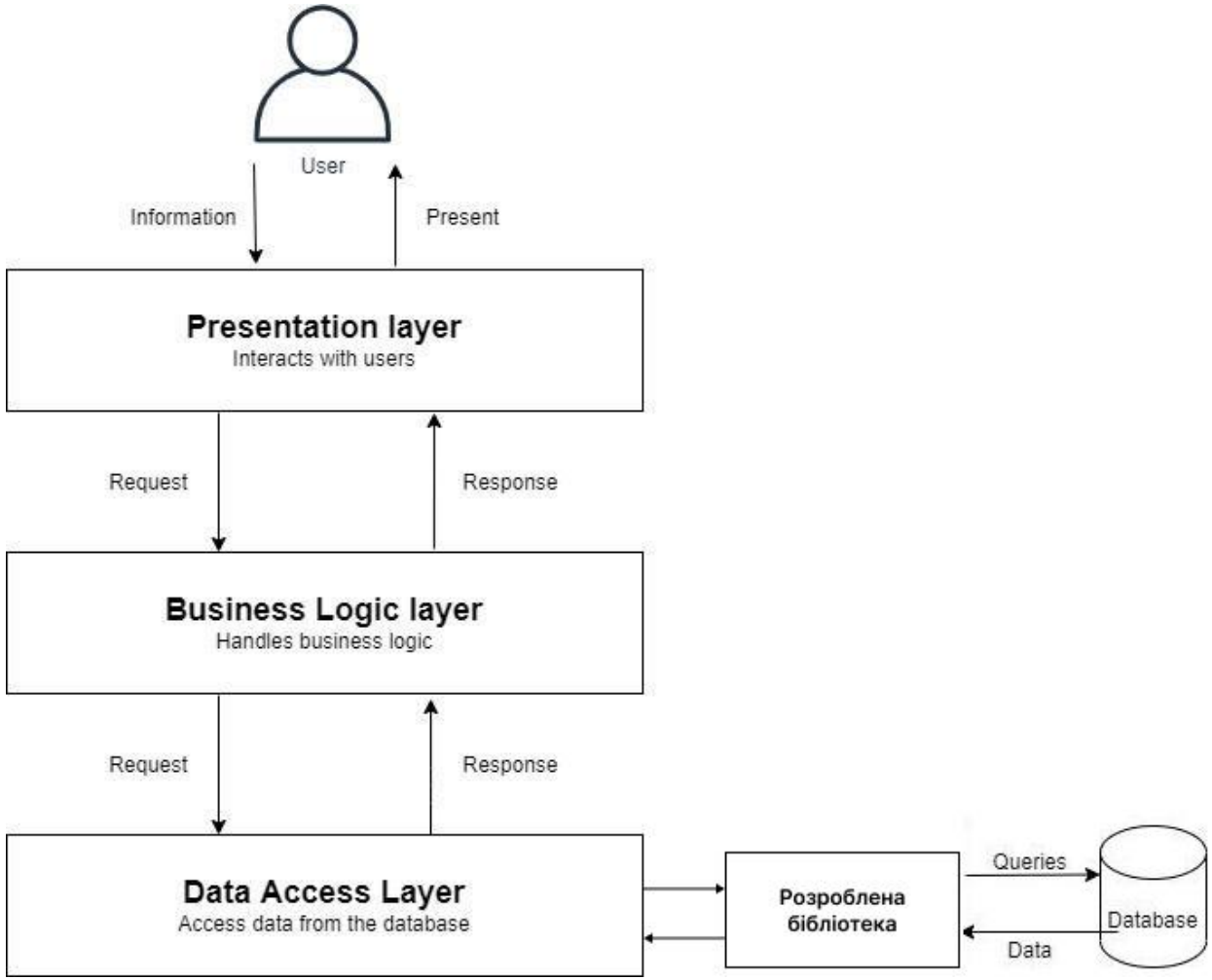
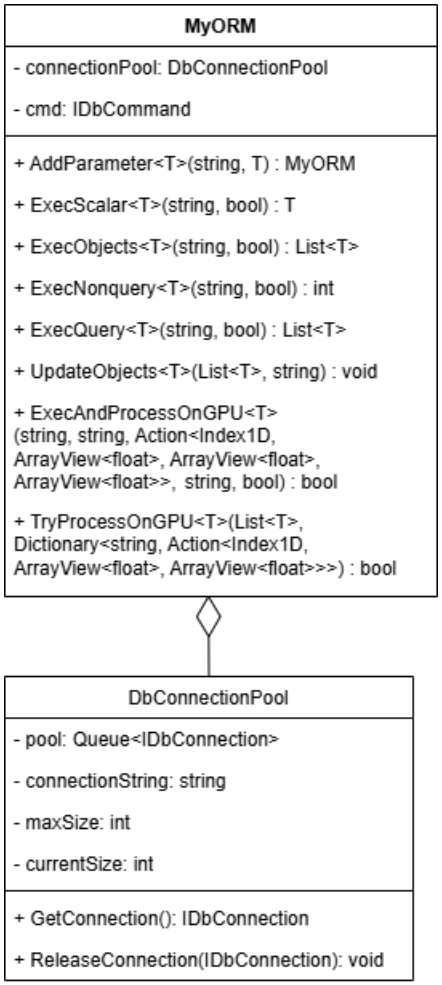


Схема роботи пула підключень
Поваров Є.В., КП-31мп



Архітектура додатку
Поваров Є.В., КП-31мп



Діаграма класів розробленої ORM
Поваров Є.В., КП-31мп

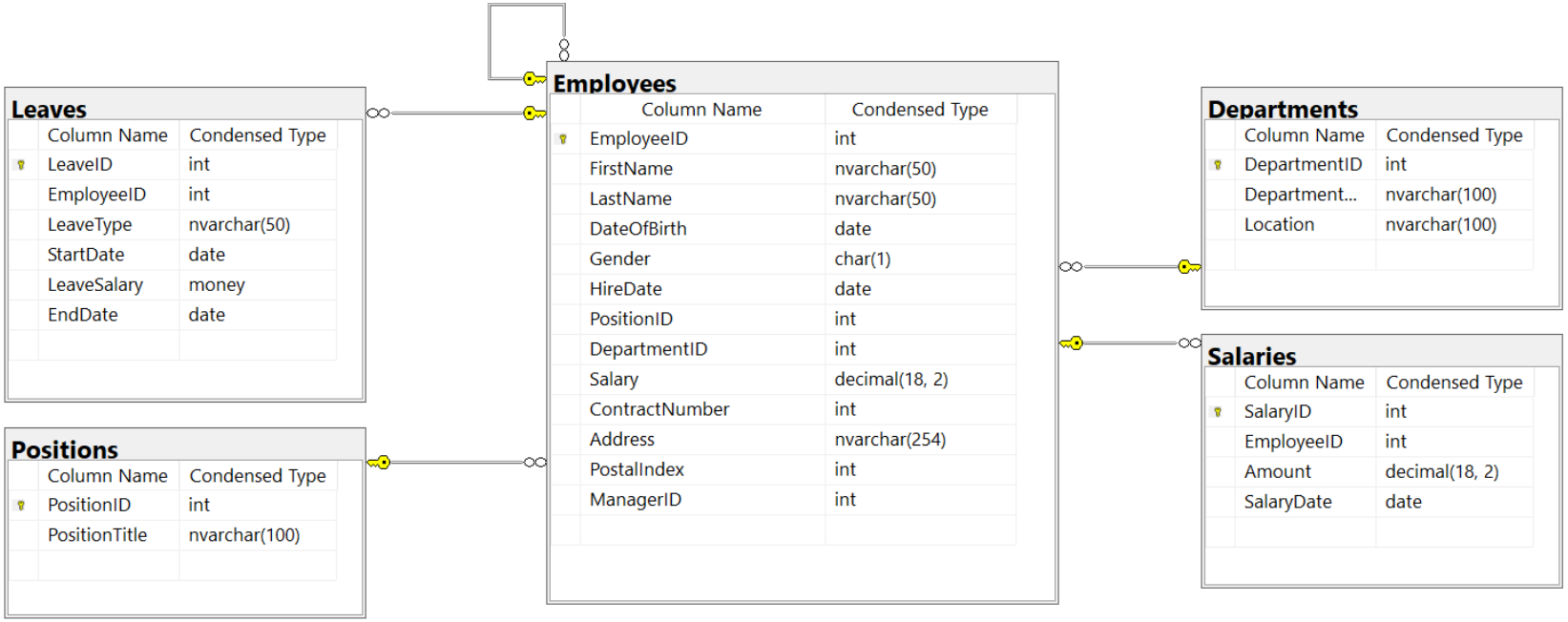
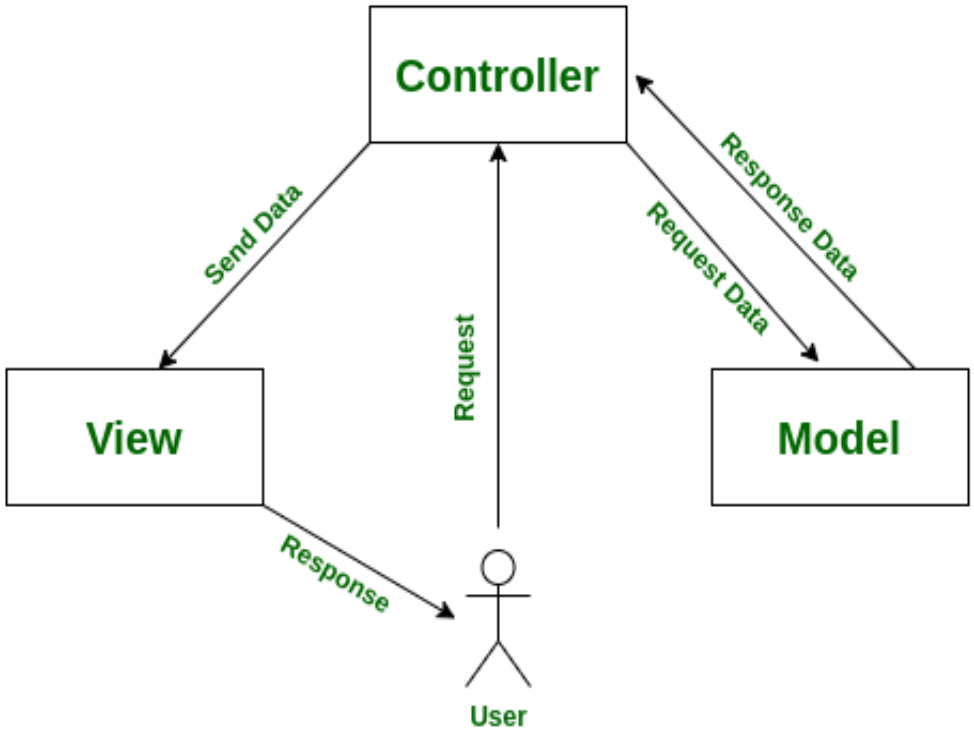
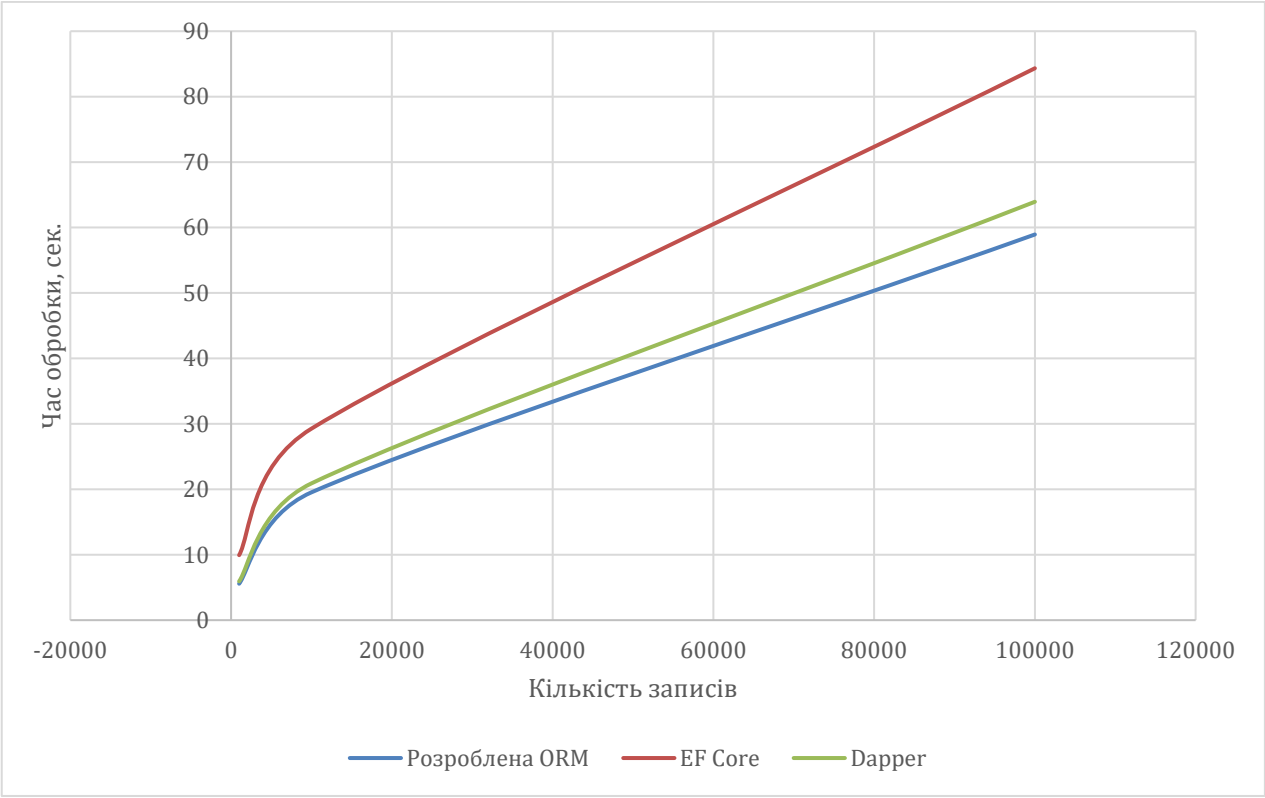


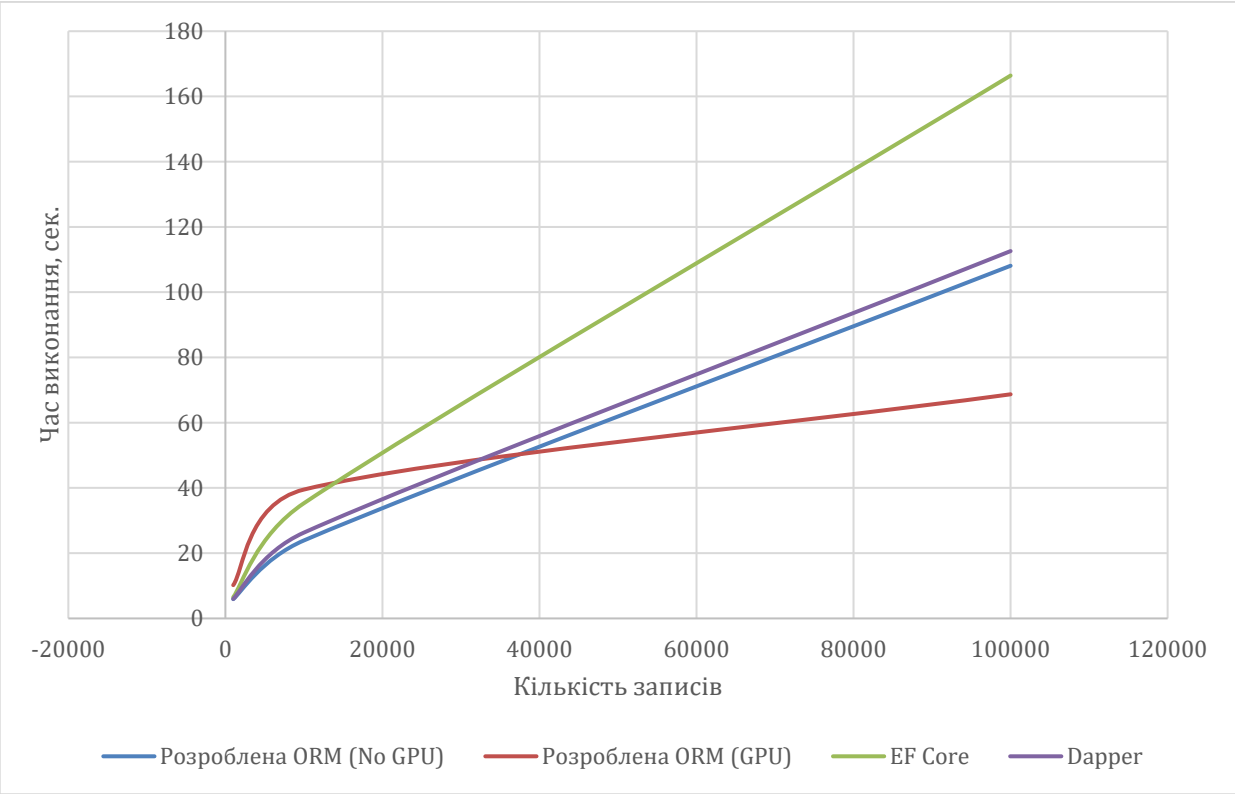
Схема бази даних
Поваров Є.В., КП-31мп



Взаємодія користувача з MVC системою
Поваров Є.В., КП-31мп



Графік порівняння швидкості розробленого підходу та аналогів
Поваров Є.В., КП-31мп



Графік швидкості обробки з та без використання апаратного прискорення
Поваров Є.В., КП-31мп

Дерево проблем
Поваров Є.В., КП-31мп

Додаток 2

Лістинг програми

Лістинг 1. MyORM.cs

```
using ILGPU;
using ILGPU.Runtime;
using ILGPU.Runtime.Cuda;
using Microsoft.Data.SqlClient;
using System;
using System.Buffers;
using System.Data;
using System.Data.Common;
using System.Reflection;

namespace MyORM
{
    public class MyORM
    {
        private readonly DbConnectionPool _connectionPool;
        private IDbConnection _connection;
        private IDbCommand _cmd;

        public MyORM(DbConnectionPool connectionPool)
        {
            _connectionPool = connectionPool;
            _connection = _connectionPool.GetConnection();
            _cmd = _connection.CreateCommand();
        }

        #region ORM Methods

        public MyORM ClearParameters()
        {
            _cmd.Parameters.Clear();
            return this;
        }

        public MyORM AddParameter<T>(string parameterName, T value)
        {
            SqlParameter parameter = new SqlParameter(parameterName, value);
            _cmd.Parameters.Add(parameter);

            return this;
        }

        public T ExecuteScalar<T>(string query, bool isStoredProc = false)
where T : struct
        {
            T result = default;


            if (isStoredProc)
            {
                _cmd.CommandType = CommandType.StoredProcedure;
            }

            _cmd.CommandText = query;
            result = (T)_cmd.ExecuteScalar();


            return result;
```

```
        }

        public int ExecuteNonQuery(string query, bool isStoredProc = false)
        {
            int nRows = 0;

            if (isStoredProc)
            {
                _cmd.CommandType = CommandType.StoredProcedure;
            }

            _cmd.CommandText = query;
            nRows = _cmd.ExecuteNonQuery();

            return nRows;
        }

        public IEnumerable<T> ExecuteQuery<T>(string query, bool isStoredProc
= false)
        {
            IList<T> result = new List<T>();

            Type t = typeof(T);

            if (isStoredProc)
            {
                _cmd.CommandType = CommandType.StoredProcedure;
            }

            _cmd.CommandText = query;
            var reader = _cmd.ExecuteReader();
            while (reader.Read())
            {
                T obj = (T)Activator.CreateInstance(t);
                t.GetProperties().ToList().ForEach(p => p.SetValue(obj,
reader[p.Name]));

                result.Add(obj);
            }

            return result;
        }

        public T ExecObject<T>(string query, bool isStoredProc = false) where
T : new()
        {
            T result = new T();

            if (isStoredProc)
            {
                _cmd.CommandType = CommandType.StoredProcedure;
            }

            _cmd.CommandText = query;

            using var reader = _cmd.ExecuteReader();
            if (reader.Read())
            {
                MapReaderToObject(reader, result);
```

```
            }

            return result;
        }

        public List<T> ExecObjects<T>(string query, bool isStoredProc = false)
where T : new()
        {
            var resultList = new List<T>();

            if (isStoredProc)
            {
                _cmd.CommandType = CommandType.StoredProcedure;
            }

            _cmd.CommandText = query;

            using var reader = _cmd.ExecuteReader();
            while (reader.Read())
            {
                T obj = new T();
                MapReaderToObject(reader, obj);
                resultList.Add(obj);
            }

            return resultList;
        }

        public void UpdateObjects_Old<T>(List<T> objects, string fieldName)
        {
            var tableName = GetTableName<T>();

            var primaryKey =
typeof(T).GetCustomAttribute<PrimaryKeyAttributes>()?.KeyName
                ?? throw new InvalidOperationException($"Primary key not
defined for class {typeof(T).Name}.");

            var idProperty = typeof(T).GetProperty(primaryKey);
            if (idProperty == null)
            {
                throw new InvalidOperationException($"Primary key property
'{primaryKey}' not found in class {typeof(T).Name}.");
            }

            var property = typeof(T).GetProperty(fieldName);
            if (property == null || !property.CanRead)
            {
                throw new InvalidOperationException($"Field '{fieldName}' not
found or is not readable in class {typeof(T).Name}.");
            }

            foreach (var (obj, index) in objects.Select((obj, index) => (obj,
index)))
            {
                var id = idProperty.GetValue(obj);
                var value = property.GetValue(obj);

                var valueParamName = $"@value{index}";
                var idParamName = $"@id{index}";
```

```
                var query = $"UPDATE {tableName} SET {fieldName} =
{valueParamName} WHERE {primaryKey} = {idParamName}";

                AddParameter(valueParamName, value)
                    .AddParameter(idParamName, id)
                    .ExecuteNonQuery(query);
            }
        }

        public void UpdateObjects<T>(List<T> objects, string fieldName)
        {
            var tableName = GetTableName<T>();

            var primaryKey =
typeof(T).GetCustomAttribute<PrimaryKeyAttributes>()?.KeyName
                ?? throw new InvalidOperationException($"Primary key not
defined for class {typeof(T).Name}.");

            var idProperty = typeof(T).GetProperty(primaryKey)
                ?? throw new InvalidOperationException($"Primary key property
'{primaryKey}' not found in class {typeof(T).Name}.");

            var fieldProperty = typeof(T).GetProperty(fieldName);
            if (fieldProperty == null || !fieldProperty.CanRead)
            {
                throw new InvalidOperationException($"Field '{fieldName}' not
found or is not readable in class {typeof(T).Name}.");
            }

            var arrayPool = ArrayPool<(object id, object value)>.Shared;
            var buffer = arrayPool.Rent(objects.Count);

            try
            {
                for (int i = 0; i < objects.Count; i++)
                {
                    buffer[i] = (idProperty.GetValue(objects[i]),
fieldProperty.GetValue(objects[i]));
                }

                var span = buffer.AsSpan(0, objects.Count);

                foreach (var (id, value) in span)
                {
                    var valueParamName = $"@value_{id}";
                    var idParamName = $"@id_{id}";

                    var query = $"UPDATE {tableName} SET {fieldName} =
{valueParamName} WHERE {primaryKey} = {idParamName}";

                    AddParameter(valueParamName, value)
                        .AddParameter(idParamName, id)
                        .ExecuteNonQuery(query);
                }
            }
            finally
            {
                arrayPool.Return(buffer, clearArray: true);
```

```
            }
        }

        #region async version

        public async Task<T> ExecuteScalarAsync<T>(string query, bool
isStoredProc = false) where T : struct
        {
            if (_cmd is not DbCommand dbCommand)
                throw new InvalidOperationException("Command must be of type
DbCommand to support async operations.");

            T result = default;

            if (isStoredProc)
            {
                dbCommand.CommandType = CommandType.StoredProcedure;
            }

            dbCommand.CommandText = query;
            var scalarResult = await dbCommand.ExecuteScalarAsync();
            if (scalarResult != null)
            {
                result = (T)Convert.ChangeType(scalarResult, typeof(T));
            }

            return result;
        }

        public async Task<int> ExecuteNonQueryAsync(string query, bool
isStoredProc = false)
        {
            if (_cmd is not DbCommand dbCommand)
                throw new InvalidOperationException("Command must be of type
DbCommand to support async operations.");

            if (isStoredProc)
            {
                dbCommand.CommandType = CommandType.StoredProcedure;
            }

            dbCommand.CommandText = query;
            return await dbCommand.ExecuteNonQueryAsync();
        }

        public async Task<IEnumerable<T>> ExecuteQueryAsync<T>(string query,
bool isStoredProc = false) where T : new()
        {
            if (_cmd is not DbCommand dbCommand)
                throw new InvalidOperationException("Command must be of type
DbCommand to support async operations.");

            var result = new List<T>();

            if (isStoredProc)
            {
                dbCommand.CommandType = CommandType.StoredProcedure;
            }
```

```
            dbCommand.CommandText = query;

            using var reader = await dbCommand.ExecuteReaderAsync();
            var properties = typeof(T).GetProperties();

            while (await reader.ReadAsync())
            {
                var obj = new T();
                foreach (var prop in properties)
                {
                    if (!reader.IsDBNull(reader.GetOrdinal(prop.Name)))
                    {
                        prop.SetValue(obj, reader[prop.Name]);
                    }
                }
                result.Add(obj);
            }

            return result;
        }

        public async Task<T> ExecObjectAsync<T>(string query, bool
isStoredProc = false) where T : new()
        {
            if (_cmd is not DbCommand dbCommand)
                throw new InvalidOperationException("Command must be of type
DbCommand to support async operations.");

            T result = new T();

            if (isStoredProc)
            {
                dbCommand.CommandType = CommandType.StoredProcedure;
            }

            dbCommand.CommandText = query;

            using var reader = await dbCommand.ExecuteReaderAsync();
            if (await reader.ReadAsync())
            {
                MapReaderToObject(reader, result);
            }

            return result;
        }

        public async Task<List<T>> ExecObjectsAsync<T>(string query, bool
isStoredProc = false) where T : new()
        {
            if (_cmd is not DbCommand dbCommand)
                throw new InvalidOperationException("Command must be of type
DbCommand to support async operations.");

            var resultList = new List<T>();

            if (isStoredProc)
            {
                dbCommand.CommandType = CommandType.StoredProcedure;
            }
```

```
            dbCommand.CommandText = query;

            using var reader = await dbCommand.ExecuteReaderAsync();
            while (await reader.ReadAsync())
            {
                T obj = new T();
                MapReaderToObject(reader, obj);
                resultList.Add(obj);
            }

            return resultList;
        }

        public async Task UpdateObjectsAsync<T>(List<T> objects, string
fieldName)
        {
            var tableName = GetTableName<T>();

            var primaryKey =
typeof(T).GetCustomAttribute<PrimaryKeyAttributes>()?.KeyName
                ?? throw new InvalidOperationException($"Primary key not
defined for class {typeof(T).Name}.");

            var idProperty = typeof(T).GetProperty(primaryKey);
            if (idProperty == null)
            {
                throw new InvalidOperationException($"Primary key property
'{primaryKey}' not found in class {typeof(T).Name}.");
            }

            var property = typeof(T).GetProperty(fieldName);
            if (property == null || !property.CanRead)
            {
                throw new InvalidOperationException($"Field '{fieldName}' not
found or is not readable in class {typeof(T).Name}.");
            }

            foreach (var (obj, index) in objects.Select((obj, index) => (obj,
index)))
            {
                var id = idProperty.GetValue(obj);
                var value = property.GetValue(obj);

                var valueParamName = $"@value{index}";
                var idParamName = $"@id{index}";

                var query = $"UPDATE {tableName} SET {fieldName} =
{valueParamName} WHERE {primaryKey} = {idParamName}";

                await AddParameter(valueParamName, value)
                    .AddParameter(idParamName, id)
                    .ExecuteNonQueryAsync(query);
            }
        }

        #endregion

        ~MyORM()
```

```
        {
            _connectionPool.ReleaseConnection(_connection);
        }

        #endregion

        #region GPU Methods

        public static bool TryProcessOnGPU<T>(int dataSize, float[] data,
Action<Index1D, ArrayView<float>, ArrayView<float>> action)
        {
            try
            {
                using var context = Context.CreateDefault();
                using var accelerator = context.CreateCudaAccelerator(0);

                using var buffer = accelerator.Allocate1D<float>(data.Length);
                using var resultBuffer =
accelerator.Allocate1D<float>(data.Length);

                buffer.CopyFromCPU(data);

                var kernel = accelerator.LoadAutoGroupedStreamKernel(action);

                kernel((int)data.Length, buffer.View, resultBuffer.View);

                accelerator.Synchronize();

                //var results = resultBuffer.GetAsArray1D();
                resultBuffer.CopyToCPU(data);

                return true;
            }
            catch (Exception)
            {
                return false;
            }
        }

        /// <summary>
        /// Multiple Rows
        /// </summary>
        public static bool TryProcessOnGPU<T>(List<T> objects,
Dictionary<string, Action<Index1D, ArrayView<float>, ArrayView<float>>>
fieldActions)
        {
            try
            {
                using var context = Context.CreateDefault();
                using var accelerator = context.CreateCudaAccelerator(0);

                foreach (var fieldAction in fieldActions)
                {
                    var fieldName = fieldAction.Key;
                    var action = fieldAction.Value;

                    var property = typeof(T).GetProperty(fieldName);
                    if (property == null || !property.CanRead ||
!property.CanWrite)
```

```
                    {
                        throw new InvalidOperationException($"Field
'{fieldName}' is not accessible or doesn't exist.");
                    }

                    var inputData = objects
                        .Select(obj =>
Convert.ToSingle(property.GetValue(obj)))
                        .ToArray();

                    using var buffer =
accelerator.Allocate1D<float>(inputData.Length);
                    using var resultBuffer =
accelerator.Allocate1D<float>(inputData.Length);

                    buffer.CopyFromCPU(inputData);

                    var kernel =
accelerator.LoadAutoGroupedStreamKernel(action);

                    kernel(inputData.Length, buffer.View, resultBuffer.View);

                    accelerator.Synchronize();

                    var resultData = resultBuffer.GetAsArray1D();

                    for (int i = 0; i < objects.Count; i++)
                    {
                        property.SetValue(objects[i], resultData[i]);
                    }
                }

                return true;
            }
            catch (Exception ex)
            {
                Console.WriteLine($"GPU processing failed: {ex.Message}");
                return false;
            }
        }

        public bool ExecAndProcessOnGPU<T>(string query, string fieldName,
            Action<Index1D, ArrayView<float>, ArrayView<float>> gpuAction,
            bool isStoredProc = false)
            where T : class, new()
        {
            try
            {
                var objects = ExecObjects<T>(query, isStoredProc).ToList();

                if (!objects.Any())
                {
                    Console.WriteLine("No objects found to process.");
                    return false;
                }

                var property = typeof(T).GetProperty(fieldName);
                if (property == null || !property.CanRead ||
!property.CanWrite)
```

```
                {
                    throw new InvalidOperationException($"Field '{fieldName}'
is not accessible or doesn't exist.");
                }

                var inputData = objects
                    .Select(obj => Convert.ToSingle(property.GetValue(obj)))
                    .ToArray();

                using var context = Context.CreateDefault();
                using var accelerator = context.CreateCudaAccelerator(0);
                using var buffer =
accelerator.Allocate1D<float>(inputData.Length);
                using var resultBuffer =
accelerator.Allocate1D<float>(inputData.Length);

                buffer.CopyFromCPU(inputData);

                var kernel =
accelerator.LoadAutoGroupedStreamKernel(gpuAction);
                kernel(inputData.Length, buffer.View, resultBuffer.View);
                accelerator.Synchronize();

                var outputData = resultBuffer.GetAsArray1D();

                for (int i = 0; i < objects.Count; i++)
                {
                    property.SetValue(objects[i], outputData[i]);
                }

                UpdateObjects(objects, fieldName);

                return true;
            }
            catch (Exception ex)
            {
                Console.WriteLine($"Error during GPU processing or update:
{ex.Message}");
                return false;
            }
        }

        public bool ExecAndProcessOnGPU<T>(
            string query,
            string fieldName,
            Action<Index1D, ArrayView<float>, ArrayView<float>,
ArrayView<float>> gpuAction,
            string conditionField,
            bool isStoredProc = false)
            where T : class, new()
        {
            try
            {
                var objects = ExecObjects<T>(query, isStoredProc).ToList();

                if (!objects.Any())
                {
                    Console.WriteLine("No objects found to process.");
                    return false;
```

```
                }

                var fieldProperty = typeof(T).GetProperty(fieldName);
                if (fieldProperty == null || !fieldProperty.CanRead ||
!fieldProperty.CanWrite)
                {
                    throw new InvalidOperationException($"Field '{fieldName}'
is not accessible or doesn't exist.");
                }

                var conditionProperty = typeof(T).GetProperty(conditionField);
                if (conditionProperty == null || !conditionProperty.CanRead)
                {
                    throw new InvalidOperationException($"Condition field
'{conditionField}' is not accessible or doesn't exist.");
                }

                var inputData = objects
                    .Select(obj =>
Convert.ToSingle(fieldProperty.GetValue(obj)))
                    .ToArray();

                var conditionData = objects
                    .Select(obj =>
                    {
                        var conditionValue = conditionProperty.GetValue(obj);
                        if (conditionValue is DateTime hireDate)
                        {
                            return (float)((DateTime.Now - hireDate).TotalDays
/ 365.0);
                        }
                        throw new InvalidOperationException("Condition field
must be a DateTime.");
                    })
                    .ToArray();

                using var context = Context.CreateDefault();
                using var accelerator = context.CreateCudaAccelerator(0);

                using var inputBuffer =
accelerator.Allocate1D<float>(inputData.Length);
                using var conditionBuffer =
accelerator.Allocate1D<float>(conditionData.Length);
                using var resultBuffer =
accelerator.Allocate1D<float>(inputData.Length);

                inputBuffer.CopyFromCPU(inputData);
                conditionBuffer.CopyFromCPU(conditionData);

                var kernel =
accelerator.LoadAutoGroupedStreamKernel(gpuAction);
                kernel(inputData.Length, inputBuffer.View,
conditionBuffer.View, resultBuffer.View);

                accelerator.Synchronize();

                var outputData = resultBuffer.GetAsArray1D();

                for (int i = 0; i < objects.Count; i++)
```

```
                {
                    fieldProperty.SetValue(objects[i], outputData[i]);
                }

                UpdateObjects(objects, fieldName);

                return true;
            }
            catch (Exception ex)
            {
                Console.WriteLine($"Error during GPU processing or update:
{ex.Message}");
                return false;
            }
        }

        #endregion

        #region private Methods

        private string GetTableName<T>()
        {
            var tableName =
typeof(T).GetCustomAttribute<TableNameAttribute>()?.Name;
            if (string.IsNullOrEmpty(tableName))
            {
                throw new InvalidOperationException($"Table name is not
defined for class {typeof(T).Name}. Use [TableName(\"TableName\")]
attribute.");
            }
            return tableName;
        }

        private void MapReaderToObject<T>(IDataReader reader, T obj)
        {
            var properties = typeof(T).GetProperties(BindingFlags.Public |
BindingFlags.Instance)
                                      .Where(p => p.CanWrite);

            foreach (var property in properties)
            {
                if (reader.HasColumn(property.Name) &&
!reader.IsDBNull(reader.GetOrdinal(property.Name)))
                {
                    var value = reader[property.Name];
                    var propertyType = property.PropertyType;

                    if (propertyType.IsGenericType &&
propertyType.GetGenericTypeDefinition() == typeof(Nullable<>))
                    {
                        var underlyingType =
Nullable.GetUnderlyingType(propertyType);
                        property.SetValue(obj, Convert.ChangeType(value,
underlyingType));
                    }
                    else
                    {
                        property.SetValue(obj, Convert.ChangeType(value,
propertyType));
```

```
                    }
                }
                else if (property.PropertyType.IsGenericType &&
                         property.PropertyType.GetGenericTypeDefinition() ==
typeof(Nullable<>))
                {
                    property.SetValue(obj, null);
                }
            }
        }

        #endregion

    }
}
```

Лістинг 2. DbConnectionPool.cs

```
using Microsoft.Data.SqlClient;
using System.Data;

public class DbConnectionPool
{
	private readonly Queue<IDbConnection> _pool = new();
	private readonly string _connectionString;
	private readonly int _maxSize;
	private int _currentSize = 0;

	private readonly object _lock = new();

	public DbConnectionPool(string connectionString, int maxSize)
	{
		_connectionString = connectionString;
		_maxSize = maxSize;
	}

	public IDbConnection GetConnection()
	{
		lock (_lock)
		{
			if (_pool.Count > 0)
			{
				var connection = _pool.Dequeue();

				if (connection.State == ConnectionState.Closed)
				{
					connection.Open();
				}

				return connection;
			}

			if (_currentSize < _maxSize)
			{
				_currentSize++;
				return CreateConnection();
			}

			throw new InvalidOperationException("No available
connections.");
		}
	}

	public void ReleaseConnection(IDbConnection connection)
	{
		lock (_lock)
		{
			if (connection.State == ConnectionState.Open)
			{
				_pool.Enqueue(connection);
			}
			else
			{
				_currentSize--;
			}

			if (_pool.Count > 5)
			{
				var excessConnections = _pool.Count - 5;
				for (int i = 0; i < excessConnections; i++)
				{
```

```
                                var conn = _pool.Dequeue();
                                conn.Close();
                                _currentSize--;
                        }
                }
        }
    }

    private IDbConnection CreateConnection()
    {
        var connection = new SqlConnection(_connectionString);
        connection.Open();
        return connection;
    }
}
```

Додаток 3

Копія презентації

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО”

ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ

КАФЕДРА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ КОМП’ЮТЕРНИХ СИСТЕМ

МЕТОД ТА ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ ДЛЯ ЗБІЛЬШЕННЯ ШВИДКОДІЇ СИСТЕМИ УПРАВЛІННЯ ПЕРСОНАЛОМ ПІДПРИЄМСТВА

Доповідач: Поваров Євгеній Валерійович

Науковий керівник: к.т.н. Хіцко Яна Володимирівна

Київ – 2024

АКТУАЛЬНІСТЬ ДОСЛІДЖЕННЯ

- Зростання конкуренції вимагає швидкої адаптації підприємств.
- Складність бізнес-процесів вимагає використання сучасних технологій.
- Висока продуктивність системи управління персоналом дозволяє підвищити ефективність підприємства.
- Дослідження відповідають потребам сучасного бізнесу.

ПОСТАНОВКА ЗАДАЧІ

Мета роботи: Підвищити швидкість обробки запитів серверною стороною застосунку за рахунок нового підходу до розробки ORM фреймворку, з поєднанням високорівневих та низькорівненвих можливостей мови програмування C#.

Об'єкт дослідження: Процес обробки даних серверною частиною застосунку та базою даних.

Предмет дослідження: Методи, способи та програмні засоби збільшення швидкості обробки запитів системою управління персоналом підприємства.

ПОСТАНОВКА ЗАДАЧІ

Завдання:

1. Аналіз існуючих методів та програмних рішень для управління персоналом, виявлення їхніх переваг та недоліків;
2. розробка методу для збільшення швидкодії системи управління персоналом підприємства з урахуванням сучасних потреб підприємств;
3. реалізація програмного забезпечення для методу для збільшення швидкодії системи управління персоналом підприємства;
4. проведення аналізу результатів, отриманих за допомогою розробленого програмного забезпечення;
5. розробка бізнес-моделі;

ТЕРМІНОЛОГІЯ

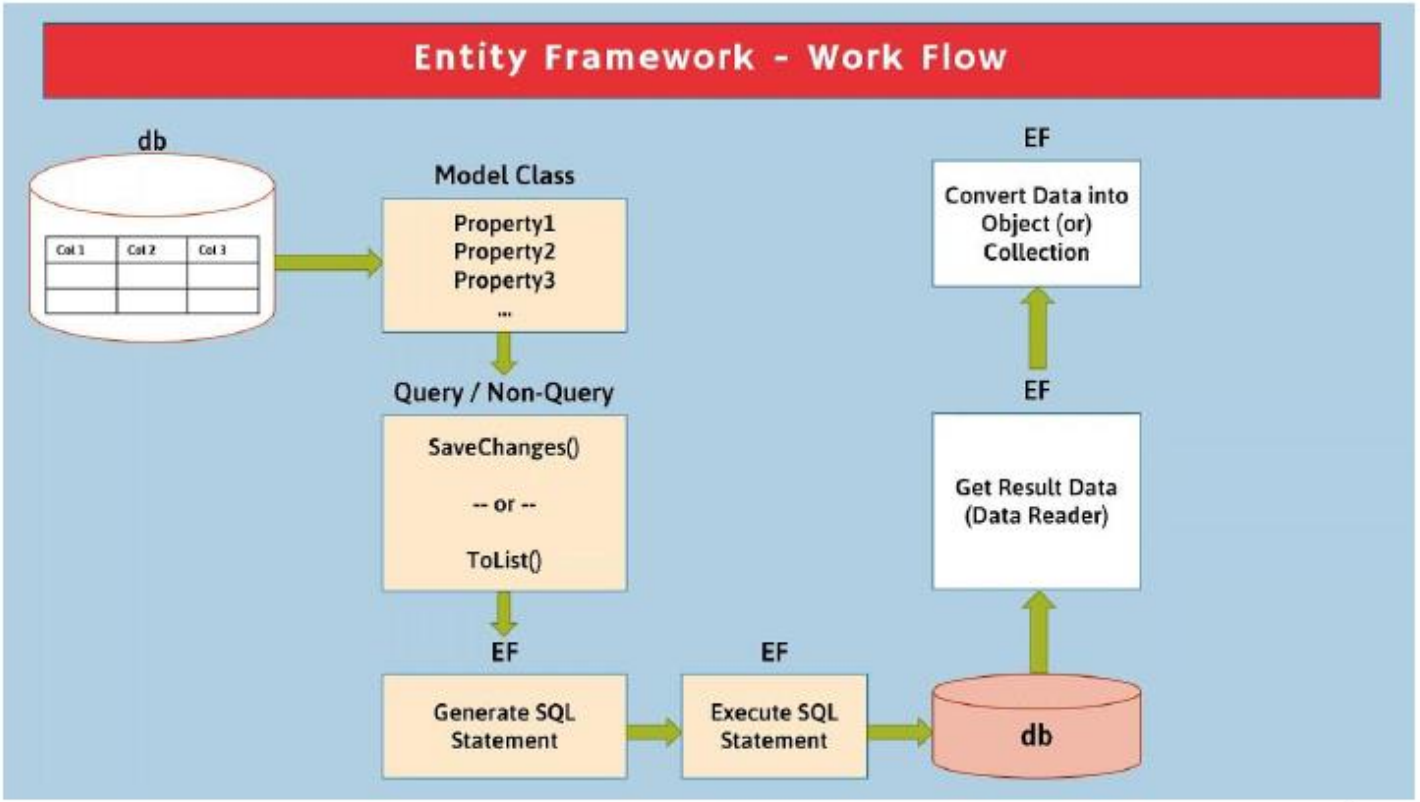
ORM (Object-relational mapping) – технологія програмування, яка зв'язує бази даних з концепціями об'єктно-орієнтованих мов програмування.

Апаратне прискорення – це використання спеціалізованого обладнання для виконання певних операцій швидше, ніж це можливо за допомогою стандартних процесорів загального призначення.

Бенчмарк – процес вимірювання продуктивності програмного або апаратного забезпечення шляхом виконання серії тестів. Він дозволяє оцінити швидкодію системи, порівняти різні рішення або оптимізації, і виявити вузькі місця в роботі програми.

ОГЛЯД ІСНУЮЧИХ ПІДХОДІВ

ОГЛЯД ІСНУЮЧИХ ПІДХОДІВ



НЕДОЛІКИ ІСНУЮЧИХ ПІДХОДІВ

- Високі витрати оперативної пам’яті при роботі з об’єктами.
- Низька швидкодії через високий рівень абстракції.
- Не розраховані на роботу з дуже великими обсягами даних

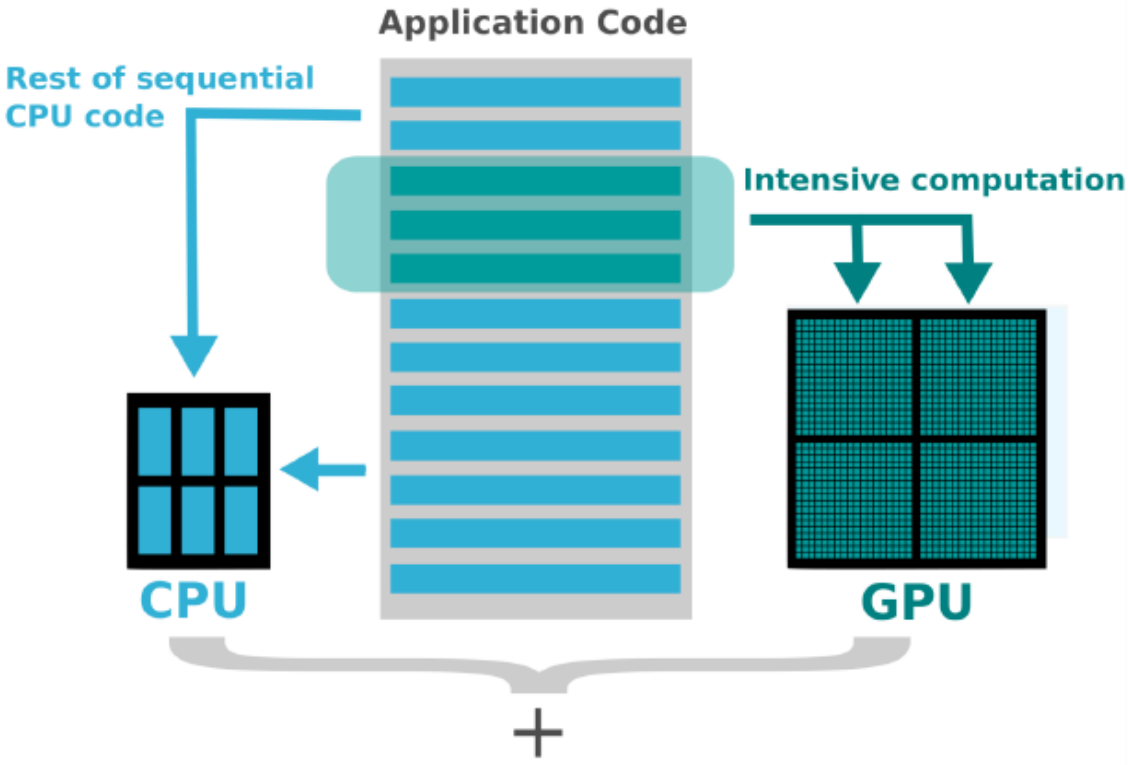
ОПИС ЗАПРОПОНОВАНОГО МЕТОДУ

Запропонований метод покликаний збільшити швидкодію обробки запитів між серверною частиною застосунку та базою даних шляхом розробки нової ORM, де вперше запропоновано використати поєднання апаратного прискорення та низькорівневих конструкцій мови C#.

Метод полягає в:

- Інтеграції апаратного прискорення
- Оптимізації коду під паралельне виконання
- Використанні низькорівневих конструкцій мови C#
- Адаптивному використанні пулів з’єднань з базою даних

АПАРАТНЕ ПРИСКОРЕННЯ



НИЗЬКОРІВНЕВІ КОНСТРУКЦІЇ

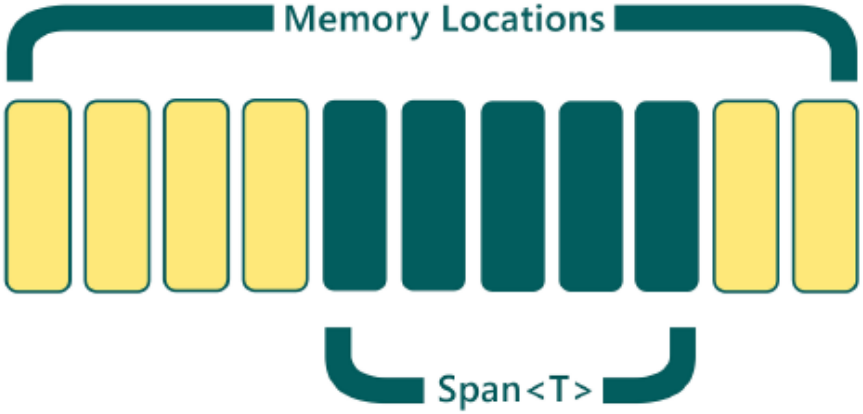
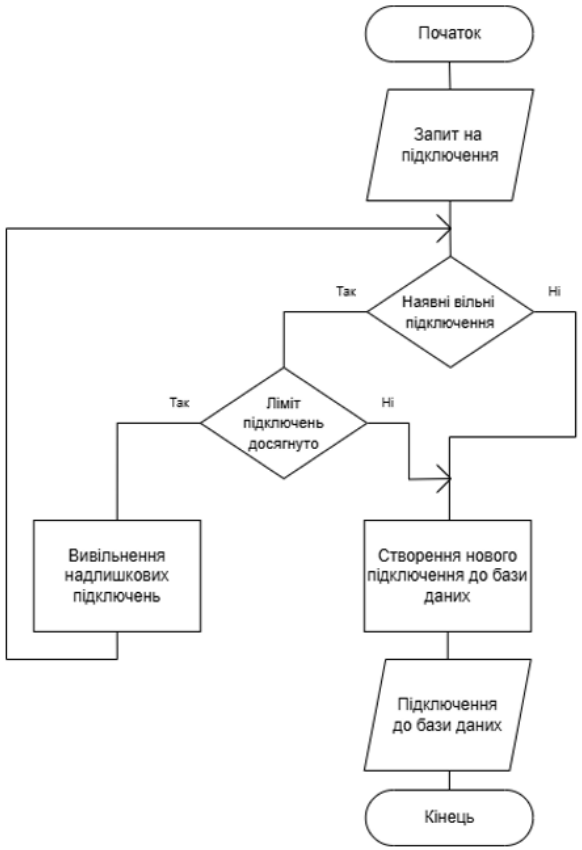
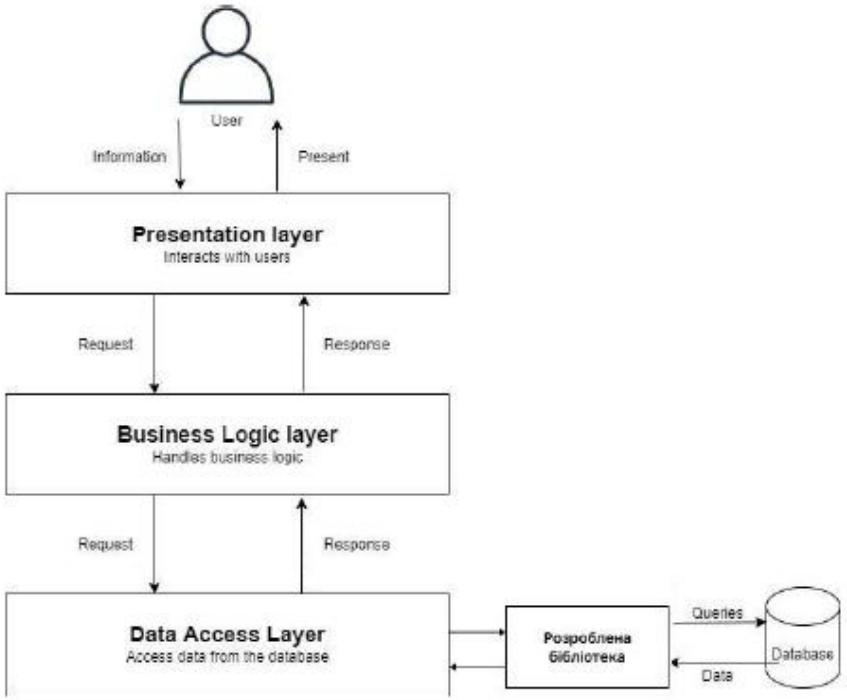


СХЕМА РОБОТИ ПУЛА ПІДКЛЮЧЕНЬ



АРХІТЕКТУРА ДОДАТКУ



MVC

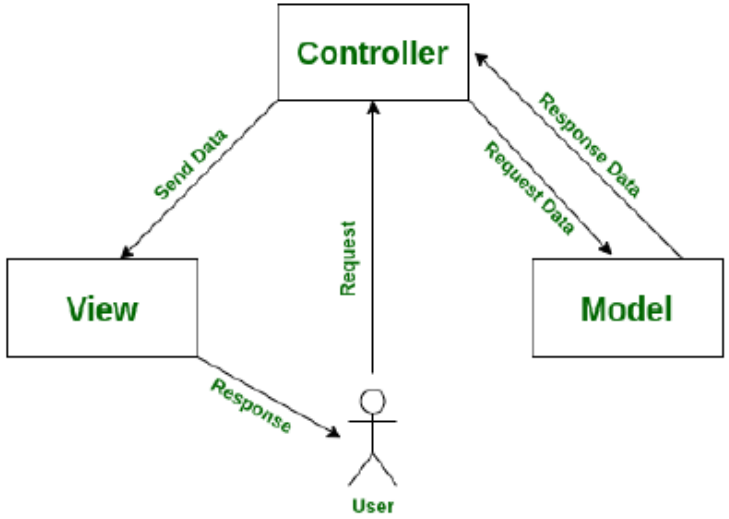
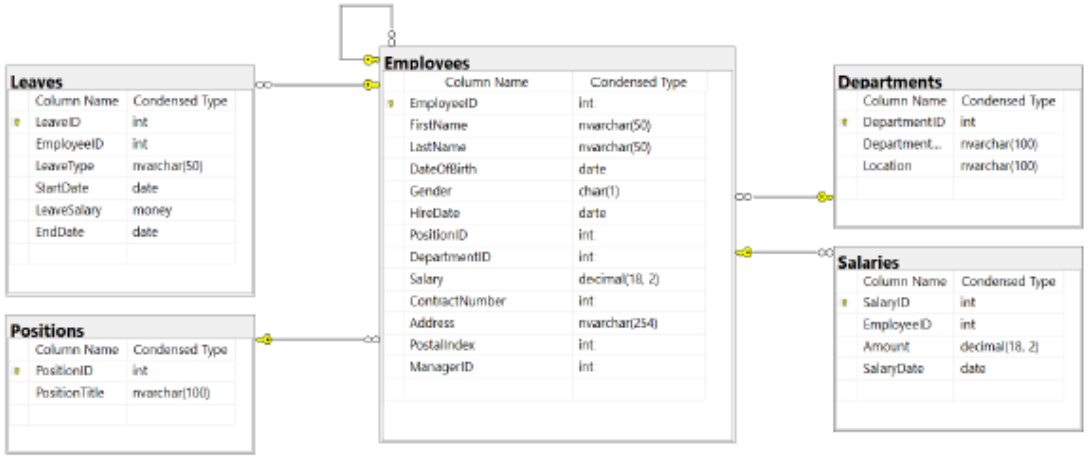
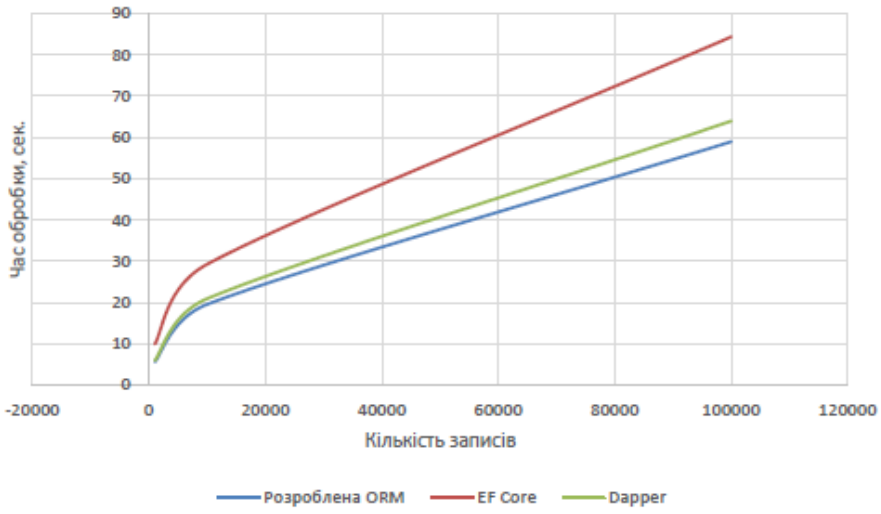

Controller
Send Data
Request
Response Data
Request Data
View
Model
Response
User

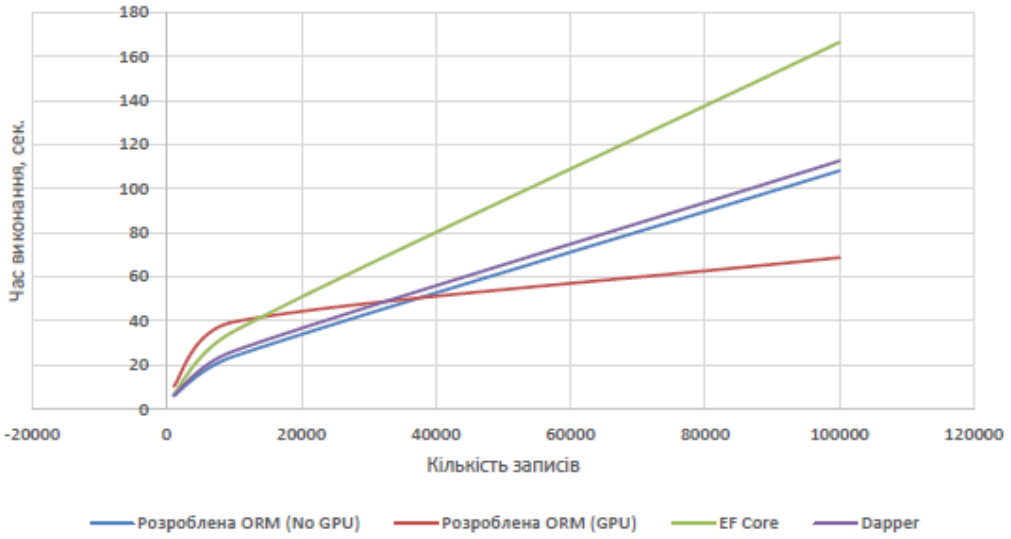
СХЕМА БАЗИ ДАНИХ



РЕЗУЛЬТАТИ БЕНЧМАРКІВ



РЕЗУЛЬТАТИ БЕНЧМАРКІВ



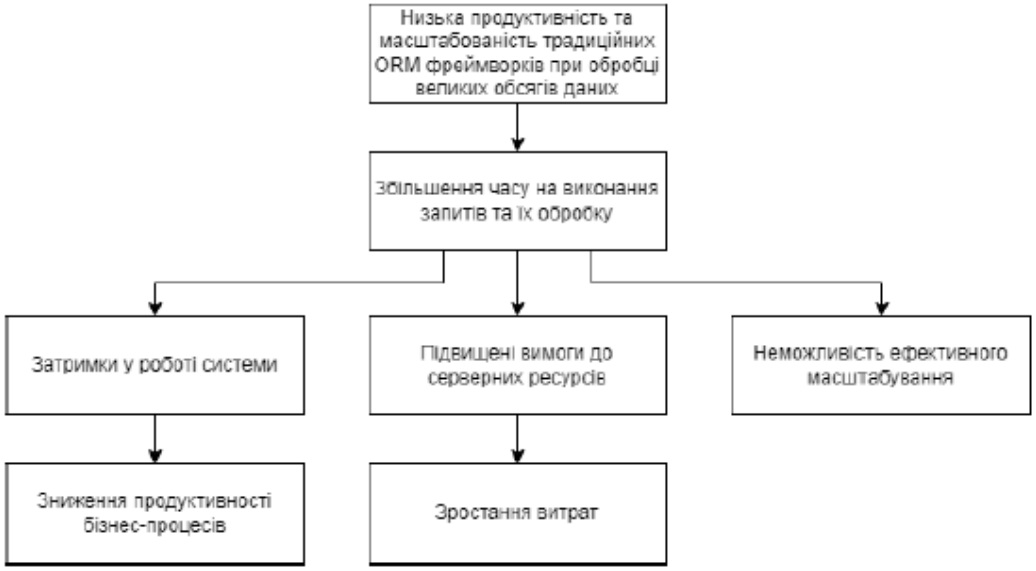
ПРАКТИЧНА ЦІННІСТЬ

Практична цінність роботи полягає в імплементації нового методу обробки запитів у програмному забезпеченні, який використовує ORM-фреймворк на мові C#. Цей метод поєднує низькорівневі конструкції мови програмування з апаратним прискоренням системи, що забезпечує значне підвищення продуктивності серверної частини додатка.

НАУКОВА НОВИЗНА

Вперше запропоновано модифікований спосіб написання ORM фреймворку, характерними рисами якого є використання низькорівневих конструкцій мови C# та апаратного прискорення системи, що дозволяє пришвидшити обробку даних сервером.

ДЕРЕВО ПРОБЛЕМ



БІЗНЕС МОДЕЛЬ

<table>
<tr>
<td rowspan="2">Проблема
Довгий час обробки великих об’ємів даних.</td>
<td>Рішення
Програмне забезпечення у вигляді веб-додатку, який використовує розроблену бібліотеку.</td>
<td rowspan="2">Унікальна ціннісна пропозиція
Програмний продукт пропонує унікальну цінність для корпоративних клієнтів, яка полягає в поєднанні високої продуктивності, масштабованості та ефективності обробки великих обсягів даних.</td>
<td>Прихована перевага
Можливість використання бібліотеки для власних розробок.</td>
<td rowspan="2">Споживачі
Великі корпорації, компанії, окремі розробники.</td>
</tr>
<tr>
<td>Ключові метрики
Кількість проданих підписок та їх термін дії, прибуток користувачів</td>
<td>Канали
Прямі продажі ліцензій на програмне забезпечення підприємствам, співпраця з системними інтеграторами для впровадження рішень у великих корпораціях.</td>
</tr>
<tr>
<td colspan="3">Структура витрат
Основними статтями витрат є розробка продукту, включаючи оплату праці розробників та тестувальників, витрати на серверні ресурси та хостинг. Значні ресурси також будуть вкладені у маркетинг, просування та технічну підтримку клієнтів. Окрім цього, частина коштів буде спрямована на дослідження та розвиток для подальшого вдосконалення технологій.</td>
<td colspan="2">Потоки доходів
Основні потоки доходів надходитимуть від ліцензування продукту, включаючи модель підписки або одноразові покупки. Крім того, додаткові доходи планується отримувати від консалтингу та надання послуг із впровадження продукту на підприємствах, а також від індивідуальної адаптації рішень для великих клієнтів.</td>
</tr>
</table>

АПРОБАЦІЯ

Основні положення і результати даної роботи були представлені на науковій конференції магістрантів та аспірантів «Прикладна математика та комп’ютинг ПМК-2024»

ВИСНОВКИ

1. Розроблено інноваційний ORM-фреймворк з використанням низькорівневих конструкцій C# та апаратного прискорення для оптимізації обробки даних на серверному рівні.

2. Порівняння запропонованого рішення з традиційними ORM-фреймворками показало значні переваги у зниженні навантаження на систему, що робить його ефективнішим для високонавантажених систем.

3. В ході аналізу ефективності та продуктивності запропонованого методу було з’ясовано, що він дозволяє досягти зменшення часу виконання класичних CRUD операцій до 50% порівняно з Entity Framework Core і до 5% порівняно з Dapper. Апаратне прискорення дало змогу скоротити час обробки складних операцій до 97%.

4. Розроблена бізнес-модель для нового програмного забезпечення, що відкриває можливості для його комерційного використання.

Дякую за увагу!

Питання?