МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Національний аерокосмічний університет ім. М. Є. Жуковського
«Харківський авіаційний інститут»

Факультет програмної інженерії та бізнесу

Кафедра інженерії програмного забезпечення

Пояснювальна записка до дипломної роботи

магістра
(освітній ступінь)

на тему «Порівняльний аналіз ефективності ORM технологій EntityFramework, Dapper та ADO.Net»

ХАІ.603.667п1.121.156352.20О

Виконав: студент 6 курсу групи № 667п2
Спеціальність 121 – Інженерія програмного забезпечення
(код та найменування)
Освітня програма Хмарні обчислення та Інтернет речей
(найменування)
Гавриленко М.С.
(прізвище й ініціали студента)
Керівник Волобуєва Л.О.
(прізвище та ініціали)
Рецензент _____
(прізвище та ініціали)

Харків – 2020

Міністерство світи і науки України
Національний аерокосмічний університет ім. М. Є. Жуковського
«Харківський авіаційний інститут»

Факультет програмної інженерії та бізнесу
(повне найменування)
Кафедра інженерії програмного забезпечення
(повне найменування)
Рівень вищої освіти другий (магістерський)

Спеціальність 121 – інженерія програмного забезпечення
(код та найменування)
Освітня програма хмарні обчислення та Інтернет речей
(найменування)

ЗАТВЕРДЖУЮ
Завідувач кафедри

_____ _____
(підпис) (ініціали та прізвище)
“_____” _____ 2020 року

З А В Д А Н Н Я
НА ДИПЛОМНУ РОБОТУ СТУДЕНТУ

Гавриленко Михайлу Сергійовичу
(прізвище, ім’я, по батькові)

1. Тема дипломної роботи Порівняльний аналіз ефективності ORM технологій EntityFramework, Dapper та ADO.Net
керівник дипломного проекту *Волобуєва Ліна Олексіївна, к.т.н, доцент*
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)
затверджені наказом вищого навчального закладу від “__” ___2020 року №
2. Термін подання студентом роботи _____
3. Вихідні дані до роботи: *рекомендації з використання різних ORM технологій.*
4. Зміст пояснювальної записки (перелік питань, які потрібно розробити)
провести аналіз сучасного стану проблеми; провести огляд існуючих підходів до розробки програм; провести огляд існуючих методів розробки; провести планування експерименту з оцінки швидкості обробки інформації; провести аналіз результатів експериментального дослідження з оцінки швидкості обробки інформації

5. Перелік графічного матеріалу
5. РПЗ – 65 стор. , рисунків – 32 шт., таблиць – 22 шт., презентація – 15 слайдів.

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
1	Волобуєва Л.О., доц. каф. 603		
2	Волобуєва Л.О., доц. каф. 603		
3	Волобуєва Л.О., доц. каф. 603		

8. Нормоконтроль ____В.А. Постернакова__ «____» ____2020 р.
(підпис) (ініціали та прізвище)

7. Дата видачі завдання _____

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проекту	Строк виконання етапів проекту	Примітка
1	Отримання і затвердження теми диплому	9.09.19	
2	Огляд і аналіз ORM технологій	10.09.19 – 29.09.19	
3	Аналіз методів агрегації данних про швидкодію ORM технологій	30.09.19 – 3.11.19	
4	Аналіз методів розрахунку і фільтрації данних	4.11.19 – 1.12.19	
5	Розробка прототипу ПЗ	2.12.19 – 15.05.20	
6	Підготовка пояснювальної записки	17.05.20 – 1.11.20	
7	Оформлення пояснювальної записки до дипломного проекту	1.11.20 - 1.12.20	
8	Передзахист дипломного проекту	1.12.20	
9	Захист дипломного проекту	21.12.20	

Студент _____ _____.____
(підпис) (прізвище та ініціали)

Керівник роботи _____ Волобуєва Л.О.
(підпис) (прізвище та ініціали)

РЕФЕРАТ

Пояснительная записка на дипломную работу (65 страниц,32 рисунков ,16 источников)

Цель дипломного проекта магистра – исследование и анализ способов работы с базами данных при помощи ORM технологий и разработка программного обеспечения для анализа быстродействия различных технологий.

Для достижения цели необходимо выполнить следующие задачи:

- Провести анализ проблем выбора той или иной технологии;
- Проанализировать доступные ORM технологии;
- Выполнить анализ особенностей различных ORM технологий;
- Выбрать методы обработки полученных данных;
- Провести анализ требований к ПО для тестирования;
- Выполнить проектирование архитектуры ПО для тестирования;
- Разработать рекомендации по использованию.

Актуальность задачи – разработка рекомендаций по использованию ORM технологий при разработке программного обеспечения, связанного с получением и обработкой информации из баз данных для языка программирования C#. Выявление различных проблем, плюсов и минусов каждой ORM технологии.

Основным практическим результатом будут рекомендации по использованию различных технологий в различных условиях разработки.

Необходимо разработать методы и модели оценки для измерения скорости обработки запросов разной сложности, размерности и назначения.

Основной целью является помощь в выборе технологии для разработки, а также выяснение проблем, связанных с разработкой. Это обеспечит следующее:

- Более качественный программный продукт по итогу разработки;
- Снижение времени на обучение новых специалистов;
- Снижение трудозатрат на разработку нового программного обеспечения;

РЕФЕРАТ

Пояснювальна записка на дипломну роботу (65 сторінок, 32 рисунка, 16 источників)

Мета дипломного проекту магістра - дослідження та аналіз способів роботи з базами даних за допомогою ORM технологій і розробка програмного забезпечення для аналізу швидкодії різних технологій.

Для досягнення мети необхідно виконати наступні завдання:

- Провести аналіз проблем вибору тієї чи іншої технології;
- Проаналізувати доступні ORM технології;
- Виконати аналіз особливостей різних ORM технологій;
- Вибрати методи обробки отриманих даних;
- Провести аналіз вимог до ПЗ для тестування;
- Виконати проектування архітектури ПЗ для тестування;
- Розробити рекомендації щодо використання.

Актуальність завдання - розробка рекомендацій по використанню ORM технологій при розробці програмного забезпечення, пов'язаного з отриманням і обробкою інформації з баз даних для мови програмування C #. Виявлення різних проблем, плюсів і мінусів кожної ORM технології.

Основним практичним результатом будуть рекомендації по використанню різних технологій в різних умовах розробки.

Необхідно розробити методи і моделі оцінки для вимірювання швидкості обробки запитів різної складності, розмірності і призначення.

Основною метою є допомога у виборі технології для розробки, а також з'ясування проблем, пов'язаних з розробкою. Це забезпечить наступне:

- Більш якісний програмний продукт за підсумком розробки;
- Зниження часу на навчання нових фахівців;
- Зниження трудовитрат на розробку нового програмного забезпечення.

ABSTRACT

Explanatory note for thesis (65 pages, 32 images, 16 sources)

The goal of the master's thesis project is to research and analyze ways of working with databases using ORM technologies and develop software for analyzing the performance of various technologies.

To achieve the goal, you must complete the following tasks:

- Analyze the problems of choosing a particular technology;
- Analyze available ORM technologies;
- Analyze the features of various ORM technologies;
- Choose methods of processing the received data;
- Analyze the requirements for software for testing;
- Design the software architecture for testing;
- Develop recommendations for use;

The relevance of the problem is the development of recommendations for the use of ORM technologies in the development of software related to the receipt and processing of information from databases for the C # programming language. Identification of various problems, pros and cons of each ORM technology.

The main practical result will be recommendations for the use of different technologies in different development conditions.

Evaluation methods and models need to be developed to measure the processing speed of requests of different complexity, dimension and purpose.

The main goal is to help in choosing the technology for development, as well as to clarify the problems associated with the development. This will ensure the following:

- Better software product based on the development results;
- Reducing the time spent on training new specialists;
- Reduction of labor costs for the development of new software.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

ORM - Object-Relational Mapping. Технологія програмування, яка зв'язує бази даних з концепціяпі об'єктно-орієнтованого програмування.

SQL – Structured Query Language. Мова програмування, що використовується для модифікації та управління даними в реляційній базі даних.

СУБД – Система Управління Базами Даних

CRUD – Базові операції додання, видалення, оновлення та получення даних з СУБД.

ВСТУП

Сьогодні дуже велике значення має робота з даними у базах даних. Для зберігання даних сьогодні використовуються різноманітні системи управління базами даних, такі як: MS SQL Server, Oracle, MySQL та безліч інших. Усі великі програмні продукти, які роботають з базами даних, так чи інакше використовують їх. Однак, для забезпечення зв’язку додатка на мові C# та базою даних, потрібен посередник.

Для роботи з даними з бази даних у додатку, нам потрібні об’єкти (сутності), а не таблиці з даними, та навпаки, для зберігання даних у базі, нам потрібні таблиці з даними, а не об’єкти. Також необхідно забезпечити інтерфейс для CRUD-операцій над даними. Загалом, потрібно позбутися необхідності писати SQL-запити та інший код мовою SQL для взаємодії з СУБД.

Рішення проблеми зберігання даних існує – це реляційні системи управління базами даних, але використання реляційної бази даних для зберігання об'єктно-орієнтованих даних змушує програмістів писати код для конвертації даних з таблиць в об’єкти, також змушує систему постійно конвертувати дані з однієї форми в другу. Це не тільки знижає продуктивність програміста, а також створює різні труднощі в обробці даних, іноді не усі типи даних з баз повністю збігаються з типами даних у додатку, наприклад розмір типу, або тип з бази не існує у додатку на C# і так далі. Це накладає обмеження на обидві сторони.

Реляційні бази даних використовують набір таблиць, що представляють прості дані. Додаткова або пов'язана інформація зберігається в інших таблицях. Часто для зберігання одного об'єкта в реляційної базі даних використовується кілька таблиць; це, в свою чергу, вимагає застосування операції JOIN для отримання всієї інформації, що відноситься до об'єкту, для її обробки. Наприклад, для зберігання даних записника, швидше за все, будуть використовуватися як мінімум дві таблиці: люди і адреси, і, можливо, навіть таблиця з телефонними номерами, коли в додатку це може бути одним об’єктом.

Так як системи керування базами даних зазвичай не реалізують реляційного подання фізичного рівня зв'язків, виконання декількох послідовних запитів (що відносяться до однієї «об'єктно-орієнтованої» структурі даних) може бути занадто витратно. Зокрема, один запит виду «знайти такого-то користувача і всі його телефони і все його адреси і повернути їх в такому форматі», швидше за все, буде виконаний швидше серії запитів виду "Знайти учасника. Знайти його адреси. Знайти його телефони ». Це відбувається завдяки роботі оптимізатора і витрат на синтаксичний аналіз запиту.

Деякі реалізації ORM автоматично синхронізують завантажені в пам'ять

об'єкти з базою даних. Для того щоб це було можливим, після створення об'єкт-в-SQL-перетворюючого SQL-запиту (класу, що реалізує зв'язок з DB) отримані дані копіюються в поля об'єкта, як у всіх інших реалізаціях ORM. Після цього об'єкт повинен стежити за змінами цих значень і записувати їх в базу даних.

Системи керування базами даних показують хорошу продуктивність на глобальних запитах, які зачіпають велику ділянку бази даних, але об'єктно-орієнтована доступ більш ефективний при роботі з малими обсягами даних, так як це дозволяє скоротити семантичний провал між об'єктної і реляційної формами даних. При одночасному існуванні цих двох різних світів збільшується складність об'єктного коду для роботи з реляційними базами даних, і він стає більш схильний до помилок.

Науково-прикладна задача - дослідження застосування різноманітних існуючих методів взаємодії з СУБД.

Об’єктом дослідження є конструювання запитів до субд.

Предметом дослідження є технології, за допомогою яких будуються запити до СУБД.

Метою даної дипломної роботи є підвищення ефективності обробки даних та підвищеня якості ПЗ, шляхом розробки практичних рекомендацій та засобів щодо вибору технологій ORM.

Методи розробки базуються на використанні мови C# сумісно з MySql.

Практичне значення отриманих результатів: в ході роботи була створено прототип програмного забезпечення для порівняння швидкості та зручності роботи ADO.NET, EntityFramework та Dapper.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ «ORM технології»

1.1Проблема взаємодії високорівневих мов програмування з таблицями баз даних

Сьогодні дуже велике значення має робота з даними у базах даних. Для зберігання даних сьогодні використовуються різноманітні системи управління базами даних, такі як: MS SQL Server, Oracle, MySQL та безліч інших. Усі великі програмні продукти, які роботають з базами даних, так чи інакше використовують їх. Однак, для забезпечення зв’язку додатка на мові C# та базою даних, потрібен посередник.

Для роботи з даними з бази даних у додатку, нам потрібні об’єкти (сутності), а не таблиці з даними, та навпаки, для зберігання даних у базі, нам потрібні таблиці з даними, а не об’єкти. Також необхідно забезпечити інтерфейс для CRUD-операцій над даними. Загалом, потрібно позбутися необхідності писати SQL-запити та інший код мовою SQL для взаємодії з СУБД.

Рішення проблеми зберігання даних існує – це реляційні системи управління базами даних, але використання реляційної бази даних для зберігання об'єктно-орієнтованих даних змушує програмістів писати код для конвертації даних з таблиць в об’єкти, також змушує систему постійно конвертувати дані з однієї форми в другу. Це не тільки знижає продуктивність програміста, а також створює різні труднощі в обробці даних, іноді не усі типи даних з баз повністю збігаються з типами даних у додатку, наприклад розмір типу, або тип з бази не існує у додатку на C# і так далі. Це накладає обмеження на обидві сторони.

Реляційні бази даних використовують набір таблиць, що представляють прості дані. Додаткова або пов'язана інформація зберігається в інших таблицях. Часто для зберігання одного об'єкта в реляційної базі даних використовується кілька таблиць; це, в свою чергу, вимагає застосування операції JOIN для отримання всієї інформації, що відноситься до об'єкту, для її обробки. Наприклад, для зберігання даних записника, швидше за все, будуть використовуватися як мінімум дві таблиці: люди і адреси, і, можливо, навіть таблиця з телефонними номерами, коли в додатку це може бути одним об’єктом.

Так як системи керування базами даних зазвичай не реалізують реляційного подання фізичного рівня зв'язків, виконання декількох послідовних запитів (що відносяться до однієї «об'єктно-орієнтованої» структурі даних) може бути занадто витратно. Зокрема, один запит виду «знайти такого-то користувача і всі його телефони і все його адреси і повернути їх в такому форматі», швидше за все, буде виконаний швидше серії запитів виду "Знайти учасника. Знайти його

адреси. Знайти його телефони ». Це відбувається завдяки роботі оптимізатора і витрат на синтаксичний аналіз запиту.

Деякі реалізації ORM автоматично синхронізують завантажені в пам'ять об'єкти з базою даних. Для того щоб це було можливим, після створення об'єкт-в-SQL-перетворюючого SQL-запиту (класу, що реалізує зв'язок з DB) отримані дані копіюються в поля об'єкта, як у всіх інших реалізаціях ORM. Після цього об'єкт повинен стежити за змінами цих значень і записувати їх в базу даних.

Системи керування базами даних показують хорошу продуктивність на глобальних запитах, які зачіпають велику ділянку бази даних, але об'єктно-орієнтована доступ більш ефективний при роботі з малими обсягами даних, так як це дозволяє скоротити семантичний провал між об'єктної і реляційної формами даних. При одночасному існуванні цих двох різних світів збільшується складність об'єктного коду для роботи з реляційними базами даних, і він стає більш схильний до помилок.

Сьогодні багато бізнес-розробників повинні використовувати два (або більше) мови програмування: високорівнева мова для бізнес-логіки і рівнів подання, і мову запитів для взаємодії з базою даних (наприклад, MySQL). Для ефективної роботи розробник повинен добре володіти кількома мовами, крім того, виникають невідповідності між мовами в середовищі розробки.

Наприклад додаток, який використовує API для доступу к даним, щоб виконати запит до бази указує весь запит у лапках. Цей запит не перевіряється компілятором на синтаксичні помилки, не має функції авто постанови, яка дуже прискорює написання коду, та перевірки типів даних і так далі.

LINQ дозволяє розробникам формувати запити на основі наборів в своєму коді програми без використання окремої мови запитів. Можна створювати запити LINQ до різних перелічуваних джерел даних (тобто до джерела даних, який реалізує інтерфейс IEnumerable), наприклад до структур даних в пам'яті, XML-документах, базах даних SQL та об’єктів DataSet. Незважаючи на те, що ці джерела даних реалізовані різними способами, у всіх них використовується однаковий синтаксис та конструкції. Через те, що запити можуть бути сформовані на мові програмування, немає необхідності використовувати іншу мову запитів, впроваджений у вигляді строкових літералів, які не можуть бути перевірені компілятором. Інтеграція запитів в мову програмування також дозволяє програмістам Visual Studio підвищити продуктивність завдяки забезпеченню перевірки типу і синтаксису під час компіляції, а також IntelliSense. Ці функції зменшують витрати на налагодження запитів і пошук помилок.

Передача даних з таблиць SQL в об'єкти в пам'яті часто буває складною і

сприяє здійсненню помилок. Постачальник LINQ, реалізований LINQ to DataSet і LINQ to SQL, перетворює вихідні дані в колекції об'єктів на основі IEnumerable. Програміст завжди бачить дані як колекції IEnumerable як при запиті, так і при оновленні. Для написання запитів до цих колекцій надана повна підтримка технології IntelliSense.

1.2Рішення проблеми взаємодії високорівневих мов програмування з базами даних

Разработано безліч пакетів, що усувають необхідність в перетворенні об'єктів для зберігання в реляційних базах даних.

Деякі пакети вирішують цю проблему, надаючи бібліотеки класів, здатних виконувати такі перетворення автоматично. Маючи список таблиць в базі даних і об'єктів в програмі, вони автоматично перетворять запити з одного виду в інший. В результаті запиту об'єкту «людина» (з прикладу з адресною книгою) необхідний SQL-запит буде сформований і виконаний, а результати «чарівним» чином перетворені в об'єкти «номер телефону» всередині програми.

З точки зору програміста система повинна виглядати як постійне сховище об'єктів. Він може просто створювати об'єкти і працювати з ними як зазвичай, а вони автоматично будуть зберігатися в реляційної базі даних.

На практиці все не так просто і очевидно. Всі системи ORM зазвичай виявляють себе в тому чи іншому вигляді, зменшуючи в деякому роді можливість ігнорування бази даних. Більш того, шар транзакцій може бути повільним і неефективним (особливо в термінах згенерованого SQL). Все це може привести до того, що програми будуть працювати повільніше і використовувати більше пам'яті, ніж програми, написані «вручну».

Але ORM позбавляє програміста від написання великої кількості коду, часто одноманітного і схильного до помилок, тим самим значно підвищуючи швидкість розробки. Крім того, більшість сучасних реалізацій ORM дозволяють програмісту при необхідності самому жорстко поставити код SQL-запитів, який буде використовуватися при тих чи інших діях (збереження в базу даних, завантаження, пошук і т. Д.) З постійним об'єктом.

1.3 ADO.NET

ADO.NET надає собою технологію роботи з даними, яка заснована на платформі .NET Framework. Ця технологія являє нам набір класів, через які ми

можемо відправляти запити до баз даних, встановлювати підключення, отримувати відповідь від бази даних і виробляти ряд інших операцій.

Причому важливо зазначити, що систем управління баз даних може бути безліч. У своїй сутності вони можуть відрізнятися. MS SQL Server, наприклад, для створення запитів використовує мову T-SQL, а MySQL і Oracle застосовують мову PL-SQL. Різні системи баз даних можуть мати різні типи даних. Також можуть відрізнятися якісь інші моменти. Однак функціонал ADO.NET побудований таким чином, щоб надати розробникам уніфікований інтерфейс для роботи з самими різними СУБД.

Основу інтерфейсу взаємодії з базами даних в ADO.NET представляють наступні об’єкти: Connection, Command, DataReader, DataSet і DataAdapter. За допомогою об'єкта Connection відбувається установка підключення до джерела даних. Об'єкт Command дозволяє виконувати операції з даними з БД. Об'єкт DataReader зчитує отримані в результаті запиту дані. Об'єкт DataSet призначений для зберігання даних з БД і дозволяє працювати з ними незалежно від БД. І об'єкт DataAdapter є посередником між DataSet і джерелом даних. Головним чином, через ці об'єкти і буде йти робота з базою даних.

Однак щоб використовувати один і той же набір об'єктів для різних джерел даних, необхідний відповідний провайдер даних. Власне через провайдер даних в ADO.NET і здійснюється взаємодія з базою даних. Причому для кожного джерела даних в ADO.NET може бути свій провайдер, який власне і визначає конкретну реалізацію вищевказаних класів.

1.3.1 Фундаментальні класи ADO.NET

Постачальник даних (data provider) - це набір класів ADO.NET, які дозволяють отримувати доступ до певної базі даних, виконувати команди SQL та видавати дані. По суті, постачальник даних - це міст між вашим додатком і джерелом даних.

У першому наближенні постачальник даних можна розглядати як набір типів, визначених в даному просторі імен, який призначений для взаємодії з конкретним джерелом даних. Однак незалежно від використовуваного постачальника даних, кожен з них визначає набір класів, що забезпечують основну функціональність.

Класи ADO.NET групуються в кілька просторів імен. Кожен постачальник має свій власний простір імен, а узагальнені класи на зразок DataSet знаходяться в просторі імен System.Data. Нижче описані найбільш важливі простору імен для базової підтримки ADO.NET:

System.Data. Містить ключові класи контейнерів даних, які моделюють стовпці, відносини, таблиці, набори даних, рядки, уявлення і обмеження. Додатково містить ключові інтерфейси, які реалізовані об'єктами даних, заснованими на з'єднаннях.

System.Data.Common. Містить базові, найбільш абстрактні класи, які реалізують деякі з інтерфейсів з System.Data і визначають ядро функціональності ADO.NET. Постачальники даних успадковуються від цих класів (DbConnection, DbCommand і т.п.), створюючи власні спеціалізовані версії.

System.Data.OleDb. Містить класи, використовувані для підключення до постачальника OLE DB, включаючи OleDbCommand, OleDbConnection і OleDbDataAdapter. Ці класи підтримують більшість постачальників OLE DB, але не ті, що вимагають інтерфейсів OLE DB версії 2.5

System.Data.SqlClient. Містить класи, використовувані для підключення до бази даних Microsoft SQL Server, в тому числі SqlDbCommand, SqlDbConnection і SqlDbDataAdapter. Ці класи оптимізовані для використання інтерфейсу TDS до SQL Server.

System.Data.OracleClient. Містить класи, необхідні для підключення до бази даних Oracle (версії 8.1.7 і вище), в тому числі OracleCommand, OracleConnection і OracleDataAdapter. Ці класи використовують оптимізований інтерфейс OCI (Oracle Call Interface - Інтерфейс викликів Oracle)

System.Data.Odbc. Містить класи, необхідні для підключення до більшості драйверів ODBC, такі як OdbcCommand, OdbcConnection, OdbcDataReader і OdbcDataAdapter. Драйвери ODBC поставляються для всіх видів джерел даних і конфігуруються через значок Data Sources (Джерела даних) панелі управління.

System.Data.SqlTypes. Містить структури, відповідні вбудованим типам даних SQL Server. Ці класи не є необхідними, але надають альтернативу застосуванню стандартних типів даних .NET, що вимагають автоматичного перетворення.

Схематично архітектуру ADO.NET можна представити таким чином (рисунок 1.1)

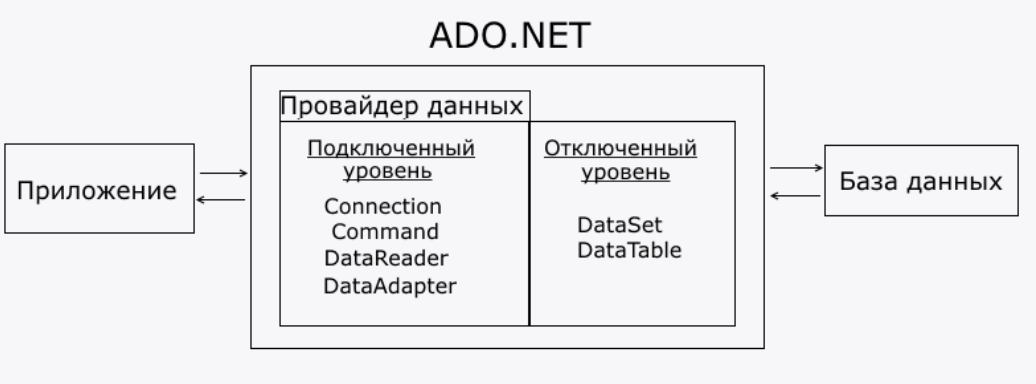

Рисунок 1.1 – Схема ADO.NET

Функціонально класи ADO.NET можна розбити на два рівня: підключений і відключений. Кожен провайдер даних .NET реалізує свої версії об'єктів Connection, Command, DataReader, DataAdapter і ряду інших, який складають підключений рівень. Тобто за допомогою них встановлюється підключення до БД і виконується з нею взаємодію. Як правило, реалізації цих об'єктів для кожного конкретного провайдера в своїй назві мають префікс, який вказує на провайдер:

Object	SQL Server	OLE DB	ODBC
Connection	SqlConnection	OleDbConnection	OdbcConnection
Command	SqlCommand	OleDbCommand	OdbcCommand
Data reader	SqlDataReader	OleDbDataReader	OdbcDataReader
Data adapter	SqlDataAdapter	OleDbDataAdapter	OdbcDataAdapter

Рисунок 1.2 – Версії об’єктів для різних провайдерів

Інші класи, такі як DataSet, DataTable, DataRow, DataColumn і ряд інших складають відключений рівень, так як після отримання даних в DataSet ми можемо працювати з цими даними незалежно від того, чи встановлено підключення чи ні. Тобто після отримання даних з БД додаток може бути відключено від джерела даних.

Бібліотеки ADO.NET можна застосовувати трьома концептуально різними способами: в підключеному режимі, в автономному режимі і за допомогою технології Entity Framework. При використанні підключеного рівня (connected layer), кодова база явно підключається до відповідного сховища даних і відключається від нього. При такому способі використання ADO.NET зазвичай

відбувається взаємодія зі сховищем даних за допомогою об'єктів підключення, об'єктів команд і об'єктів читання даних.

Автономний рівень (disconnected layer), дозволяє працювати з набором об'єктів DataTable (містяться в DataSet), який представляє на стороні клієнта копію зовнішніх даних. При отриманні DataSet за допомогою відповідного об'єкта адаптера даних підключення відкривається і закривається автоматично. Зрозуміло, що цей підхід допомагає швидко звільняти підключення для інших викликів і підвищує масштабованість систем.

Отримавши об'єкт DataSet, що викликає код може переглядати і обробляти дані без витрат на мережевий трафік. А якщо потрібно занести зміни в сховище даних, то адаптер даних (разом з набором операторів SQL) задіюється для поновлення даних - при цьому підключення відкривається заново для проведення оновлень в базі, а потім відразу ж закривається.

Після випуску .NET 3.5 SP1 в ADO.NET з'явилася підтримка нового API, яка називається Entity Framework (скорочено EF). Технологія EF показує, що багато низькорівневі деталі роботи з базами даних (наприклад, складні SQL-запити) приховані від програміста і відпрацьовуються за нього при генерації відповідного LINQ-запиту (наприклад, LINQ з Entities).

1.3.2Стандартизація в ADO.NET

На перший погляд може здатися, що ADO.NET пропонує фрагментовану модель, оскільки не включає узагальненого набору об'єктів, які працювали б з безліччю типів баз даних. В результаті, коли ви переходите від однієї реляційної СУБД до іншої, доводиться модифікувати код доступу до даних для використання іншого набору класів.

Але навіть незважаючи на те, що різні постачальники даних .NET використовують різні класи, всі вони певним чином стандартизовані. Точніше кажучи, кожен постачальник заснований на одному і тому ж наборі інтерфейсів і базових класів. Так, наприклад, об'єкт Connection реалізує інтерфейс IDbConnection, який визначає такі ключові методи, як Open () і Close (). Подібна стандартизація гарантує, що кожен клас Connection буде працювати однаковим способом мислення й надасть один і той же набір ключових властивостей і методів.

Оскільки кожен постачальник використовує одні й ті ж інтерфейси і базові класи, можна писати узагальнений код доступу до даних (з додатком невеликих додаткових зусиль), працюючи з інтерфейсами, а не класами постачальників.

Оскільки кожен постачальник реалізований окремо, він може використовувати відповідну оптимізацію (це відрізняється від моделі ADO, де кожен виклик бази даних повинен проходити через загальний рівень, перш ніж досягне що лежить в основі драйвера бази даних). Крім того, спеціалізовані постачальники можуть додавати нестандартні засоби, яких не мають інші постачальники (наприклад, можливість SQL Sever виконувати XML-запити).

ADO.NET також має інший рівень стандартизації - DataSet. Клас DataSet - це контейнер даних загального призначення, які витягуються з однієї або більше таблиць джерела даних. DataSet повністю узагальнено; іншими словами, спеціалізовані постачальники не визначають власних спеціалізованих версій класу DataSet.

Незалежно від того, який постачальник даних застосовується, можна витягати дані і розміщати їх в повністю автономний DataSet однаковим чином. Це полегшує відділення коду, що витягує дані, від коду, який займається обробкою їх. У разі зміни лежить в основі бази даних доведеться змінити тільки код, який видобуває дані, але якщо використовується DataSet, а інформація має одну і ту ж структуру, модифікувати спосіб її обробки не знадобиться.

1.3.3 Переваги і нововведення в ADO.NET

Використання роз'єднаною моделі доступу до даних. У клієнт-серверних додатках традиційно використовується технологія доступу до джерела даних при якій з'єднання з базою підтримується постійно. Однак після широкого поширення додатків, орієнтованих на Інтернет, було виявлено деякі недоліки такого підходу. Спробуємо виявити деякі з них.

З'єднання з базою даних вимагають виділення системних ресурсів, що може бути критично при великому навантаженні сервера. Хоча постійне з'єднання дозволяє дещо прискорити роботу програми, загальний збиток від розтрати системних ресурсів зводить перевагу нанівець.

Специфіка веб додатків не дозволяє серверу в кожен момент часу знати, що необхідно користувачу. Тобто до наступного запиту сервер не має уявлення, чи потрібно ще підтримувати з'єднання.

Досвід розробників показав, що додатки з постійним з'єднанням з джерелом даних надзвичайно важко піддаються масштабування. Хоча існують і інші недоліки, наведені, найбільш істотні і найбільш зустрічаються. Всі ці проблеми породжуються постійним з'єднанням з базою даних і вирішуються в ADO.NET, де використовується інша модель доступу. Тепер з'єднання

встановлюється лише на той короткий час, коли необхідно проводити операції над базою даних.

Слід визнати, що нова технологія іноді все ж програє традиційної. Для цих випадків рекомендовано використовувати ADO. Прикладами таких додатків служать програми проводять часті і об'ємні зміни змісту записів.

Зберігання даних в об'єктах DataSet. При роботі з базою даних нам найчастіше доводиться працювати не з однією, а декількома записами. Більш того, дані ці можуть збиратися з різних таблиць. У роз'єднаною моделі доступу до бази даних не має сенсу з'єднуватися і джерелом даних при кожному зверненні. Виходячи з цього, представляється логічним зберігати кілька рядків і звертатися до них при необхідності. Для цих цілей і використовується DataSet.

DataSet представляє собою, спрощену реляційну базу даних і може виконувати найбільш типові для таких баз даних операції. Тепер, на відміну від Recordset ми можемо зберігати в одному DataSet відразу кілька таблиць, зв'язки між ними, виконувати операції вибірки, видалення та оновлення даних. Безумовно, роз'єднана модель не дозволяє постійно відслідковувати зміни в базі даних, вироблені іншими користувачами. Це може привести до помилок в таких додатках, де інформація повинна оновлюватися кожен момент - замовлення квитків або продаж цінних паперів. Однак в будь-яку секунду може бути отримана свіжа інформація з бази даних через виклик методу FillDataSet. Таким чином, DataSet залишається надзвичайно зручним для найширшого класу додатків: коли необхідно отримати дані з бази і як-небудь обробити їх.

Глибока інтеграція з XML. Все більш широко поширюється XML грає найважливішу роль в ADO.NET і приносить ще кілька переваг у порівнянні з традиційним підходом.

Зауважимо для початку, що практично будь-який XML файл може бути використаний як джерело даних і на його основі може бути створений DataSet. точно також при передачі даних між компонентами або збереженні їх в файл використовується XML.

Програміст, що працює з ADO.NET не обов'язково повинен мати досвід роботи з XML або пізнання в цій мові. Всі операції залишаються прозорими для розробника.

Так як XML має текстове представлення, це дозволяє передавати його по протоколах типу HTTP через брандмауери. Справа в тому, що системи захисту зазвичай налаштовані на фільтрацію двійкової інформації, текстова ж легко пропускається, що полегшує створення розподілених додатків.

XML являє собою промисловий стандарт, який підтримують майже будь-якої сучасної платформою, що дозволяє передавати дані будь-якого компонента, який вміє працювати з XML, і виконувати під будь операційною системою.

1.4 Entity Framework

Entity Framework являє спеціальну об'єктно-орієнтовану технологію на базі фреймворка .NET для роботи з даними. Якщо традиційні засоби ADO.NET дозволяють створювати підключення, команди та інші об'єкти для взаємодії з базами даних, то Entity Framework являє собою більш високий рівень абстракції, який дозволяє абстрагуватися від самої бази даних і працювати з даними незалежно від типу сховища. Якщо на фізичному рівні ми оперуємо таблицями, індексами, первинними і зовнішніми ключами, але на концептуальному рівні, який нам пропонує Entity Framework, ми вже працює з об'єктами.

Перша версія Entity Framework – версія 1.0 представляла дуже обмежену функціональність, базову підтримку ORM і один єдиний підхід до взаємодії з бд - Database First. З виходом версії 4.0 багато чого змінилося - з цього часу Entity Framework став рекомендованою технологією для доступу до даних, а в сам фреймворк були введені нові можливості взаємодії з бд - підходи Model First і Code First.

Додаткові поліпшення функціоналу пішли з виходом версії 5.0. І нарешті, був випущений Entity Framework 6.0, що володіє можливістю асинхронного доступу до даних.

Центральної концепцією Entity Framework є поняття сутності або entity. Сутність представляє набір даних, асоційованих з певним об'єктом. Тому дана технологія передбачає роботу не з таблицями, а з об'єктами і їх наборами.

Будь-яка сутність, як і будь-який об'єкт з реального світу, має свій набір властивостей. Наприклад, якщо сутність описує людину, то ми можемо виділити такі властивості, як ім'я, прізвище, зріст, вік, вага ті інші. Властивості необов'язково представляють прості дані типу int, а й можуть представляти більш комплексні структури даних. І у кожної сутності може бути одна або кілька властивостей, які будуть відрізняти цю сутність від інших і будуть унікально визначати її. Подібні властивості називають ключами.

При цьому суті можуть бути пов'язані асоціативної зв'язком один-ко-многим, один-ко-одному і багато-до-багатьох, подібно до того, як в реальній базі даних відбувається зв'язок через зовнішні ключі.

Відмінною рисою Entity Framework є використання запитів LINQ для вибірки даних з БД. За допомогою LINQ ми можемо не тільки отримувати певні

рядки, що зберігають об'єкти, з бд, а й отримувати об'єкти, пов'язані різними асоціативними зв'язками.

Іншим ключовим поняттям є Entity Data Model. Ця модель зіставляє класи сутностей з реальними таблицями в БД. Entity Data Model складається з трьох рівнів: концептуального, рівень сховища і рівень зіставлення (маппінга).

На концептуальному рівні відбувається визначення класів сутностей, які використовуються в додатку.

Рівень сховища визначає таблиці, стовпці, відносини між таблицями і типи даних, з якими порівнюється використовувана база даних.

Рівень зіставлення (маппінга) служить посередником між попередніми двома, визначаючи зіставлення між властивостями класу суті і стовпцями таблиць.

Таким чином, ми можемо через класи визначені у додатку взаємодіяти з таблицями з бази даних.

Entity Framework передбачає три можливі способи взаємодії з базою даних:

Database first: Entity Framework створює набір класів, які відображають модель конкретної бази даних

Model first: спочатку розробник створює модель бази даних, по якій потім Entity Framework створює реальну базу даних на сервері.

Code first: розробник створює клас моделі даних, які будуть зберігатися в бд, а потім Entity Framework за цією моделлю генерує базу даних і її таблиці.

При роботі з Entity Framework, як і з будь-яким іншими ORM, часто виникають питання, пов'язані з його продуктивністю. Багато розробники через незнання нюансів роблять помилки, що призводять до поганих результатів. Потім, під час аналізу проблем і пошуку рішень, недостатньо розібравшись в питанні, приходять до висновку, що поліпшити ситуацію можна лише переходом на інший ORM або відмовою від нього взагалі. Хоч в деяких ситуаціях таке рішення може виявитися розумним, часто не все так погано - просто потрібно знати нюанси.

Включений трекінг змін, коли це не потрібно. Припустимо, в нашому проекті є такий фрагмент коду, призначений для вичитки деякого списку сутностей і передачі їх потім на клієнт:

```
using (var context = new EntityDataContext ())
{
   found = context.Entities.Where (e => e.Name.StartsWith (filter)). ToList ();
}
```

З першого погляду, ніяких проблем немає, але якщо в конструкторі не приховано ніяких спеціальних налаштувань, цей код призведе до зайвих витрат обчислювальних ресурсів. У кожного запиту є атрибут MergeOption, який вказує, як завантажувати прочитані запитом об'єкти в контекст. За замовчуванням цей атрибут дорівнює AppendOnly, який говорить про те, що сутність, яких ще немає в контексті, повинні бути в нього додані.

Якщо об'єкти, вичитані з допомогою Entity Framework контексту, не будуть змінені і не братимуть участі в модифікаціях інших об'єктів, у відповідному запиті потрібно викликати AsNoTracking () для колекції:

```
using (var context = new EntityDataContext ())
{
   found = context.Entities.AsNoTracking (). Where (e => e.Name.StartsWith (filter)). ToList ();
}
```

Ця функція встановлює атрибут MergeOption в значення NoTracking, тим самим виключаючи дії по додаванню прочитаних об'єктів в контекст.

Інша потенційна проблема, пов'язана з надмірним трекингом змін, в першу чергу стосується сценаріїв з додаванням і зміною даних. У конфігурації контексту є властивість AutoDetectChangesEnabled яка вказує чи треба автоматично викликати метод DetectChanges перед виконанням деяких операцій. До таких операцій відносяться додавання об'єкта в контекст, збереження змін в базу, пошук об'єктів через метод Find та ішні. Виклик DetectChanges потрібен для визначення що саме змінилось, видалилось, додалось, для поновлення зв'язків між об'єктами та інше.

Бездумне вимикання властивості AutoDetectChangesEnabled може привести до небажаних наслідків, наприклад втрата даних, тому просте правило - це якщо код не передбачає подальшого зміни доданих в контекст об'єктів в межах тієї ж сесії, то це властивість можна сміливо відключати. Така ситуація зустрічається досить часто - типовий CRUD API зазвичай отримує об'єкт ззовні і або просто його додає, або ще визначає, які були зроблені зміни з моменту вичитки, і відповідним чином оновлює інформацію про стан об'єкта в контексті, сам об'єкт при цьому не змінюється.

Починаючи з Entity Framework 5, запити автоматично кешуються після компіляції, що дозволяє значно прискорити їх подальші виконання - текст SQL запиту буде взятий з кеша, залишається тільки підставити настройки на власний вибір. Але є кілька ситуацій, в яких компіляція буде виконуватися при кожному виконанні.

Велика кількість Include в одному запиті. Очевидно, найпростіший спосіб прочитати дані з бази разом з дочірніми колекціями та іншими навігаційними властивостями - це використовувати метод Include(). Незалежно від кількості Include() в LINQ запиті, за підсумком буде сформований один SQL запит, який повертає всі зазначені дані. Може скластися враження, що в рамках Entity Framework такий підхід для вичитки складних об'єктів буде найбільш оптимальним в будь-якій ситуації. Але це не зовсім так.

Логічно було б припустити, що Include () просто додає ще один JOIN в запит. Але Entity Framework поводиться інакше. Якщо включається навігаційне властивість - одиничний об'єкт, а не колекція, то буде просто ще один JOIN. Якщо колекція - то під кожну буде сформований окремий підзапит, де батьківська таблиця з'єднується з дочірньою, а всі такі підзапити будуть об'єднані в загальний UNION ALL. Очевидно, що якщо потрібна тільки одна дочірня колекція, то UNION ALL не буде.

Зроблено це для боротьби з проблемою перемноження результатів. Припустимо, у об'єкта є три дочірніх колекції по 10 елементів в кожній. Якщо всі три додати через OUTER JOIN безпосередньо в «головний» запит, то в результаті буде 10 * 10 * 10 = 1000 записів. Якщо ж піти шляхом Entity Framework, і ці три колекції збирати в один запит через UNION, то отримаємо 30 записів. Чим більше колекцій і елементів в них, тим виграш підходу з UNION очевидніше.

Але проблема в тому, що при великій складності самих сутностей і критеріїв вибірки, побудова та оптимізація такого запиту дуже трудомісткі для Entity Framework, як і виконання його на рівні сервера бази даних. Тому якщо результати профілювання показують незадовільну продуктивність запитів, що містять Include, а з індексами в базі все в порядку - є сенс задуматися про альтернативні рішення.

Основна ідея альтернативних рішень - це вичитування кожної колекції окремим запитом. Найбільш простий варіант можливий, якщо об'єкти при вибірці додаються в контекст, тобто без використання AsNoTracking().

Виходить, що для кожної дочірньої колекції ми вичитуємо всі об'єкти, які мають відношення до батьківських сутностей, які підпадають під критерій запиту. Після виклику Load() об'єкти додаються в контекст. Під час вичитки батьківських сутностей Entity Framework знайде всі дочірні, що знаходяться в контексті, і відповідним чином додасть на них посилання.

Основний недолік тут - то, що на кожен запит йде окреме звернення до сервера бази даних. На щастя, є спосіб вирішити і цю проблему. У бібліотеці EntityFramework.Extended є можливість створювати «майбутні» запити. Основна ідея в тому, що всі запити, у яких був викликаний extension method Future (),

будуть послані в одному зверненні до сервера, коли у будь-якого з них буде викликаний термінальний метод.

1.5 Dapper

Dapper є технологією зіставлення (маппінга) результатів sql-запитів з класами с #. В цьому плані Dapper трохи схожий на Entity Framework. У той же час за рахунок своєї легковажності Dapper забезпечує більшу продуктивність і швидше дозволяє виконувати запити, ніж Entity Framework. В принципі, це не повний ORM, але якщо потрібно просто запускати запити без необхідності боротися з ORM, це дуже хороше рішення.

Для здійснення запитів Dapper надає для об'єктів IDbConnection метод розширення Query <T>, який в якості параметра приймає sql-вираз і може повертати об'єкт типу T, з яким зіставляються результати запиту.

Dapper не створюэ моделі класу, не генерує сам запити та не відслюдковує об’єкті та їх зміни, але через його легковесність він дуже швидко працює з даними, также як ADO.NET, тому що спосыб генерацыъ запитів ідентичний.

1.6 Висновки з розділу 1

У цьому розділі було розглянуто 3 ORM технології: Entity Framework, Dapper та ADO.NET. Були описані їх основні задачі, визначення, переваги та недоліки кожної з них.

2 ПЛАНУВАННЯ ЕКСПЕРИМЕНТУ

Експериментами у даному дипломному проекту будуть визначення швидкодії трьох ORM технологій. Швидкість буде оцінена програмою Postman.

Об'єктом дослідження є процес конструювання запитів до СУБД з використанням технології ORM.

Предметом дослідження є технології побудови запитів до СУБД.

Метою даної дипломної роботи є підвищення ефективності обробки даних та підвищення якості ПЗ, шляхом розробки практичних рекомендацій та засобів щодо вибору технологій ORM.

Методи досліджень базуються на використанні методів планування експерименту, статистичних методів, методів системного аналізу.

Критерієм оцінювання буде швидкість обробки запиту.

Для збору тестових даних по швидкості обробки буде використаний метод «чорного ящику». Цей метод тестування дозволить нам абстрагуватися від конкретних реалізацій алгоритму. На нього будуть впливати фактори які будуть змінювати наш результат. Вхідні величини прийнято називати факторами, а вихідні – відгуками. У даному випадку ми проводимо активний експеримент, який дозволить нам одержати вихідні дані шляхом проведення сукупності дій над об'єктом.

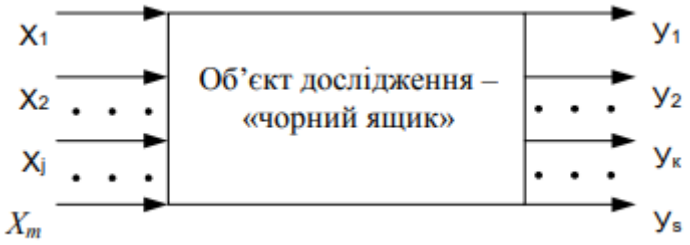

Рисунок 2.1 – схема чорного ящику

Де:

x – вхідні параметри (дані)

y - вихідні параметри (результати), мілісекунди

План експерименту буде зображено у таблиці що буде містить у собі умови проведення дослідів. План m-факторного експерименту, що складеться з n-дослідів зображено нижче (таблиця 2.1).

Таблиця 2.1

Номер досліду	Умови дослідів			Результати дослідів		
i	x_{ji}	…	x_{mi}	y_{ji}	…	y_{si}

Де:
i - номер досліду;
j - номер параметру;
m - максимальне число параметрів досліду;
s - максимальне число результатів досліду.

2.1 Конфігурація системи для тестування

Дані отримані у ході цього експерименту можуть відрізнятися на різних конфігураціях, але це буде впливати на усі технології однаково.

Комп'ютер на якому проводиться тестування має наступну конфігурацію:

- Процесор Intel Core i5 3570 @ 3.4 GHz;
- Відеокарта Radeon RX580;
- 12 гігабайтів оперативної пам'яті;
- SSD Kingston 500GB;
- ОС Windows 10;
- Visual Studio Community Edition 2019;
- EntityFramework 6 версії;
- Dapper NET.Core extension 3.2.0;
- ADO.NET для версії .NET 4.5.

2.2 Postman

Postman - зручний HTTP-клієнт для тестування веб-сайтів, входить в число кращих розширень каталогу Chrome Web Store в категорії «інструменти для роботи» (productivity tools). Також існує desktop-версія прогарми.

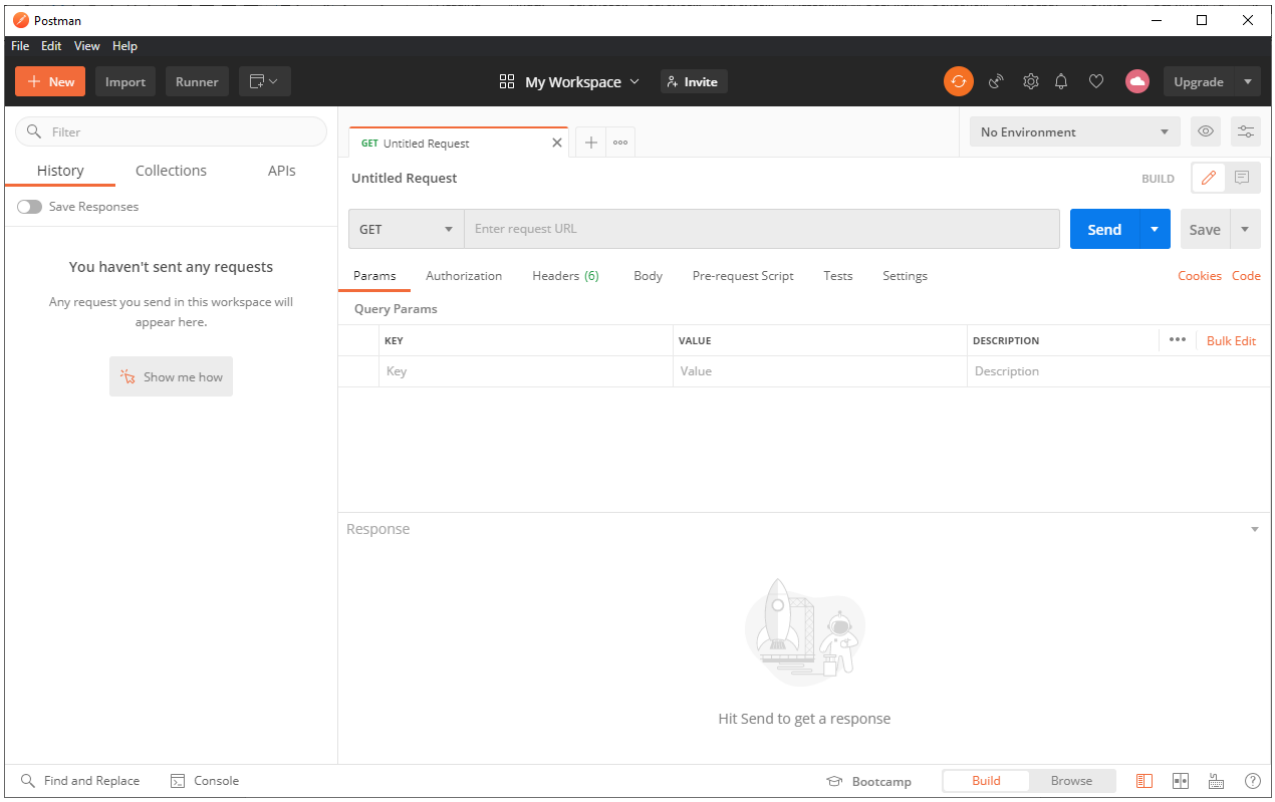

Рисунок 2.1 – вікно програми Postman

За допомогою розширення можна складати і редагувати прості або складні HTTP-запити. Складені запити автоматично зберігаються на майбутнє для повторного застосування. Відповіді від сервера можна теж зберігаються як файли на жорсткому диску. Таким чином, Postman економить купу часу веб-розробникам та тестерам.

У програмі є вбудований редактор запитів, з можливостями кодування запитів, завантаження з файлу і відправки бінарних даних.

С GET запитом все просто: задаємо url, додаємо параметри (можна «вмикати» і «вимикати» параметри), натискаємо send і в нижній половині екрану бачимо відповідь (на сервері у данному прикладі просто повертається html сторінка і виводяться всі вхідні параметри в json форматі) . Перемикаючись між вкладками на області відповіді ми можемо знайти будь-яку інформацію і навіть перемикатися між уявленнями і форматами виведення.

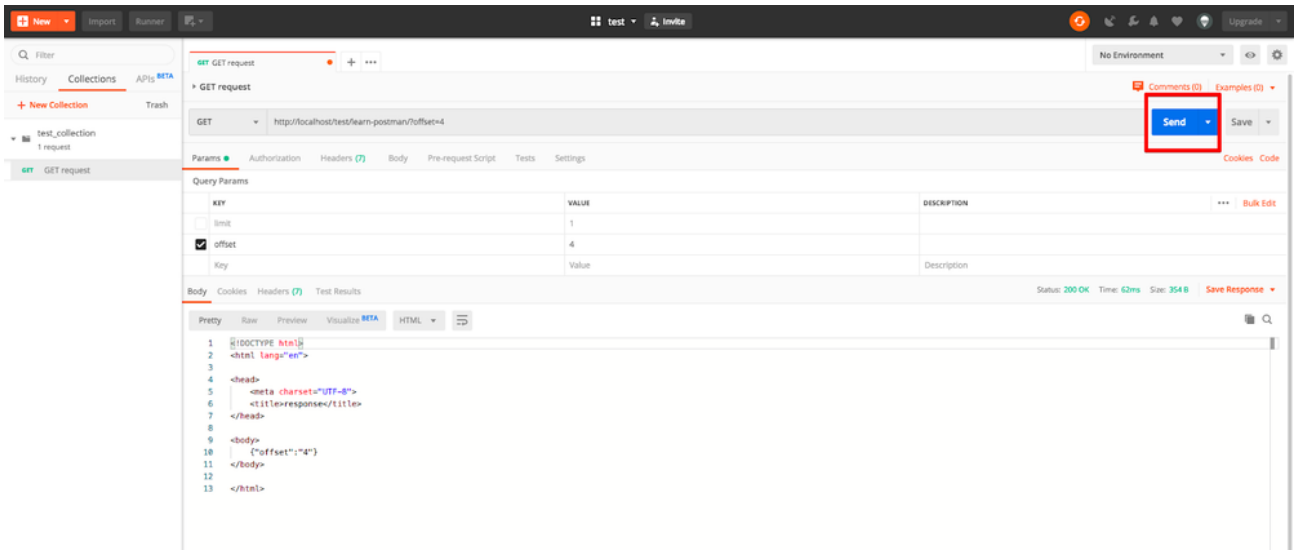

Рисунок 2.2 – простий запит у Postman

Праворуч від кнопки Send є кнопка Save. При спробі збереження Postman попросить вас вказати / створити колекцію. Колекція це групування запитів в робочому просторі (наприклад по проектам). Також Postman відображує час виконання запиту, на який ми будемо орієнтуватися при аналізі швидкості виконання запитів у різних технологіях.

2.3 Експеримент перший. Швидкість операції додавання запису у базу даних

Для кожної ORM будуть проведені тести з наступними параметрами.

Вхідні дані: сутність для запису у базу.

Вихідні дані: час, за який сервер обробить інформацію та верне її користувачу.

Буде проведено 10 тестів для виявлення середнього, максимального та мінімального часу відгуку від сервера.

2.4 Експеримент другий. Модифікація запису в базі даних

Для кожної ORM будуть проведені тести з наступними параметрами.

Вхідні дані: Модифікована сутність.

Вихідні дані: час, за який сервер обробить інформацію та верне її користувачу.

Буде проведено 10 тестів для виявлення середнього, максимального та мінімального часу відгуку від сервера.

2.5 Експеримент третій. Вилучення 1 запису с бази даних

Для кожної ORM будуть проведені тести з наступними параметрами.

Вхідні дані: ID сутності для вилучення.

Вихідні дані: час, за який сервер обробить інформацію та верне її користувачу.

Буде проведено 10 тестів для виявлення середнього, максимального та мінімального часу відгуку від сервера.

2.6 Експеримент четвертий. Швидкість операції видалення з бази даних

Для кожної ORM будуть проведені тести з наступними параметрами.

Вхідні дані: ID сутності для видалення.

Вихідні дані: час, за який сервер обробить інформацію та верне її користувачу.

Буде проведено 10 тестів для виявлення середнього, максимального та мінімального часу відгуку від сервера.

2.7 Експеримент п’ятий. Вилучення багатьох записів з фільтрацією з декількох таблиць.

Для кожної ORM будуть проведені тести з наступними параметрами.

Вхідні дані: кількість таблиць, кількість записів для вилучення.

Вихідні дані: час, за який сервер обробить інформацію та верне її користувачу.

Буде проведено 10 тестів для виявлення середнього, максимального та мінімального часу відгуку від сервера.

2.8 Висновки з розділу 2

У цьому розділі було описано процес проведення експерименту, його основні положення, мета, задачі, критерії оцінки та програми для тестування.

3 ПРОВЕДЕННЯ ЕКСПЕРИМЕНТУ

3.1 Структура БД та середовища розробки ПЗ

Прототип програмного забезпечення (ППЗ) має бути представлений у вигляді двох взаємодіючих компонентів: серверу, що реалізовуватиме роботу з СУБД та кліенту, шо буде інтерфейсом користувача.

Основні функції ПЗ мають бути наступними:

– CRUD-операції с даними;
– перетворення сутностей в дані таблиці для сберігання;
– перетворення даних таблиці у сутності для роботи с зими;
– перетворення сутностей у моделі для відправки на кліент;
– перетворення моделей у сутності для роботи з ними;
– відображення результату операцій;
– повідомлення про помилки в ході виконання.

У solution входить три основні проекти: Asp.NetCoreWebApi і два .NetCoreClassLibrary (слоібізнес логіки і роботи з даними). База даних для проекту формується з допомогою EntityFramework на основі сутностей (DAL.Entities). Основна логіка додатка зосереджена в класах сервісів шару BLL. Взаємодія з таблицями БД відбувається в репозиторіях шару DAL, за допомогою яких в проекті реалізується репозиторій-патерн. Сервіси мають доступ до репозиторіїв, а контролери-до сервісів. Клієнтська частина проекту представлена Angular-додатком, яке разом з WebApi утворюють шар уявлення. Оскільки шар уявлення не має доступу до шару роботи з даними, в шарі бізнес логіки зберігаються моделі (BLL.Models), які необхідні для візуального представлення інформації з БД. Дані, отримані з репозиторіїв, перетворюються з сутностей в моделі, після чого використовуються бізнес логікою і поданням для відображення в браузері.

Структура бази даних являє собою набір таблиць, кожна з яких зберігає дані про конкретні сутності (Entity). Кожна таблиця обов'язково має поле ідентифікатора (Id). Орієнтовна схема БД з організацією зовнішніх зв'язків представлена на рис.1.

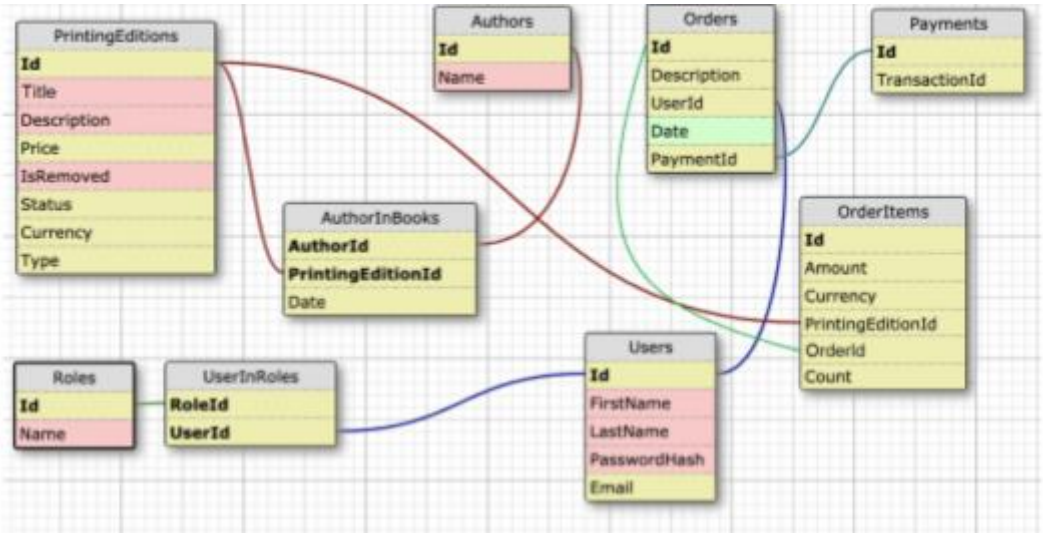

Рисунок 3.1 – структура бази даних

Зберігання даних в БД здійснюється згідно з такими принципами:

Users, UserInRoles, Roles, додані за допомогою IdentityFramework; необхідно використовувати функціонал Identity для додавання і використання даних таблиць;

1.PrintingEditions містить в собі все ПІ, до яких відповідно до типу відносяться книги, журнали і газети; поля Currency і Status є перерахуваннями;

2.Базова суть проекту повинна містити поляId, CreationData і IsRemoved (для позначки про те, що об'єкт був видалений)

3.PrintingEditions містить в собі все ПІ, до яких відповідно до типу відносяться книги, журнали і газети; поля Currency і Status є перерахуваннями;

4.Таблиця AuthorInPrintingEditions (як і AspNetUserRoles) є проміжною і застосовується для зв'язку Author і PrintingEdition по типу «багато до багатьох»;

5.Таблиці Orders і OrderItems містять в собі необхідні даних для оформлення замовлення; дані з кошика використовуються для формування замовлення, проведення оплати (Payments) і зв'язування їх з допомогою полів OrderId, UserId і PaymentId.

В ході реалізації dapper репозиторіїв необхідно виконати наступні завдання:

1. Створення базового <Generic> сховища з основним функціоналом по роботі з БД (Dapper + CRUD);

2. Додавання user сховища із застосуванням Dapper (відповідно до функціоналом проекту), використання Identity функціоналу;

Створення printingEdition сховища (Dapper + CRUD для printingEdition);

3. Створення author сховища (Dapper + CRUD для author);

4. Створення authorInPrintingEditions сховища (Dapper + CRUD для authorInPrintingEditions);

5. Створення payment сховища (Dapper + CRUD длярayment);

6. Створення order сховища (Dapper + CRUD для order);

7. Створення orderItem сховища (Dapper + CRUD для orderItem);

8. Перевірка правильності виконання завдання, порівняння швидкості виконання запитів з використанням EntityFramework і Dapper;

3.2 Формування структури проекту

Проект буде сформований 3 частинами, де кожна частиа буде відповідати за свої функції.

DataAccessLayer (далі – DAL) – частина, яка працює безпосередньо з СУБД, наприклад MySql, відповідає за структуру та формування таблиць.

BusinessLogicLayer (далі – BLL) – частина, яка займається обробкою та конвертацією інформації у різні форми

PresentationLayer – частина, яка відправляє або получає дані з зовнішньюго джерела, наприклад тестова програма Postman або сайт.

3.2.1 Data Access Layer

Для створення БД при використанні подходу Model-first потрібно описати структуру БД у програмі наступним образом:

```
public class ApplicationDbContext : IdentityDbContext<ApplicationUser, ApplicationRole,
long>
    {

        public DbSet<Payment> Payments { get; set; }
        public DbSet<PrintingEdition> PrintingEditions { get; set; }
        public DbSet<Order> Orders { get; set; }
        public DbSet<OrderItem> OrderItems { get; set; }
        public DbSet<AuthorInPrintingEdition> AuthorInPrintingEditions { get; set; }
        public DbSet<Author> Authors { get; set; }

        public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
            : base(options)
        {

        }

        protected override void OnModelCreating(ModelBuilder builder)
        {
            builder.Entity<AuthorInPrintingEdition>()
```

```
            .HasKey(c => new { c.AuthorId, c.PrintingEditionId });

        builder.Entity<AuthorInPrintingEdition>()
             .HasOne(sc => sc.Author)
             .WithMany(s => s.AuthorInPrintingEditions)
             .HasForeignKey(sc => sc.AuthorId);

        builder.Entity<AuthorInPrintingEdition>()
             .HasOne(sc => sc.PrintingEdition)
             .WithMany(c => c.AuthorInPrintingEditions)
             .HasForeignKey(sc => sc.PrintingEditionId);

        builder.Entity<OrderItem>()
            .HasKey(c => new { c.OrderId, c.PrintingEditionId });

        builder.Entity<OrderItem>()
             .HasOne(sc => sc.Order)
             .WithMany(c => c.OrderItems)
             .HasForeignKey(sc => sc.OrderId);

        builder.Entity<OrderItem>()
             .HasOne(sc => sc.PrintingEdition)
             .WithMany(c => c.OrderItems)
             .HasForeignKey(sc => sc.PrintingEditionId);

        builder.Entity<Order>()
             .HasOne(u => u.User)
             .WithMany(o => o.Orders)
             .HasForeignKey(f => f.UserId);

        base.OnModelCreating(builder);
    }
```

Це потрібно для автоматичного створення зв’язку між програмою та СУБД, також для автоматичного оновлення полів таблиць.Далі робляться базові та успадковані від неї сутності (Рис 3.1). Вони потрібні для автоматичного створення та оновлення полей у таблицях БД.

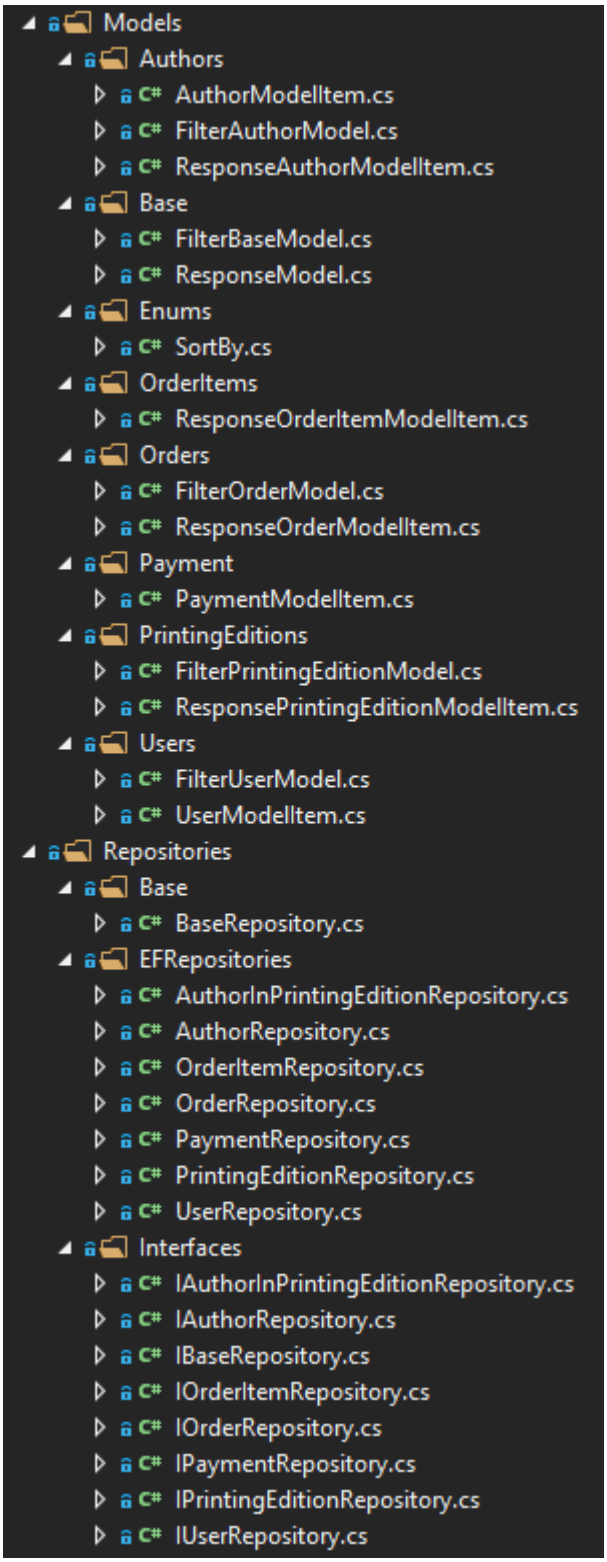

Рисунок 3.2 – базові сутності програми та репозиторії для роботи з базою

А також репозиторії для роботи з таблицями (Рис. 3.2) , які містять у собі усі базові функції БД, такі як створення, видалення, запис та оновлення даних.

3.2.2 Business Logic Layer

Частина бізнес-логіки займається обробкою, та конвертацією даних з виду у якому вони зберігаються у БД, в вид у якому вони будуть показані користувачу. Ця частина не має доступу напряму до БД, вона обробляє дані що повертаюсь репозиторії з частини DAL.

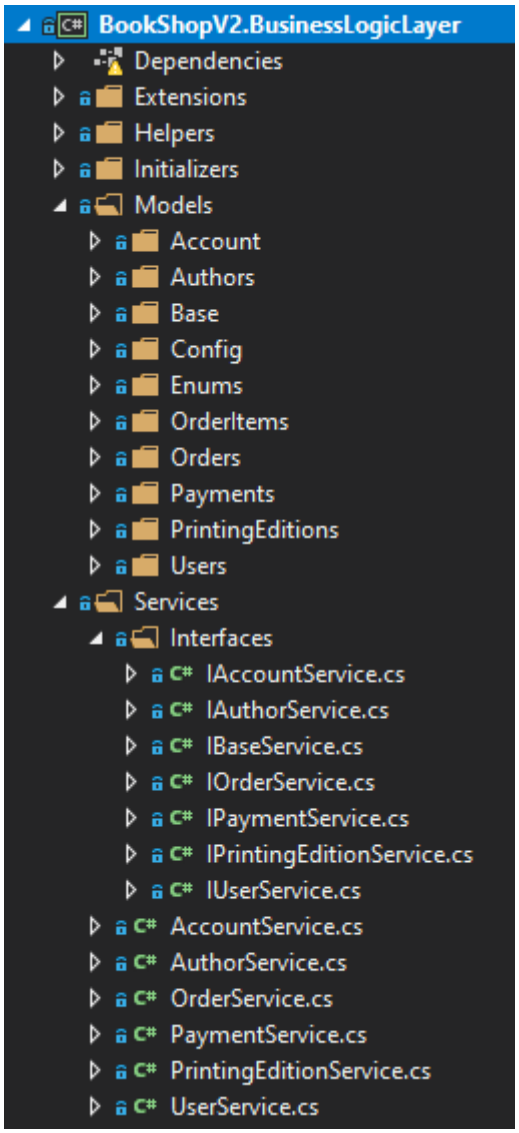

Рисунок 3.3 – сервіси конвертації та обробки даних

3.2.3 Presentation Layer

Ця частина відповідає за зв'язок основної програми з зовнішніми програмами, надає адреса для звернення до частин програми, містить у собі конфігурацію проекту.

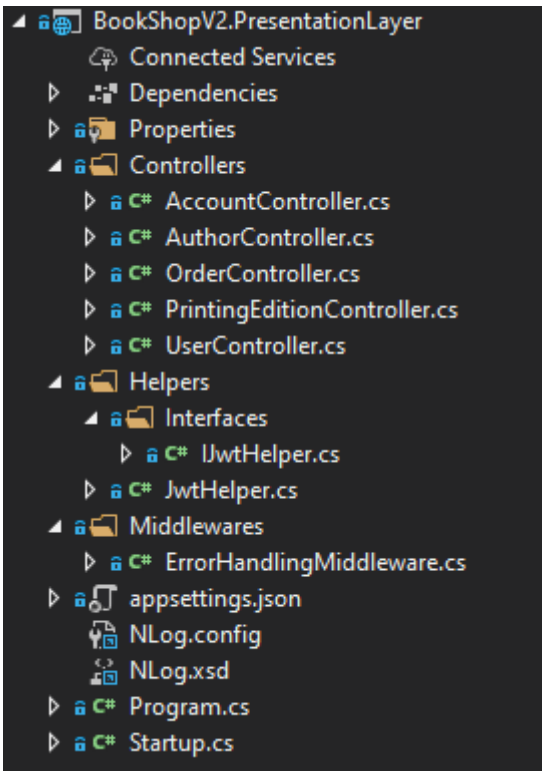

Рисунок 3.4 – контролери та конфігурація проекту

3.3 Тестування Entity Framework

3.3.1 Додавання запису у базу

Для тестування будемо добавляти у базу сутність PrintingEdition (рисунок 3.1). Конкретні значення полей запису не впливають на швидкість обробки інформації. Головне дотримуватися наступних типів даних у полей:

1) ID - string
2) Title - string
3) Description - string
4) Price - decimal
5) isRemoved - boolean
6) Status - enum
7) Currency - enum

8) Type – enum

Заповнимо нашу базу 10 випадковими записами, та зафіксуймо час за котрий сервер поверне нам результат. Результати представлені у таблиці 3.1.

Таблиця 3.1

Номер досліду	Вхідні параметри	Вихідні параметри
	Сутності	Швидкість обробки, мілісекунди (ms)
1	Сутність 1	118
2	Сутність 2	61
3	Сутність 3	60
4	Сутність 4	63
5	Сутність 5	66
6	Сутність 6	78
7	Сутність 7	70
8	Сутність 8	71
9	Сутність 9	70
10	Сутність 10	74
Середнє значення		73,1

Рисунок 3.5 – Швидкість додавання запису у технології Entity Framework

3.3.2 Модифікація запису у базі даних

Для тестування будемо модифікувати сутність PrintingEdition (рисунок 3.1). Конкретні значення полей запису не впливають на швидкість обробки інформації. Головне дотримуватися наступних типів даних у полей:

1) ID - string
2) Title - string
3) Description - string
4) Price - decimal
5) isRemoved - boolean
6) Status - enum
7) Currency - enum
8) Type – enum

Модифікуємо 10 записів, доданих у першому тесті. Результати представлені у таблиці 3.2.

Таблиця 3.2

Номер досліду	Вхідні параметри	Вихідні параметри
	Сутності	Швидкість обробки, мілісекунди (ms)
1	Сутність 1	109
2	Сутність 2	80
3	Сутність 3	66
4	Сутність 4	61
5	Сутність 5	68
6	Сутність 6	78
7	Сутність 7	75
8	Сутність 8	60
9	Сутність 9	69
10	Сутність 10	79
Середнє значення		74,5

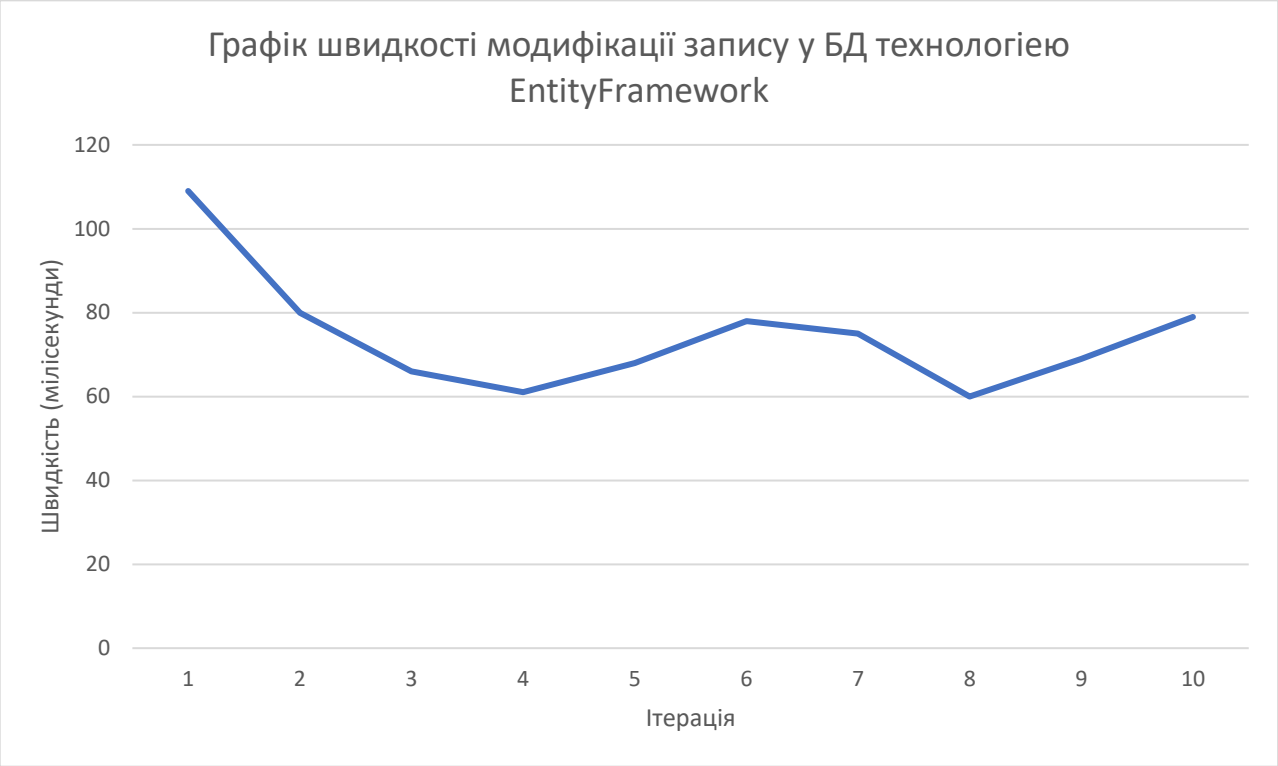

Рисунок 3.6 – графік швидкості модифікації запису у БД технологією EntityFramework

3.3.3 Вилучення 1 запису з бази даних

Для тестування будемо вилучати записи з попередніх тестів, вхідним параметром буде ID об'єкту. Результати представлені у таблиці 3.3.

Таблиця 3.3

Номер досліду	Вхідні параметри	Вихідні параметри
	ID об’єкту	Швидкість обробки, мілісекунди (ms)
1	1	131
2	2	79
3	3	78
4	4	77
5	5	64
6	6	69
7	7	64
8	8	67
9	9	63
10	10	60
Середнє значення		73,1

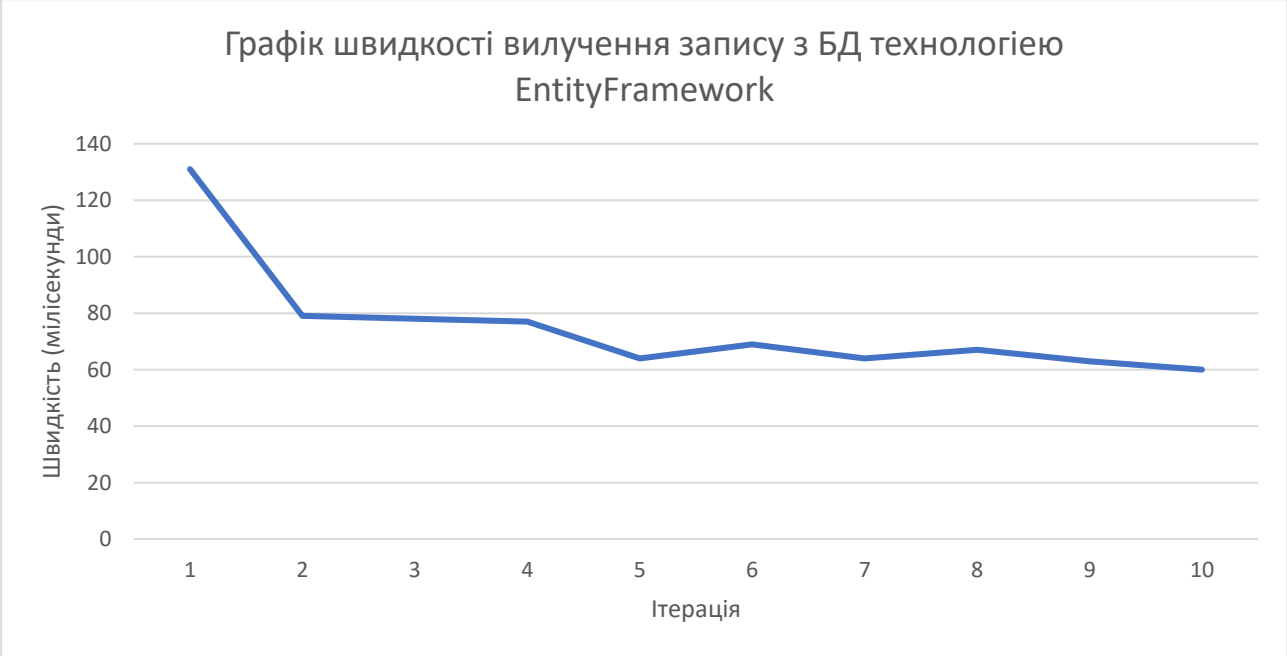

Рисунок 3.7 – графік швидкості вилучення даних з БД технологією EntityFramework

3.3.4 Видалення запису з бази даних

Для тестування будемо вилучати записи з попередніх тестів, вхідним параметром буде ID об'єкту. Результати представлені у таблиці 3.4.

Таблиця 3.4

Номер досліду	Вхідні параметри	Вихідні параметри
	ID об’єкту	Швидкість обробки, мілісекунди (ms)
1	1	116
2	2	63
3	3	79
4	4	77
5	5	64
6	6	78
7	7	64
8	8	80
9	9	60
10	10	60
Середнє значення		74,1

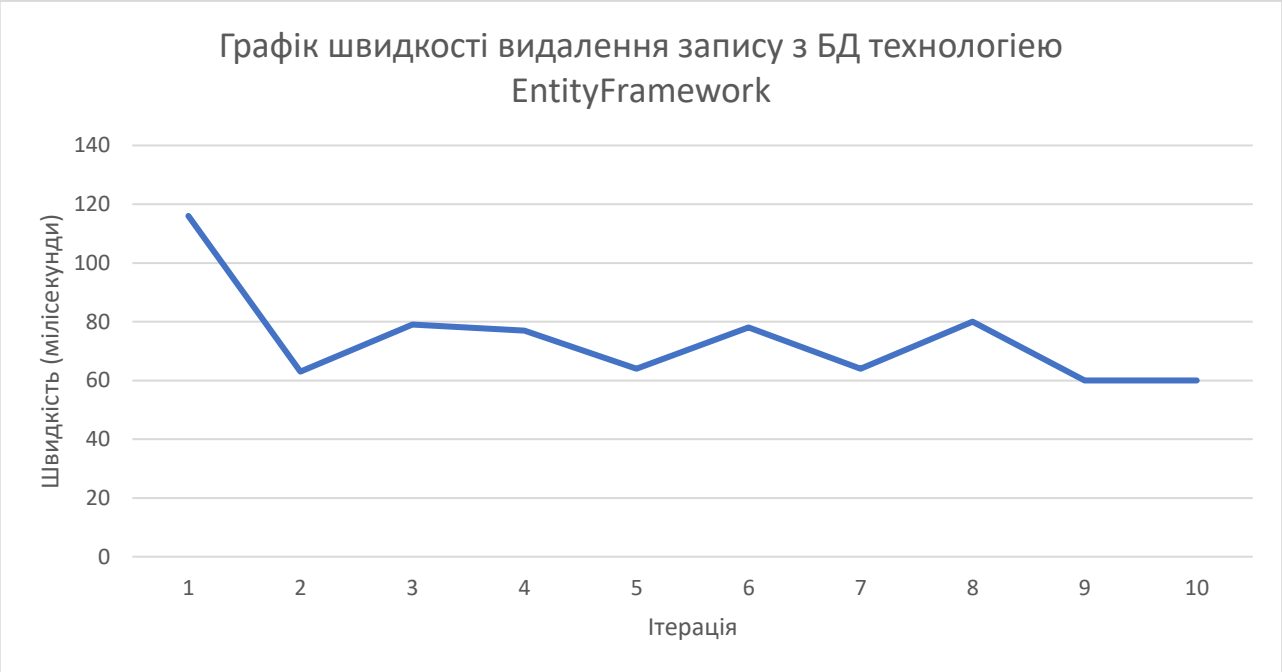

Рисунок 3.8 – графік швидкості видалення запису з БД технологією EntityFramework

3.3.5 Вибірка з декількох таблиць

Для тестування будемо перевіряти швидкість обробки для декількох таблиць з різною кількістю записів для вибірки. Для 100 500 та 1000 записів перевіриться швидкість на витяг з 1-5 таблиць. Результати у таблиці 3.5.

Таблиця 3.5

Номер досліду	Вхідні параметри		Вихідні параметри
	Кількість таблиць для вибірки	Число записів для вибірки	Швидкість обробки, мілісекунди (ms)
1	1	100	153
2	2	100	187
3	3	100	203
4	4	100	281
5	5	100	353
6	1	500	213
7	2	500	243
8	3	500	323
9	4	500	425
10	5	500	553
11	1	1000	302
12	2	1000	352
13	3	1000	503
14	4	1000	681
15	5	1000	971

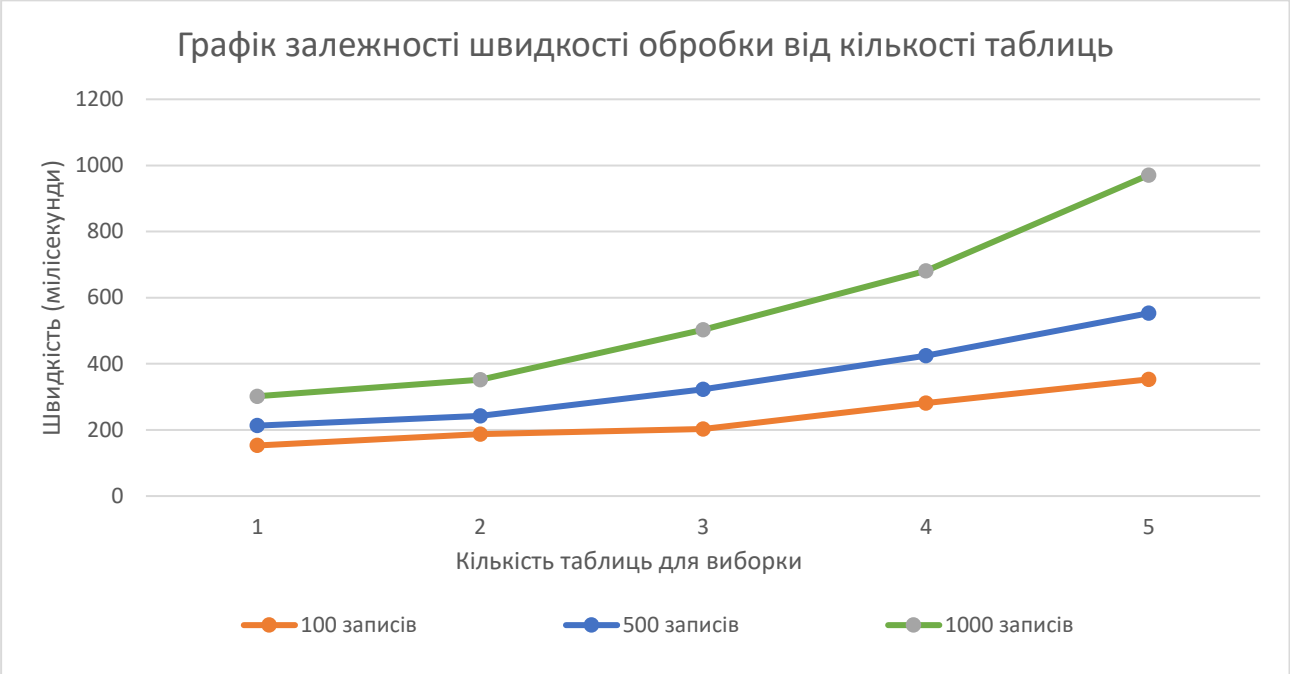

Рисунок 3.9 – графік залежності швидкості обробки від кількості таблиць технологією EntityFramework

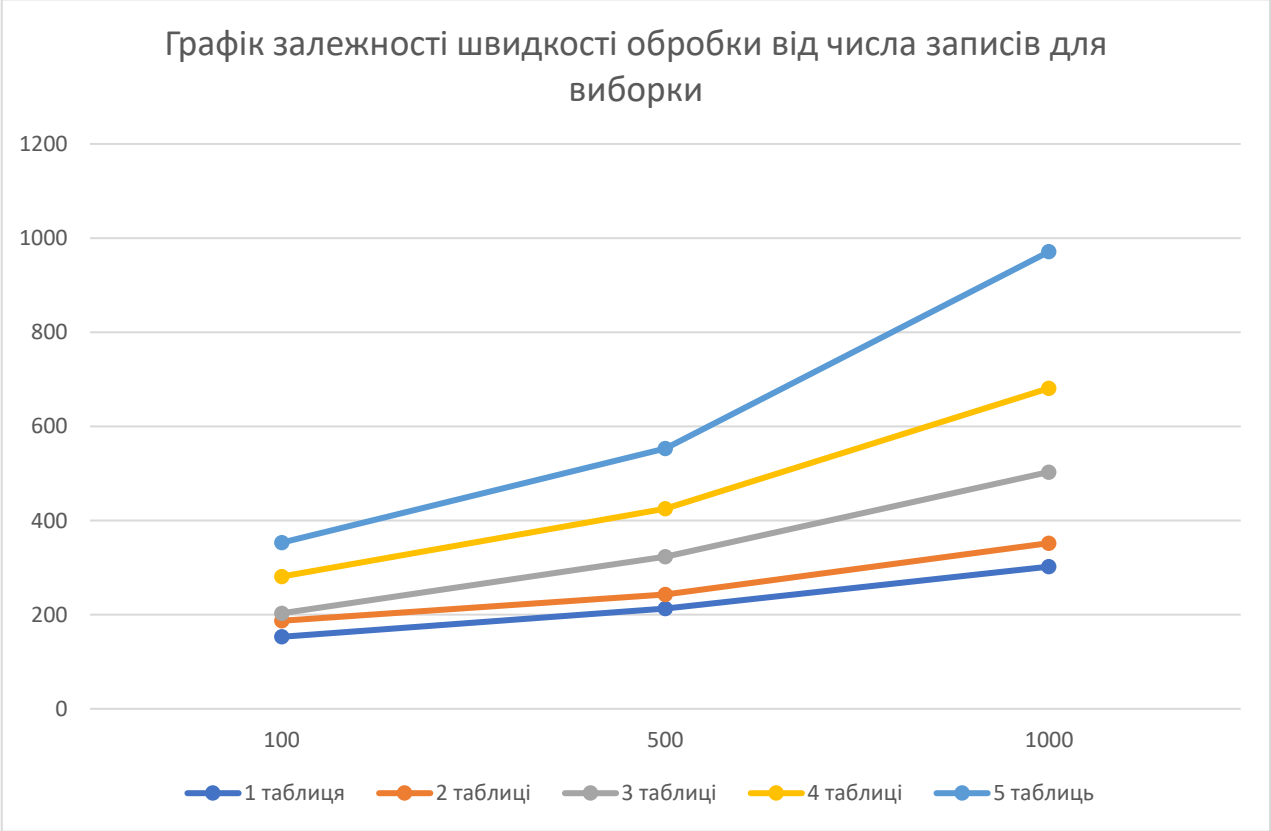

Рисунок 3.10 – графік залежності швидкості обробки від числа записів для вибірки технологією EntityFramework

3.4 Тестування Dapper

3.4.1 Додавання запису у базу

Для тестування будемо добавляти у базу сутність PrintingEdition (рисунок 3.1). Конкретні значення полей запису не впливають на швидкість обробки інформації. Головне дотримуватися наступних типів даних у полей:

1) ID - string
2) Title - string
3) Description - string
4) Price - decimal
5) isRemoved - boolean
6) Status - enum
7) Currency - enum
8) Type – enum

Заповнимо нашу базу 10 випадковими записами, та зафіксуймо час за котрий сервер поверне нам результат. Результати представлені у таблиці 3.6.

Таблиця 3.6

Номер досліду	Вхідні параметри	Вихідні параметри
	Сутності	Швидкість обробки, мілісекунди (ms)
1	Сутність 1	40
2	Сутність 2	35
3	Сутність 3	49
4	Сутність 4	45
5	Сутність 5	46
6	Сутність 6	50
7	Сутність 7	49
8	Сутність 8	40
9	Сутність 9	41
10	Сутність 10	38
Середнє значення		43,3

Рисунок 3.11 – Швидкість додавання запису у технології Dapper

3.4.2 Модифікація запису у базі даних

Для тестування будемо модифікувати сутність PrintingEdition (рисунок 3.1). Конкретні значення полей запису не впливають на швидкість обробки інформації. Головне дотримуватися наступних типів даних у полей:

9) ID - string
1) Title - string
2) Description - string
3) Price - decimal
4) isRemoved - boolean
5) Status - enum
6) Currency - enum
7) Type – enum

Модифікуємо 10 записів, доданих у першому тесті. Результати представлені у таблиці 3.7.

Таблиця 3.7

Номер досліду	Вхідні параметри	Вихідні параметри
	Сутності	Швидкість обробки, мілісекунди (ms)
1	Сутність 1	47
2	Сутність 2	46
3	Сутність 3	39
4	Сутність 4	41
5	Сутність 5	44
6	Сутність 6	38
7	Сутність 7	39
8	Сутність 8	41
9	Сутність 9	50
10	Сутність 10	49
Середнє значення		43,4

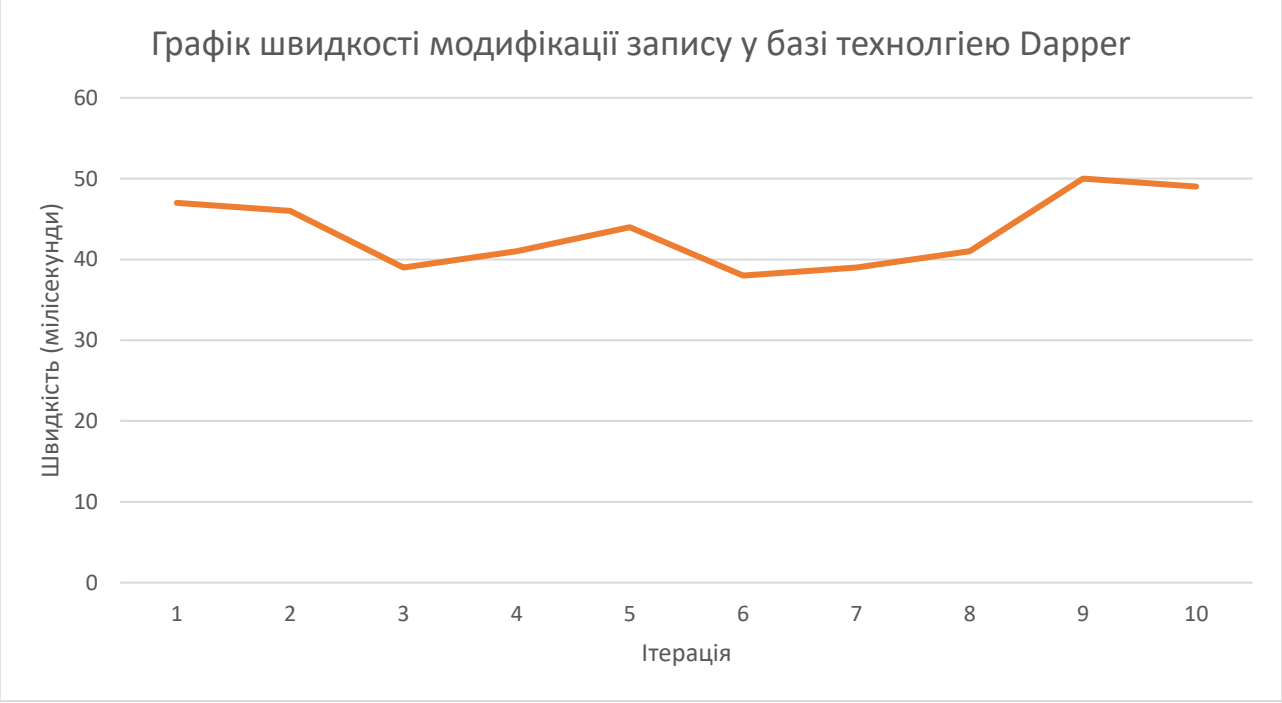

Рисунок 3.12 – графік швидкості модифікації запису у БД технологією Dapper

3.4.3 Вилучення 1 запису з бази даних

Для тестування будемо вилучати записи з попередніх тестів, вхідним параметром буде ID об'єкту. Результати представлені у таблиці 3.8.

Таблиця 3.8

Номер досліду	Вхідні параметри	Вихідні параметри
	ID об’єкту	Швидкість обробки, мілісекунди (ms)
1	1	42
2	2	30
3	3	38
4	4	34
5	5	30
6	6	49
7	7	44
8	8	48
9	9	38
10	10	34
Середнє значення		38,7

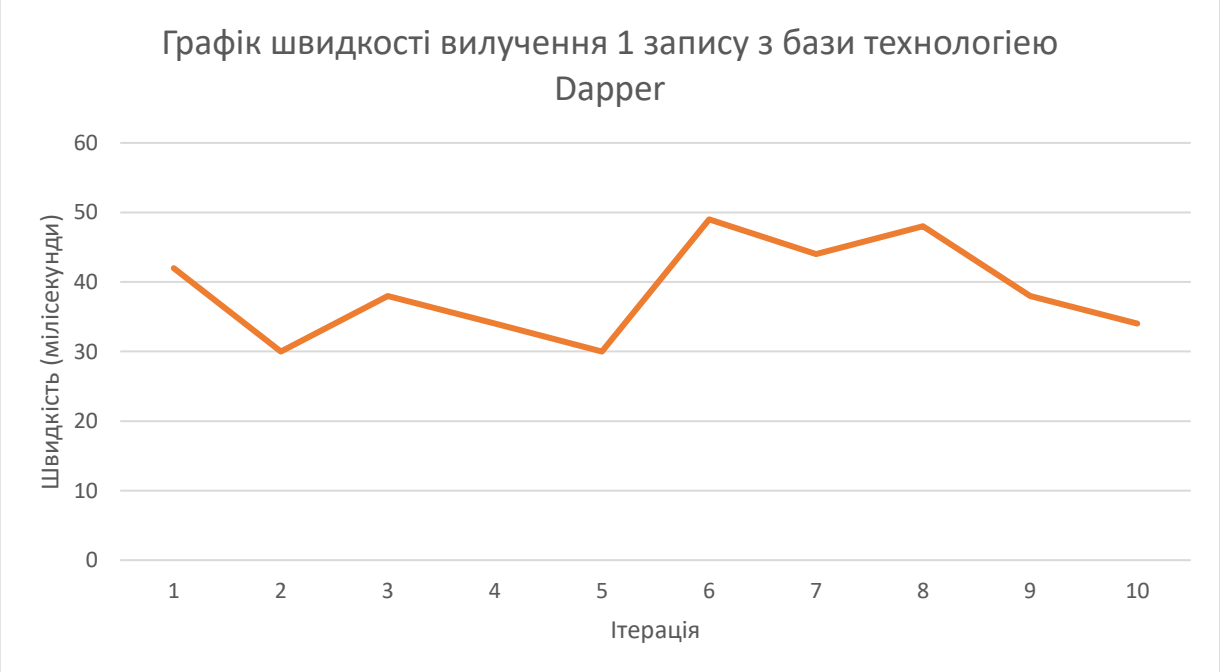

Рисунок 3.12 – графік швидкості вилучення даних з БД технологією Dapper

3.4.4 Видалення запису з бази даних

Для тестування будемо вилучати записи з попередніх тестів, вхідним параметром буде ID об'єкту. Результати представлені у таблиці 3.9.

Таблиця 3.9

Номер досліду	Вхідні параметри	Вихідні параметри
	ID об’єкту	Швидкість обробки, мілісекунди (ms)
1	1	50
2	2	32
3	3	44
4	4	30
5	5	39
6	6	31
7	7	34
8	8	40
9	9	41
10	10	32
Середнє значення		37,3

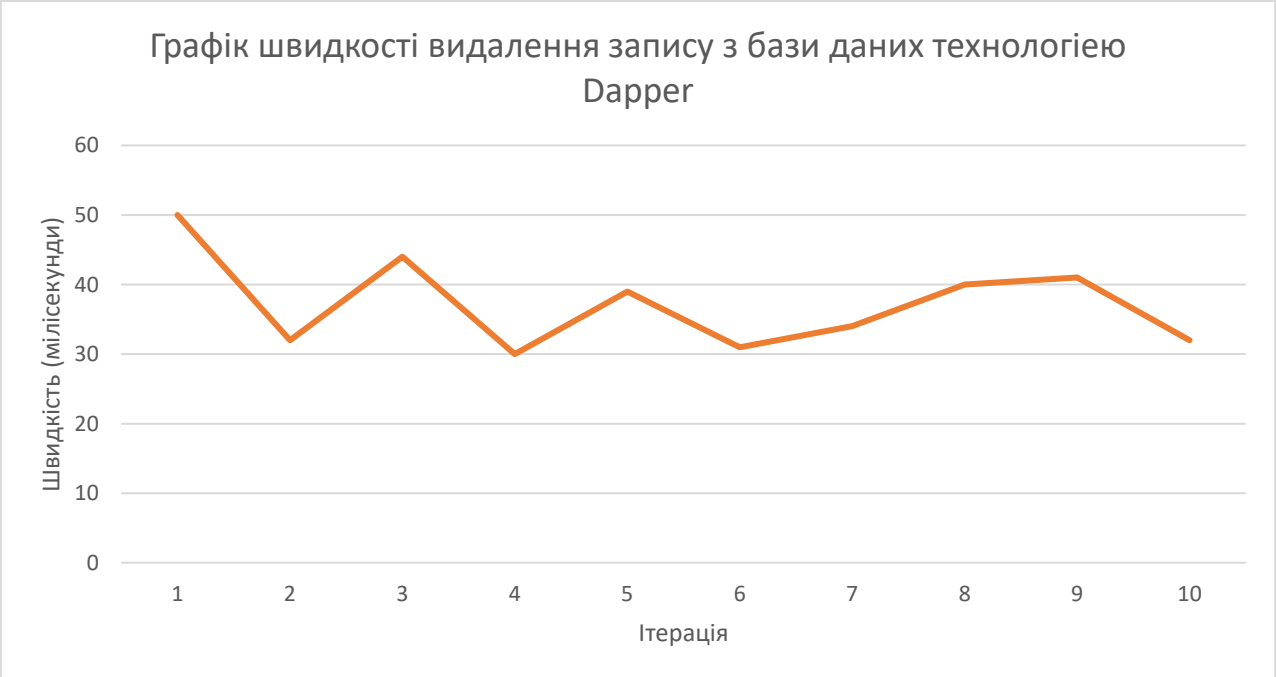

Рисунок 3.13 – графік швидкості видалення запису з БД технологією Dapper

3.4.1 Вибірка з декількох таблиць

Для тестування будемо перевіряти швидкість обробки для декількох таблиць з різною кількістю записів для вибірки. Для 100 500 та 1000 записів перевіриться швидкість на витяг з 1-5 таблиць. Результати у таблиці 3.10.

Таблиця 3.10

Номер досліду	Вхідні параметри		Вихідні параметри
	Кількість таблиць для вибірки	Число записів для вибірки	Швидкість обробки, мілісекунди (ms)
1	1	100	102
2	2	100	123
3	3	100	139
4	4	100	157
5	5	100	172
6	1	500	134
7	2	500	158
8	3	500	172
9	4	500	193
10	5	500	223
11	1	1000	162
12	2	1000	184
13	3	1000	224
14	4	1000	255
15	5	1000	278

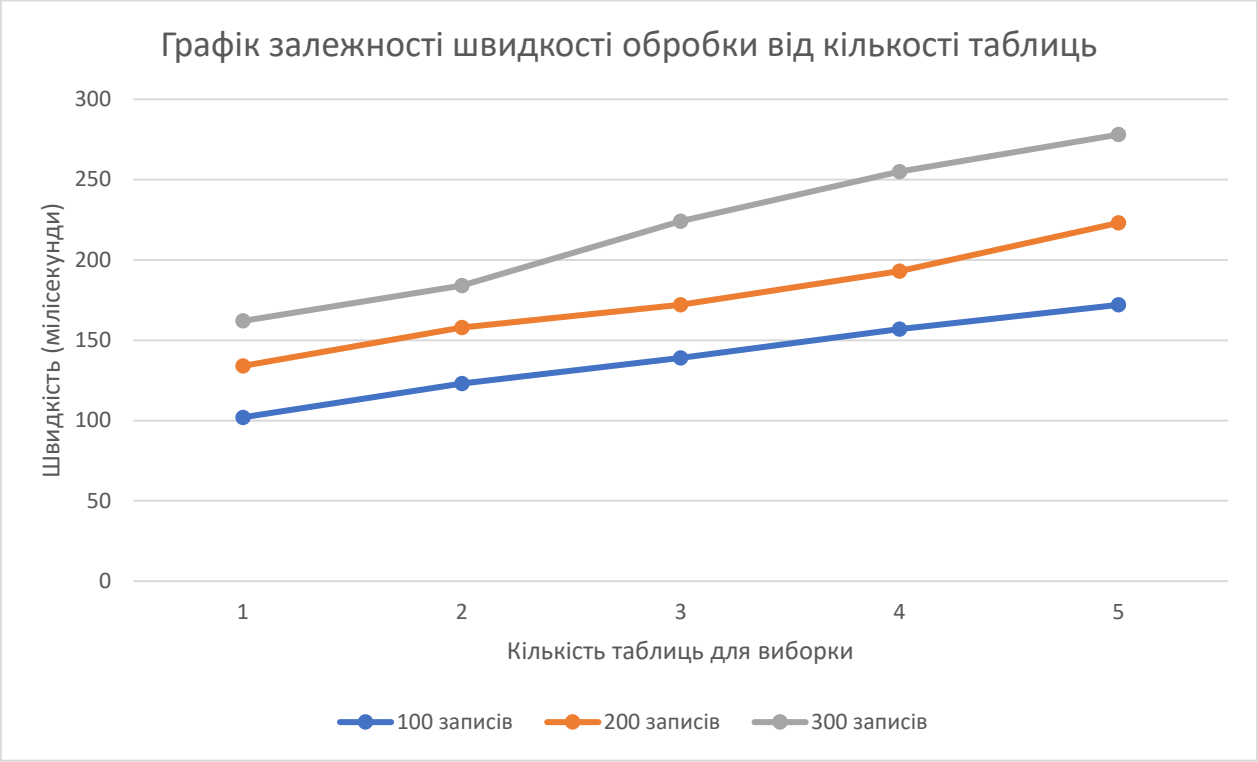

Рисунок 3.14 – Графік залежності швидкості обробки від кількості таблиць технологією Dapper

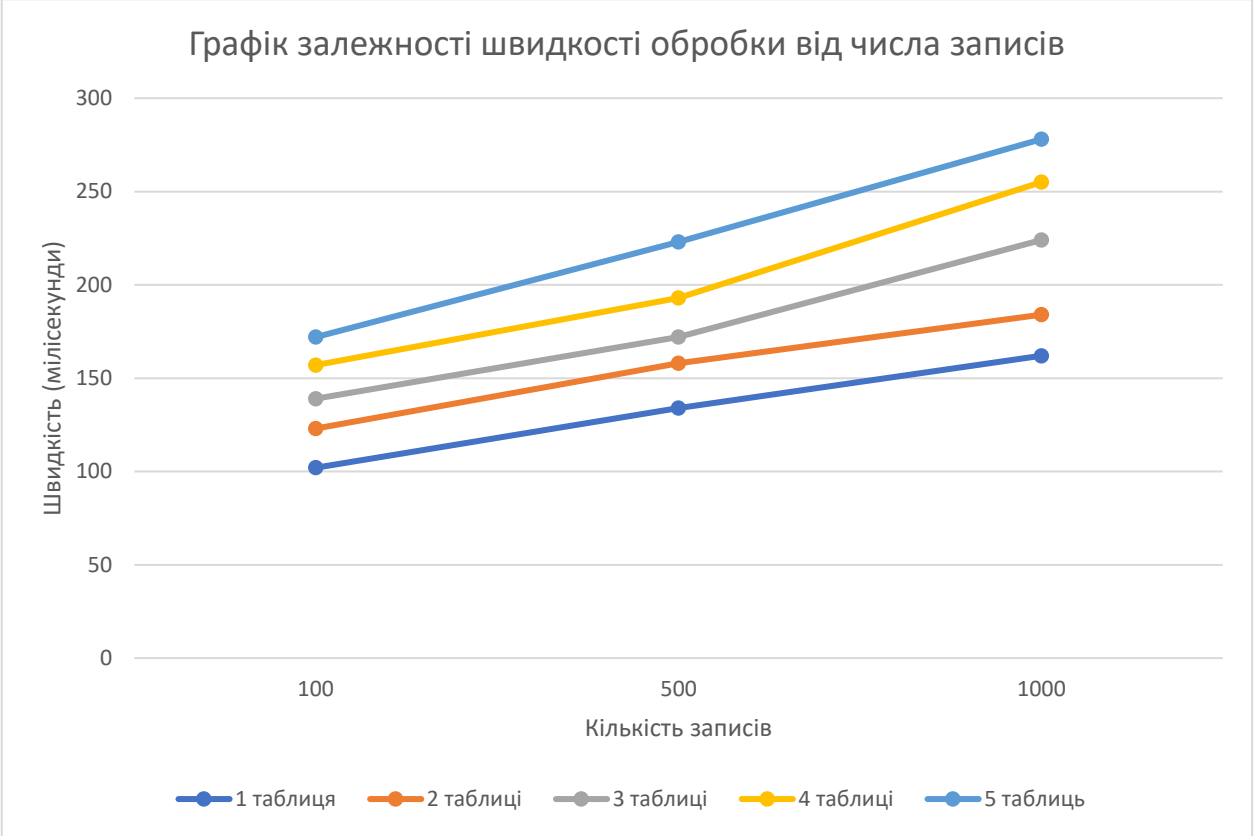

Рисунок 3.15 – Графік залежності швидкості обробки від числа записів технологією Dapper

3.5 Тестування ADO.NET

3.5.1 Додавання запису у базу

Для тестування будемо добавляти у базу сутність PrintingEdition (рисунок 3.1). Конкретні значення полей запису не впливають на швидкість обробки інформації. Головне дотримуватися наступних типів даних у полей:

1) ID - string
2) Title - string
3) Description - string
4) Price - decimal
5) isRemoved - boolean
6) Status - enum
7) Currency - enum
8) Type – enum

Заповнимо нашу базу 10 випадковими записами, та зафіксуймо час за котрий сервер поверне нам результат. Результати представлені у таблиці 3.11.

Таблиця 3.11

Номер досліду	Вхідні параметри	Вихідні параметри
	Сутності	Швидкість обробки, мілісекунди (ms)
1	Сутність 1	36
2	Сутність 2	26
3	Сутність 3	44
4	Сутність 4	45
5	Сутність 5	37
6	Сутність 6	39
7	Сутність 7	45
8	Сутність 8	30
9	Сутність 9	43
10	Сутність 10	36
Середнє значення		38,1

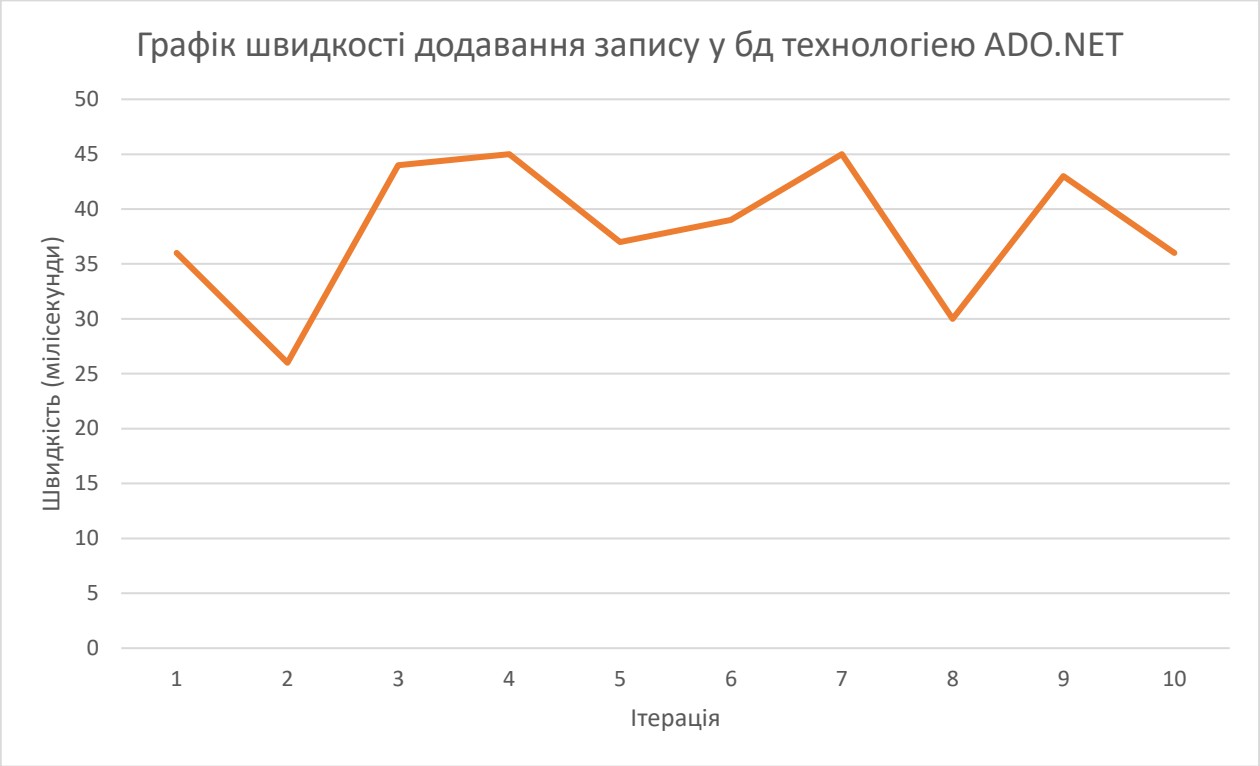

Рисунок 3.16 – Швидкість додавання запису у технології ADO.NET

3.5.2 Модифікація запису у базі даних

Для тестування будемо модифікувати сутність PrintingEdition (рисунок 3.1). Конкретні значення полей запису не впливають на швидкість обробки інформації. Головне дотримуватися наступних типів даних у полей:

1) ID - string
2) Title - string
3) Description - string
4) Price - decimal
5) isRemoved - boolean
6) Status - enum
7) Currency - enum
8) Type – enum

Модифікуємо 10 записів, доданих у першому тесті. Результати представлені у таблиці 3.12.

Таблиця 3.12

Номер досліду	Вхідні параметри	Вихідні параметри
	Сутності	Швидкість обробки, мілісекунди (ms)
1	Сутність 1	29
2	Сутність 2	37
3	Сутність 3	45
4	Сутність 4	26
5	Сутність 5	33
6	Сутність 6	25
7	Сутність 7	39
8	Сутність 8	37
9	Сутність 9	39
10	Сутність 10	42
Середнє значення		35,2

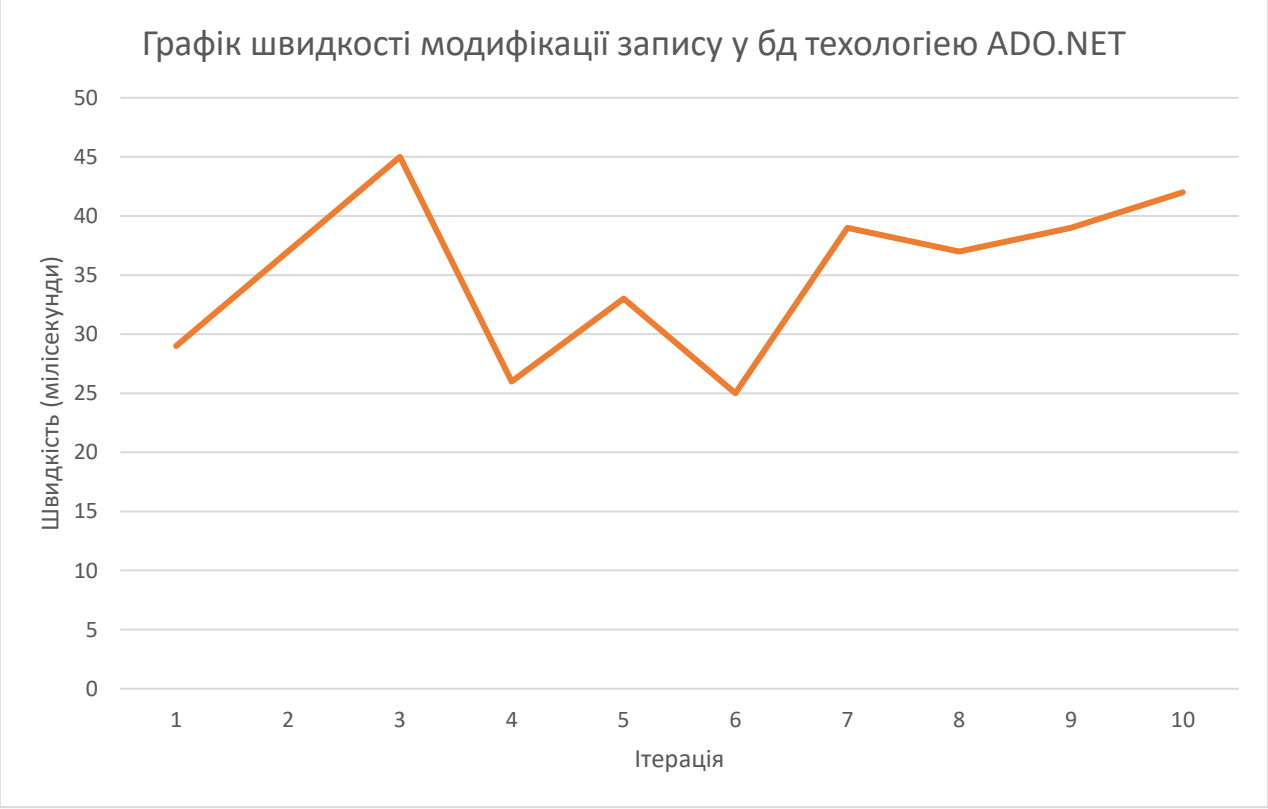

Рисунок 3.17 – графік швидкості модифікації запису у БД технологією ADO.NET

3.5.3 Вилучення 1 запису з бази даних

Для тестування будемо вилучати записи з попередніх тестів, вхідним параметром буде ID об'єкту. Результати представлені у таблиці 3.13.

Таблиця 3.13

Номер досліду	Вхідні параметри	Вихідні параметри
	ID об’єкту	Швидкість обробки, мілісекунди (ms)
1	1	38
2	2	44
3	3	34
4	4	38
5	5	29
6	6	36
7	7	36
8	8	42
9	9	45
10	10	27
Середнє значення		36,9

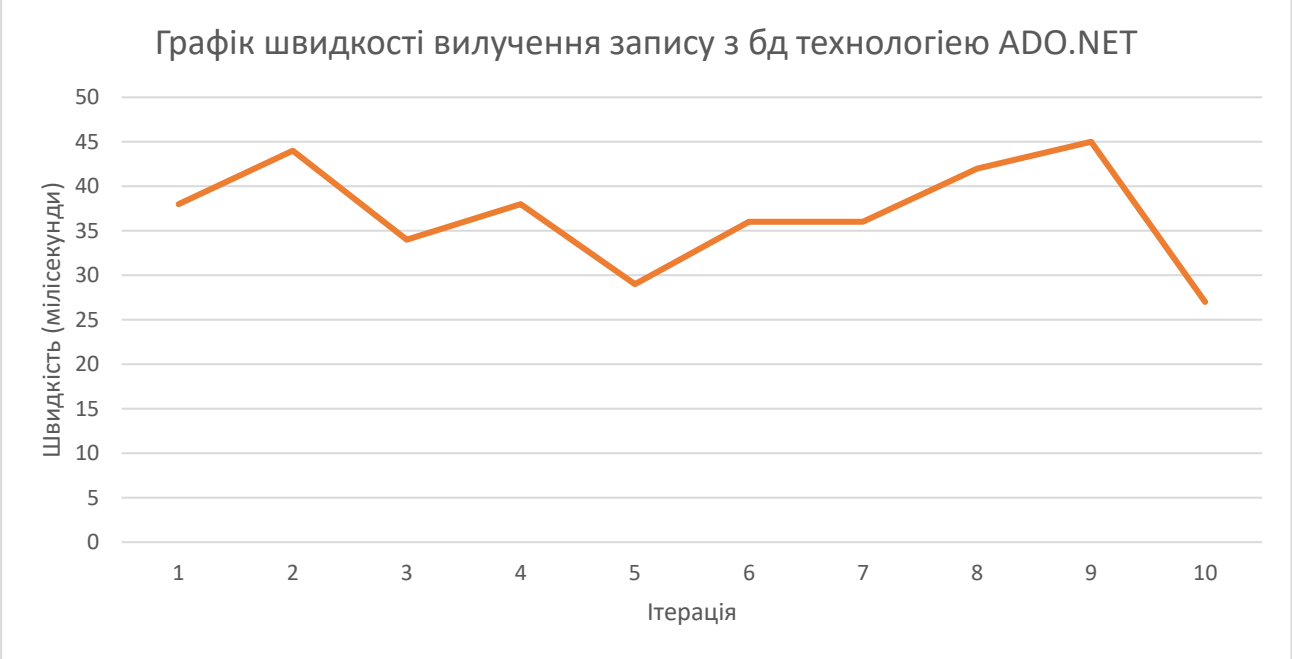

Рисунок0 3.18 – графік швидкості вилучення даних з БД технологією ADO.NET

3.5.4 Видалення запису з бази даних

Для тестування будемо вилучати записи з попередніх тестів, вхідним параметром буде ID об'єкту. Результати представлені у таблиці 3.14

Таблиця 3.14

Номер досліду	Вхідні параметри	Вихідні параметри
	ID об’єкту	Швидкість обробки, мілісекунди (ms)
1	1	30
2	2	39
3	3	35
4	4	44
5	5	36
6	6	41
7	7	30
8	8	30
9	9	26
10	10	27
Середнє значення		33,8

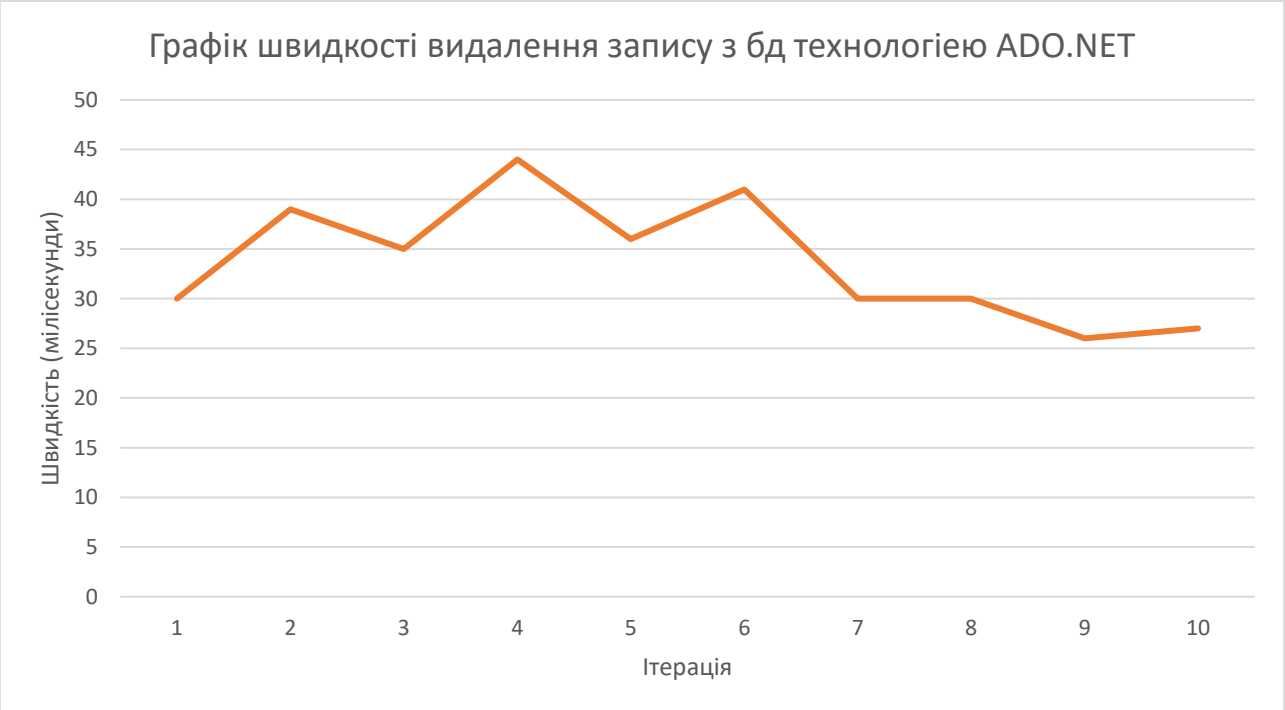

Рисунок 3.19 – графік швидкості видалення запису з БД технологією ADO.NET

3.5.1 Вибірка з декількох таблиць

Для тестування будемо перевіряти швидкість обробки для декількох таблиць з різною кількістю записів для вибірки. Для 100 500 та 1000 записів перевіриться швидкість на витяг з 1-5 таблиць. Результати у таблиці 3.15.

Таблиця 3.15

Номер досліду	Вхідні параметри		Вихідні параметри
	Кількість таблиць для вибірки	Число записів для вибірки	Швидкість обробки, мілісекунди (ms)
1	1	100	97
2	2	100	115
3	3	100	133
4	4	100	150
5	5	100	164
6	1	500	129
7	2	500	152
8	3	500	164
9	4	500	188
10	5	500	215
11	1	1000	154
12	2	1000	176
13	3	1000	216
14	4	1000	247
15	5	1000	270

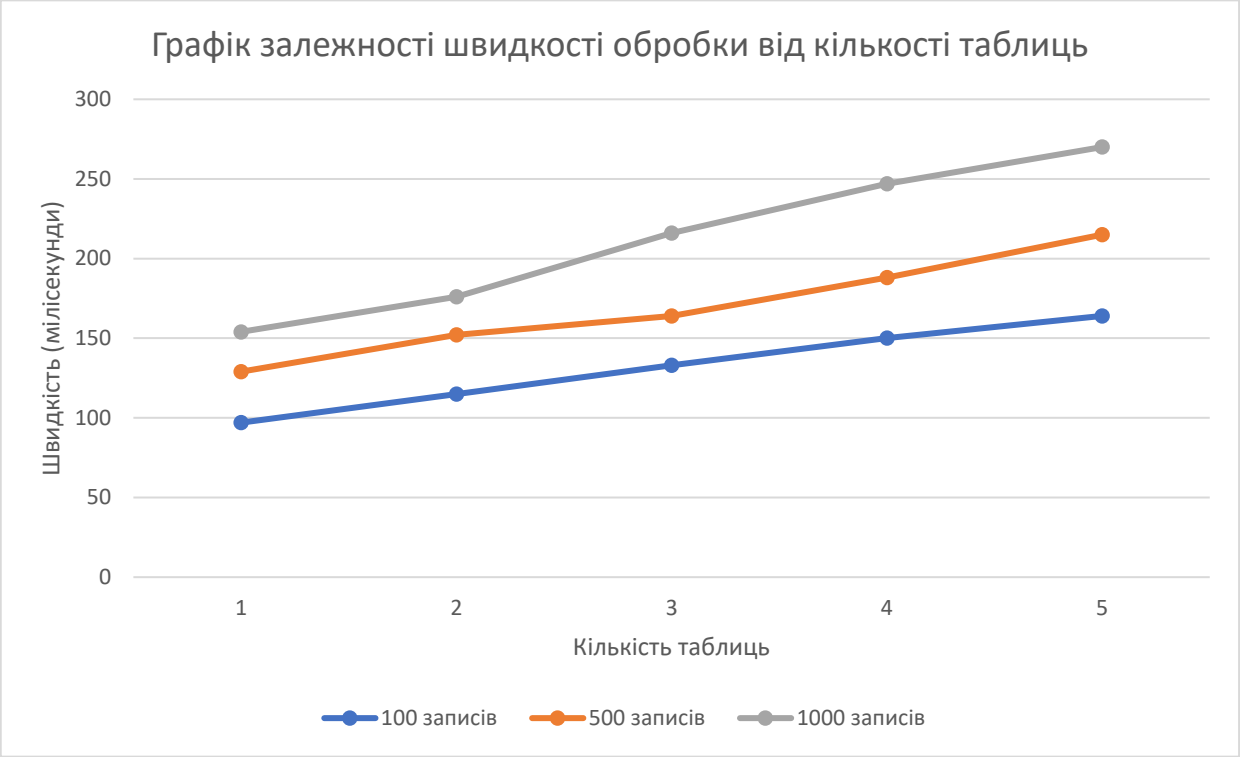

Рисунок 3.20 – графі залежності швидкості обробки від кількості таблиць у технології ADO.NET

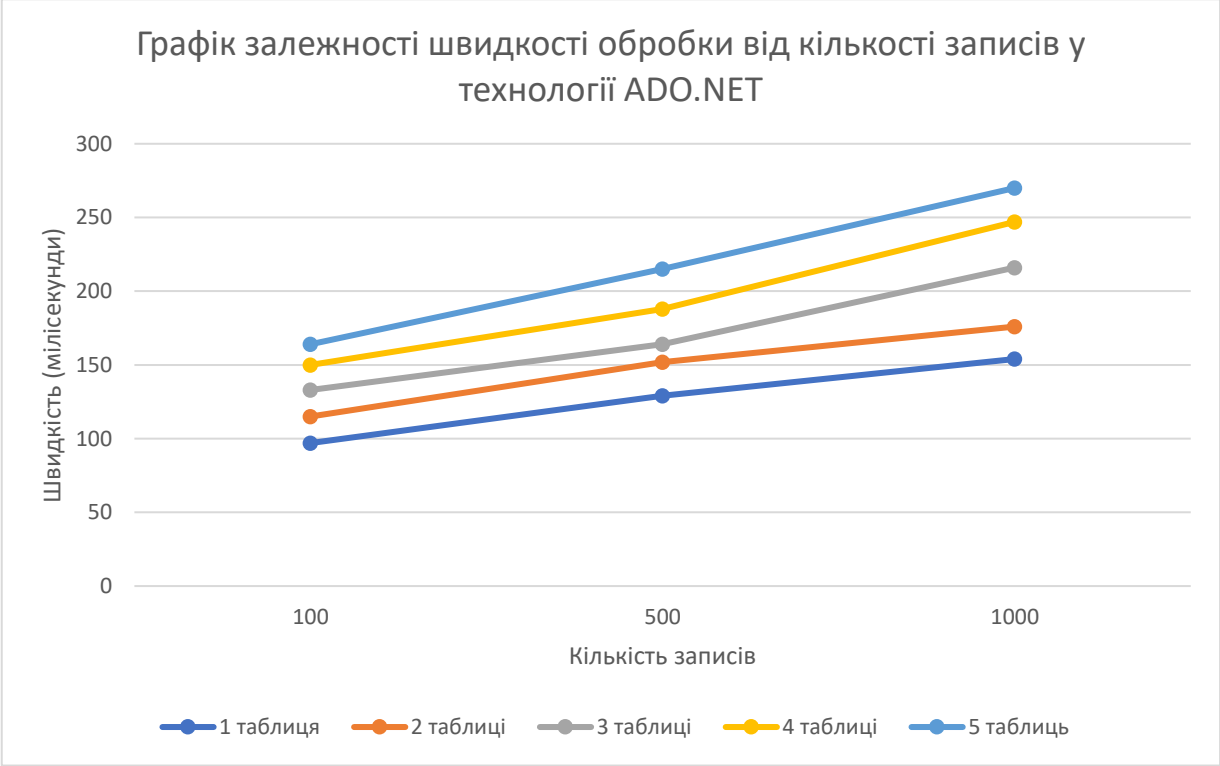

Рисунок 3.21 – графік залежності швидкості обробки від кількості записів у технології ADO.NET

3.6 Порівняння швидкості роботи технологій

3.6.1 Порівняння операції додавання даних

Об’єднаємо результати тестів додавання запису у єдину таблицю для порівняння. Порівняння швидкості технологій представлено у таблиці 3.16.

Таблиця 3.16

Технологія	Мінімальна швидкість	Максимальна швидкість	Середня швидкість
Entity Framework	60	118	73,1
Dapper	35	50	43,3
ADO.NET	26	45	38,1

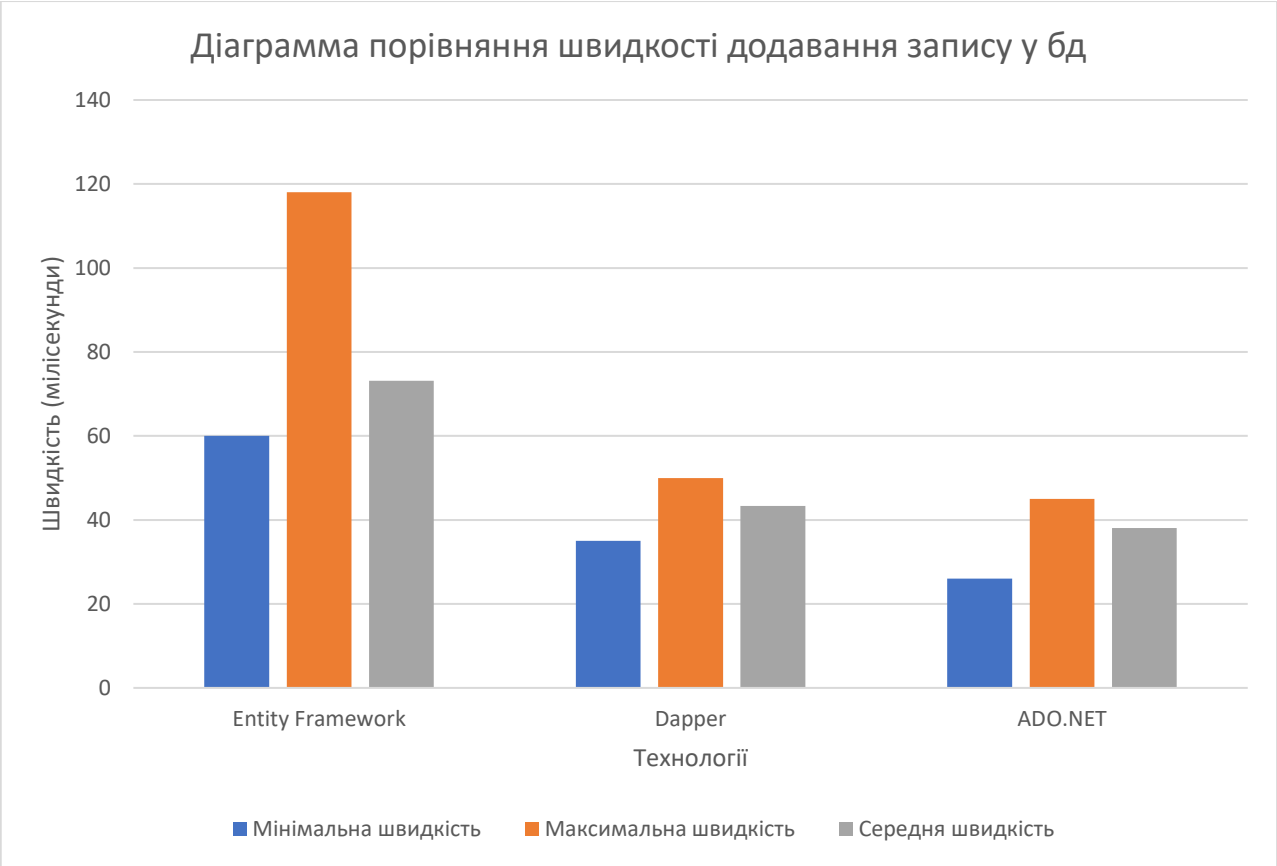

Рисунок 3.22 – Діаграма порівняння швидкості додавання запису у 3 технологій

3.6.2 Порівняння операції модифікації даних

Об’єднаємо результати тестів модифікацій записів у єдину таблицю для порівняння. Порівняння швидкості технологій представлено у таблиці 3.17.

Таблиця 3.17

Технологія	Мінімальна швидкість	Максимальна швидкість	Середня швидкість
Entity Framework	60	109	74,5
Dapper	38	50	43,4
ADO.NET	25	45	35,2

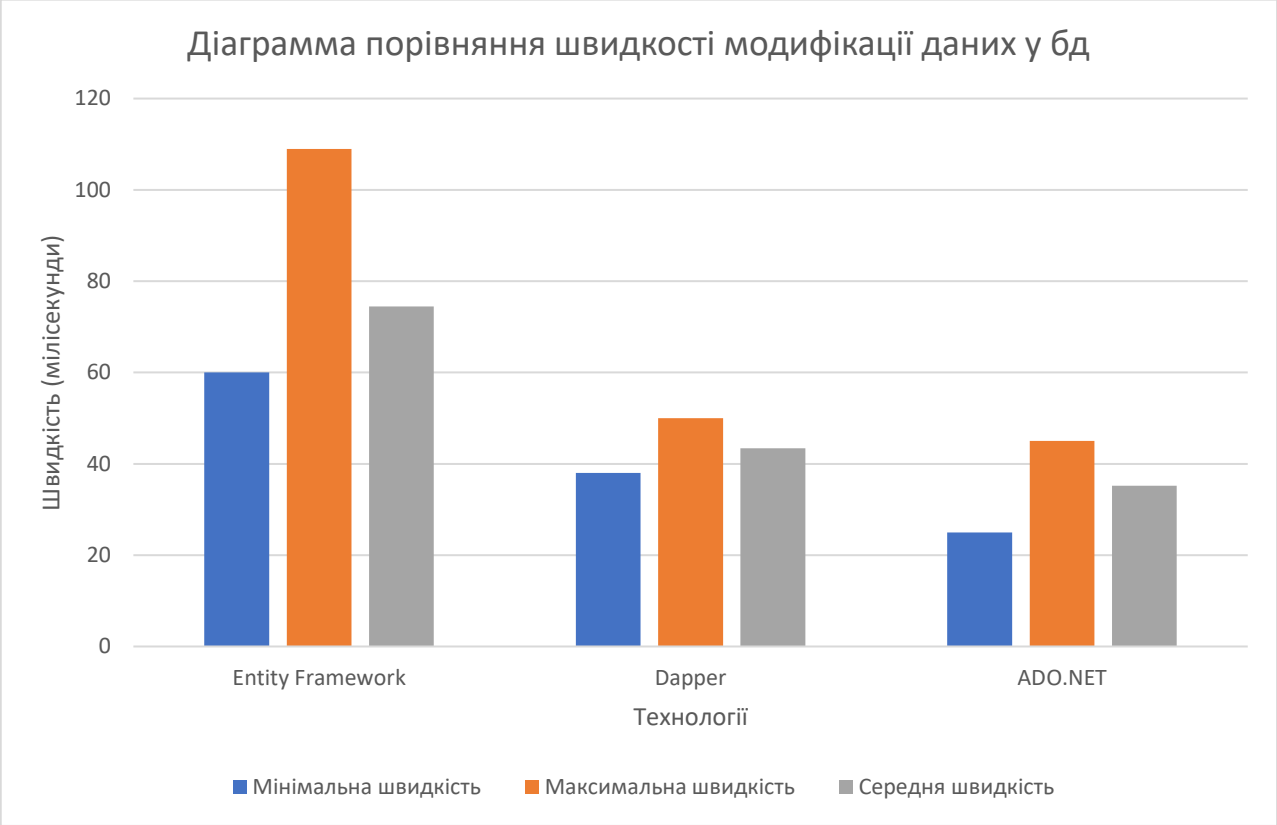

Рисунок 3.23 – Діаграма порівняння швидкості модифікації запису у 3 технологій

3.6.3 Порівняння операції вилучення даних з 1 таблиці

Об’єднаємо результати тестів вилучення запису з 1 табилці у єдину таблицю для порівняння. Порівняння швидкості технологій представлено у таблиці 3.18.

Таблиця 3.18

Технологія	Мінімальна швидкість	Максимальна швидкість	Середня швидкість
Entity Framework	60	131	73,1
Dapper	30	49	38,7
ADO.NET	27	45	36,9

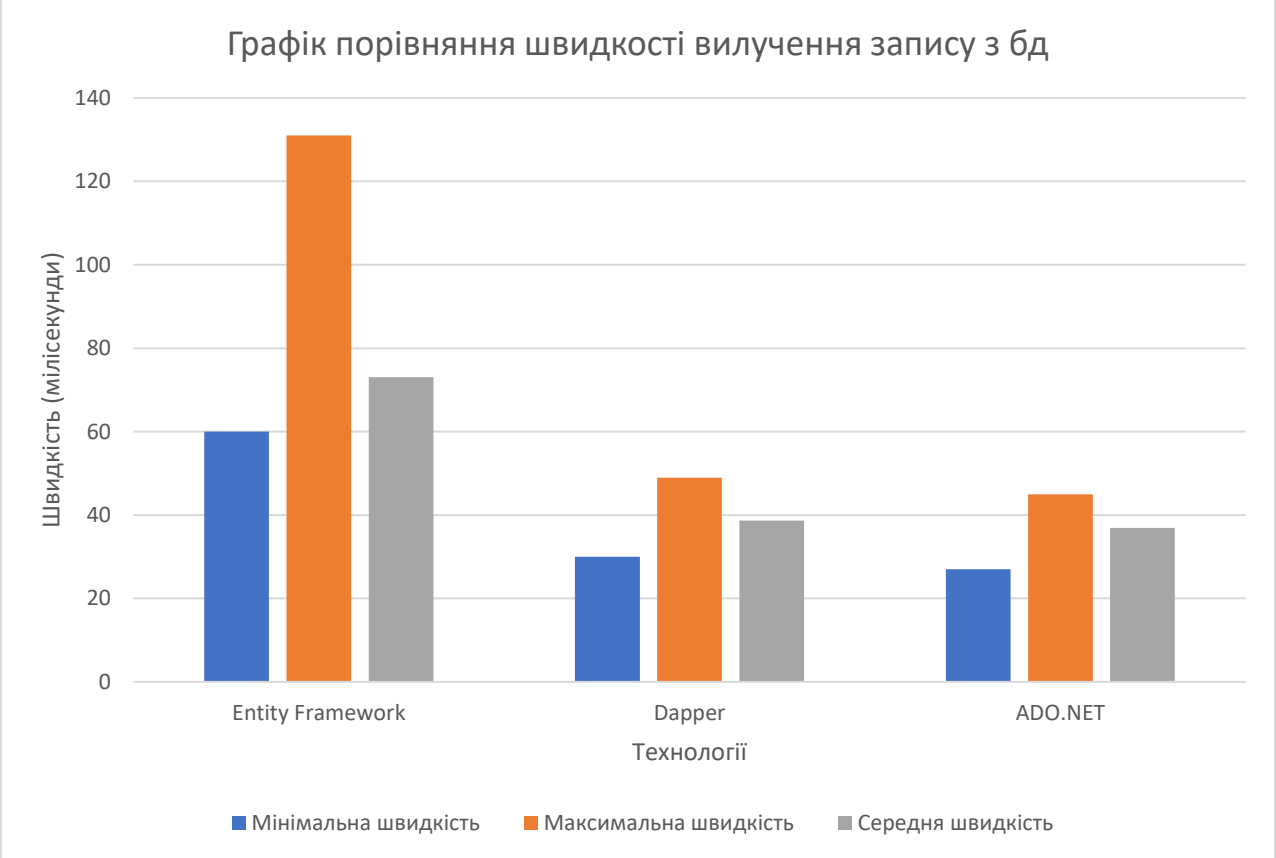

Рисунок 3.24 – Діаграма порівняння швидкості вилучення запису у 3 технологій

3.6.4 Порівняння операції видалення даних

Об’єднаємо результати тестів видалення запису у єдину таблицю для порівняння. Порівняння швидкості технологій представлено у таблиці 3.19.

Таблиця 3.19

Технологія	Мінімальна швидкість	Максимальна швидкість	Середня швидкість
Entity Framework	60	131	73,1
Dapper	30	49	38,7
ADO.NET	27	45	36,9

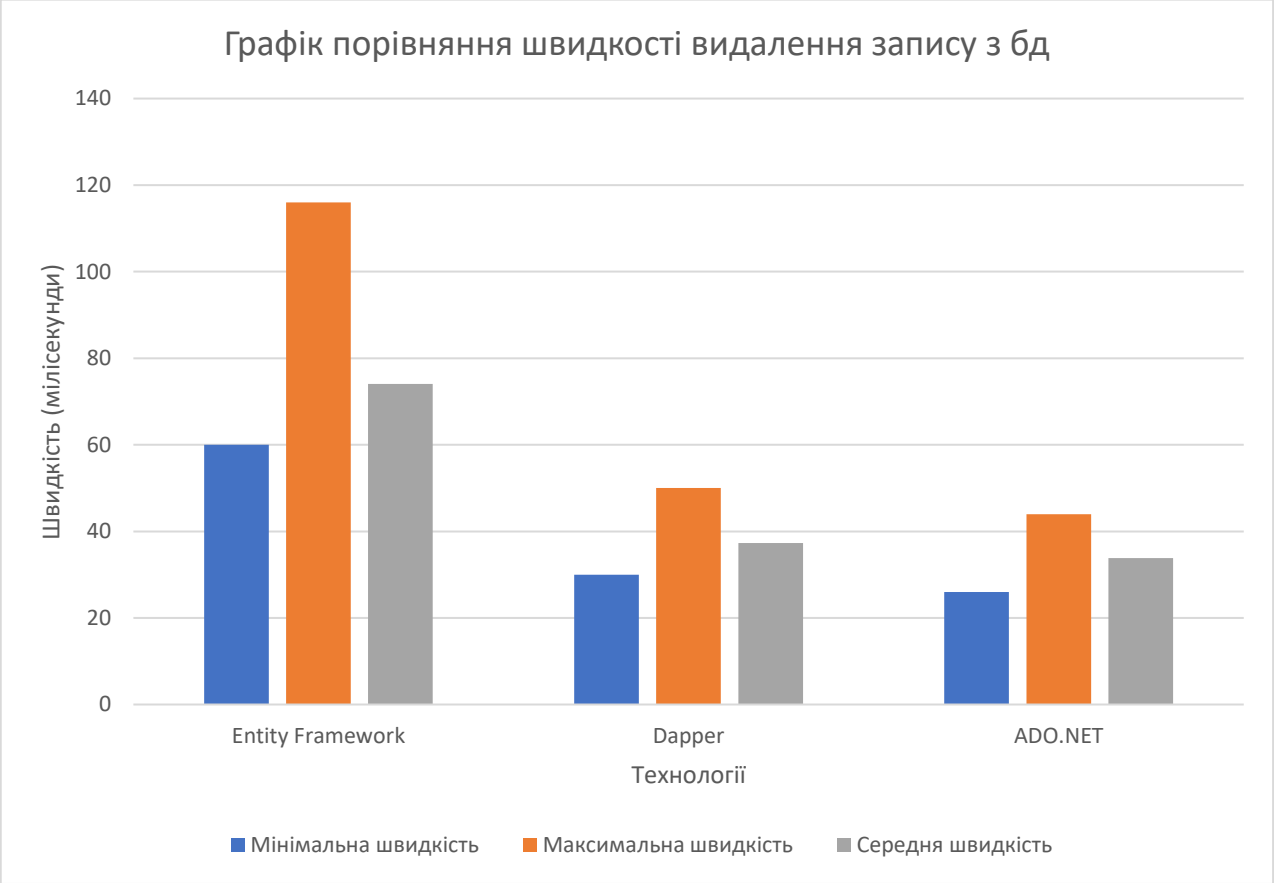

Рисунок 3.25 – Діаграма порівняння швидкості вилучення запису у 3 технологій

3.6.5 Порівняння операції вилучення даних з декількох таблиць з фільтрацією

Об’єднаємо результати тестів вилучення даних з декількох таблиць з фільтрацією у єдину таблицю для порівняння. Дані будуть взяті з таблиць 3.5, 3.10 та 3.15. Для наочності різниці між технологіями візьмемо запити з 5 таблицями.

Таблиця 3.20

Технологія	Кількість запитів	Швидкість (мілісекунди)
Entity Framework	100	353
	500	553
	1000	971
Dapper	100	172
	500	223
	1000	278
ADO.NET	100	164
	500	215
	1000	270

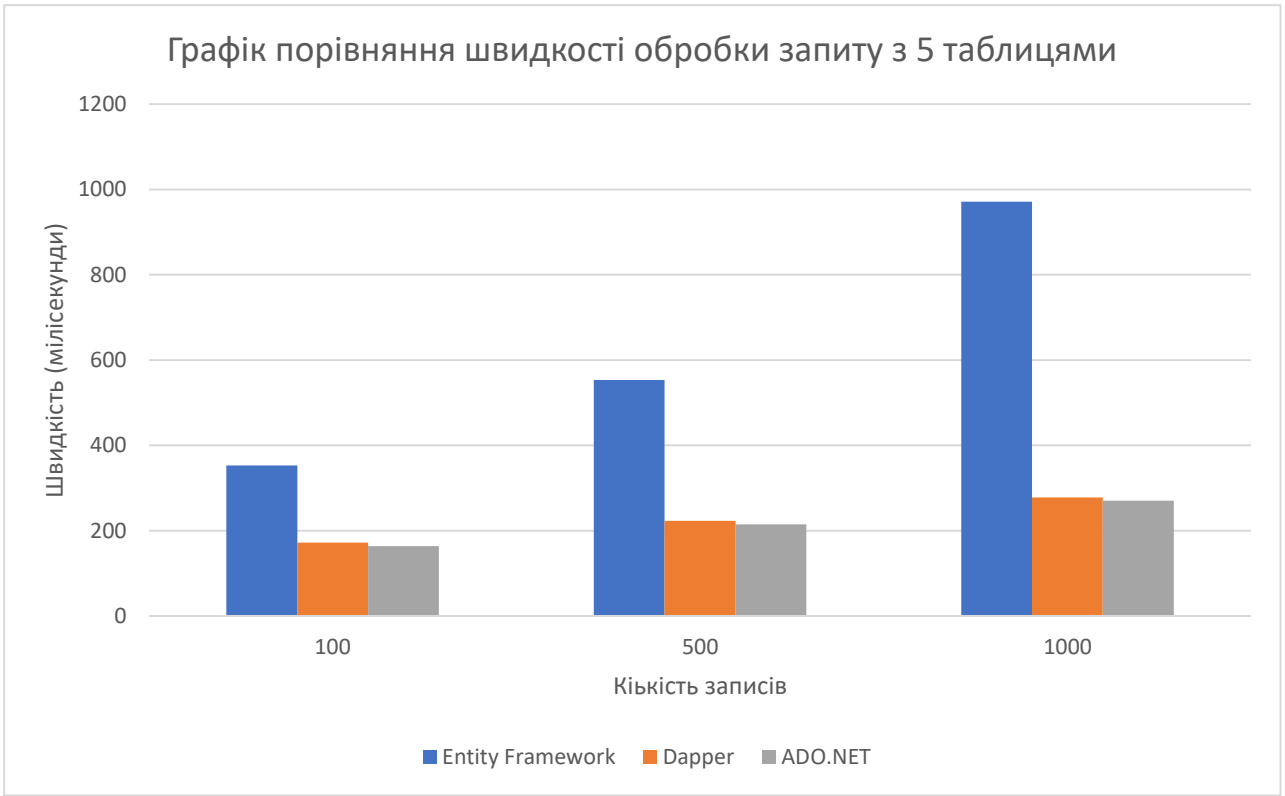

Рисунок 3.26 – Графік порівняння швидкості обробки запису з 5 таблицями

3.7 Висновки з розділу 3

	Entity Framework	Dapper	ADO.NET
Швидкість роботи з малими запитами	3	5	5
Швидкість роботи з великими запитами	3	5	5
Швидкість CRUD-операцій	3	5	5
Зручність роботи читання та редагування коду	5	4	3
Кількість функцій доступних користувачу	5	2	3
Актуальність на сьогоднішній день	5	5	3

Після ознайомлення з усіма трьома технологіями та проведення тестів на швидкість обробки даних, є деякі рекомендації до використовування цих технологій. Як показав тест швидкості обробки інформації, можна зробити висновок, що ADO.NET буде найкращим вибором, але його функціонал значно меньше по зрівнянню з Entity Framework, що очевидно, тому що Entity Framework являєтсья оберткою над ADO.NET, але це і є причина його низької швидкості обробки даних. Entity Framework зачастую виповняє багато інших дій, які не дуже благосклонно впливають на швидкість обробки інформації. Але зручність використання Entity Framework та більша швидкість написання коду по зрівнянню з ADO.NET це з лихою компенсує.

Якщо потрібно написати швидко та якісно програму, приорітет у вибору технології на стороні Entity Framework, але його швидкість роботи, як було сказано вище, не дуже велика. Тут нам на виручку приходить Dapper. У нього не великий об’єм функцій, але у комбінації з Entity Framework, можно поєднати швидкість написання програми Entity Framework та швидкість обробки інформації Dapper. Наприклад можна написати усі основні CRUD-операції на Entity Framework, а складні запити і ті що займають багато часу на обробку, використовуючи Dapper. Але для використовування Dapper потрібно мати дуже добре володіння мовою програмування SQL, в той же час усі запити Entity Framework пишуться мовою C#.

Підсумовуючи вище написане, оптимальним вибором буде написати основну структуру використовуючи Entity Framework та складні велики запити використовуючи Dapper для оптимального баланса зручності та швидкості.

4 ВИСНОВКИ

У процесі виконання дипломної роботи магістра були розглянуті основні способи взаємодії серверу з СУБД, 3 найбільш актуальні технології ORM: Entity Framework, Dapper та ADO.NET. Описані та розглянуті основні положення кожної технології, їх переваги та недоліки.

Було проведено планування експерименту для визначення швидкості обробки інформації у п’яти сценаріях трьома технологіями. Для проведення експерименту було розроблене програмне забезпечення, та обрано спосіб перевірки швидкості обробки інформації.

Отримані результати експериментів були проаналізовані та зроблені висновки на їх основі. Після ознайомлення з трьома технологіями були зроблені висновки що до їх використання у різних ситуаціях та описані основні проблеми кожної з них.

ПЕРЕЛІК ПОСИЛАНЬ

1. Опис технології Entitfy Framework [Електронний ресурс] – Режим доступу до ресурсу: https://metanit.com/sharp/entityframework/.

2. Опис технології Entitfy Framework [Електронний ресурс] – Режим доступу до ресурсу: https://metanit.com/sharp/entityframework/1.1.php.

3. Опис технології ADO.NET [Електронний ресурс] – Режим доступу до ресурсу: https://ru.wikipedia.org/wiki/ADO.NET_Entity_Framework.

4. Опис технології Entitfy Framework [Електронний ресурс] – Режим доступу до ресурсу: https://habr.com/ru/post/262023/

5. Опис технології ADO.NET [Електронний ресурс] – Режим доступу до ресурсу: https://ru.wikipedia.org/wiki/ADO.NET.

7. Опис технології ADO.NET [Електронний ресурс] – Режим доступу до ресурсу: https://metanit.com/sharp/adonet/.

8. Опис технології ADO.NET [Електронний ресурс] – Режим доступу до ресурсу: https://metanit.com/sharp/adonet/1.1.php.

9. Опис технології ADO.NET [Електронний ресурс] – Режим доступу до ресурсу: https://docs.microsoft.com/en-us/dotnet/framework/data/adonet/ado-net-overview.

10. Опис технології ADO.NET [Електронний ресурс] – Режим доступу до ресурсу: https://professorweb.ru/my/ADO_NET/base/level1/1_1.php.

11. Опис технології ADO.NET [Електронний ресурс] – Режим доступу до ресурсу: http://www.codenet.ru/db/other/ado-dot-net/

12. Опис технології Dapper [Електронний ресурс] – Режим доступу до ресурсу: https://metanit.com/sharp/articles/mvc/dapper.php.

13. Опис технології Dapper [Електронний ресурс] – Режим доступу до ресурсу: https://metanit.com/sharp/aspnet5/26.1.php.

14. Опис технології Dapper [Електронний ресурс] – Режим доступу до ресурсу: https://en.wikipedia.org/wiki/Dapper_ORM.

15. Опис технології Entitfy Framework та Dapper [Електронний ресурс] – Режим доступу до ресурсу: https://docs.microsoft.com/ru-ru/archive/msdn-magazine/2016/may/data-points-dapper-entity-framework-and-hybrid-apps.

16. Опис технології Dapper [Електронний ресурс] – Режим доступу до ресурсу: https://www.infoworld.com/article/3025784/how-to-use-the-dapper-orm-in-c.html.

17. Порівняння швидкості Entity Framework та Dapper [Електронний ресурс] – Режим доступу до ресурсу: https://www.exceptionnotfound.net/dapper-vs-entity-framework-vs-ado-net-performance-benchmarking/

18. Процес планування експерименту [Електронний ресурс] – Режим доступу до ресурсу: http://web.kpi.kharkov.ua/ea/wp-content/uploads/sites/25/2017/02/OND-Ukr.pdf

19. Проблеми Entity Framework [Електронний ресурс] – Режим доступу до ресурсу: https://habr.com/ru/post/459716/