ДЕРЖАВНИЙ УНІВЕРСИТЕТ ТЕЛЕКОМУНІКАЦІЙ

НАВЧАЛЬНО–НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Кафедра інженерії програмного забезпечення

Пояснювальна записка

до магістерської роботи
на ступінь вищої освіти магістр

на тему: «**ОПТИМІЗАЦІЯ ORM ТЕХНОЛОГІЙ ENTITY FRAMEWORK CORE ЗА ДОПОМОГОЮ РОЗШИРЕННЯ T-SQL**»

Виконав: студент 6 курсу, групи ПДМ–61
спеціальності
121 Інженерія програмного забезпечення
(шифр і назва спеціальності/спеціалізації)

______________________ Гнатюк В.І.
(прізвище та ініціали)

Керівник _________ Бондарчук А.П.
(прізвище та ініціали)

Рецензент ________________________________
(прізвище та ініціали)

Київ –2022

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ТЕЛЕКОМУНІКАЦІЙ

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Кафедра Інженерії програмного забезпечення
Ступінь вищої освіти -«Магістр»
Спеціальність підготовки – 121 «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Інженерії програмного забезпечення
_____________Негоденко О.В.
"____" ___________2022 року

З А В Д А Н Н Я
НА МАГІСТЕРСЬКУ РОБОТУ СТУДЕНТА

ГНАТЮКА ВЛАДИСЛАВА ІВАНОВИЧА
(прізвище, ім'я, по батькові)

1. Тема роботи: «Оптимізація ORM технологій Entity Framework Core за допомогою розширення T-SQL»

Керівник роботи: Бондарчук А.П., д.т.н., професор
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

Затверджені наказом вищого навчального закладу від «11» жовтня 2021 року №170.
2. Строк подання студентом роботи «15» грудня 2021 року
3. Вхідні дані до роботи
Моделі оптимізації SQL запитів;
Науково-технічна література з питань, пов'язаних з програмним забезпеченням щодо оптимізації SQL запитів;
Алгоритми оптимізації SQL запитів;

4. Зміст розрахунково-пояснювальної записки(перелік питань, які потрібно розробити).
4.1 Архітектура програмного забезпечення.

4.2 Вимоги та оцінка якості ПЗ та його тестування.
4.3 Реалізація моделей та алгоритмів для оптимізації виконання SQL запитів.
4.4 Тестування програмного забезпечення.

5. Перелік демонстраційного матеріалу (назва основних слайдів)
1. Актуальність проблеми та існуючі моделі та алгоритми їх вирішення
2. Існуючі аналоги розширення ORM технології Entity Framework Core
3. Принцип роботи програмного забезпечення
4. Оптимізація SQL запитів
5. Архітектура програмного забезпечення
6. Демонстрація тестування SQL запитів з розширеннями та без

6. Дата видачі завдання____

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів магістерської роботи	Строк виконання етапів роботи	Примітка
1	Підбір науково-технічної літератури		
2	Аналіз існуючих рішень		
3	Аналіз моделей та алгоритмів оптимізації		
4	Розробка архітектури ПЗ		
5	Розробка програмного забезпечення		
6	Вступ, висновки, реферат		
7	Опис систем, технологій та вимог до ПЗ		
8	Опис моделей, алгоритмів реалізації ПЗ		
9	Опис процесу тестування ПЗ		
10	Розробка обов‘язкових демонстраційних матеріалів		
11	Попередній захист роботи		
12	Здача роботи		

Студент ___________ ______________________
(підпис) (прізвище та ініціали)

Керівник роботи __________ __________________
(підпис) (прізвище та ініціали

РЕФЕРАТ

Текстова частина магістерської роботи 82 с., 18 рис., 24 джерел.

ОПТИМІЗАЦІЯ SQL ЗАПИТІВ, ORM, ENTITY FRAMEWORK CORE, SQL, T-SQL, LINQ, ENTITY FRAMEWORK PLUS, SQL SERVER, SQL DAPPER, .NET CORE, CLASS LIBRARY, NUGET PACKAGE, ARCHITECTURE PATTERNS, REPOSITORY, UNITOFWORK, DAO PATTERN, DATABASE

Об'єкт дослідження – розширення функціоналу ORM технології Entity Framework Core за допомогою аналізу та модифікація SQL запитів.

Предмет дослідження – програмне забезпечення для розширення функціоналу ORM технологією Entity Framework Core за допомогою аналізу та модифікація SQL запитів.

Мета роботи – оптимізація швидкості та зменшення витрат пам'яті на виконання SQL запиту.

Методи дослідження – методи аналізу та модифікації SQL запитів.

У роботі проведено аналіз існуючого програмного забезпечення, метою якого є оптимізація швидкості виконання SQL запитів та економія пам'яті на створенні безлічі об'єктів. Єдиним ліцензійованим доступним аналогом є бібліотека Entity Framework Plus. Зазвичай, компанії створюють для внутрішнього користування набір розширень, які не поширюються за їх межі.

Загальною проблою програмного забезпечення такого типу є відсутність постійного аналізу виконуваних запитів та неоптимальні пакетні запити, які можуть закінчити своє виконання помилкою, якщо пакет має забагато даних.

Особливістю програмного забезпечення є аналіз створюваних розробником програмного забезпечення SQL запитів та їх подальшою оптимізацією, якщо це необхідно. Описано архітектуру, вимоги до безпеки та тестування. Програмне забезпечення написано и вигляді набору бібліотек на платформі .NET Core. В якості технологій використовує: Entity Framework Core та SQL Dapper. Для зручності

встановлення, оновлення версії програмного забезпечення – публікується в NuGet, в захищеному сховищі.

Отже, розроблено та описано набір бібліотек, завданням яких є аналіз та модифікація створенних ORM технологією Entity Framework Core SQL запитів за допомогою використання мови запитів SQL і її процедурного розширення T-SQL.

Дане програмне забезпечення може бути використано у сфері розробки програмного забезпечення на платформі .NET Core з використанням ORM технології Entity Framework Core, яке потребує швидкого виконання SQL запитів та економії пам’яті.

Галузь використання – розробка програмного забезпечення.

ЗМІСТ

ВСТУП **11**

1. АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ **13**

1.1. Entity Framework Plus 13

2. ORM ТЕХНОЛОГІЇ **15**

2.1. ORM – Object relation mapping 15

2.2. Реліаційні бази даних. Проблеми їх взаємодії з об'єктами 16

2.3. Типи ORM систем 18

2.4 Переваги та недоліки ORM систем 20

2.5. Технологія Entity Framework Core 22

2.6. Інфраструктура Entity Framework Core 23

2.7. Взаємодія розробників програмного забезпечення з ORM технологією Entity Framework Core 26

2.8. SQL Dapper 32

2.9. Переваги та недоліки Dapper 33

2.10. Порівняння і проблеми SQL Dapper та Entity Framework Core. 35

2.11. Висновок 38

3. ВИМОГИ ТА ОЦІНКА ЯКОСТІ РОЗШИРЕНЬ ТЕХНОЛОГІЇ **40**

3.1. Загальний опис 40

3.2. Вимоги до безпеки 41

3.3. Опис архітектури розширень 43

3.3.1. Архітектура аналізаторів SQL запитів побудованих ORM технологією Entity Framework Core 44

3.3.2. Архітектура виконання перевизначених методів проекції та перехрещеного пошуку даних 47

3.3.3. Архітектура виконання операції оновлення, видалення та додавання даних при роздільних операціях 49

3.3.4 Архітектура виконання операції оновлення, видалення та додавання даних партіями 51

3.4. Випробування та тестування 53

3.4.1. Випробування вимог до безпеки 53

3.4.2. Тестування програмного забезпечення ... 54
4. МОДЕЛІ ТА АЛГОРИТМИ ... 58
4.1. Модель вдосконалення виконання операції проекції ... 58
4.2. Висновок ... 60
4.3. Модель оптимізації виконання оператору перехресного пошуку даних ... 61
4.4. Висновок ... 64
4.3. Алгоритм для оптимізації масового оновлення/видалення/додавання даних одиничним шляхом ... 66
4.4. Алгоритм для оптимізації масового оновлення/видалення/додавання даних пакетами ... 68
5. РЕАЛІЗАЦІЯ РОЗШИРЕНЬ ... 70
5.1. Створення програмного забезпечення та його конфігорування ... 70
5.2. Розробка нової моделі для розширення Entity Framework Core ... 71
5.3. Розробка абстракції аналізаторів та побудовників SQL запитів ... 74
5.4. Розробка розширень операції проекції та перехресних з’єднань ... 76
5.4.1. Реалізація моделі аналізу для оптимізації операції проекції ... 78
5.4.2. Реалізація моделі аналізу та побудування SQL запиту для операції перехресного пошуку даних ... 78
5.5. Розробка розширень для масового оновлення, видалення та додавання даних до таблиць ... 80
5.6. Розробка розширень для масового оновлення, видалення та додавання даних до таблиць пакетами ... 81
5.7. Публікація програмного забезпечення у вигляді NuGet пакету ... 81
6. ТЕСТУВАННЯ ... 83
6.1. Тестування ПЗ на SQL ін’єкції ... 83
6.2. Тестування програмного забезпечення ... 84
ВИСНОВКИ ... 89
ПЕРЕЛІК ПОСИЛАНЬ ... 91
ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ ... 93

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

ORM – Object-Relation-Mapping

EF Core – Entity Framework Core

SQL – Structured Query Language

T-SQL – Transact Structured Query Language

API – Application Programming Interface

Linq – Language Integrated Query

СУБД – Система Управління Базами Даних

DTO – Data Transfer Object

CRUD – Create Read Update Delete

ВСТУП

Обґрунтування вибору теми та її актуальність: Під час розробки програмного забезпечення розробникам програмного забезпечення доводиться взаємодіяти з даними. Щоб не описувати програмний код для роботи з базою даних, вони використовують ORM техноголії. Проте, більшість технологій надаючи готове програмне рішення для роботи з даними нехтує оптимальним створенням запитів для викнання та потребляє багато ресурсів.

Для оптимізації процесу створення та виконання SQL запитів необхідно розробити програмне забезпечення, яке буде аналізувати створюванні розробником запити та модифікувати їх за допомогою мови запитів SQL та її процедурного розширення T-SQL.

Ступінь вивчення проблеми: На даний момент існує не значна кількість програмного забезпечення, яке займається оптимізацією SQL запитів генеруємих ORM технологіями. Внаслідок того, що зазвичай майже кожна компанія вирішує дану проблему на своему рівні створюючи необхідне програмне забезпечення у вигляді розширень ORM систем власним програмним кодом.

У зв’язку з широким використанням розробниками ORM технологій для спрощення роботи з даними, проблема виконання неоптимальних та повільних SQL запитів може виникати в будь-якому програмному забезпечені. На даний момент в більшості випідків проблему вирішують локально – створюючи збережені процедури або функції, які виконують роботу лише з конкретною проблему.

Об’єктом дослідження є розширення функціоналу ORM технологією Entity Framework Core за допомогою аналізу та модифікація SQL запитів.

Предметом роботи є програмне забезпечення для розширення функціоналу ORM технологією Entity Framework Core за допомогою аналізу та модифікація SQL запитів.

Метою роботи є оптимізація швидкості та зменшення витрат пам’яті на виконання SQL запиту.

Методика дослідження: Перш за все, необхідно було визначити найпроблемніші запити генеруємі ORM системою. Після, можна було зробити висновки, які саме SQL запити необхідно аналізувати та модифікувати. Виходячи з цього, потрібно було розробити правильну методику аналізу та модифікації SQL запитів, розробити архітектуру програмного забезпечення з можливістю подальшого розширення.

Для модифікації SQL запитів потрібно було визначити моделі та алгоритми, які зможуть оптимізувати генеруємі ORM системою запити. Через неможливість модифікації ядра СУБД, необхідно вносити зміни на етапі формування SQL запиту, перед його виконанням.

Беручи до уваги вимоги до програмного забезпечення, необхідно створити набір бібліотек, які розробник зможе підключати до свого рішення. Для зручного використання, підтримки версіонності продукту, можливості оновлень необхідно розроблені бібліотеки додати до NuGet пакетів.

Наукова новизна роботи: Таким чином, наукова новизна полягає в створенні розширень для ORM технології Entity Framework Core з метою аналізу та модифікації SQL запитів за допомогою мови запитів SQL та її процедурного розширення T-SQL.

Практична значущість результатів: Дане програмне забезпечення можна використовувати у розробці будь-якого типу додатків, які працюють з базами даних через використання ORM технології Entity Framework Core.

1. АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ

1.1. Entity Framework Plus

Entity Framework Plus [1] – це програмне забезпечення у вигляді набору бібліотек, які можна завантажити через NuGet пакети. Дане програмне забезпечення слугує для розширення стандартного функціоналу ORM технологій Entity Framework та Entity Framework Core. Дане програмне забезпечення існує з 2014 року. Воно не піддається модифікації та оновлень на момент написання данної роботи. Entity Framework Plus підтримує досить мало версій ORM технології Entity Framework від Microsoft:

- Entity Framework 5
- Entity Framework 6.x
- Entity Framework Core 2.x

На момент написання даної роботи остання версія Entity Framework Core 6.0. Це означає, що програмне забезпечення Entity Framework Plus для розширень може бути несумісне з новими версіми ORM системи.

На відміну від кількості підтримуємих версій, бібліотека розширень має досить великий спектр SQL постачальників даних:

- SQL Server 2008+
- SQL Azure
- SQL Compact
- Oracle
- MySQL
- PostgreSQL
- SQLite

Це дозволяє використовувати використовувати бібліотеку розширень в контексті Entity Framework Core з різними SQL постачальниками даних. Це являє як

перевагу, так і недолік. Адже, кожен постачальник підтримує свій власний діалект мови запитів SQL та його процедурного розширення.

Бібліотека розширень має широкий спектр функціоналу для допомоги в розробці програмного забезпечення. Серед основного функціоналу можна виділити:

- Виконання масових операцій. Насамперед, це операції оновлення, видалення та додавання даних до таблиць.
- Також, в бібліотеці розширень присутній влаштований механізм кешування даних. Тобто, розробник програмного забезпечення може забути про створення не тільки програмного коду для роботи з даними, а й коду для створення механізмів кешування.
- Entity Framework Plus має можливість логування виконання кожного SQL запиту «з коробки». Логування може бути сконфігуровано: які сутності протрібно логувати, які властивості сутностей, в який час, при якому навантажені системи, яке програмне забезпечення виконало запит.
- Наявність можливості створювати тригери в пам’яті додатку з можливістю їх включення та виключення.
- Можливості розробки фільтрів.
- Використання данімачного Linq. Дозволяє генерувати сутності під час виконання додатку без створення самій класів сутностей.

Бібліотека Entity Framework Plus має широкий спектр можливостей та багатий функціонал необхідний розробникові програмного забезпечення для оптимальної роботи з даними. Проте, через застарілість версії ORM технології Entity Framework Core, з якою вона взаємодіє, вона не буде сумісна для програмного забезпечення нових версій. Також, при використанні цієї бібліотеки з’являється залежність від використаного програмного коду, що ускладнює процес міграції на інші бібліотеки.

2. ORM ТЕХНОЛОГІЇ

2.1. ORM – Object relation mapping

ORM [2] (Object-Relational Mapping, Об’єктно-орієнтоване перетворення) – це технологія (фреймворк) програмування, яка дозволює взаємодіяти бази даних з об’єктно-орієнтованими мовами програмування. Іноді результатом використання цієї технології називають створення віртуальної об'єктно-орієнтованної бази даних, хоча насправді база даних є реліаційною.

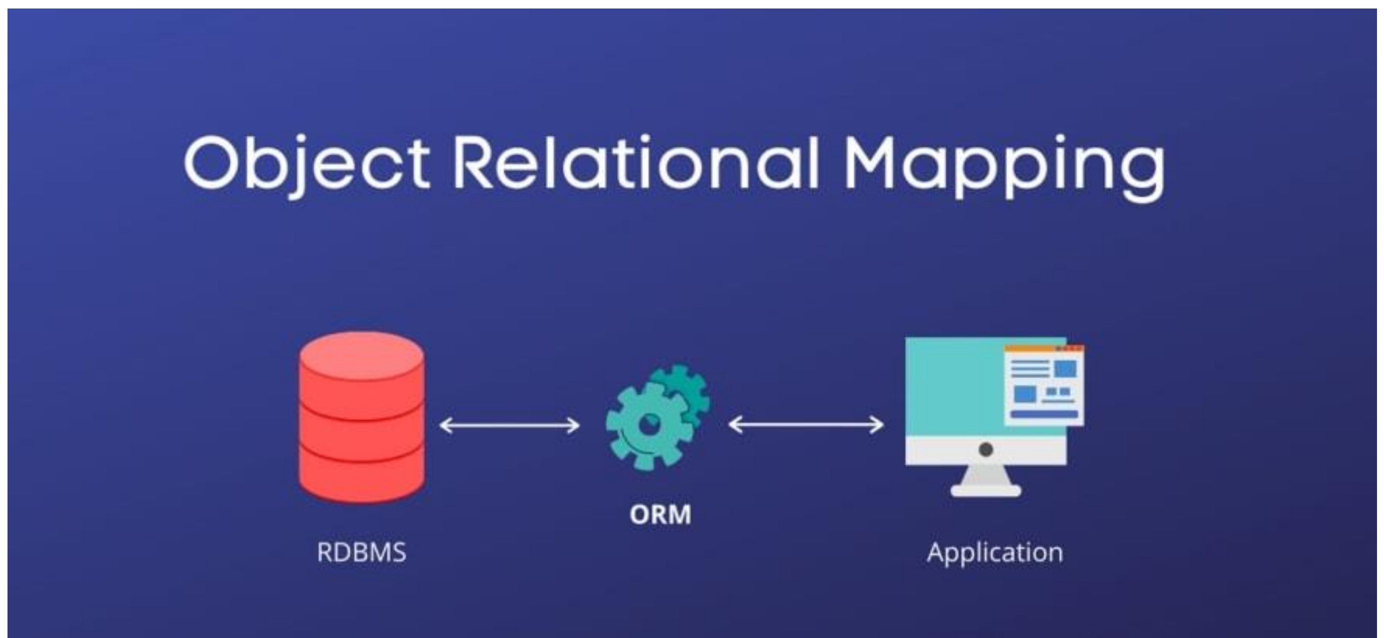


Рис. 2.1. – Взаємодія додатку з базою даних з використанням ORM технології

ORM системи почали свій шлях з 2003 року, коли Мартін Фаулер описав у своїй книзі «Шаблони та архітектери корпоративних додатків» архітектурні шаблони проектування програмного забезпечення. Він визначив, що необхідно автоматизувати процес розробки інтерфейсу для взаємодії з даними, щоб розробники не витрачали час на його постійне написання.

ORM технологія представляє собою набір пакетів або бібліотек, які розробники під’єднують до свого додатку з цілью спрощення взаємодії з даними із баз даних.

Технологія використовує внутрішнью конвертацію даних з бази данних у об'єкти та навпаки. Для цього в бібліотеках описують аналізатори об'єктів та структури бази даних. Вони аналізують імена, тип даних, обмеження властивостей та намагаються співставити властивості об'єктів з полями бази даних. Іноді, для кращого співставлення використовуються налаштування, які може надати розробник для того, щоб навчити ORM технологію співставляти властивості об'єктів з полями бази даних явно.

За допомогою використання ORM технології розрозбникам програмного забезпечення не потрібно замислюватись на написанні SQL команд та використовувати звичні для них об'єкти, щоб отримати доступ до даних, зберігаємих у базі даних.

2.2. Реліаційні бази даних. Проблеми їх взаємодії з об'єктами

Найшпоширинішим способом зберігання даних є реліаційні бази даних [3]. Вони зберігають інформацію у вигляді таблиць, які зв'язані між собою. Кожен стовпець таблиці являє собою якийсь конкретний тип даних для зберігання. Наприклад: дата, вік, ім'я користувача, податковий номер та інше. Кожен рядок таблиці являє собою конкретний запис інформації. З релійними базами даних працювати напряму досить легко. Проте, коли справа доходить до взаємодії коду додатку з базою даних – починають виникати труднощі. Їх вирішує використання технології ORM. Щоб краще зрозуміти, чому пряма взаємодія реляційних баз даних з дадатком супроводжується труднощами – давайте спочатку розглянемо чому вибір розробників падає на реляційні бази даних та як вони працюють.

Цей спосіб отримав свою популярність завдяки простоті зберігання та доступу до даних. Дані декомпозуються в набір атомарних елементів, які розташовуються у відповідних стовпцях однієї або декількох таблиць. В якості прикладу можна

розібрати запис інформації про реєстрацію користувача. Візьмемо лише основну (базову) інформацію про користувача: логін, email, пароль та дату реєстрації. Дані будуть збережені у три таблиці:

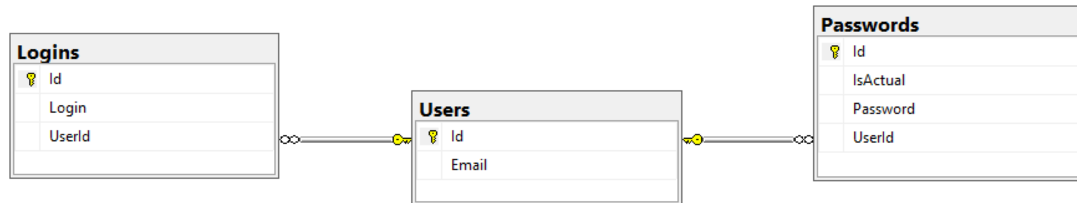


Рис. 2.2. – Схема таблиць особистих даних користувача

Чому саме 3 таблиці, а не одна? Уся суть у бізнес правилах до роботи з даними. Одними з основних правил є можливість користувача змінювати свій пароль та логін. Основним правилом безпечного пароля є його неповторюванність. Для досягнення цієї мети потрібно зберігати старі паролі користувача, тому під паролі виділена окрема таблиця, яка зберігає всі паролі користувачів. Також, ця таблиця має поле «IsActual», яке дозволяє дізнатись активний пароль на даний момент часу кожного користувача. Також, ми не маємо дозволяти користувачам видавати себе за інших користувачів. Для цього створюємо окрему таблицю «Logins», яка буде зберігати всі логіни кожного користувача. Таблиці зв'язані між собою, за допомогою поля «Id» таблиці «Users». Тим самим з будь-якої таблиці ми можемо перейти до інформації потрібної таблиці. Потрібно лише знати ідентифікатор зв'язку.

Для взаємодії з даними таблиць потрібно знати мову запитів SQL (Structured Query Language). Кожна СУБД (Система Управління Базами Даних) має свій діалект мови запитів. Проте, зазвичай вони досить схожі і для досвідчених розробників не є проблемою досить швидко переключатись між діалектами.

Як ми бачимо, робота з даними в реліаційних базах даних доволі проста і потребує лише знання мови SQL, логічного мислення та предметної області, дані якої будуть збережені. Проблеми починають виникати при взаємодії з додатком. Додатки

представлені у вигляді об’єктів, які взаємодіють один з одним. Тобто, розробники додатків звикли взаємодіяти з об’єктами, а не з таблицями, рядками та стовпчиками. Розробникам доводиться створювати додатки, які обробляють дані в об’єктно-орієнтованому стилі, а зберігають інформацію у реліаційних базах даних. Через це потрібно постіно перетворювати два стилі роботи з даними між собою. Це змушує розробників тратити свій час на написання трансляторів від об’єкту до табличних даних та навпаки. Також, сюди додаются обмеження по роботі з об’єктами та реліаційними даними, які часто конфліктують один з одним.

Саме вишевказані проблеми вирішує технологія ORM. Роботу по трансляції об’єктів у реліаційні дані та навпаки бере на себе ORM, а розробник працює з реліаційними даними через звичні йому об’єкти.

2.3. Типи ORM систем

ORM системи поділяються на 2 основні різновидності [4], які являють собою реалізацію відомих шаблонів проектування: «Active Record Pattern» та «Data Mapper Pattern». Кожен з цих шаблонів проектування має суттєвий недолік, він зав'язує програмне забезпечення на його подальше використання. Це означає, що якщо ви захочете його позбутись – доведеться переписати програмний код вашого додатку. Щоб вийти з цієї ситуації, потрібно скористатись шаблоном проектування «DAO». Кожна конкретна реалізація технології ORM надає один із вказаних шаблонів проектування або одразу усі. Винятком є лише шаблон проектування DAO, він являє собою лише абстракцію по роботі з ORM системою.

Active Record Pattern [5] (Шаблон активного запису) – представляє собою один із шарів коду, об’єкти якого описують таблиці бази даних. Такі об’єкти принято називати сутності. Кожна сутність має набір властивостей, які мають свій власний тип даних та встановлені обмеження. З переваг можна виділити повне налаштування, створення та редагування структури бази данних та її таблиць з коду. Тобто,

розробник лише за допомогою звичних йому об’єктів взаємодіє з базою даних та її інформацією. Серед недоліків можна виділити повну залежність програмного забезпечення від кодової бази сутностей та конфігурацій бази даних. Це може затруднити процес видалення бази даних або її переносу.

Data Mapper Pattern [6] (Шаблон відображення даних) – представляє собою один із шарів програмного забезпечення, який займається конвертацією об’єктів у реліаційні дані та навпаки. Тобто, властивості сутностей програмного забезпечення при передачі інформації у базу данних конвертуються за допомогою маперів у реліаційні дані. При отриманні реліаційних данних з бази даних проходить зворотній процес – дані конвертуються за допомогою маперів у властивості сутностей. З переваг можна виділити простоту зміни бази даних з використанням тієї ж самої логіки. Серед недоліків можна виділити повну залежність вашого додатку від шару маперів, що може створити складності переходу від однієї ORM технології (яка підтримує мапери) до іншої. Також, кожна ORM технологія реалізує шар маперів своїм шляхом, що заставить розробників написати нові мапери для роботи з новою ORM технологією.

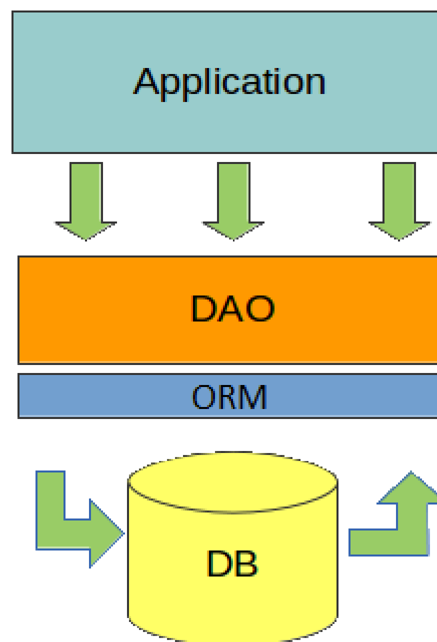


Рис. 2.3. – Приклад програмного забезпечення з шаблоном DAO

Також, варто згадати шаблон проектування DAO [7], або Data Access Object (Шаблон доступу до даних) – не являє собою шаблон проектування, який використовується для реалізації ORM технології. Рекомендований для використання над будь-якою ORM у вашому програмного забезпечені. Він представляє собою додатковий шар абстракції, який має свій набір DTO [8] (Data transfer object), репозиторіїв, конфігураторів баз даних та фабрик. Він інтегрується з ORM технологією одним із више вказаних способів (реалізація шаблону проектування), а саме програмне забезпечення, яке використовує доступ до даних, про цю інтеграцію нічого не знає. Тому, при зміні ORM технології, або шаблону проектування при роботі з нею не доведеться переписувати програмний код всього програмного забезпечення, а лише шар абстракції, який реалізує шаблон проектування DAO. Можна сказати, що цей шаблон проектування являє собою лише оболонку, яка лягає на конкретну реалізацію. Цей підхід використовують зазвичай у складному та великому програмному забезпечені, де зміна ORM технології призведе до глобальної модифікації кода та залучення великої кількості ресурсів у вигляді розробників програмного забезпечення. Приклад роботи шаблону зображено на рисунку 2.3.

2.4 Переваги та недоліки ORM систем

ORM технологія є досить популярною серед розробників програмного забезпечення з використанням ООП через мінімізацію обсягу знань SQL, що необхідні для взаємодії програмного забезпечення з базою даних. ORM технологія автоматично генерує код SQL, що дозволяє розробникам програмного забезпечення зосередитися на генерації бізнес-логіки. Серед основних переваг [9] можна виділити: продуктивність, управління базою даних на рівні програмного забезпечення, повторне використання коду та просте тестування.

Продуктивність – вирішує проблему написання коду для доступу до даних. Написання такого коду займає досить багато часу та не додає майже ніякої великої

цінності до функціональності програмного забезпечення. Робота з даними це так називаємий «must have». Використання ORM технології дозволяє замінити ручне написання коду для доступу до даних на автоматичну генерацію кода, що заощаджує величезний час на розробку. Проте так само не додає великої цінності.

Управління базою даних на рівні програмного забезпечення – дозволяє заздалегідь не продумувати дизайн та структуру бази даних. Тому що, ви можете легко змінити існуючу структуру. Також, це стосується підключення вже існуючої бази даних до програмного забезпечення. Не потрібно її підготовлювати до підключення, всі модифікації можна буде зробити через програмний код. Використання ORM дозволяє використовувати лише потрібні вам сутності (таблиці) в вашому програмному забезпеченні, відкидуючі всі інші.

Повторне використання коду – вам не потрібно постійно писати новий програмний код, щоб отримувати доступ до кожної таблиці окремо. Використовуючи стандартні інтерфейси ORM технології ви можете створити вашу власну бібліотеку для взаємодії з базою даних та повторно її використовувати у ваших інших проектах.

Спрощене тестування – через те, що більшість програмного коду згенерований за допомогою технології ORM був перевірений компанією розробником, вам не потрібно витрачати ресурси та час на його тестуванні. Все, що створюєте ви – це об’єкти сутностей та об’єкти для налаштування бази даних та під’єднання до неї. Тим самим розробники програмного забезпечення можуть зосередитися на тестувані бізнес-логіки та програмного коду.

Наскільки б чудовим не була технологія ORM, вона як і будь-яка система має власні недоліки [9]. Зачасту це пов’язано із складністю програмного забезпечення. Якщо програмне забезпечення просте, то наявність високого рівня абстракції допомогає при розробці. Проте, для складного програмного забезпечення приховання деталей реалізації за великими шарами абстракції ускладнює вирішення проблем для

кожної конкретної задачі. Серед основних недоліків ми можемо виділити: продуктивність, необхідність знання SQL та поганий мапінг.

Продуктивність – найпошириніша проблема кожної ORM технології це створюваний нею додатковий програмний код. Він уповільнює продуктивність програмного забезпечення через необхідність його виконання перед доступом до даних.

Необхідність знання SQL – ORM системи не завжди генерують «найкращі» SQL запити до бази даних. ORM системи досить гарно генерують SQL запити на простий доступ до даних, сортування, групування даних. Проте, при більш складних запитах, розробники програмного забезпечення повинні відстежувати які SQL запити генерує ORM система.

Поганий мапінг – іноді ORM системи можуть неправильно зіставляти сутності об'єктного шару із реліаційними даними з бази даних. Розробникам програмного забезпечення доводиться конфігурувати такі вузькі місця за допомогою написання програмного коду, який пояснює ORM системі як потрібно зіставляти ті чи інші властивості сутності із полями таблиць бази даних.

2.5. Технологія Entity Framework Core

Entity Framework (EF) Core [10] — це легка, розширювана, кросплатформна версія популярної технології доступу до даних Entity Framework. Вона дозволяє розробникам програмного забезпечення на платформі .NET працювати з базою даних за допомогою об'єктів .NET. Усуває потребу в написані руками більшості програмного коду доступу до даних. Entity Framework Core підтримує багато механізмів баз даних, може процювати із різними постачальниками даних.

Своему створенню Entity Framework Core завдячує своїй платформо-залежній версії Entity Framework. Свою історію технологія Entity Framework починає з 2008 року, коли з редакцією Visual Studio 2008 був включений пакет оновлень для .NET

Framework 3.5. До нього було внесено бібліотеки для використання Entity Framework v1.0. Протягом декількох років технологія активно розвивалась та в 2013 досягла свого піку. Саме в цей час популярності набуває відкритий вихідний програмний код та кросплатформеність бібліотек для можливості використання під різними операційними системами. Entity Framework Core була написана з нуля, на зовсім новій кодовій базі з можливістю підтримки не тільки реліаційних баз даних, а й NoSQL. Вона отримала можливість взаємодіяти з програмним забезпеченням написаному на платформі .NET Core, які можуть розгортуватися під операційними системами: Windows, Linux, macOS. Великим кромом для розвитку технології був повний «open-source». Всю кодову базу можна переглянути у відкритому репозиторію на GitHub від компанії Microsoft. Таким чином, технологія Entity Framework Core завдяки розробникам програмного забезпечення із усього світу вдосконаляє свої механізмі та алгоритми. Також, кожен розробник може ознайомитись з внутрішньою інфраструктурою та модифікувати її під себе.

2.6. Інфраструктура Entity Framework Core

Внутрішня інфраструткура технології Entity Framework Core складається з великого фасаду та високого рівня абстракцій. Кожен елемент системи являє собою інтерфейс, тобто контракт. Це дозволяє розробнику програмного забезпечення реалізувати частину функціоналу потрібним йому способом.

Основним елементом технології Entity Framework Core є модель. Модель – представляє собою набір об’єктів. По-перше, це об’єкти сутностей, які представляють таблиці бази даних. По-друге, це контекст взаємодії за базою даних, який реалізує два шаблони проектування: «Repository» та «UnitOfWork». Зазвичай, прийнято надавати контексту суфікс «DbContext». Цей суфікс розшифровується як контекст для роботи з базою даних.

Об'єкти сутностей по синтаксису представлені звичайними класами з наборами властивостей, які описують поля таблиць чи колекцій. Кожна властивість позначається зазвичай одним із простих типів даних (число, строка, дата та інші). Також, можливо використовувати комплексні типи даних, які є іншими об'єктами. Проте, з такими типами даних модель не зможе працювати по замовчуванню. Потрібно буде зробити ручне конфігурування за допомогою будівників моделі або спеціальних атрибутів «DataAnnotations».

Контекст взаємодії з базою даних (об'єкти «DbContext») представляє керуючий об'єкт для роботи з базою даних, який реалізує два шаблони проектування. Цим він збирає в себе керуванням доступом усіма сутностями та управлінням сесіями. Також, до контексту взаємодії з базою даних було долучено фасад над базою даних, який керує порційним виконанням SQL запитів. Під цим розуміються транзакції – неперервна одиниця виконання запитів.

Шаблон «Repository» [11] – представляє собою одиницю доступу до конкретних даних. Завдяки репозиторіям ми отримуємо доступ до даних таблиці. Для кожної таблиці може бути створений як свій репозиторій, так і використовуватись стандартний. Стандартний репозиторій надає базовий «CRUD» механізм доступу до даних. Це такі операції як читання даних, їх оновлення, видалення та додавання до таблиць. Свій персональний репозиторій має такий же «CRUD» механізм доступу до даних та додатковий функціонал, який потрібний лише у випадку роботи з певним доменом даних.

Шаблон «UnitOfWork» [12] – представляє собою «з'єднання» з базою даних або іншими словами сесію. Тобто, при створенні об'єкту «UnitOfWork» відкривається сесія для взаємодії з базою даних і всі операції по роботі з даними будуть проводитися в контексті однієї сесії. Основною операцією є фіксація змін у базі даних. Великою особливістю ORM системи Entity Framework Core є відстеження змін з даними. Тобто, коли розробник програмного забезпечення працює з даними: змінює якісь значення

властивості, додає нові дані до бази або їх видаляє. Це не означає, що ці зміни в ту ж мить будуть застосовані до бази даних. Вони лише фіксуються системою відстеження змін технології Entity Framework Core. Лише після використання операції фіксації змін, вони будуть застосовані до бази даних. Це є оптимальним рішенням, тому що надає розробнику програмного забезпечення можливість контролювати в який саме час необхідно зробити запити до бази даних. Ця операція в моделі називається «SaveChanges». Завдяки такому контролю, при кожному зчитуванню компілятором операцій, написаних за допомогою Linq, запит не виконується до фіксації. Основна проблема, яка вирішується – це можливість не створювати десятки SQL запитів замість одного.

Система відстеження змін фіксує будь-які дії над сутностями бази даних. Кожна операція зміни, додавання чи видалення даних записується до системи. Це надає можливість логувати будь-які зміни даних. Тобто, будь-яка зміна даних не буде не поміченою – можна дізнатись який користувач зробить ці зміни, щоб в подальшому на них відреагувати. Система відстежування є абстрактною, що дозволює змінити реалізацію та додати власний програмний код розробником програмного забезпечення для реалізації власних цілей.

Фасад бази даних дозволяє підтримувати транзакції на рівні програмного коду розробником програмного забезпечення. Транзакції [13] – це неперервне виконання SQL запитів. Зазвичай всі прості операції, такі як оновлення, вставка та видалення даних є атомарними, тобто неперервними. Це означає, що підтримка транзакцій при виконанні вище вказаних операцій підтримується відразу на початковому рівні виконання. Проте, розробники програмного забезпечення можуть виділяти декілька роздільних SQL запитів як одну атомарну операцію. Де збій в одному з SQL запитів повинен приводити до відміни виконання всіх інших та поверненю даних до початкового вигляду або їх видалення, якщо до моменту виконання їх не існувало. За допомогою фасаду бази даних розробник програмного забезпечення може

відтворювати початок транзакції та в потрібному йому моменті зробити фіксацію (Commit) транзакції, що застосує зміни до бази даних та зафіксує транзакцію. При провалі виконання – можна зробити відкат (Rollback) виконаних дій до початку виконання транзакцій або до створеної раніше точки збереження.

2.7. Взаємодія розробників програмного забезпечення з ORM технологією Entity Framework Core

Розробники програмного забезпечення для взаємодії з ORM системами на платформі .NET Core використовуються Language Integrated Query [14] (LINQ). Технологія LINQ – це об’єктно-орієнтована версія мови запитів SQL, яка оброблює набори даних. В платформі .NET Core є декілька різновидів мови LINQ:

- Linq to objects – мова для виконання запитів до колекцій чи масивів у пам’яті.
- Linq to XML – мова для виконання запитів до тегів XML дерева.
- Linq to Html – мова для виконання запитів до тегів Html дерева.
- Linq to Sql – мова для виконання прямих SQL запитів в базі даних.
- Linq to Entities – мова для створення дерева запитів на стороні додатку та подальше їх виконання на стороні SQL серверу.
- Parallel Linq – мова для виконання запитів до колекцій чи масивів у пам’яті з розпаралелуванням обробки даних згідно апаратних потужностей пристрою на якому виконується запит.
- Async Linq – мова для виконання запитів до колекцій чи масивів у пам’яті з неблокуючим очікуванням виконання обробки даних. Підходить для ситуацій з асинхроним введененням та виведенням даних. Загалом, під виконання чистого асинхронного програмування підпадає ввод-вивод.

Для роботи технології Entity Framework Core використувується діалект мови Linq to Entities [15]. Вона складається з набору статичних методів, які реалізують технологію розширюваних методів в платформі .NET Core. Кожен статичний метод

LINQ представляє собою один із операторів SQL або скалярну чи агрегатну функцію, яку можна виконати над набором даних у базі даних. Щоб використовувати Linq to Entities необхідно встановити декілька бібліотек. Звичайна версія цієї мови дозволяє використовувати стандартні оператори. Проте, якщо необхідно використовувати додатковий функціонал операторів – потрібно встановити NuGet пакет «Microsoft.EntityFrameworkCore». Він дозволяє додати розширені версії операторів, які не використовуються для роботи інших різновидів Linq. У запиті, створюємим Linq to Entities, точно вказується, дані якої сутності треба отримати із бази даних. У запиті можна також вказати, як слід сортувати, групувати та формувати інформацію, що повертається. У Linq запит зберігається у змінній. Ця змінна запит не виконує, вона лише зберігає інформацію про запит. Після створення запиту його потрібно виконати, щоб отримати дані.

Таблиця 3.1. Набір основних операторів SQL.

Статичний метод Linq	**Оператор SQL**	**Дія**
Select	SELECT	Проводить вибірку даних з джерела даних.
Where	WHERE	Фільтрує результуючий набір даних за вказаним предикатом.
Order by	ORDER BY	Сортує дані за вказаними атрибутами.
Group by	GROUP BY	Групує дані за вказаними атрибутами.

Основні оператори SQL дозволяють розробникам програмного забезпечення проводити вибірку даних з таблиць з корегуванням вибірки даних за допомогою

предикатів які фільтрують результуючий набір. При фільтрації можна використовувати будь-які логічні операції, що доступні в ядрі SQL серверу. При необхідності можна відсортувати дані за одним чи декількома атрибутами або їх згрупувати, якщо була використана хоча б одна агрегована функція.

Таблиця 3.2. Набір операторів з'єднання даних з декількох таблиць.

Статичний метод Linq	**Оператор SQL**	**Дія**
Join	INNER JOIN	Перекресний пошук даних з двох таблиць по рівністю двох ідентифікаторів таблиць.
Include	LEFT OUTER JOIN	Перекрестний пошук даних з двох таблиць по рівністю двох ідентифікаторів таблиць. Якщо порції даних з лівої таблиці не буде порції з правої, то для вибірки правої буде встановлено значення NULL.

Оператори з'єднання допомагають розробникам програмного забезпечення виконувати вибірку даних більше ніж з однієї таблиці бази даних. Статичний метод Include окрім використання «під капотом» SQL оператора LEFT OUTER JOIN виконує також «жадібне завантаження». Це означає автоматичне завантаження вказаних властивостей за допомогою SQL оператора LEFT OUTER JOIN. Потрібно бути обережним цією операцію, так як ви можете затягнути в оперативну пам'ять занадто

багато даних, що може перевантажити систему. Особливо, якщо ця операція може виконуватись паралельно для декількох користувачів. Завантаження проводиться відразу, як проходить вибірка сутностей. Щоб уникнути завантаження всього набору даних можна використовувати пряме завантаження або «ліниве завантаження. Воно завантажує дані властивості лише при необхідності.

Таблиця 3.3. Набір агрегатних функцій

Статичний метод Linq	**Агрегатна функція SQL**	**Дія**
Sum	Sum	Сумує значення вказаного атрибуту.
Min	Min	Мінімальне значення для вказаного атрибуту.
Max	Max	Максимальне значення для вказаного атрибуту.
Count	Count	Підраховує кількість даних за вказаним атрибутом.
Avarage	AVG	Середнє арефметичне вказаного атрибуту.

Агрегатні функції мови запитів SQL в платформі .NET Core представлені звичайними статичними методами, які виконують на стороні серверу агрегатну функцію. Функції Min та Max можуть виконуватися над будь-яким типом даних, маючи свої нюансі. Наприклад: виконючи Min та Max функції над числами 1,2,3,5 та 10 ми отримаємо очевидний результат. Min поверне значення 1, а Max 10. Проте, якщо ми будемо працювати із символьними рядками, то там буде визначатися порядковий номер кожного символу в таблицях кодування. При функції Min випаде те слово, яке

має найменші символи, при функції Max навпаки, найбільше. Функції Sum та Avg виконуються лише над числами та датами. Функція Count виконуюється над будь-яким типом даних. Дана функція повинна лише повернути кількість вказаних атрибутів. При виконанні агрегатних функцій необхідно групувати повторюємі дані. Адже, агрегатна функція повертає єдине значення, яке автоматично ядро SQL серверу не може з'єднати з табличними даними.

Таблиця 3.4 Набір Linq методів для отримання результату виконання.

Статичний метод Linq	**Дія**
First	Витягує перший елемент результуючого набору. Якщо елементів немає – буде помилка.
FirstOrDefault	Витягує перший елемент результуючого набору. Якщо елементів немає – буде повернуто значення NULL.
Single	Витягує єдиний елемент результуючого набору. Якщо елементів немає або їх більше за один – буде помилка.
SingleOrDefault	Витягує єдиний елемент результуючого набору. Якщо елементів немає або їх більше за один – буде повернуто значення NULL.
ForEach	Циклічний перебір результуючих даних.
ToList, ToArray, ToDictonary….	Витягування даних та їх розміщення в колекціяї, масивах, словниках та інше.

Також, в платформі .NET Core мова запитів Linq має також оператори, які не мають аналогів у мові запитів SQL. Насамперед, це зв'язано виконанням Linq запитів. Мова Linq виконується у відкладеному стані. Коли ви створите Linq запит – це не означає, що він виконається в цю ж секунду. В цей момент будується дерево запитів з використаних статичних методів. Для виконання побудованого дерева запитів Linq потрібно здійснити ініціацію доступу до результатів запиту. Це можна зробити одним із способів вказаних в таблиці вище, в залежності від мети обробки елементів.

Інтерфейсом (контрактом) для її роботи являє собою інтерфейс IQueryable. За цим інтерфейсом ховається реалізація побудовниками дерева виразів. Мова запитів SQL – це і є дерево виразів, де запит складається з частин, які виконуються один за одним. У платформі .NET Core за побудовання дерева виразів відповідає клас Expression. Він може під час виконання будувати методи, їх тіло, вхідні параметри та результат виконання. Прямо під час виконання програмного забезпечення вираз компілюється та виконується на запит програмного коду.

Саме завдяки наявності дерева виразів для побудування Linq запиту – платформа .NET Core та її основна мова програмування C# пропонує розробникам 3 варіанта написання Linq запитів.

```
// static style:
var res1 = Enumerable.Select(
    Enumerable.Where(employees, x => x.Age > 18), x => new { FullName = $"{x.Name} {x.Surname}" });

// extension style:
var res2 = employees
    .Where(x => x.Age > 18)
    .Select(x => new { FullName = $"{x.Name} {x.Surname}" });

// query style:
var res3 = from emp in employees
           where emp.Age > 18
           select new { FullName = $"{emp.Name} {emp.Surname}" };
```

Рис. 3.1. – Стилі написання Linq запитів

Розробник програмного забезпечення, може використовувати виклики статичних методів, методів розширення або синтаксичний стиль написання Linq запитів. Насправді, існує лише один варіант – це виклики статичних методів. Проте, для розробника на етапі написання програмного коду доступні всі 3 варіанти. Синтаксичний стиль написання Linq запитів після компіляції програмного коду перетворюється на виклики методів розширення. Які в свою чергу на низькому рівні перетворюються на виклики статичних методів. Вибір стилю написання Linq запитів ніяк не впливає на продуктивність програмного забезпечення. Тому, вибір конкретного варіанту повністю лежить на розробникові програмного забезпечення або на правилах написання програмного коду в компанії.

2.8. SQL Dapper

Dapper [16] – це «полегшена» версія ORM технології для платформи .NET Core. Основна її задача збільшити продуктивність при роботі з даними. Dapper забезпечує відображення об’єктно-реліаційної моделі, домену (сутностей) та реліаційну базу даних. Це ніби гібрид звичайної ORM технології, як Entity Framework Core та прямих SQL запитів.

Dapper дозволяє звільнити розробника програмного забезпечення від написання більшості програмного коду. Сюди входить:

1. Відкриття з’єднання з базою даних. Мається на увазі, шо увесь програмний код вже написаний розробниками бібліотеки Dapper. Його потрібно лише використати викликом одного програмного методу.
2. Готові об’єкти SQL параметрів, які будуть необхідні для передачі з SQL запитом. Дані об’єкти мають властивості для встановлення їм імен, значень та конкретного типу даних. Можливе налаштування об’єктів складними типами даних – такі як колекції, об’єкти, перерахування та інше.

3. Методи для виконання SQL запитів. Проте, сам SQL запит розробник програмного забезпечення повинен написати власними руками.
4. Влаштованні мапери даних. Методи для виконання SQL запиту проекції мають вказівник заповнювача типу (Generic type). Він дозволяє вказати тип сутності в яку необхідно конвертувати отриманні дані від операції проекції. За допомогою технології рефлексії платформи .NET Core бібліотека зчитує назви та типи даних властивостей сутності та шукає збіги із даними, отриманими від операції проекції.
5. Відсутність менеджеру для відслідковування змін. У розробника програмного забезпечення не має необхідності в постійній фіксації змін над сутностями.

2.9. Переваги та недоліки Dapper

При роботі з Dapper можна виділити як переваги, так і недоліки. Насамперед, слід розуміти, що вся робота Dapper будується на основі влаштованого в .NET Core платформу інтерфейсу (контракту) IDbConnection. Цю особливість можна вважати як перевагою, так і недоліком. Якщо подивитися зі сторони переваги, то наявність реалізації інтерфейсу IDbConntection дозволяє поеднувати роботу системи Dapper як з іншими ORM технологіями, які реалізують даний інтерфейс та с платформою .NET Core в цілому. Зі сторони недоліку, реалізація даного інтерфейсу робить залежним ваш програмний код від ORM технологій, які його реалізують. Більшість ORM технологій, які існують у екосистемі платформи .NET Core реалізують даний інтерфейс але не всі. Проте, в даній роботі розглядається ORM система Entity Framework Core, вона не реалізує даний інтерфейс взаємодії об’єктів. Тому, для поєднання роботи цих двох технологій доведеться скористатися патерном проектування «Адаптер».

Основними перевагами технології Dapper є:

1. Швидкість та висока продуктивність. Так, як у Dapper відсутні будь-які об'єктні модулі для аналізу LINQ запитів та генерації на основі аналізу SQL запитів, а розробник повинен писати сам SQL запит власними руками.
2. Можливість статичної або динамічної прив'язки об'єкту. При виконанні операції проекції є можливість як вказати об'єкт сутності, який буде заповнений даними при вибірці. Або, можна вказати універсальний стандартий тип даних dynamic. Він дозволить системі Dapper використати об'єкт ExpandoObject, який динамічно створює властивості для об'єкта. Використання типу dynamic доволі спірний момент, тому що втрачається основна потужність мови C# - строга типізація. Також, Dapper дозволяє використовувати анонимні типи для генерації набору властивостей, які можна заповними необхідними даними з таблиць.
3. Підтримка кількох запитів. Система Dapper дозволяє вам за раз виконувати не один пакет SQL запитів, а стільки, скільки вкаже розробник програмного забезпечення при написанні запиту.
4. Підтримка та обробка збережених процедур/функцій. Система Dapper надає можливість використовувати збережені процедури та функції із мапінгом результатом процедур та функцій у сутності.
5. Підтримує можливість масової вставки даних. Система Dapper, на відміну від інших ORM систем надає можливість вставити великий об'єм даних єдиним пакетом або пакетами.

Основні недоліки технології Dapper є:

1. Відсутність підтримки мови запитів LINQ для написання запитів SQL в об'єктному стилі, більш зручному для розробника програмного забезпечення.
2. Відсутність загальних об'єктів та моделей для конфігурування мапінгу отриманих даних від операції проекції. Це змушує розробника програмного забезпечення надавати імена та типи даних властивостей, які повністю співпадають із схемою таблиць бази даних. Або, розробнику програмного

забезпечення доведеться використовувати конфігурування Dapper, яке не має загального інтерфейсу та не використовує інтерфейси платформи .NET Core.

3. Потреба в написанні SQL запитів руками. Розробнику програмного забезпечення потрібно досить гарно знати SQL та його процедурне розширення для написання запитів. Також, це унеможливлює глобальні зміні в усіх SQL запитах одночасно. Потрібно переписувати кожен запит окремо.
4. Відсутність вбудованих адаптерів для SQL запитів. Кожен SQL запит через Dapper необхідно створювати таким чином, щоб результат його роботи можна було отримати в об'єктно-орієнтованому вигляді.

2.10. Порівняння і проблеми SQL Dapper та Entity Framework Core.

Для написання програмного коду для роботи з даними найбільш ефективним є технологія Entity Framework Core. Вона дозволяє розробнику програмного забезпечення забути на 95% про реалізація шару програмного забезпечення, який відповідає за роботу з даними. Проте, об'єктно-орієнтована складова технології Entity Framework Core завдяки побудування всіх запитів на основі об'єктних побудовників SQL запиту набагато повільніший та витрачає більше ресурсів за SQL Dapper. Останній, в свою чергу не використовує ніяких побудовників. Таким чином, в пам'яті не створюється об'єктів, які займають ресурси. Їх непотрібно відслідковувати, їх непотрібно видаляти з пам'яті після використання. Це значно пришвидшує роботу програмного забезпечення. SQL запит надає сам розробник програмного забезпечення, тому його не потрібно будувати. Таким чином, швидкість виконання запиту повністю залежить від розробника. Якщо запит складений без помилок та найбільш ефективним способом, його виконання буде занадто швидше за запит який створить ORM технологія Entity Framework Core.

ORM технології SQL Dapper та Entity Framework Core мають різні інтерфейси для конфігурування сутностей для роботи з даними. Якщо програмне забезпечення

буде використовувати одразу обидві ORM системи, то йому доведеться мати два набори об’єктів для конфігорування сутностей. Це збільшує об’єм роботи, який доведеться виконувати розробнику програмного забезпечення при зміні схеми таблиць. Тому що доведеться провидити зміни одразу в двох місцях. По-перше, це об’єкти для конфігурування сутностей Entity Framework Core. По-друге, це об’єкти для конфігурування сутностей Dapper. Цю проблему можна вирішити – написавши адаптери для об’єктів конфігурування. Щоб зміни необхідно було проводити лише в одному об’єкті. Проте, це буде додатковою роботою розробника програмного забезпечення. Який повинен зосередитися на розробці бізнес-логіки замість шару даних.

Entity Framework Core не вміє ефективно генерувати основні операції DML: вибірка, оновлення, додавання та видалення даних. Це стосується лише ситуацій при роботі з великою кількістю даних. Навіть при роботі з найпростішою операцією вибірки у Entity Framework Core настають проблеми у продуктивності.

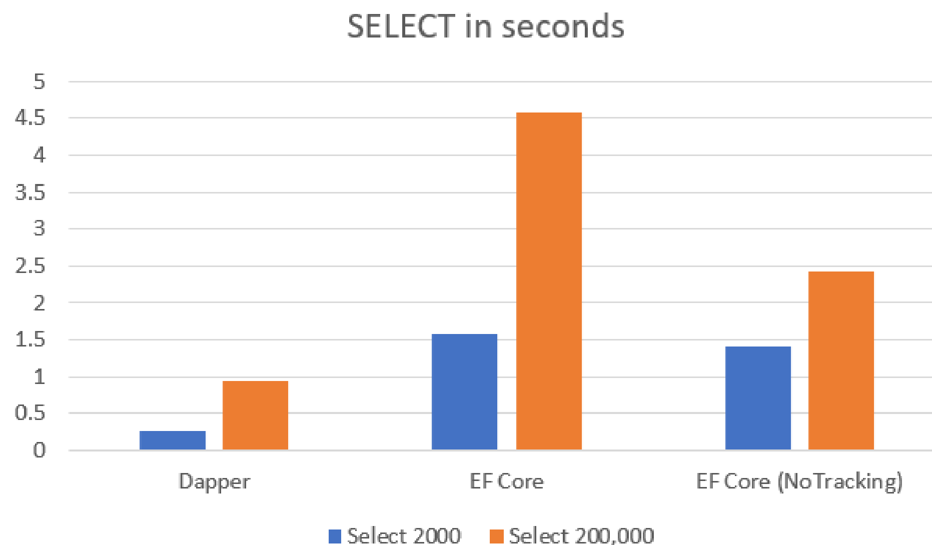


Рис 4.1. – Продуктивність вибірки SQL Dapper VS Entity Framework Core[17]

З рисунку 4.1 видно, що навіть при вимкнутому відстежуванні змін над об’єктом, технологія SQL Dapper буде більш оптимальним за часом виконання понад чим 80%. Чим більша кількість елементів сутностей для обробки, тим більше програє Entity Framework Core. Це настає через велику об’єктно-орієнтовану оболонку усієї ORM системи. Для обробки результатів запитів створюється доволі велика кількість об’єктів, що уповільнює роботу. При цьому, працюють додаткові механізми Entity Framework Core, що також уповільнюють час обробки. До проблеми с продуктивністю додається проблема з виділенням значної кількості пам’яті на обслуговування об’єктів для роботи. Це також надає вплив на кінцевий результатат виконання SQL запиту.

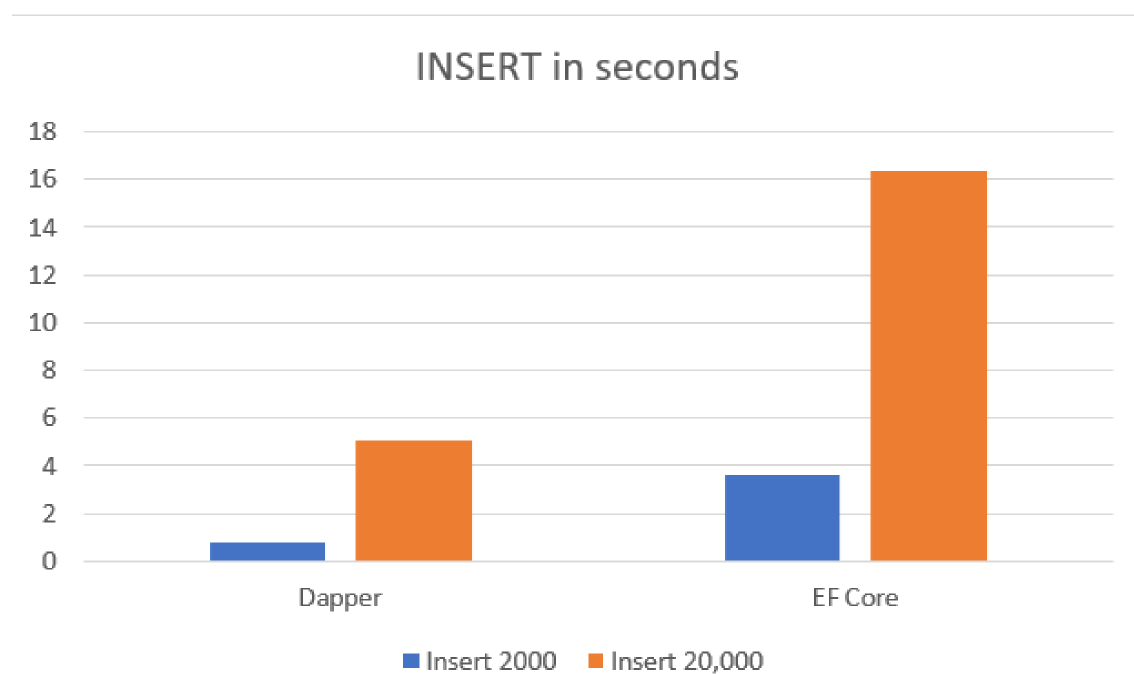


Рис. 4.2. – Продуктивність додавання сутностей до таблиці SQL Dapper VS Entity Framework Core [17]

Для прикладу роботи операцій додавання, оновлення та видалення можна розглянути найважчу операцію – додавання даних. На рисунку 4.2 можна побачити, що продуктивність виміряна у часі SQL Dapper у 4 рази більша, ніж у Entity Framework

Core. Це пов’язано насамперед у неефективних запитах та заповненню великої кількості пам’яті даними. Наприклад, якщо в циклічній конструкції ви почнете додавати до моделі Entity Framework Core – система буде створювати на кожен елемент додавання сутності новий SQL запит. Цей SQL запит та його налаштування будуть збережені у вигляді об’єктів. Через це програмне забезпечення за один раз буде намагитися передати до таблиць бази даних велику порцію роздрібнених команд. Пам’ять програмного забезпечення буде переповнена об’єктами, які потребують відслідковування свого стану та знищення при необхідності. Кожен з цих факторів уповільнює виконання об’єктно-орієнтованих команд. Для корегування цієї проблеми можна створити аналізатор, який буде створювати єдиний SQL запит для певної (n) кількості сутностей, що зменшить кількість роздрібнених SQL запитів у n раз.

2.11. Висновок

Згідно з розглянутою технологією Entity Framework Core можна зробити висновок, що вона надає безліч особливостей та покращень для розробників програмного забезпчення. Насамперед, це простота написання програмного коду для роботи з даними. Більшість програмного коду вже написана, а на плечі розробника випадає написання лише 5% програмного коду. Розробнику не доводиться витрачати на це час, він може його витратити на розробку бізнес-логіки програмного забезпечення. Це і є основним рішенням данної ORM технології.

Окрім простоти можна виділити відсутність в необхідності глибоких знань мови запитів SQL та її процедурних розширень. Розробник програмного забезпечення використовує звичну йому роботу з об’єктами та влаштованою мовою запитів Linq.

ORM технологія Entity Framework Core повністю написана на основі інтерфейсів (контрактів) та підтримує будь-які розширення свого функціоналу, що дозволяє перевизначити необхідний розробникові функціонал.

Entity Framework Core має певні проблеми з продуктивністю виконання запитів та пам'яті. Ця проблема поширюється майже на всі об'єктно-орієнтовані системи.

Проте, за допомогою перевизначення функціоналу можна об'єднати потужності кожної з розглянутих систем. В цій роботі були розглянуті ORM технології Entity Framework Core та SQL Dapper. Перша більш ефективна для розробника програмного забезпечення в швидкості написання програмного коду, його комплектації, повній абстракції та можливості перевизначення та розширення функціоналу. Друга більш ефективна у продуктивності виконання SQL запитів та економії оперативної пам'яті. Необхідно поєднати можливості цих двох систем. За допомогою можливості простого виконання SQL та T-SQL операцій розширити функціонал ORM технології Entity Framework Core з використанням SQL Dapper для перевизначення проблемних операцій. Для оптимізації даних операцій будуть використанні існуючі моделі та алгоритми, які будуть описані в розділі №4.

Внаслідок пристуності в ORM технології Entity Framework Core системи відстеження змін – з'являється можливість повноцінно аналізувати усі проведені операції. Це дозволить об'єднувати розгалуджені запити до пакетів для єдиного виконання. Повна об'єктна структура дозволить проаналізувати тип сутності, які властивості змінюються та що за операція проводиться. Таким чином, можна об'єднувати лише так звані однотипні сутності.

Влаштований побудовник SQL запитів ORM технології Entity Framework Core дозволяє розширити свої можливості для генерації нових запитів з можливістю їх подальшого виконання через SQL Dapper.

3. ВИМОГИ ТА ОЦІНКА ЯКОСТІ РОЗШИРЕНЬ ТЕХНОЛОГІЇ

3.1. Загальний опис

Розширення ORM технології Entity Framework Core складають з себе використання ORM технології SQL Dapper, мови запитів SQL та її процедурного розширення T-SQL. При розробці даної роботи будуть розширені наступні запити:

1. Запит проекції – вилучення даних з таблиць бази даних. Найбільш ефективним рішенням для оптимізації цього процесу є наявність некластерного індексу в атрибутах таблиці, які використовуються для пошуку.
2. Запит оновлення даних – оновлення атрибутів для сутності. Основною проблемою є масове оновлення даних. За допомогою використання SQL Dapper можна пришвидшити цей процес використовуючи оновлення партіями.
3. Запит видалення даних – видалення записів із таблиці. Як і з операціє оновлення даних – основною проблемою є велика кількість даних для видалення. Використовуючи той самий принцип попереднього пункту можна пришвидшити процес.
4. Запит додавання даних – додавання записів до таблиць. Ця операція є найбільш особливою, тому що вона сповільнюється через можливу наявність індексів в атрибутах таблиці. На відміну від попередніх описаних операцій проблеми можуть початися навіть при невеликій кількості даних. Тому, окрім використання SQL Dapper для додавання даних до таблиць партіями – необхідно аналізувати кількість додаваємих даних для розуміння використовування прямого додавання окремими операціми або їх об’єднання в одну.
5. Перехресний пошук даних – використання операторів об’єднання для пошуку даних з двох та більше таблиць. В наявності є теоретичні моделі, які дозволяють пришвидшувати дану операцію внаслідок наявності індексів, операцій

агрегування та сортування даних. Кожен з цих факторів може вплинути на вибір алгоритму створення SQL запиту та на кінцевий результат операції.

Окрім розширення наявних SQL запитів необхідно надати можливість виконання цикличніх та умовних операцій на стороні SQL серверу.

3.2. Вимоги до безпеки

При створенні розширень для ORM технології Entity Framework Core не можна забувати про небезпеку вставок SQL ін'єкцій [18]. Щоб зрозуміти необхідність вимог до безпеки – потрібно розглянути даний тип атаки. SQL ін'єкція (Впровадження SQL коду) – це досить поширений тип атак для зламу веб-сайтів та програмного забезпечення, працюючих з базами даних, заснований на впроваджені в запит додаткового (небезпеченого) SQL коду.

```
Console.WriteLine("Welcome to admin console.");
Console.Write("Enter login: ");
var login = Console.ReadLine();

Console.Write("Enter password: ");
var password = Console.ReadLine();

var adminSearchQuery = "SELECT TOP(1) * FROM Admins WHERE Login = '{0}' AND Password = '{1}'";

var admin = SqlExecutor.Execute<Admin>(string.Format(adminSearchQuery, login, password));
```

Рис. 3.1. – Приклад вразливого до SQL ін'єкції програмного коду

На рисунку 3.1. можна побачити вразливий програмний код до атаки у вигляді SQL ін'єкції. Змінна adminSearchQuery зберігає SQL запит для пошуку сутності адміністратора за уведенем логіном та паролем. Дані атрибути представлені флагами форматування, які в подальшому повинні бути заповнені справжніми значеннями. Ці значення, згідно прикладу, повинен ввести користувач. Якщо він введе логін та пароль все відпрацює як і було заплановано. Тобто, якщо будуть збіги, то користувач буде

авторизований як адміністратор або якщо збігів не буде, то користувач отримає повідомлення, що дані введенні невірно. Проте, є безліч зловмисників-шукачів, які шукають на різних веб-сайтах та програмних забезпеченнях вразливі місця куди можна впровадити SQL ін’єкцію. Не кажучи про автономне програмне забезпечення, яке проводить такі дії автоматично. Заміть звичайного варіанту логіку, зловмнисник введе, щось нашталк наступної команди: «admin’ OR 1=1--». Рядком «admin’», зловмисник додасть значення для параметру логін. Далі він додає оператор OR, що дозволяє виконати другу частину виразу навіть, якщо значення для логіну було невірним. Другою частиною варизу вказано довести рівність між числами 1 та 1. Без сумніву вираз «1=1» надасть в результаті свого виконання значення істини. Залишились два рядки «--». В мові SQL ці два символи підряд означають коментар. Тобто, зловмисник після свого істинного виразу закоментує ввесь подальший запит. Тому навіть не доведеться нічого вводити для параметра пароль. Після SQL ін’єкції запит буде виглядати як вказано на рисунку 3.2. Тобто, таким чином вказавши будь-який логін та пароль за допомогою SQL ін’єкції зловмисник міг авторизуватися в системі з правами адміністратора.

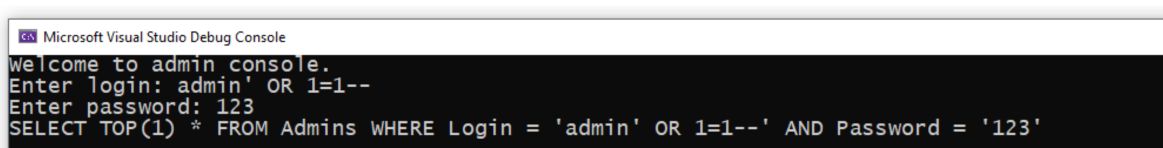


Рис 3.2. – Приклад запиту після враження SQL ін’єкцією

Даний тип атак був досить популярний в час написання SQL запитів розробниками програмного забезпечення руками. Тобто, замість використання об’єктів для побудування запиту, розробники писали запити самостійно у вигляді рядка або в окремому файлі з розширенням «.sql». З приходом ORM технології Entity Framework Core про SQL ін’єкції забули через використання екранування для параметрів. Тобто, параметри тепер не можуть трактуватися як частина SQL коду, а

лише як значення. Екранування використовується під капотом при написанні будь-якого SQL запиту через об’єктно-орієнтований стиль Linq або навіть запиту написаного руками через використання об’єкта параметр.

Через перевизначення стандартних механізмів роботи ORM технології Entity Framework на використання в деяких випадках розширень за допомогою прямих SQL та T-SQL запитів може знову з’явитись потенційна небезпека використання SQL ін’єкцій. Тому потрібно внести наступні вимоги до безпеки:

1. Заборона використання форматування рядків при створенні SQL запитів.
2. Використання об’єктів SqlParameter [19] для впровадження необхідних даних для виконання SQL запиту. Таким чином, буде проводитися екранування кожного параметру, який потрапляє до SQL запиту.
3. Заборона приймати для виконання SQL з параметрами у вигляді рядка.

3.3. Опис архітектури розширень

Архітектура будь-якого програмного забезпечення є одним з найважливішим елементів процесу розробки. Розробкою архітектури досить часто нехтують. Це приводить до непрацездатності системи або її некоректної роботи. Щоб розроблене програмне забезпечення виконувало свою роботу коректно та безперебійно, необхідно продумати архітектуру роботи. Насамперед, необхідно розробити архітектуру аналізаторів створених ORM системою SQL запитів. Після чого переходити до аналітичних моделей генерації запитів, побудовників SQL запитів та механізмів їх виконання. Архітектура розширень буде представлена у вигляді UML діаграм послідовностей. Так як система розроблюється з можливістю розширення свого функціоналу та з підтримкою патернів проектування – на ній в деяких випадках будуть зображенні лише абстракції або інтерфейси (контракти) замість конкретних реалізацій об’єктів. З метою зробити текст на зображеннях видимим для читання у

всіх методів будуть відсутні вхідні параметри та їх типи даних. До виклику кожного методу прикладується опис під зображенням архітектури з детальним описом того, що приймає в якості параметрів метод та який результат повертається.

3.3.1. Архітектура аналізаторів SQL запитів побудованих ORM технологією Entity Framework Core

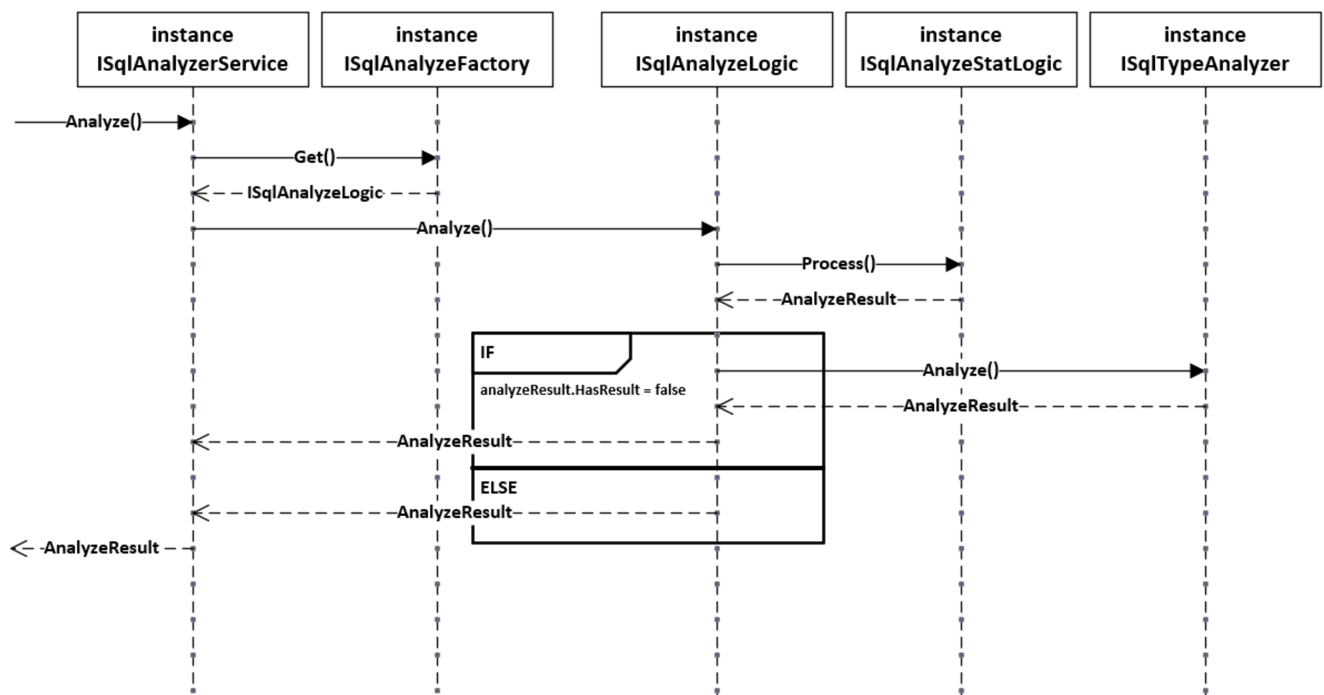


Рис. 3.3. – Діаграма послідовності аналізаторів SQL запитів

На рисунку 3.3 зображена UML діаграма послідовності. На ній можна побачити аналіз SQL запитів за допомогою послідовних викликів методів. При виклику розширюючого методу, який перевизначений – буде викликатись метод «Analyze» інтерфейсу сервісу «ISqlAnalyzerService». Проте, даний метод матиме два параметри. Перший – це SQL запит у вигляді об’єкту. Другий – це тип SQL операції, яка підтримується системою перевизначених методів (проекція, оновлення, видалення, додавання та перехресний пошук данних). Сервіс першим своїм кроком має зробити

виклик методу «Get» фабрики «ISqlAnalyzeFactory». Даному методу потрібно передати в якості параметра тип SQL операції. За допомогою переданого типу SQL операції фабрика із внутрішньої колекції знайде готовий екземпляр об’єкту «ISqlAnalyzeLogic». Результатом виконання фабрики буде повернення знайденого екземпляру об’єкта. Другим кроком, сервіс виконає виклик методу «Analyze» на отриманому екземпляру об’єкту «ISqlAnalyzeLogic». Даний екземпляр представляє собою конкретну реалізацію, яка вміє аналізувати лише вказаний тип SQL операції. Вище вказаному методу необхідно передати в якості параметру SQL запит у вигляді об’єкту. Метод «Analyze» першим кроком проводить виклик методу «Process» на екземплярі об’єкту «ISqlAnalyzeStatLogic». Даний екземпляр повинен витягувати з бази даних статистику по виконанню останніх подібних SQL запитів. Якщо для даного типу SQL операції доступна можливість ведення аналітичної інформації щодо запитів. Для покращення продуктивності взаємодії – при першому викликові екземпляру об’єкту «ISqlAnalyzeStatLogic» остання 1000 записів буде витягнута в пам’ять та розміщення в кешовому сховищі програмного забезпечення. Параметер кількості записів для кешування можна налаштовувати за допомогою впровадження залежностій у програмне забезпечення. Після витягування аналітичних даних – метод «Process» проведе їх обробку. А саме буде шукати повне співпадіння запиту з виключенням унікальних значень параметрів. Якщо не буде співпадінь, то буде повернутий пустий результат з поміткою, що інформації про запит такої форми та набору операторів не існує й необхідно буде додаті аналітичну інформацію до бази даних. Це робиться для досягнення отримання готового SQL запиту в наступний раз для виконання такого ж SQL запиту з такими ж або іншими значеннями для параметрів. Якщо буде співпадіння, то «ISqlAnalyzeStatLogic» зробить запит до бази даних для вилучення готового об’єкту модифікованого SQL запиту, який був створений при попередньому аналізі. Саме цей об’єкт буде повернутий до методу

«Analyze». Далі, поведінку методу можна розділити на дві частини за допомогою умовного оператора «IF-ELSE»:

1. Якщо даний метод отримав пустий результат, то він почне проводити аналіз виконуваного типу операцій. Саме тут будуть викликаний метод «Analyze» інтерфейсу «ISqlTypeAnalyzer», куди буде переданий в якості параметру SQL запит у вигляді об’єкту. Конкретна реалізація «ISqlTypeAnalyzer» підбирається в залежності від вказаного раніше типу SQL операції. Тобто, у кожного «ISqlAnalyzeLogic» власний екземпляр об’єкту «ISqlTypeAnalyzer». Даний метод виконує різні в залежності від вибраного типу SQL операції. Він може використовувати реалізацію як моделей так і алгоритмів, вказаних в розділі №4 «Моделі та алгоритми». Результатом виконання методу «Analyze» інтерфейсу «ISqlTypeAnalyzer» буде об’єкт модифікованого SQL запиту. Перед поверненням результату виконання операції необхідно зберегти його до бази даних. Цей результат буде збережений в якості аналітичної інформації для використання в подальшому без необхідності повторного аналізу. Подібне виконання допоможе викреслити із подальшого використання подібних запитів повний аналіз. Після чого результат буде повернутий через всі виклики методів для його подальшого виконання. Також, можлива ситуація, що згідно аналізу SQL запиту розширення не може його оптимізувати. Тоді буде повернути результат щодо необхідності використовувати стандартний варіант SQL операції ORM системиs Entity Framework Core.
2. Якщо метод отримав готовий результат. А саме об’єкт модифікованого SQL запиту. Цей отриманий об’єкт буде повернутий через всі виклики методів для його подальшого виконання.

3.3.2. Архітектура виконання перевизначених методів проекції та перехрещеного пошуку даних

На рисунку 3.4 зображена UML діаграма послідовності. На ній можна побачити процес виконання перевизначених методів операції проекції та перехресного пошуку даних з декількох таблиць. Через абстрагування реалізацій у вигляді інтерфейсів немає різниці на данній діаграмі яка саме буде операція: проекція чи перехресний пошук даних. Першим етапом є виклик розробником програмного забезпечення одного з вище вказаних методів. Перевизначені методи знаходяться в статичному класі «EfCoreExtensions».

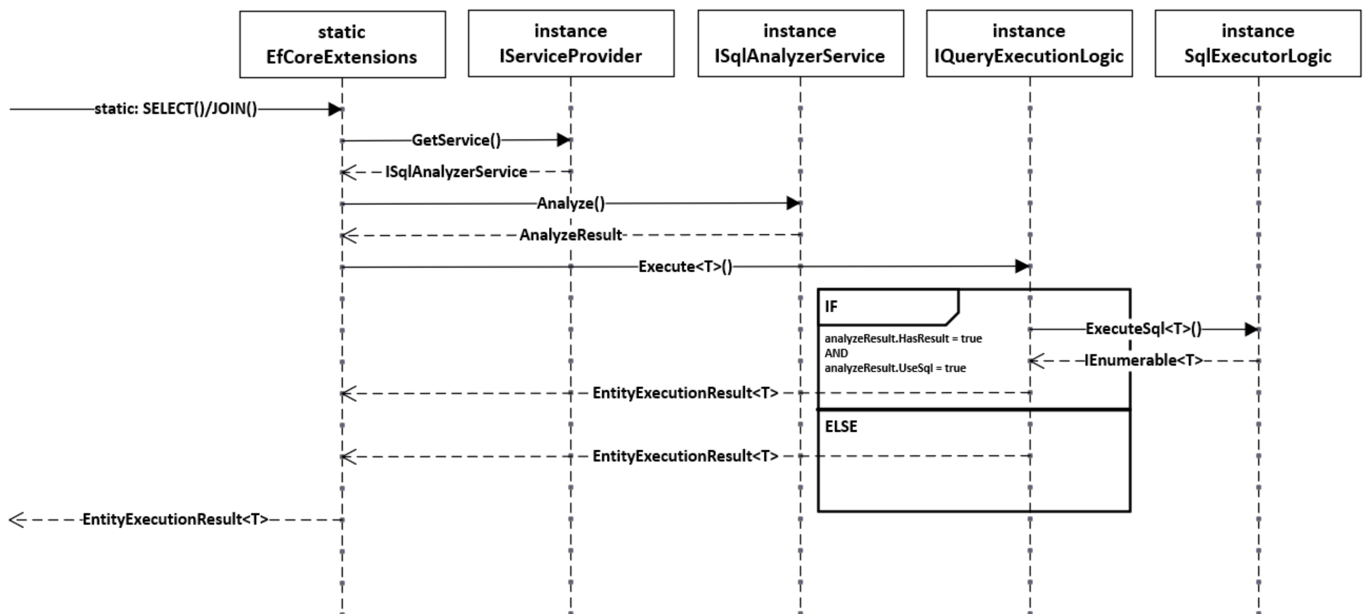


Рис. 3.4. – Діаграма послідовностей виконання перевизначених методів

«Під капотом», перевизначені методи використовуючи метод «GetService» інтерфейсу «IServiceProvider» для отримання екземпляру об’єкту «ISqlAnalyzerService». На отриманому екземплярі буде викликаний метод «Analyze», який поверне результат аналізу SQL запиту написаного розробником програмного забезпечення. В якості параметрів буде переданий об’єкт SQL запиту та тип SQL операції. Перший параметр буде отриманий за допомогою QueryableExtensions. Це

додатковий вкладений приватний статичний об'єкт. Основним його методом є «GetSqlObject», який приймає в якості параметра IQueryable<T> та повертає результат у вигляді екземпляру об'єкту «SqlCommand». Це метод використовуючи механізми рефлексії витягує усі елементи SQL запиту, сформованим розробником програмного забезпечення. Другий параметр буде переданий «хардкодом» в залежності від вибору операції проекції або перехресного пошуку. Архітектура аналізу запитів була описана у попередньому підрозділі.

Далі, після проведеного аналізу необхідно виконати SQL запит. Для цього використовується виклик метод «Execute<T>» інтерфейсу «IQueryExecutionLogic». Даний метод перевіряє отриманий результат від аналітичного сервісу. Для перевірки результату необхідно використати умовний оператор «IF-ELSE», який розділить подальше виконання на два різних виконання:

1. Якщо результат аналізу буде мати значення істини для властивостей «HasResult» та «UseSql», то сформований аналітичним сервісом SQL запит буде виконаний через об'єкт «SqlExecutorLogic». Даний об'єкт використовує «під капотом» технологію SQL Dapper. Робота цієї технології винесена до об'єкту для необхідності її заміни у випадку необхідності. Результат «SqlExecutorLogic» буде повернутий у вигляді колекції, яка складається з екземплярів об'єктів сутностей.
2. Якщо результат аналізу буде мати значення хиби хоча б для однієї властивості з наступних властивостей: «HasResult» та «UseSql». То сформований розробником програмного забезпечення SQL запит за допомогою мови Linq буде запакований у змінну результату у вигляді об'єкту типу IQueryable<T>. Таким чином, повернутий об'єкт зможе виконатись згідно стандартного виконання ORM технологією Entity Framework Core.

Результат виконання отриманий від будь-якого з фасаду по виконанню SQL запиту буде повернутий через усі шари до початкової точки виклику.

3.3.3. Архітектура виконання операції оновлення, видалення та додавання даних при роздільних операціях

На рисунку 3.5. зображено використання стандартного механізму для додавання, оновлення або видалення даних з таблиць. Робота починається з виклику методу «Method» екземпляру об’єкту «UsageClass». Даний метод та екземпляр об’єкту представляє будь-який метод та об’єкт, який буде виконувати операції додавання, видалення та оновлення даних.

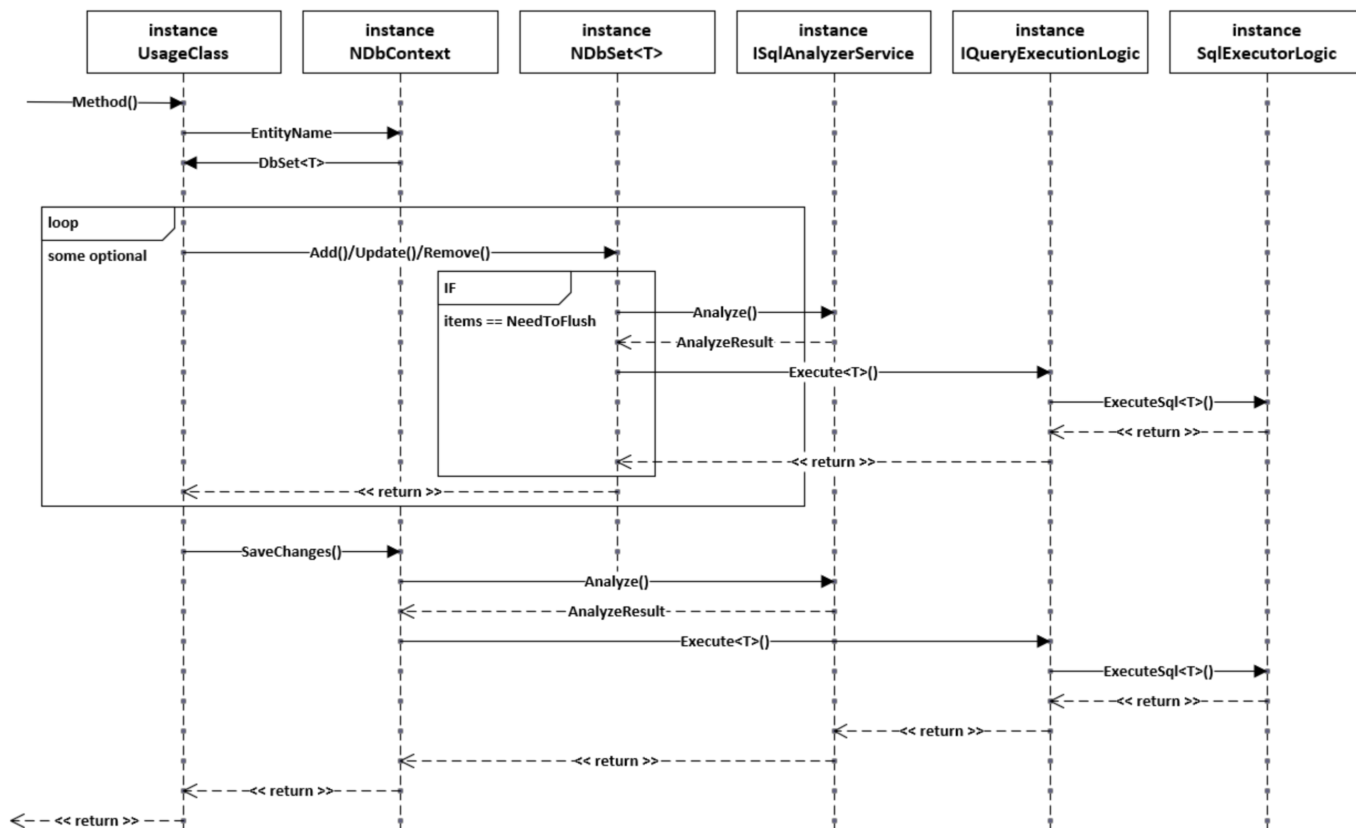


Рис 3.5. – Діаграма послідовностей виконання перевизначених методів

Для початку, в методі буде отриманий екземпляр класу «NDbSet<T>» працюючий з вибраною сутністю через виклик властивості із назвою «EntityName». Це також абстрактна властивість, ім’я та тип якої може змінюватись в залежності від кожного конкретного випадку. Об'єкт «NDbSet<T>» являє собою клас, який перевизначає методи для додавання, оновлення та видалення даних з таблиць. Далі, за допомогою циклічної конструкції, яка виконується поки умова виконання циклічної

конструкції повертає істину. В тілі циклу відбувається виклик будь-якої з операцій додавання, оновленення або видалення даних. Виклик будь-якої з цих операцій приводить до зміну стану кожної окремої сутності. Це відбувається за допомогою спеціального вбудованого об’єкту «ChangeTracker» ORM технології Entity Framework Core. При виконані будь-якої з вище перечислених операцій до внутрішнього сховища DbSet<T> додається спеціальний об’єкт «EntityEntry<T>» із інформацією про сутність та своїм статусом:

- Для SQL операції додавання даних – статус «Added».
- Для SQL операції оновлення даних – статус «Modified».
- Для SQL операції видалення даних – статус «Deleted».

Під час виконання кожної з операцій всередині об’єкту «NDbSet<T>» спрацьовує перевизначена версія методу. В ньому проходить перевірка кількості вже опрацьованих елементів з урахуванням типу операції. Якщо кількість елементів досягає вказаній в налаштуваннях при впроваджені залежностей, то система починає очищати буфер.

Під очисткою буферу розуміється виконання вказаних SQL команд на стороні серверу з метою розвантаження системи та економії пам’яті. Для цього використовується виклик аналізатору, а саме методу «Analyze» на інтерфейсі «ISqlAnalyzerService». Він повертає результат аналізу запиту, в якому перебувати згенерований SQL запит для виконання операції. Далі, використовується виклик методу «Execute<T>» інтерфейсу «IQueryExecutionLogic» для виконання SQL запиту. Який, в свою чергу, використовує виклик методу «ExecuteSql<T>» абстрактного об'єкту «SqlExecutorLogic». Даний об’єкт «під капотом» виконує SQL запит за допомогою ORM технології Dapper. Після виконання запиту управління повертається через усі шари до моменту виконання методу операції. Оброблені елементи видаляються із черги на оновлення ORM системою Entity Framework Core з черги на оновлені після виконання операції фіксації відстежених змін даних.

Після виконання циклічної конструкції викликається метод «SaveChanges» об'єкту «NDbContext». Цей метод займається фіксацією змін. Так, як при останьому виклику методу операції кількість елементів могла не дійти до порового значення очистки буферу – необхідно зафіксувати залишки змін над сутностями. Саме тому необхідно завжди виконувати метод «SaveChanges». Даний метод витягує з внутрішнього сховища змін над сутностями всі залишки та починає виконувати процес аналізу SQL запиту. Для цього використовується функціонал інтерфейсу «ISqlAnalyzerService», який в результаті виконання методу «Analyze» поверне сформований SQL запит. Він буде виконаний за допомогою виклику методу «Execute<T>» на інтерфейсі «IQueryExecutionLogic». Який в свою чергу виконає SQL запит через Dapper використовуючи в якості фасаду метод «ExecuteSql<T>» абстрактного об'єкта «SqlExecutorLogic». Після виконання SQL запиту увесь контроль буде повернуто до початкового виклику методу «Method», який завершить свою роботу та віддасть контроль далі по стеку викликів.

3.3.4 Архітектура виконання операції оновлення, видалення та додавання даних партіями

На відміну від попереднього підрозділу тут буде розглянута архітектура виконання операцій оновлення, видалення та додавання даних однією групою. Це підходить для ситуації, коли розробник програмного забезпечення знає про можливість отримання досить великої порції даних, стан яких потрібно перенести до бази даних. Завдяки цьому, розробник може відразу використату одну операцію для роботи із сутностями. Це суттєво зменшує та покращує написання кодової бази. Адже, для виконання операції потрібно буде виконати лише мапінг даних до сутностей та саме виконання операції. На рисунку 3.6. зображено використання механізму для додавання, оновлення або видалення даних з таблиць партіями.

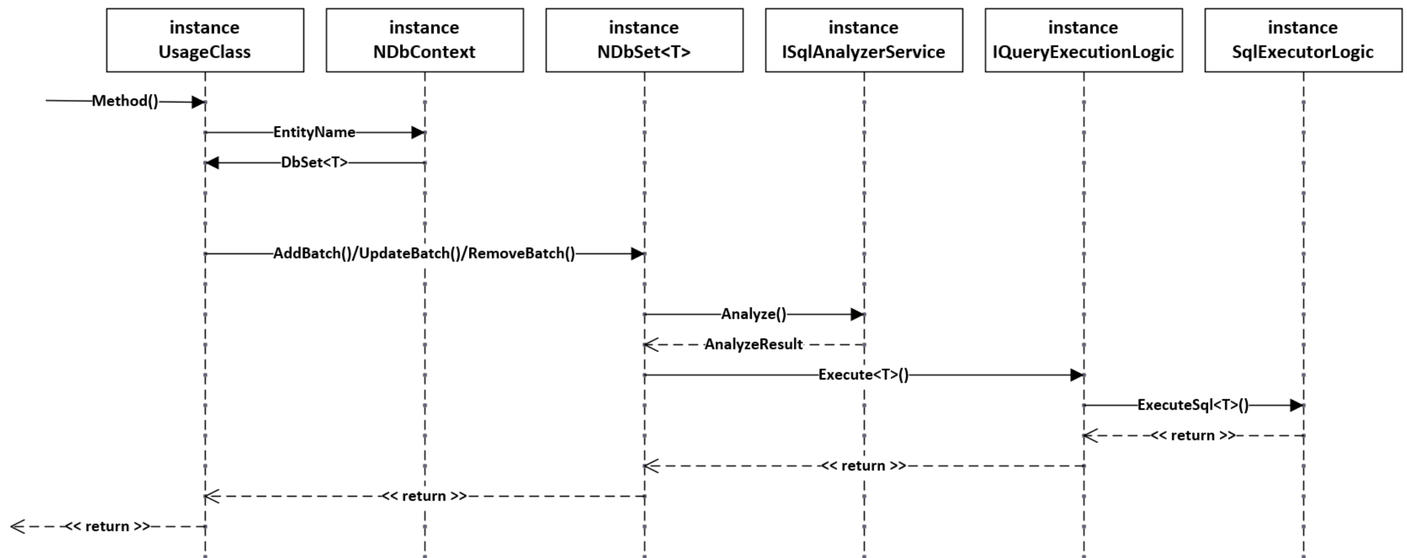


Рис 3.6. – Діаграма послідовностей виконання операцій партіями

Робота знову починається з виклику методу «Method» екземпляру об’єкту «UsageClass». Нагадую, що даний метод та екземпляр об’єкта представляє будь-який метод та об’єкт, який буде виконувати операції додавання, видалення та оновлення даних партіями. В методі буде отриманий екземпляр класу «NDbSet<T>» працюючий з вибраною сутністю через виклик властивості із назвою «EntityName». Це абстрактна властивість, ім’я та тип якої може змінюватись в залежності від кожного конкретного випадку використання та конкретної доменної області програмного забезпечення. Об'єкт «NDbSet<T>» являє собою клас, який має методи для додавання, оновлення та видалення даних партіями з таблиць. Кожен з цих методів приймає в якості вхідного параметру масив елементів типу T з модифікатором змінної кількості аргументів. Це дозволяє передавати елементи напряму, у вигляді масивів та колекцій. Опис методів:

- AddBatch() – метод для додавання партії даних до таблиці бази даних.
- UpdateBatch() – метод для оновлення партії даних в таблиці бази даних.
- RemoveBatch() – метод для видалення партії даних з таблиці бази даних.

Кожен з вище вказаних методів всередині викликає метод «Analyze» інтерфейсу «ISqlAnalyzerService» для отримання аналітичного результата. Цей результат має

сформований оптимальний SQL запит для подальшого виконання. Яке проходить через використання методу «Execute<T>» інтерфейсу «IQueryExecutionLogic». Даний контракт використовує абстрактний об’єкт у вигляді фасаду для виконання SQL запиту через використання ORM технології Dapper. Фасад являє собою абстрактний об’єкт під назвою «SqlExecutionLogic», який має метод для виконання під назвою «ExecuteSql<T>».

3.4. Випробування та тестування

3.4.1. Випробування вимог до безпеки

В підрозділі №3.2. були виявлені потенційні небезпеки, описана їх технічна реалізація та способи уникнення. Після реалізації програмного забезпечення для розширення ORM технології Entity Framework Core необхідно провести контрольний запит на вразливість до SQL ін’єкцій. Для цього буде створена тимчасова таблиця з назвою «Admins». Серед атрибутів, таблиця буде мати лише ідентифікатор, логін та пароль. За допомогою онлайн ресурсу «GenerateData» будуть згенеровані дані для додавання до таблиці в кількості 1000 записів. Для проведення контрольного запиту буде створений Linq запит в програмному коді для вибірки даних за таблиці за умовою, що введений користувачем програмного забезпечення логін та пароль співпадають з існуючим записом

В якості інтерфейсу для введення даних буде створений консольний додаток, де користувачеві буде пропонуватися ввести логін та пароль для пошуку сутності адміністратора у базі даних. Якщо користувач введе коректні дані адміністратора, він отримає повідомлення, що його наділено правами адміністратора. Якщо дані будуть некоректні, то отримає повідомлення, що введений логін чи пароль невірний. При введенні логіну буде введенений наступний рядок: «admin’ OR 1=1--». Якщо повернеться повідомлення, що логін та пароль невірний – це буде критерієм для успішної оцінки безпеки програмного забезпечення до SQL ін’єкцій.

3.4.2. Тестування програмного забезпечення

Після розробки програмного забезпечення необхідно провести його тестування для підтвердження збільшення ефективності його роботи. Перед описом процесу тестування необхідно визначити критерії оцінювання:

- Успішна оцінка прийняття – виконання перевизначених SQL операцій в 4 рази швидше за стандартний аналог.
- Задовільна оцінка прийняття – виконання перевизначених SQL операцій в 1-2 рази швидше за стандартний аналог.
- Незадовільна оцінка прийняття – виконання перевизначених SQL операцій за той же самий час або менше за стандартний аналог.

При тестуванні, необхідно буде приблизно оцінити виконання кожної операції. Так як при кожному старті програмного забезпечення для тестування, можна отримувати новий результат – критерії оцінювання будуть визначатися саме приблизно. Це дозволить сфокусуватися на розробці програмного забезпечення замість приведення результатів виконання до ідеалу шляхом шліфування кожного конкретного алгоритму.

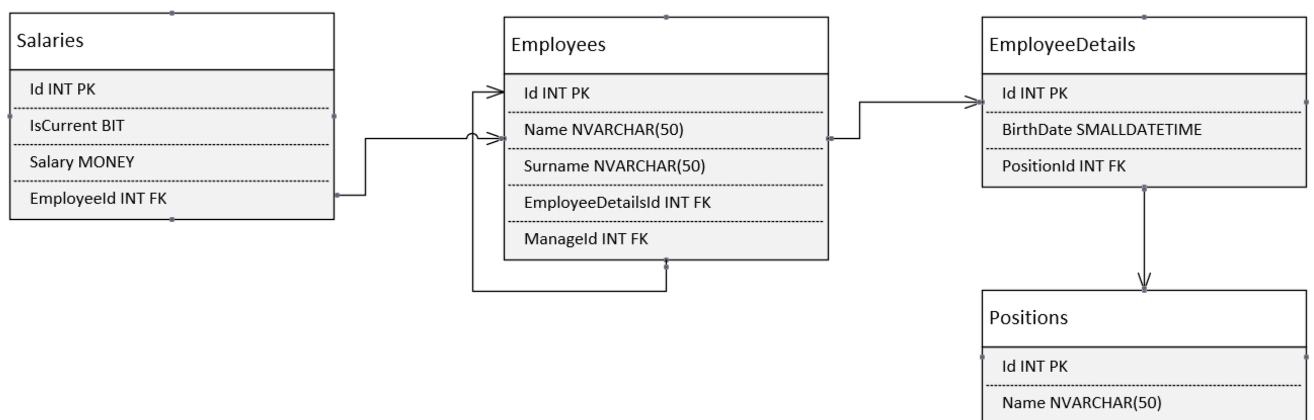


Рис. 3.7. – Схема тестової бази даних

Для тестування необхідно створити базу даних з набором таблиць, які сполучені одна з одною за допомогою реляційних зв'язків. Розробити SQL запити для виконання в стандартній формі та за допомогою перевизначених операцій. На рисунку 3.7. зображена схема тестової бази даних. Вона складається з чотирьох таблиць: «Employees», «EmployeeDetails», «Positions» та «Salaries». Кожна з цих таблиць має різний набір атрибутів, конфігурації та зв'язані між собою реляційним відношенням «один до одного» або «один до багатьох».

Таблиця «Positions» має наступні атрибути:

1. Id цілого числового типу. Має налаштування первинного ключа. Зберігає унікальний ідентифікатор у вигляді авто-інкрементованого числа.
2. Name рядкового типу. Має обмеження в кількості 50 символів для збереження даних. Зберігає інформацію про посади, які можуть обіймати співробітники.

Таблиця «EmployeeDetails» має наступні атрибути:

1. Id цілого числового типу. Має налаштування первинного ключа. Зберігає унікальний ідентифікатор у вигляді авто-інкрементованого числа.
2. BirthDate типу дати та часу. Зберігає інформацію про дату та час народження співробітника.
3. PositionId цілого числового типу. Має налаштування зовнішнього ключа. Зберігає унікальний ідентифікатор зовнішньої таблиці «Positions» для створення реляційного зв'язку «один до багатьох». Через зовнішній ключ можна дізнатись посаду співробітника.

Таблиця «Employees» має наступні атрибути:

1. Id цілого числового типу. Має налаштування первинного ключа. Зберігає унікальний ідентифікатор у вигляді авто-інкрементованого числа.
2. Name рядкового типу. Має обмеження в кількості 50 символів для збереження даних. Зберігає ім'я співробітника.

3. Surname рядкового типу. Має обмеження в кількості 50 символів для збереження даних. Зберігає фамілію співробітника.
4. EmployeeDetailsId цілого числового типу. Має налаштування зовнішнього ключа. Зберігає унікальний ідентифікатор зовнішньої таблиці «EmployeeDetails» для створення реліаційного зв'язку «один до одного». Через зовнішній ключ можна дізнатись детальну інформацію співробітника.
5. ManageId цілого числового типу. Має налаштування зовнішнього ключа. Зберігає унікальний ідентифікатор цієї ж таблиці «Employees» для створення реліаційного зв'язку «один до багатьох». Через зовнішній ключ можна дізнатись кому співробітник підпорядковується.

Таблиця «Salaries» має наступні атрибути:

1. Id цілого числового типу. Має налаштування первинного ключа. Зберігає унікальний ідентифікатор у вигляді авто-інкрементованого числа.
2. IsCurrent типу біт – еквівалент логічного значенню істини або хиби. Позначає, що інформація про заробітну плату є активною або ні. Зберігає значення 1, якщо цей цей запис є активним. Зберігає значення 0, якщо цей запис не є активним.
3. Salary типу «грошей». Зберігає заробітну плату співробітника.
4. EmployeeId цілого числового типу. Має налаштування зовнішнього ключа. Зберігає унікальний ідентифікатор зовнішньої таблиці «Employees» для створення реліаційного зв'язку «один до багатьох». Через зовнішній ключ можна дізнатись якому співробітнику належить заробітна плата.

Згідно із схемою бази даних можна розробити SQL запити для перевірки працездатності та ефективності як стандартного, так перевизначених операцій. Кожну з операцій необхідно виконати 3 рази для чистого тестування, щоб можливе навантаження апаратного забезпечення не могло вплинути на процес тестування. Середовище для тестування повинно бути очищене від сторонніх додатків та програмного забезпечення, яке витрачатиме ресурси для виконання SQL запитів. Для

створення середовища необхідно використати віртуальну машину, де будуть відсутні будь-які додатки для особистого користування. На дану машину буде встановлений Windows сервер 2016 року та необхідне програмне забезпечення для розгортання СУБД SQL Server.

Для операції проекції необхідно виконати простий Linq запит проекції до кожної з існуючої таблиць: «table.Select().ToList()».

Для операції перехресного пошуку необхідно виконати наступні Linq операції:

1. «employees.Join(employeeDetails, emp => emp.EmployeeId, ed => ed.Id, (emp, ed) => new { Employee = emp, EmployeeDetails = ed })» необхідний для перевірки моделі перевизначення SQL запиту для перехресного пошуку за індексами.
2. «employees.Select(emp => new { Name = emp.Name, Surname = emp.Surname, PromotionNumbers = emp.Salaries.Count() })» необхідний для перевірки моделі перевизначення SQL запиту для перехресного пошуку з наявністю агрегованих функцій у SQL запиті.
3. «employees.Join(employeeDetails, emp => emp.EmployeeId, ed => ed.Id, (emp, ed) => new { Employee = emp, EmployeeDetails = ed }).OrderByDescending(emp => emp.EmployeeDetails.BirthDate)» необхідний для перевірки моделі перевизначення SQL запиту перехресного пошуку з наявністю операції сортування даних.

Для перевірки оптимізації операції додавання даних до таблиць необхідно використати два способи: додавання по одному Linq запиту через циклічну конструкцію та додавання даних партіями. Те ж саме необхідно зробити з операціями оновлення та видалення даних.

Щоб зробити тестування об’єктивним з погляду кількості даних до запитів. В залежності від кількості одиниць обробки буде визначений найоптимальніший спосіб для виконання кожної перевизначеної SQL операції. Кожне тестування буде проводитись над двома різними кількостями даних: 1000, та 400000 одиниць даних.

4. МОДЕЛІ ТА АЛГОРИТМИ

4.1. Модель вдосконалення виконання операції проекції

Дані кожної з таблиць можуть знаходитися на різних сторінках пам’яті. Кожна із сторінок зберігає 8 кілобайт даних. При пошуку даних за допомогою операції проекції – системі доведеться шукати дані через всі сторінки даних, щоб витягнути інформацію потрібну розробникові програмного забезпечення. Операція фільтрації даних підвищує кількість оброблюваних даних через необхідність пошуку лише певних записів з таблиці. СУБД SQL Server має декілька алгоритмів [20] для виконання операції проекції, які можна використати для пришвидшення обробки даних. Кожен з представлених алгоритмів має власну «вартість» свого виконання. Під «вартістю» розуміється кількість затрачених ресурсів та часу.

Лінійний пошук [20] – зчитування кожного запису даних файлів із подальшим порівнянням з критеріїв підбору операції фільтрації. «Вартість» виконання лінійного пошуку за неключовим атрибутом дорівнює формулі (1.1). Якщо пошук буде проводитися за ключовим атрибутом середня «вартість» становитиме формулі (1.2). В найгіршому випадку становитиме повний обсяг, у вигляді формули (1.1).

$$a_r \tag{1.1}$$

де a – це кількість блоків у файлі;

r – представляє відношення

$$a_r/_2 \tag{1.2}$$

де a_r – це кількість блоків у файлі, що представляють відношення r

2 – стале значення, що представляє половину об’єму файлу з даними

Бінарний (двійковий) пошук за первинним ключем [20] – пошук рівності, що виконується за атрибутом з налаштуванням первинного ключа. Що означає, що сторінки пам’яті будуть впорядковані за первинним ключем. Це значно ефективніше

за лінійний пошук та зважди буде дорівнювати виконанню формули (1.2). В найгіршому випадку «вартість» становитиме формулі (1.3).

$$[lg(a_r)] \quad (1.3)$$

де a_r – це кількість блоків у файлі, що представляють відношення r

Пошук з використанням кластерного індексу [20]. Якщо таблиця має кластеризований індекс – її структура збережуваних даних перетворюється на бінарне дерево. Тобто, табличні записи в пам’яті будуть відсортовані за принципом бінарного дерева. Де ключом пошуку буде сам індекс, а значеннями всі інші атрибути таблиці.

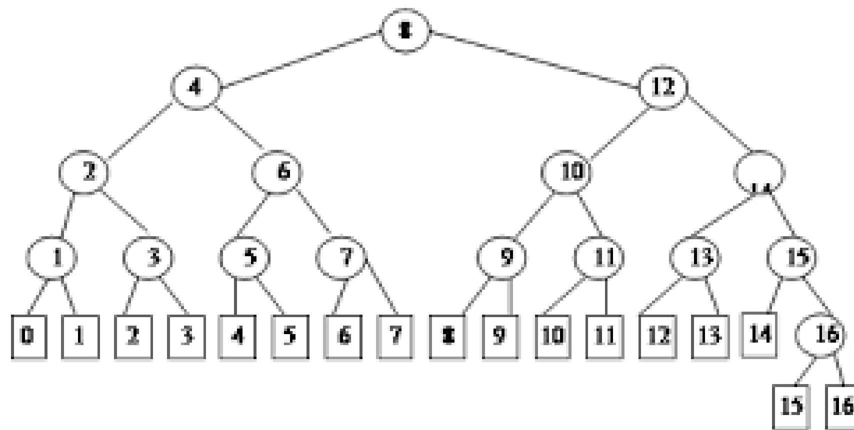


Рис. 4.1. – Приклад бінарного дерева

Корневий вузол дерева має серединє значення для елементів кластерного індексу. Кожна наступна гілка, яки створюється з корню та наступних гілок ділить свій диапазон значень на по середині. Самі значення зберігаються у листових вузлах дерева. Із кожним значенням індексу в листовому вузлі одразу зберігається всі значення його атрибутів. Для пошуку використовуються операції порівнення значень (<, >, <=, >=). Найгірший пошук рівності з використанням значення кластерного індексу для пошуку інформації за алгоритмом обхіду бінарного дерева буде дорівнювати формулі (1.4)

$$h + 1 \quad (1.4)$$

де h – це висота бінарного дерева. Тобто, середнє значення між найменшим та найвищим.

1 – це костанте значення, що представляє одну дію для переходу листового вузлу дерева.

Пошук з використанням одразу двох різновидів індексів [20]. В таблицяї окрім кластерного індексу, також може бути присутній некластерний індекс. Некластерний індекс схожим чином сортує дані таблиці за єдиним виключенням, що в його дереві на листових вузлах знаходиться лише значення некластерного індексу та посилання на комірку в пам’яті, яка зберігає необхідні значення атрибутів таблиці. Якщо таблиця буде індексована кластерним індексом, то некластерний індекс буде вказувати не на комірку в па’мяті, на кластерний індекс бінарного дерева.

4.2. Висновок

Розглянуті алгоритми операції проекції дозволяють зрозуміти, що в залежності від конфігурування таблиці вибарається відповідний алгоритм виконання пошуку необхідний даних за вказаним фільтром. Найефективніший результат надає наявність кластерного індексу в парі з некластерними індексами. Зазвичай, кластерний індекс встановлюють за замовчуванням на атрибут ідентифікуючий запис в таблиці використовуючи обмеження первинного ключа. Проте, пошук даних для вибірки не завжди включає в себе використання в якості фільтра атрибуту первинного ключа, що може досить сильно сповільнити виконання запиту. Настільки, що може бути вибраний лінійний алгоритм пошуку даних. Ця проблема вирішується наданням, часто використовуємих для пошуку атрибутів в якості фільтра, некластерного індексу. Він дозволяє використати пошук через бінарне дерево для знаходження значення кластерного індекса та вилучення через нього необхідних значень для атрибутів

пошуку. Але, наявність великої кількості некластерних індексів негативно впливає на операцію додавання записів до таблиці. Тому що, після кожного додавання даних необхідно перерахувати некластерні індекси згідно доданих нових значень.

Це дозволяє створити статистичний аналізатор, який у змозі буде аналізувати виконувані SQL запити проекції та можливі SQL операції додавання записів до таблиць. Якщо співвідношення проекції до додавання записів до таблиць буде 1:8, можна наділити систему розширень правами автоматичної генерації некластерних індексів до постійно використовуючих атрибутів таблиці в якості операнду операції фільтрації. Таким же чином, якщо співвідношення проекції до додавання записів до таблиць вийде за вказане значення (1:8) система автоматично знищить некластерний індекс.

4.3. Модель оптимізації виконання оператору перехресного пошуку даних

За реалізацією операцій перехресного пошуку даних лежить реліаційна алгебра [21]. Вона досить сильно схожа на звичайну алгребру, окрім того, що вона використовує замість чисел відносини як значення. Також, відрізняються оператори та операції. Деякі розширені SQL запити потребують явних операцій реліаційної алгебри для свого виконання. Мова запитів SQL є декларативною. Це означає, що при створені запиту ви вказуєте, що ви хочете отримати. Проте, не вказуєте як це має бути обчислено. Тому, для зміни принципів виконання оператору перехресного з’єднання слід шукати шляхи для побудування SQL запиту таким чином, щоб це підвищило його ефективність в залежності від умови перехрещення, функцій та операторів, які присутні для використання у запиті.

Перехрещенний пошук за допомогою наявності кластерних та некластерних індексів [21]. Він оброблюється алгоритмом реліаційної алегребри представлений у формулі (1.5).

$$
\begin{aligned}
&R3 = join(R1, Ai, R2, Bj)\\
&\quad for\ each\ tuple\ t\ in\ R1\ do\\
&\quad\quad for\ each\ tuple\ s\ in\ R2\ at\ index(t.Ai)\ do\\
&\quad\quad\quad insert(R3, t.A1, t.A2, \dots, t.AN,\\
&\quad\quad\quad\quad s.B1, \dots, s.B(j-1), s.B(j+1), \dots, s.BM)
\end{aligned} \tag{1.5}
$$

де R3 – це результуюча вибірка, куди будуть поміщені результата перехресного пошуку.

R1 – це «ліва» таблиця, з якої починається пошук даних

R2 – це «права» таблиця, дані якої зрівнюються з даними, отриманими від лівої таблиці

Ai – це пошуковий індекс елементу з «лівої» таблиці.

Bi – це пошуковий індекс елементу з «правої» таблиці.

t – це елемент для отримання доступу до елементу з «лівої таблиці».

s – це елемент для отримання доступу до елементу з «правої таблиці».

Даний алгоритм використовує циклічний перебір індексів кожної з таблиці з необхідністю знаходження співпадінь. Таким чином, доступ до даних за допомогою такого перехресного пошуку буде відбуватися значно швидше через використання індексованих атрибутів. Щоб даний спосіб спрацював умовою для виконання завжди буде наявність індексованих атрибутів в умовах для перехресного пошуку даних.

Перехрещений пошук за допомогою наявності агрегованих операцій [21]. Основою алгоритму оптимізації даного типу пошуку є відокремлення виконання операції агрегації та операції перехресного пошуку один від одного. Алгоритм складається з декількох кроків:

1. Виявлення операції агрегування даних.
2. Якщо вона присутня, необхідно відокремети агрегацію до окремого SQL запиту. Якщо операцій агрегації немає перейти до виконання кроку №4.

3. Створення вкладеного SQL запиту для основного, який поверне результат у табличній формі із атрибутом для перевіреки умови перехресного пошуку та атрибутами, які несуть в собі значення результату виконання агрегованної функції.
4. Виконання основного SQL запиту.

При цьому, при створені вкладеного SQL запиту потрібно використовувати оператор для групування незгрупованних даних. Адже вкладений SQL запит поверне згрупований результат.

Перехрещений пошук за допомогою хеш-з'єднання [22]. Вони доволі ефективно обробляють несортовані, великі та неіндексовані з'єднання даних. Хеш-зєднання досить корисні для виконання у складних проміжних SQL запитах. Декілька переваг використання:

- Проміжні результати складних запитів не піддаються індексації і зазвичай не сортуються для наступної операції в плані запиту.
- Оптимізатори SQL запитів оцінюють лише проміжні розміри результатів. Оскільки оцінки дуже неточні для складних SQL запитів, алгоритми, які обробляють проміжні результати мають бути ефективними. А також, в ситуації коли проміжний результат виявляється набагато більшим, ніж очікувалося – алгоритм повинен погіршити обчислення, щоб не втратити всю пам'ять на його виконання.

Використання хеш-з'єднання зменшує використання процесу денормалізації. Вона використовується для досягнення високої продуктивності за рахунок скорочення операцій з'єднання. Окрім зменшення потреби в денормалізації, хеш-з'єднання надають можливість використовувати для фізичного проектування бази даних вертикальний розподіл. Це представлення групи атрибутів з однієї таблиці в окремих файлах або навіть індексах.

Хеш-з’єднання для своєї роботи має два входи. Першний – це вхід для побудови. Другий – це вхід для перевірки. Оптимізатор SQL запитів визначає ці ролі, щоб найменший серед двох вхідних даних був вхідним для збірки в цілому.

Хеш-з’єднання може бути використаним для багатьох типів операцій перехресного пошуку:

- INNER JOIN – внутрішнє перехресне з’єднання
- OUTER LEFT|RIGHT|FULL JOIN – зовнішне ліве/праве/повне перехресне з’єднання
- LEFT|RIGHT SEMI JOIN – ліве/праве перехресне напівз’єднання
- INTERSECT – знаходження елементів, що перетинаються між двома множинами даних
- UNION – об’єднання елементів з двох множин даних
- EXCEPT – включення до результату результатів множини даних, що не входять до даних правої множини.

Окрім очевидних операторів перехресного пошуку та операторів роботи над множинами – хеш-з'єднання може ефективно виконувати видалення і групування дублікатів. Наприклад, SQL запит: «SUM(Indexing) GROUP BY Position». Дана модифікація використовує лише один вхід як для ролі збірки, так і для проведення тестування.

4.4. Висновок

Розглянуті алгоритми та моделі викоання операції перехрещеного пошуку даних дозволяють зрозуміти, що в залежності від використання атрибутів та операторів можна переробити SQL запит для більш ефективного виконання. Найшвидше оператори перехресного пошуку будуть відпроцювувати, якщо в умові для пошуку будуть використовуватись кластерні або некластерні індекси. Це

досягається за допомогою використання внутрішніх алгоритмів, які побудовані на основі реліаційної алегбри. Дані алгоритми циклічно переберають зв’язки двох таблиць з ціл’ю додавання індексів до результуючого набору. Таким чином, після виконання знаходження всіх індексів пройде їх вибірка. З попереднього підрозділу відомо, що найефективніша вибірка даних проходить саме над індексованими даними. При наявності агрегованих операторів разом з операцією перехрещення даних краще не використовувати їх разом. Для покращення необхідно розділити ці операції на два етапи. Перший – створення вкладеного SQL запиту для вибірки агрегованих даних з ідентифікатором, який використовується для перехресного пошуку в умові. Другий – вкладений SQL запит необхідно приєднати до основного перехресного оператору пошуку. Тобто, спочатку виконається агрегування даних, а лише потім операція з’єднання цих даних с даними іншої таблиці. При виконанні складних запитів найкраще використовувати хеш-з’єднання. Спочатку воно сканує або обчислює весь вхідний збір, а потім створює хеш-таблицю в пам'яті, якщо вона відповідає наданню пам'яті. Кожен рядок вставляється в хеш-відро відповідно до хеш-значення, обчисленого для хеш-ключа, тому для створення хеш-таблиці потрібна пам’ять. Якщо всі вхідні дані збірки менші за доступну пам’ять, усі рядки можна вставити в хеш-таблицю. За цією фазою збирання слідує фаза тестування. Весь вхід датчика сканується або обчислюється по одному рядку за раз, і для кожного рядка зондування обчислюється значення хеш-ключа, відповідний сегмент хешування сканується і збігається. Проте, якщо результат виконання стає значно більший за очікуваний – алгоритм буде погіршувати його виконання, щоб не втратити всю пам’ять на виконання SQL запиту.

Необхідно створити аналізатор для модифікації SQL запиту згідно з отриманою інформацією. Для початку необхідно проаналізувати, як саме модифікувати SQL запит. Потім перебудувати SQL запит згідно однієї з моделей чи алгоритмів. Проте, якщо аналіз покаже, що оптимізація згідно розширених моделей та алгоритмів

неможлива – необхідно виконати SQL запит існуючим шляхом через сформований Entity Framework Core SQL запит

Якщо в таблиць для перехресного пошуку є індексовані атрибути, необхідно переписати SQL запит з їх використанням для збільшення продуктивності виконання запиту.

Якщо SQL запит для перехресного пошуку має використання агрегації даних – необхідно переписати запит. Виконання агрегатного оператору винести у вкладений запит, який під'єднати до основного запиту, що виконає перехресний пошук даних. Спосіб перехресного з'єднання даних через наявність агрегатної функції буде аналізуватись на основі кількості агрегатів та атрибутів для групування та складності функцій агрегатів. Якщо кількість атрибутів для групування буде більше за 5 – це збільшує вірогідність використання цього способу на 35%. Більше за 10 – збільшує вірогідність на 55%. Якщо використовуються агрегатні функції Sum, Avg – це додає 20% за кожну. Якщо Count, Min, Max – це додасть по 10% за кожну. Коли коефіцієнт буде більше за 0.75 (75%) – тоді буде використаний даний спосіб перехресного з'єднання.

Якщо SQL запит для перехресного пошуку має більше, ніж 10 операторів для виконання – необхідно модифікувати SQL запит для явного використання оператору HASH JOIN. Даний оператор вбудований в СУБД SQL Server, що дозволить не затрачувати багато зусиль на зміну SQL запиту, а лише замінити звичайні операторі JOIN на їх хеш версію.

4.3. Алгоритм для оптимізації масового оновлення/видалення/додавання даних одиничним шляхом

Проблема роботи масових операцій оновлення, видалення та додавання даних до таблиць полягає у поглинанні великої кількості ресурсів та не оптимальному

виконанні великої кількості запитів. Більшість оптимізацій цього процесу пропонують методи для проведення «масових» операцій, які можуть оптимально генерувати SQL запити. ORM технологія Entity Framework Core дозволяє розширювати власний функціонал та його перевизначати. Алгоритм дій повинен бути наступним:

1. Створення налаштування для відслідковування кількості даних виконуваних в одному пакеті інструкцій. Це необхідно, адже кожна СУБД має свої обмеження щодо кількості використовуємих параметрів в контексті одного SQL запиту. При масовій операції таких параметрів може бути тисячі.
2. Створення розширюваних методів для реєстрації залежностей. Це надасть змогу розробнику програмного забезпечення вказати всі налаштування для використання можливостей розширення ORM системи.
3. Створення реалізації класу «DbSet<T>» під назвою «NDbSet<T>».
4. Перевизначити методи для оновлення, видалення та додавання даних в класі «NDbSet<T>». В них необхідно перевіряти кількість отриманих даних. Якщо ця кількість співпадає з вказаним значенням у налаштуванні (створеному у кроці №1), то необхідно звільнити буфер.
5. Необхідно згенерувати SQL запит, який буде виконувати всі визначені операції в констекті одного пакету даних. Тобто, однієї SQL команди.
6. Виконання згенеровано SQL запиту.
7. Знищення виконаних даних з внутрішнього сховища відстежуваних сутностей, щоб запобігти їх повторному виконанню.
8. Якщо сутностей у внітрішьому сховищі недостатньо для виконання звільнення буферу, необхідно виконати базову реалізацію методу.
9. Створення реалізації класу «DbContext» під назвою «NDbContext».
10. Перевизначити метод «SaveChanges» в класі «NDbContext». Необхідно всі сутності, які оновлюються/видаляються або додаються то таблиць – вибрати та

об'єднати для генерації SQL запиту. Він буде виконувати агреговані дані в контексті одного пакету даних.

11. Виконати базову реалізацію методу «SaveChanges».

Зазвичай, розробники програмного забезпечення використовують циклічні конструкції для створення операції оновлення, видалення та додавання даних до бази одиничними SQL командами через невідому кількість приймаємих даних. Це може бути як 5 сутностей, так 5 мільйонів. Даний алгоритм дозволить оптимізувати процес виконання операцій незважаючи на кількість даних.

4.4. Алгоритм для оптимізації масового оновлення/видалення/додавання даних пакетами

Проблеми, що описані в попередньому розділі стосуються й цього. Як було сказано раніше: «Більшість оптимізацій цього процесу пропонують методи для проведення «масових» операцій, які можуть оптимально генерувати SQL запити». Це саме так. Проте, необхідно брати до уваги можливість пропускної здатності СУБД. Кожна із існуючих СУБД має власний максимум для одночасного використання параметрів у контексті одного пакету. Що означає, якщо розробник програмного забезпечення виконає дану операцію без урахування цього аспеткту – він отримає помилку при виконанні SQL запиту. На відміну від попереднього випадку, в цій ситуації розробник програмного забезпечення явно знає, що кількість даних для передачі до бази даних буде досить об'ємною. Тому, помилка при досяганні ліміту – це досить погана поведінка описаної оптимізації, що використовують деякі ORM системи для виконання масових операцій. Алгоритм, представлений в цьому підрозділі має знешкодити описану небезпеку. Алгоритм дій повинен бути наступним:

1. Використання створеного в попередньому підрозділі налаштування для відслідковування кількості даних виконуваних в одному пакеті інструкцій.

2. Використання створених в попередньому підрозділі розширюваних методів для реєстрації залежностей у програмному забезпеченні.
3. Використання створеного в попередньому підрозділі класу «NDbContext» для розширення його методами «UpdateBatch», «RemoveBatch» та «AddBatch».
4. Кожен новостворений метод має реалізувати алгоритм для аналізу повної кількості отриманих даних для розбиття усієї купи даних на партії згідно з максимальним розміром налаштування.
5. Виконати SQL запит

5. РЕАЛІЗАЦІЯ РОЗШИРЕНЬ

5.1. Створення програмного забезпечення та його конфігорування

Для реалізації розширень по-перше необхідно створити набір бібліотек, які будуть використовувати розробники програмного забезпечення. Найкращим інструментов для роботи з платформою.NET Core це Visual Studio. Тип програмного забезпечення необхідно вибрати «.NET Core Class Library». Це бібліотека класів – яку майбутній користувач (розробник програмного забезпечення) буде використовувати для впродження розширень для свого програмного забезпечення.

Після створення проектів, необхідно встановити NuGet пакети для підключення всіх необхідних бібліотек для роботи ORM технології Entity Framework Core. Встановлення пакетів можна зробити як через графічний інтерфейс так і через консоль пакетів NuGet [23]. Для встановлення на момент написання наукової роботи останньої версії Entity Framework Core необхідно змінити версію проекту на .NET 5.0 або вище.

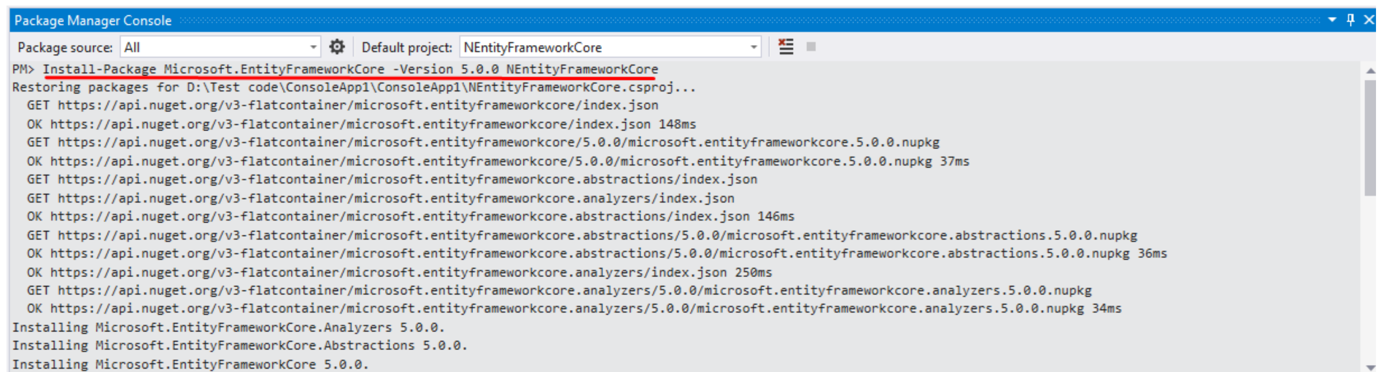


Рис. 5.1. – Встановлення пакетів Entity Framework Core

На рисунку 5.1. зображено використання налаштованої інструкції для встановлення необхідних пакетів Entity Framework Core. Необхідно вказати команду для встановлення пакету «Install-Package». Після чого вказується назва пакету бібліотек, який необхідно встановити. Далі використовуються параметер «-Version»,

щоб вказати після назви параметру версію встановлюємого пакету. Останньою частиною інструкції є вказання проекту, у який буде встановлення пакету бібліотек.

Рішення для створення розширень ORM технології Entity Framework Core буде складатися з більше, ніж одного проекту бібліотек класів платформи .NET Core. Після створення майже кожного проекту – він буде сконфігурований вказаним способом в цьому розділі для можливості взаємодії з ORM системою.

5.2. Розробка нової моделі для розширення Entity Framework Core

Для початку необхідно створити проект бібліотеки класів платформи .NET Core під назвою «NEFCore.NModel». До даного проекту необхідно під’єднати NuGet пакети Entity Framework Core, як описано у попередньому підрозділі. Даний проект буде розширювати стандартну модель Entity Framework Core шляхом створення нащадків для класів «DbContext» та «DbSet<T>». Окрім моделі, проект матиме метод, що розширює тип «IServiceCollection» для підключення всіх необхідних залежностей в програмне забезпечення, що буде використовувати даний набір бібліотек. При роботі з розширенням ORM системи розробникові програмного забезпечення доведеться встановити саме цю бібліотеку, яка надаватиме всі необхідні класи та об’єкти для оптимізації роботи.

По-перше, необхідно створити клас під назвою «NDbSet<T>». Він буде наслідувати базовий клас DbSet<T>. Даний клас буде розширений новим функціоналом в наступних розділах. Проте, знаходитись він має саме в цьому проекті, тому що він являється основною точкою входу для роботи з оптимізованим функціоналом.

По-друге, необхіжно створити клас під назвою «NDbContext». Даний клас матиме набір стандартних таблиць у вигляді екземплярів класу «NDbSet<T>». Ці таблиці необхідні для збереження статистики виконаних SQL запитів. На рисунку 5.2. зображена схема аналітичних таблиць.

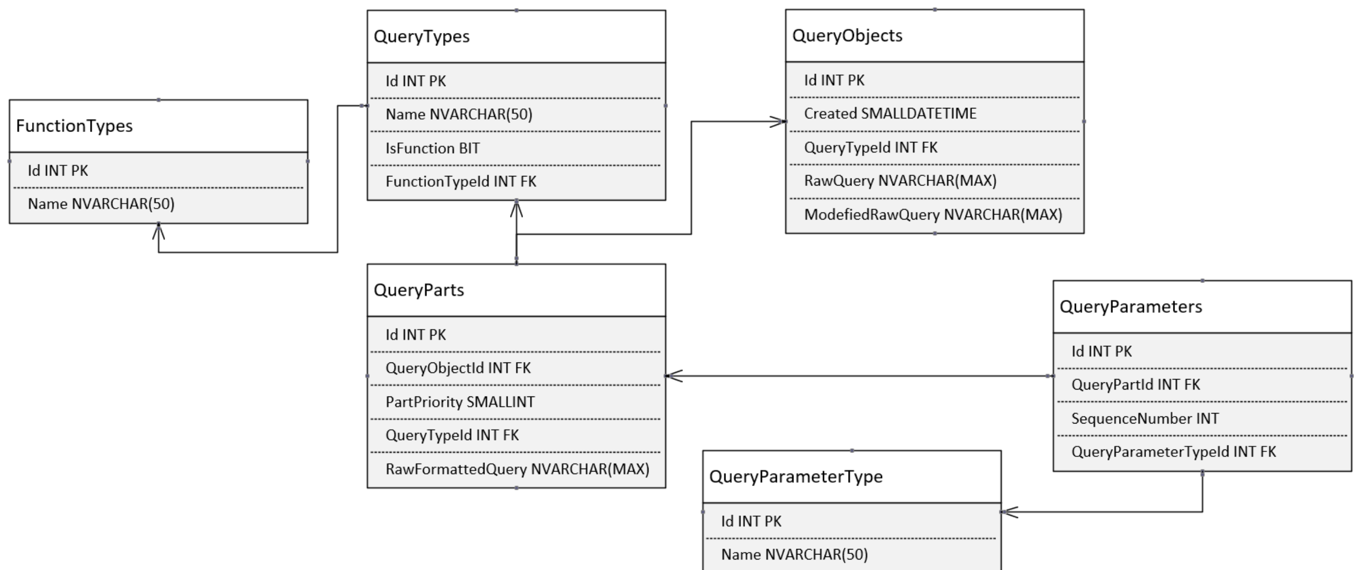


Рис 5.2. – Структура таблиць для збереження аналітичних даних SQL запитів

Таблиця «FunctionTypes» зберігає інформацію про всі можливі різновиди функцій. Доступні наступні різновиди функцій: скалярні, табличні та агрегатні. Дана таблиця необхідна для подальшого аналізу з метою визначення що частина запиту являє собою один із різновидів функції.

Таблиця «QueryTypes» зберігає необхідну інформацію про всі можливі типи операторів SQL запитів та стандартних функцій (SELECT, WHERE, JOIN, SUM…). Дана таблиця необхідна для аналізування частини запиту з цілью дізнатися, що саме буде виконувати дана частина SQL запиту. Таблиця має реліаційний зв’язок із таблицею «FunctionTypes», щоб дізнаватись тип функції.

Таблиця «QueryParts» зберігає необхідно інформацію про частину SQL запиту. По-перше, в таблиці зберагіється форматована частина SQL запиту дослівно у вигляді рядку. Завдяки використанні форматування, в рядку SQL запиту знаходиться маркер підстановки, який в подальшому буде заповнений необхідним значенням параметру. По-друге, це зберігання пріоритету частини SQL запиту. Кожен оператор SQL виконується згідно свого приорітету. Тому, необхідно зберігати даний приоритет для подальшого відтворення запиту з метою його аналізу На рисунку 5.3. зображений

порядок виконання SQL операторів. По-третє, в таблиці знаходиться реліаційний зв'язок до таблиці «QueryTypes», який надає інформацію про тип частини SQL запиту. Так, як таблиця представляє лише частину SQL запиту, вона має реліаційне відношення до таблиці «QueryObjects», яка представляє увесь SQL запит повністю. Тобто, декілька записів даної таблиці складають повний SQL запит.

1. FROM
2. ON
3. JOIN
4. WHERE
5. GROUP BY
6. WITH CUBE or WITH ROLLUP
7. HAVING
8. SELECT
9. DISTINCT
10. ORDER BY
11. TOP

Рис. 5.3. – Порядок виконання операторів SQL Server

Таблиця «QueryParameterTypes» зберігає інформацію про тип даних кожного параметру на стороні SQL серверу (INT, TINY, BIT, NVARCHAR…). Дана інформація необхідна для подальшого відтворення типу даних SQL параметрів у запиті.

Таблиця «QueryParameters» зберігає інформацію про параметри, які використовуються для відновлення частини SQL запиту. В даній таблиці зберігається інформація про порядковий номер параметру в частині запиту та саме значення. Також, таблиця має два реліаційні зв'язки. Перший – це ідентифікатор частини SQL запиту, до якого відноситься параметр. Другий – це ідентифікатор типу даних параметру, який допоможе при парсингу значення до відповідного типу даних.

Таблиця «QueryObjects» зберігає інформацію про повний SQL запит. По-перше, це дата створення запиту. По-друге, це вихідна версія SQL запиту в рядковому вигляді. По-третє, це модифікована версія SQL запиту в рядковому вигляді, що була сторевена побудовником SQL запиту. Основна задача даної таблиці – це збирання частин SQL запиту в єдиний запит. Всі реліаційні зв'язки зрештою сходяться саме в цій таблиці.

Окрім набору екземплярів NDbSet<T>, які будуть представляти сутності для зберігання аналітичних даних, класс «NDbContext» матиме перевизначення методу «OnModelCreating». В реалізації цього методу будуть встановлені налаштування для кожної сутності таблиць аналітичних даних. Всі налаштування вказані на рисунку 5.2. у вигляді позначок біля типу даних атрибуту та за допомогою вказаний реліаційних зв'язків. Найменування атрибутів таблиць із властивостями сутностей повністю збігається, тому не потребує додаткових налаштувань.

5.3. Розробка абстракції аналізаторів та побудовників SQL запитів

Для роботи з аналізаторами необхідно створити проект бібліотеки класів платформи .NET Core під назвою «NEFCore.SqlAnalyzation». До даного проекту необхідно під'єднати NuGet пакети Entity Framework Core, як описану у першому підрозділі даного розділу. Даний проект буде описувати абстракції та інтерфейси (контракти). По-перше, це будуть абстракції для аналізу згенерованих ORM системою SQL запитів. По-друге, це генератори та редактори SQL запитів з метою їх оптимізації. Архітектура аналізу та побудування SQL запиту описана у розділі №3.3.1. Деякі з інтерфейсів будуть реалізовані на даному етапі розробки. Проте, задля зменшення обсягу інформації – ціль та функціонал кожного буде описаний на рівні інтерфейсу.

По-перше, необхідно створити інтерфейс «ISqlTypeAnalyzer». Суть даного контракту у проведені аналізу SQL запиту для конкретного типу SQL операції. Тобто, кожна розширена операція даної бібліотеки матиме власну реалізацію. Інтерфейс

матиме єдиний метод «Analyze», який прийматиме на вхід один параметр у вигляді об'єкту SQL запиту. Метод повертає результат обробки SQL запиту. Він оброблює отриманий SQL запит з метою його модифікації для більш оптимального виконання. Якщо обробка буде успішною – він поверне модифікований SQL запит та інформацію про те, що запит був успішно модифікований.

По-друге, необхідно створити інтерфейс «ISqlAnalyzeStatLogic». Даний контракт буде проводити аналіз вже існуючих подібних запитів у базі даних. У нього є единий метод «Process», який приймає в якості параметрів об'єкт SQL запиту та тип SQL операції. Якщо буде знайдений успішно-модифікована версія даного SQL запиту, то буде повернутий модифікований SQL запит. Якщо модифікація не буде знайдена – буде вернутий пустий об'єкт.

По-третє, необхідно створити інтерфейс «ISqlAnalyzeLogic». Цей контракт реалізує логіку для аналізу існуючих запитів на можливу підходящу модифікацію або виконання процесу модифікації SQL запиту, якщо це можливо. В даному інтерфейсі присутній метод «Anyalyze», який приймає в якості вхідного параметру об'єкт SQL запиту, створеного розробником програмного забезпечення. А також, повертає результат аналізу. Це, при успішному виконанні – модифікований запит. Або, при провальному виконанні – вихідний запит, сформований системою Entity Framework Core. Якщо модифікація присутня він розпарсить SQL запит до об'єкту та підставить необхідні параметри для викликаємого запиту. Якщо модифікація відсутня – в роботу вступить побудовник SQL запиту («ISqlTypeAnalyzer»), який проаналізує запит на можливість модифікації та якщо це можливо – його модифікує. Якщо модифікація SQL запиту була створена на даному етапі, то інформація про це буде збережена до аналітичних таблиць з метою подальшого використання.

У четвертих, необхідно створити інтерфейс «ISqlAnalyzeFactory». Даний контракт буде віддавати готові об'єкти «ISqlAnalyzeLogic». Він має лише один метод під назвою «Get», який приймає у якості вхідного параметру тип SQL запиту та

повертає готовий об’єкт «ISqlAnalyzeLogic». На основі переданого типу SQL запиту проходить вибірка необхідного об’єкту.

У п’ятих, необхідно створити інтерфейс «ISqlAnalyzerService». Даний контракт являє собою лише фасад для виклику бізнес-логіки аналізу та модифікації SQL запиту. Він має лише один метод «Analyze», який приймає у якості вхідного параметру об’єкт SQL запиту та повертає в якості результата модифікований або вихідний SQL запит. Об’єкт для проведення аналізу буде отримуватися з фабрики «ISqlAnalyzeFactory».

5.4. Розробка розширень операції проекції та перехресних з’єднань

Для розробки розширень операції проекції та перехресних з’єднань необхідно розширити раніше створену бібліотеку «NEFCore.NModel». В даному випадку додати два класи: «SelectNExtensions» та «JoinNExtensions». Кожен з цих класів буде розширювати свою операції. Хід виконання у кожного з цих SQL операторів однаковий, за винятком конкретної (власної) реалізації аналізатора та побудовника SQL запитів. Архітектура аналізу та побудування SQL запиту описана у розділі №3.3.2. Деякі з інтерфейсів будуть реалізовані на даному етапі розробки.

По-перше, необхідно створити клас «SqlExecutorLogic». Даний клас являє собою фасад над використанням технології SQL Dapper. В класі є метод під назвою «ExecuteSql<T>», який приймає в якості вхідного параметру рядок для виконання SQL запиту та повертає отриманий результат. Даний метод використовує Dapper для виконання SQL запиту на стороні серверу та мапінгу результату у сутність.

По-друге, необхідно створити інтерфейс «IQueryExecutionLogic». Він має один метод під назвою «Execute<T>», який має в якості вхідних параметрів результат аналізу та повертає результат виконання модифікованого SQL запиту або сформований стандартний SQL запит у вигляді IQueryable<T>. Даний метод обробляє аналіз SQL запиту з метою виконання його модифікованої версії на місці або

повернення стандартної версії запиту. Модифікована версія SQL запиту виконується за допомогою створеного раніше фасаду «SqlExecutorLogic».

Кроки виконання операцій проекції та перехресного пошуку даних:

1. Кожен метод розширення за допомогою статичного модулю «DependencyProvider» отримує екземпляр об'єкту «IServiceProvider». Даний об'єкт спроможний видавати будь-яку із зареєстрованих залежностей у програмному забезпечені.
2. За допомогою отриманого постачальника сервісів методи розширення отримують екземпляр об'єкту «ISqlAnalyzerService».
3. На отриманому екземплярі виконується виклик методу «Analyze» з метою можливого отримання модифікованого SQL запиту.
4. За допомогою отриманого постачальника сервісів методи розширення отримують екземпляр об'єкту «IQueryExecutionLogic».
5. На отриманому екземплярі виконується виклик методу «Execute<T>», куди передається результат аналізу SQL запиту.
6. Метод оброблює результат аналізу. Перевіряє чи аналіз був проведений успішно та наявність модифікованого SQL запиту для виконання. Якщо обидві властивості будуть мати істіну то виконується крок №7 та №8. Якщо хоча б одна властвість буде мати хибу то виконується крок №9.
7. Передача модифікованого запиту до фасаду з метою виконання його через Dapper та повернення результату виконання.
8. Упаковка готових результатів у вигляді IEnumerable<T> до результату виконання обробки запиту.
9. Упоковка стандартного SQL запиту сформованого Entity Framework Core до результату виконання обробки з метою його виконання на стороні клієнта (програмний код, написаний розробником програмного забезпечення).
10. Повернення результату виконання.

5.4.1. Реалізація моделі аналізу для оптимізації операції проекції

Для підвищення ефективності виконання SQL запиту проекції згідно розділу №4.1. необхідно налаштування сутностей за допомогою некластерних індексів. Це значно збільшує процес проведення проекції з операцією фільтрації. Згідно проведених досліджень необхідно розробити модель для автоматичної генерації та виделення некластерних індексів.

По-перше, необхідно створити налаштування яке довзолить вмикати та вимикати даний функціонал, якщо розробник програмного забезпечення хоче отримати повний контроль над схемою бази даних.

По-друге, внаслідок додавання індексів уповільнюється операція додавання даних до таблиці – необхідно створити статичний клас для конфігурування відношення операції проекції до операції додавання даних. Це дозволить не нашкодити таблицям, в які досить часто постачаються нові дані. Необхідно встановити співвідношення 1 до 8. Тобто, одна операція додавання до восьми операцій проекції. Аналізатор для операції проекції повинен проаналізувати, чи відповідає поточне співвідношення вказаному налаштуваннях. Якщо дане співвідношення буде зберігатись на протязі 1000 запитів, то буде створений некластерний індекс для атрибутів фільтрації. Проте, якщо співвідношення буде порушено – некластерний індекс буде видалений.

5.4.2. Реалізація моделі аналізу та побудування SQL запиту для операції перехресного пошуку даних

Для підвищення ефективності виконання SQL запиту перехресного пошуку даних згідно розділу №4.2. необхідно розробити аналізатор та побудовник SQL запитів для можливої модифікації. Надається 3 моделі для реалізації різного виконання перехресного пошуку даних.

1. Перехресний пошук даних за індексами. Внаслідок використання реліаційної алгребри СУБД реалізують окремий алгоритм для виконання операції з’єднання над таблицями із індексами. Необхідно розробити аналізатор, який зможе проаналізувати атрибути таблиць та визначити чи є в них обох в наявності індекси. Якщо вони є – необхідно модифікувати SQL запит з цілью додати у якості умови для з’єднання індексовані атрибути. Це значно підвищить швидкість обробки операції перехресного з’єднання.
2. Перехресний пошук даних з наявністю операцій агрегації. Реалізація аналізатора SQL запиту повина шукати в об’єкті запиту наявність агрегованих операторів. При наявності даних операторів далі аналізуватиме необхідність проведення перехресного з’єднання за даним алгоритмом. За основу потрібно взяти кількість агрегатів та атрибутів для групування та визначення складності функцій агрегатів. Якщо кількість атрибутів для групування буде більше за 5 елементів – це збільшує вірогідність використання цього способу на 35%. Більше за 10 елементів – збільшує вірогідність на 55%. Якщо використовуються агрегатні функції Sum, Avg – це додає 20% за кожну. Якщо Count, Min, Max – це додасть по 10% за кожну. Коли коефіцієнт буде більше за 0.75 (75%) – тоді буде використаний даний спосіб перехресного з’єднання. Він включає в себе подальше вилучення виконання агрегатних функцій на вкладений запит проекції. Який матиме в якості атрибутів результати виконання агрегатних функцій та елемент, який використовувався в умові для перехресного пошуку. Результат викнання вкладеного запиту повинен буде з’єднаний із зовнішнім запитом. Зовнішній запит все також повинен виконати операцію групування необхідних атрибутів.
3. Складний перехресний пошук даних. Якщо SQL запит для перехресного пошуку має більше, ніж 10 операторів для виконання – необхідно модифікувати SQL запит для явного використання оператору HASH JOIN. Даний оператор

вбудований в СУБД SQL Server, що дозволить не затрачувати багато зусиль на зміну SQL запиту, а лише замінити використання оператору JOIN на його хеш версію.

5.5. Розробка розширень для масового оновлення, видалення та додавання даних до таблиць

Для розробки розширень масового оновлення, видалення та додавання даних необхідно перевизначити в класі «NDbSet<T>» методи «Add», «Update» та «Remove». Згідно, з представленим алгоритмом у розділі №4.3. у перевизначені методи необхідно додати логіку для аналізу та побудування SQL запиту для оновлення, видалення та додавання даних партіями. Для цього до «NDbSet<T>» необхідно додати впровадження залежності «ISqlAnalyzerService». Необхідно створити налаштування, яке буде вказувати на порогове значення для максимальної кількості елементів та параметрів однієї партії SQL запиту. Виклик операції для аналізу та побудування SQL запиту буде виконуватися лише за умови, що кількість сутностей для оновлення, видалення або додавання даних досягне створеного налаштування. Реалізація аналізатору полягає в отримані інформації про кількість параметрів всіх елементів сутностей, стан яких (Modefied, Removed, Added) перейде до виконання на стороні бази даних. Підрахунок усіх параметрів проводиться через аналіз об'єкту SQL запиту. Далі побудовник SQL запиту генерує єдиний пакет оновлення, видалення чи додавання даних до таблиці. Це досягається за допомогою виклику методів об'єкту «SqlBuilder», який влаштований в Entity Framework Core. Якщо кількість параметрів не влазить до одного пакету, то для залишених елементів сутностей сформується новий пакет. Для якого будуть повторені ті ж самі дії. Після отримання необхідних пакетів SQL запитів, вони будуть направлені на виконання до методу «Execute<T>» інтерфейсу «IQueryExecutionLogic». Який в свою чергу через фасад «SqlExecutorLogic» виконає SQL запити.

5.6. Розробка розширень для масового оновлення, видалення та додавання даних до таблиць пакетами

Для розробки розширень масового оновлення, видалення та додавання даних пакетами необхідно створити в класі «NDbSet<T>» наступні методи: «UpdateBatch», «RemoveBatch», «AddBatch». Кожен з цих методів буде приймати в якості вхідного параметру масив елементів сутностей для проведення операції масового оновлення, видалення та додавання даних. Аналізатор для обробки створеного SQL запиту використовує обробку та модифікацію запиту описану в попередньому розділі. Виконання сформованих пакетів, також, використовує виклик методу «Execute<T>» інтерфейсу «IQueryExecutionLogic». Єдина відмінність від попереднього розділу в тому, що в цьому випадку не проходить жодних перевірок на кількість отриманих сутностей та їх параметрів. Методи відразу отримують великий обсяг елементів сутностей, які розподіляються по пакетам SQL запитів.

5.7. Публікація програмного забезпечення у вигляді NuGet пакету

Необхідно опублікувати створений набір біблотек розширень ORM технології Entity Framework Core [24]. Це дозволить спросити установку та використання розширення. Необхідно виконати наступний алгоритм дій:

1. Для початку, необхідно створити маніфест для біблотек (опис, версії, копірайти, ліцензції, торгова марка та ідентифікатор).
2. Наступним кроком буде завантаження з офіційного сайту Microsoft додатку під назвою «nuget.exe».
3. Необхідно створити власний акаунт на сайті «nuget.org» для подальшої публікаї пакетів.

4. За допомогою встановленого додатку виконати команду «nuget spec». Це створить спеціальний файл з розширенням «.nuspec». Це необхідно для включення спеціальних токенів заміни та отримання відомостей про проект.
5. Потрібно перейти до папки, яка зберігає файл з розширенням «.nuspec» та виконати команду «nuget pack» за допомогою встановленого додатку «nuget.exe». Ця дія створить файл з розширенням «.nupkg», що і являється NuGet пакетом.
6. Отримання API ключа за допомогою створеного акаунту на сайті.
7. Публікація створеного NuGet пакету та встановлення йому прав доступності.

Перед публікацію пакетів, вони всі перевіряються на віруси. У випадку, якщо вони будуть знайдені – публікація буде відхилина.

Щоб отримати API ключ необхідно виконати наступні кроки:

1. Увійти до свого акаунту до nuget.org та вибрати ім’я свого користувача.
2. Вибрати пункт меню «API ключі».
3. Натиснути на команду Create (Створити), ввести ім’я ключа, після чого вибрати елементи Select Scopes (Вибір областей) та натиснути Push (Надіслати). Далі необхідно написати до поля Glob pattern (Стандартна маска), після чого натисніть на клавішу Create (Створити).
4. Після створення API ключа натисність на клавішу Copy (Копіювати) для отримання ключа доступа, який буде потрібен для введення в інтерфейсі командного рядка.

Ключі мають термін дії. Крім того, їх можна прив'язати до певних пакетів (або стандартних масок). Кожен ключ прив'язаний до конкретних операцій: надсилання нових пакетів та оновлень, надсилання лише оновлень або видалення зі списку. Використовуючи визначення області, можна створювати ключі API різним користувачам, які керують пакетами. Це дозволить надавати їм лише потрібні дозволи.

6. ТЕСТУВАННЯ

6.1. Тестування ПЗ на SQL ін'єкції

Для тестування програмного забезпечення на вразливість до атак типу SQL ін'єкція було розроблено консольний додаток. Даний консольний додаток просить ввести логін та пароль для авторизації адміністратора в системі. Якщо логін та пароль співпадуть – користувач буде наділений правами адміністратора та отримає повідомлення «Welcome to admin panel». Якщо логін або пароль буде введено невірно, користувач отримає повідомлення «Login or password are wrong».

Для тестування використаємо наступну SQL ін'єкцію: «admin' OR 1=1--». В якості значення для поля паролю буде ведений рядок «123». Тобто, якщо SQL ін'єкція спрацює, то на екран консолі буде відображене повідомлення щодо успішнлї авторизації. Якщо система буде захищена від атак SQL ін'єкцій, то повідомлення, що логін та пароль невірні.

На рисунку 6.1. зображено, що при введенні SQL ін'єкції до поля логін та будь-якого значення паролю система не авторизувала користувача. Це демонструє, що система захищена від даного типу атак.

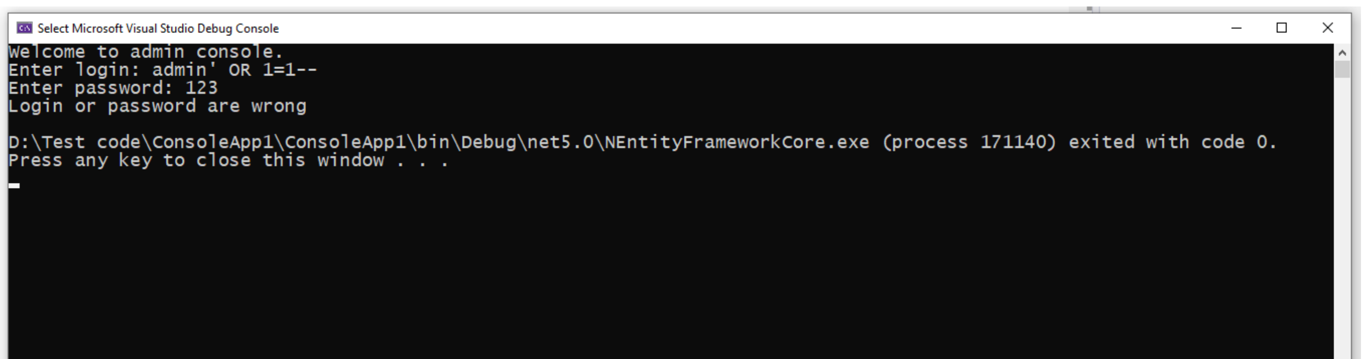


Рис. 6.1. – Результат тестування на SQL ін'єкції

6.2. Тестування програмного забезпечення

Для оцінки якості програмного забезпечення були розроблені критерії оцінювання та спеціальний алгоритм тестування з підготовленими Linq запитами. Їх необхідно виконати визначену кількість разів над декількома порціями даних. Детальний процес проведення тестування описаний у розділі №3.4.2.

Першим запитом для тестування буде операція проекції. Необхідно провести вибірку із тестових таблиць: «EmployeeDetails», «Employees» та «Salaries».

Таблиця 6.1. – Тестування виконання запиту проеції. Вказане середню значення для обробки всіх таблиць

	Entity Framework Core		**Розширення за допомогою T-SQL**	
К-ть елементів:	1000	400 000	1000	400 000
1 виклик в сек	2,16	9,81	1,51	7,86
2 виклик в сек	2,34	9,24	1,79	7,79
3 виклик в сек	2,29	9,19	1,61	7,87

Згідно із зазначеними результатами тестування в таблиці 6.1. можна спостерігати, що розширення ORM технології Entity Framework Core за допомогою T-SQL – привело до незначного пришвидшення виконання операції проекції над малим об’ємом даних у 1000 елементів сутностей. Над великим об’ємом, 400 000 елементів сутностей, розширення привело до пришвидшення в один раз. Можна зробити висновок, що за допомогою впровадження аналізаторів та модифікаторів SQL запитів, виконання операції проекції пришвидшується. Згідно вказаних критеріїв оцінювання можна надати оцінку «Задовільно».

Наступне тестування буде проводитись над операцією перехресного пошуку даних. В контексті цієї роботи було представлено три різних варіанти SQL операції перехресного пошуку в залежності від обставин вихідного SQL запиту.

Таблиця 6.2. – Тестування виконання перехресного пошуку за індексами

	Entity Framework Core		**Розширення за допомогою T-SQL**	
К-ть елементів:	1000	400 000	1000	400 000
1 виклик в сек	2,94	11,54	1,11	6,41
2 виклик в сек	2,84	10,94	1,79	5,82
3 виклик в сек	2,75	11,13	1,81	6,24

Згідно із зазначеними результатами тестування в таблиці 6.2. спостерігаємо, що розширення ORM технології Entity Framework Core за допомогою T-SQL – привело до незначного пришвидшення виконання над малим об’ємом даних у 1000 елементів сутностей. Проте, над великим об’ємом даних, у 400 000 елементів сутностей – пришвидшується виконання майже в 2 рази. Можна зробити висновок, що впровадження аналізаторів та модифікаторів SQL запитів, які модифікують запит додаванням до обробки умови індексованих атрибутів – виконання операції перехресного пошуку даних за індексами значно пришвидшується. Можливо, при більшому наборі даних (1 000 000 елементів сутностей та більше) дана оптимізація перехресного пошуку за індексами спрацює краще. Згідно вказаних критеріїв оцінювання можна надати оцінку «Успішно».

Таблиця 6.3. – Тестування виконання перехресного пошуку з агрегованими даними

	Entity Framework Core		**Розширення за допомогою T-SQL**	
К-ть елементів:	1000	400 000	1000	400 000
1 виклик в сек	3,12	32,74	2,75	27,16
2 виклик в сек	3,24	30,56	2,69	26,82
3 виклик в сек	3,16	31,62	2,71	26,97

На відміну від попердніх проведених тестувань, згідно із зазначеними результатами в таблиці 6.3. можна бачити не досить успішний результат. Розширення ORM технології Entity Framework Core за допомогою T-SQL – привело до надзвичайно незначного пришвидшення, як при невеликій кількості елементів сутностей (1000), так і при великому об’ємі (400 000). Можна зробити висновок, що впровадження аналізаторів та модифікаторів SQL запитів, які модифікують запит розділенням виконання агрегованих функцій та перехресного пошуку на вкладений та зовнішній запит відповідно – пришвидшує роботу приблизно на 20%. Це незначний результат оптимізації, проте, по критеріям оцінювання він отримає оцінку «Задовільно»..

Таблиця 6.4. – Тестування виконання перехресного пошуку з використанням хеш-з’єднання

	Entity Framework Core		**Розширення за допомогою T-SQL**	
К-ть елементів:	1000	400 000	1000	400 000
1 виклик в сек	2,16	17,21	1,85	13,86
2 виклик в сек	2,34	17,02	1,79	13,60
3 виклик в сек	2,29	17,13	1,81	14,08

Згідно із зазначеними результатами тестування в таблиці 6.4. спостерігаємо, що розширення – привело до незначного пришвидшення виконання над малим об'ємом даних у 1000 елементів сутностей. Над великим об'ємом даних, у 400 000 елементів сутностей можна спостерігати такий же результат. Тобто, пришвидшення приблизно в 1 раз. Можна зробити висновок, що впровадження аналізаторів та модифікаторів SQL запитів, які модифікують запит додаванням хеш-з'єднання – пришвидшують обробку складних запитів, які мають більше 10 операторів. Згідно вказаних критеріїв оцінювання можна надати оцінку «Задовільно».

Таблиця 6.5. – Тестування виконання додавання даних до таблиці різними запитами

	Entity Framework Core		**Розширення за допомогою T-SQL**	
К-ть елементів:	1000	400 000	1000	400 000
1 виклик в сек	2,11	328,61	1,50	176,11
2 виклик в сек	2,09	329,02	1,44	169,87
3 виклик в сек	2,14	326,16	1,40	172,46

Згідно із зазначеними результатами тестування в таблиці 6.5. спостерігаємо, що розширення – привело до значного пришвидшення виконання операції додавання даних до таблиць як з невиликом об'ємом даних (1000 елементів сутностей), так і з великим (400 000 елементів сутностей). Тобто, пришвидшення приблизно в 2 рази. Можна зробити висновок, що впровадження аналізаторів та модифікаторів SQL запитів, які модифікують запит додавання елементів до таблиць за допомогою групування визначної кількості даних для додавання в один пакет даних значно пришвидшують процес. Згідно вказаних критеріїв оцінювання можна надати оцінку «Успішно».

Таблиця 6.6. – Тестування виконання додавання даних до таблиці пакетами

	Entity Framework Core		**Розширення за допомогою T-SQL**	
К-ть елементів:	1000	400 000	1000	400 000
1 виклик в сек	1,09	помилка	1,15	157,24
2 виклик в сек	1,17	помилка	1,19	162,44
3 виклик в сек	1,14	помилка	1,12	160,19

Згідно із зазначеними результатами тестування в таблиці 6.5. спостерігаємо, що розширення – привело до значного пришвидшення виконання операції додавання даних до таблиць з великим обсягом даних (400 000 елементів сутностей). Робота операції додавання даних до таблиць пакетами має майже рівний результат. Проте, додавання даних до таблиць пакетами у великому обсязі даних від Entity Framework Core, згідно стандартних налаштувань серверу, виконалось з помилкою. Ця помилка виникла внаслідок неможливості отримання такої великої порції за один раз. Можна зробити висновок, що впровадження аналізаторів та модифікаторів SQL запитів, які модифікують запит додавання елементів до таблиць за допомогою пакетів значно пришвидшують процес. Згідно вказаних критеріїв оцінювання можна надати оцінку «Успішно», назважаючи на результат обробки малого об’єму даних (1000 елементів сутностей). Внаслідок аналізу виконуваних раніше SQL запитів уповільнюється час на обробку. Тому, при тестуванні не невеликому обсязі даних продуктивність знаходиться майже на тому же рівні, що і робота без розширень. Проте, якщо виключити можливість аналізу для пакетних операцій – вони будуть працювати значно швидше.

ВИСНОВКИ

Для оптимізації процесу створення та виконання SQL запитів було розроблене програмне забезпечення. Воно розширює стандартний функціонал ORM технології Entity Framework Core за допомогою використання мови SQL запитів та її процедурного розширення T-SQL. Програмне забезпечення складається з набору бібліотек, які залежать одна від одної. Всі створені бібліотеки були опубліковані у вигляді NuGet пакету.

Програмне забезпечення аналізує створюванні розробником запити, створювані за допомогою мови запитів Linq. Аналізуємими є SQL запити проекції, перехресного пошуку, оновлення, видалення та додавання даних. Для кожного із запитів був створений окремий аналізатор, який вміє розділяти та розпізнавати кожну частину створеного запиту. Для модифікації SQL запитів були створені спеціальні об’єкти побудовники. Кожен побудовник SQL запиту вміє модифікувати конкретний тип SQL операції з метою його оптимізації для більш ефективного виконання. Для розробки аналізаторів та побудовників SQL запитів були використані моделі та спеціальні алгоритми.

Для впровадження ефективності виконання SQL запитів було додано до системе можливість зберігання аналітичної інформації. Аналітичні дані використовуються з метою пошуку вже виконаних схожих запитів, щоб не запускати повний механізм аналізу та побудування запиту. Адже, вже готовий та модифікований варіант запиту готовий. Лише потрібно провести заміну параметрів для кожного конкретного випадку. Для вбудування створених аналітичних таблиць були створені спеціальні об’єкти «NDbContext» та «NDbSet<T>». Перший перевизначає ряд свого функціоналу для можливості виконання відправки SQL операцій пакетами та має доступ до сутностей, які представляють аналітичні таблиці. Другий перевизначає методи оновлення, видалення та додавання даних, щоб обновити функціонал для виконання пакетної віправки SQL операцій. Також, був створений функціонал для прямого

пакетного виконання SQL операцій оновлення, видалення та додання даних до таблиць без отримання помилок та з оптимізаціює швидкості виконання.

ПЕРЕЛІК ПОСИЛАНЬ

1. entityframework-plus [Електронний ресурс]. – Режим доступу: https://entityframework-plus.net/ – Entity Framework Plus. A FREE & Open Source library to enhance EF6 and EF Core.
2. Makers institute gitbooks [Електронний ресурс]. – Режим доступу: https://makersinstitute.gitbooks.io/asp-net-mvc/content/net-project-3/importing-sql-tables-into-visual-studio/mapping-the-tables.html – Object Relational Mapping
3. IBM [Електронний ресурс]. – Режим доступу: https://www.ibm.com/cloud/learn/relational-databases – Relational Databases
4. Thoughtfulcode [Електронний ресурс]. – Режим доступу: https://www.thoughtfulcode.com/orm-active-record-vs-data-mapper/ – ORM Patterns: The Trade-Offs of Active Record and Data Mappers for Object-Relational Mapping
5. Karoldabrowski [Електронний ресурс]. – Режим доступу: https://karoldabrowski.com/blog/active-record-pattern-or-anti-pattern-overview/ – Active Record pattern (or anti-pattern) – overview
6. Martin Fowler blog [Електронний ресурс]. – Режим доступу: https://martinfowler.com/eaaCatalog/dataMapper.html – Data Mapper
7. Tutorialspoint [Електронний ресурс]. – Режим доступу: https://www.tutorialspoint.com/design_pattern/data_access_object_pattern.htm – Data Access Object Pattern
8. Baeldung [Електронний ресурс]. – Режим доступу: https://www.baeldung.com/java-dto-pattern – The DTO Pattern (Data Transfer Object)
9. Altexsoft [Електронний ресурс]. – Режим доступу: https://www.altexsoft.com/blog/object-relational-mapping/ – Understanding Object-Relational Mapping: Pros, Cons, and Types
10. Microsoft docs [Електронний ресурс]. – Режим доступу: https://docs.microsoft.com/en-us/ef/core/ – Entity Framework Core
11. Microsoft docs [Електронний ресурс]. – Режим доступу: https://docs.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/infrastructure-persistence-layer-design – Design the infrastructure persistence layer – The Repository pattern
12. Microsoft docs [Електронний ресурс]. – Режим доступу: https://docs.microsoft.com/en-us/aspnet/mvc/overview/older-versions/getting-started-with-ef-5-using-mvc-4/implementing-the-repository-and-unit-of-work-patterns-in-an-asp-net-mvc-application – Implementing the Repository and Unit of

Work Patterns in an ASP.NET MVC Application (9 of 10) - Creating the Unit of Work Class

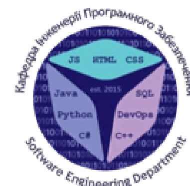
13. Microsoft docs [Електронний ресурс]. – Режим доступу: https://docs.microsoft.com/en-us/sql/t-sql/language-elements/transactions-transact-sql?view=sql-server-ver15 – Transactions (Transact-SQL)
14. Microsoft docs [Електронний ресурс]. – Режим доступу: https://docs.microsoft.com/en-us/dotnet/csharp/linq/ – Language Integrated Query (LINQ)
15. Microsoft docs [Електронний ресурс]. – Режим доступу: https://docs.microsoft.com/en-us/dotnet/framework/data/adonet/ef/language-reference/linq-to-entities – LINQ to Entities
16. Dapper tutorial [Електронний ресурс]. – Режим доступу: https://dapper-tutorial.net/dapper – Dapper
17. Kagawacode [Електронний ресурс]. – Режим доступу: https://kagawacode.medium.com/speed-benchmark-dapper-vs-ef-core-3-f3777b76dd0 – Speed Benchmark — Dapper vs EF Core
18. Portswigger [Електронний ресурс]. – Режим доступу: https://portswigger.net/web-security/sql-injection – SQL injection
19. Microsoft docs [Електронний ресурс]. – Режим доступу: https://docs.microsoft.com/en-us/dotnet/api/system.data.sqlclient.sqlparameter?view=dotnet-plat-ext-6.0 – SqlParameter Class
20. IUSB [Електронний ресурс]. – Режим доступу: https://clas.iusb.edu/computer-science-informatics/research/reports/TR-20080105-1.pdf – Introduction to Query Processing and Optimization
21. IJCSMC [Електронний ресурс]. – Режим доступу: https://www.ijcsmc.com/docs/papers/March2016/V5I3201604.pdf – Roll of Relational Algebra and Query Optimizer in Different Types of DBMS
22. Microsoft docs [Електронний ресурс]. – Режим доступу: https://docs.microsoft.com/en-us/sql/relational-databases/performance/joins?view=sql-server-ver15 – Joins (SQL Server)
23. Microsoft docs [Електронний ресурс]. – Режим доступу: https://docs.microsoft.com/en-us/nuget/quickstart/install-and-use-a-package-in-visual-studio – Quickstart: Install and use a package in Visual Studio (Windows only)
24. Microsoft docs [Електронний ресурс]. – Режим доступу: https://docs.microsoft.com/en-us/nuget/quickstart/create-and-publish-a-package-using-visual-studio?tabs=netcore-cli – Quickstart: Create and publish a NuGet package using Visual Studio (.NET Standard, Windows only)

ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ТЕЛЕКОМУНІКАЦІЙ

НАВЧАЛЬНО–НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Кафедра інженерії програмного забезпечення



Магістерська робота

«Оптимізація ORM технологій Entity Framework Core за допомогою розширення T-SQL»

Виконав: студент групи ПДМ-61,
Гнатюк Владислав Іванович
Керівник: директор ННІІТ, професор,
доктор технічних наук
Бондарчук Андрій Петрович

Київ - 2021

МЕТА, ОБ'ЄКТА ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ 2

Мета роботи: оптимізація швидкості та зменшення витрат пам'яті на виконання SQL запиту.

Об'єкт дослідження: розширення функціоналу ORM технології Entity Framework Core за допомогою аналізу та модифікації SQL запитів.

Предмет дослідження: програмне забезпечення для розширення функціоналу ORM технологієї Entity Framework Core за допомогою аналізу та модифікації SQL запитів.

РОЗГЛЯНУТІ МОДЕЛІ ТА АЛГОРИТМИ

Формули моделі аналізу ефективності таблиці

- Обхід бінарного дерева:
  $h + 1$

- Бінарний пошук з наявністю кластерного індексу:
  $[lg(a_r)]$

Формули моделі для аналізу ефективності з’єднань

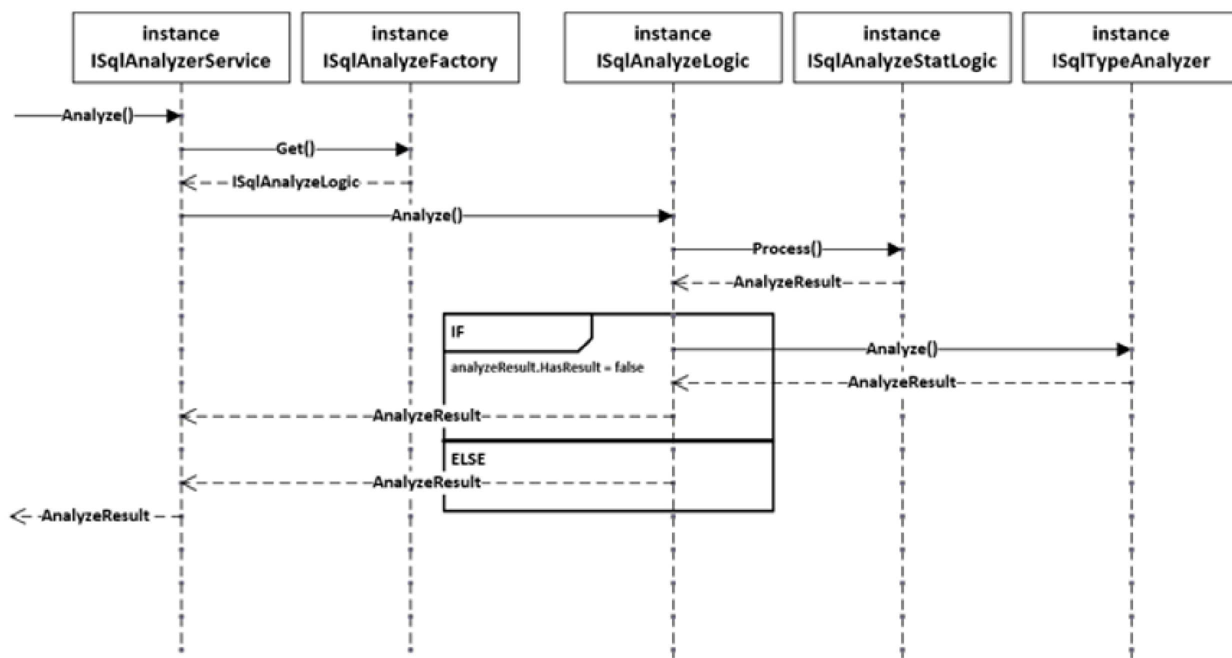
- Алгоритм реліаційної алгебри:

$$R3 = join(R1, Ai, R2, Bj)$$
$$for\ each\ tuple\ t\ in\ R1\ do$$
$$for\ each\ tuple\ s\ in\ R2\ at\ index(t.Ai)\ do$$
$$insert(R3, t.A1, t.A2, \dots, t.AN,$$
$$s.B1, \dots, s.B(j-1), s.B(j+1), \dots, s.BM)$$

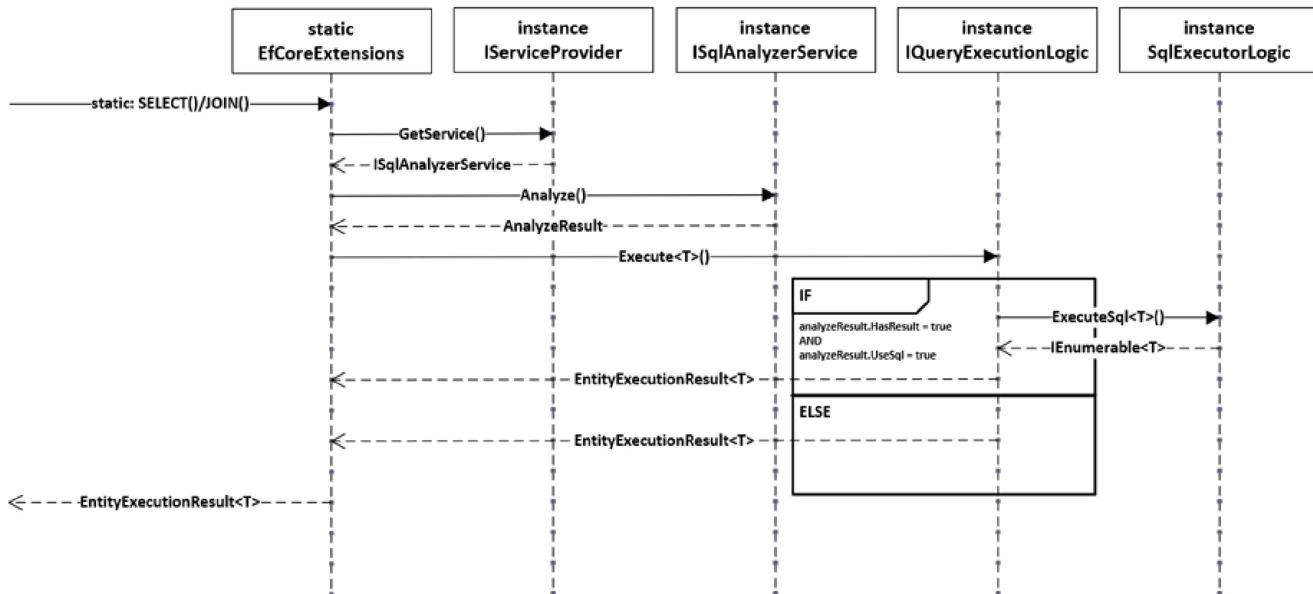
ПОРІВНЯЛЬНА ХАРАКТЕРИСТИКА ІСНУЮЧИХ ОПТИМІЗАЦІЙ ORM СИСТЕМ

	Плюси	**Мінуси**
Entity Framework Plus	Логування кожного запиту з коробки. Кешує хто, коли та які сутності змінював.	Відсутні алгоритми модифікації збережених процедур та функцій користувача
	Влаштований алгоритм кешування даних	Виконання лише DML запитів на стороні серверу
	Алгоритм створення тригеру в пам'яті додатку для подальших запитів	Не підтримує наслідування EF
	Підтримує роботу наборів	Залежність від використаного коду
NEF Core	Налаштування для використання асинхронних запитів	Відсутність динамічної генерації виразів LINQ
	Алгоритм виконання операцій пакетами	Відсутність алгоритму фільтрування залежних сутностей
	Алгоритм заміщення запитів за допомогою T-SQL	Відсутність алгоритму накладення фільтрів на сутності
	Кешування виконаних SQL запитів	Не підтримує синхронізацію стану глобальних налаштувань серверу

АРХІТЕКТУРА СИСТЕМИ АНАЛІЗУ SQL ЗАПИТУ



АРХІТЕКТУРА РОЗШИРЕННЯ ПРОЕКЦІЇ ТА З’ЄДНАНЬ



АРХІТЕКТУРА ДОДАВАННЯ, ОНОВЛЕННЯ ТА ВИДАЛЕННЯ ДАНИХ

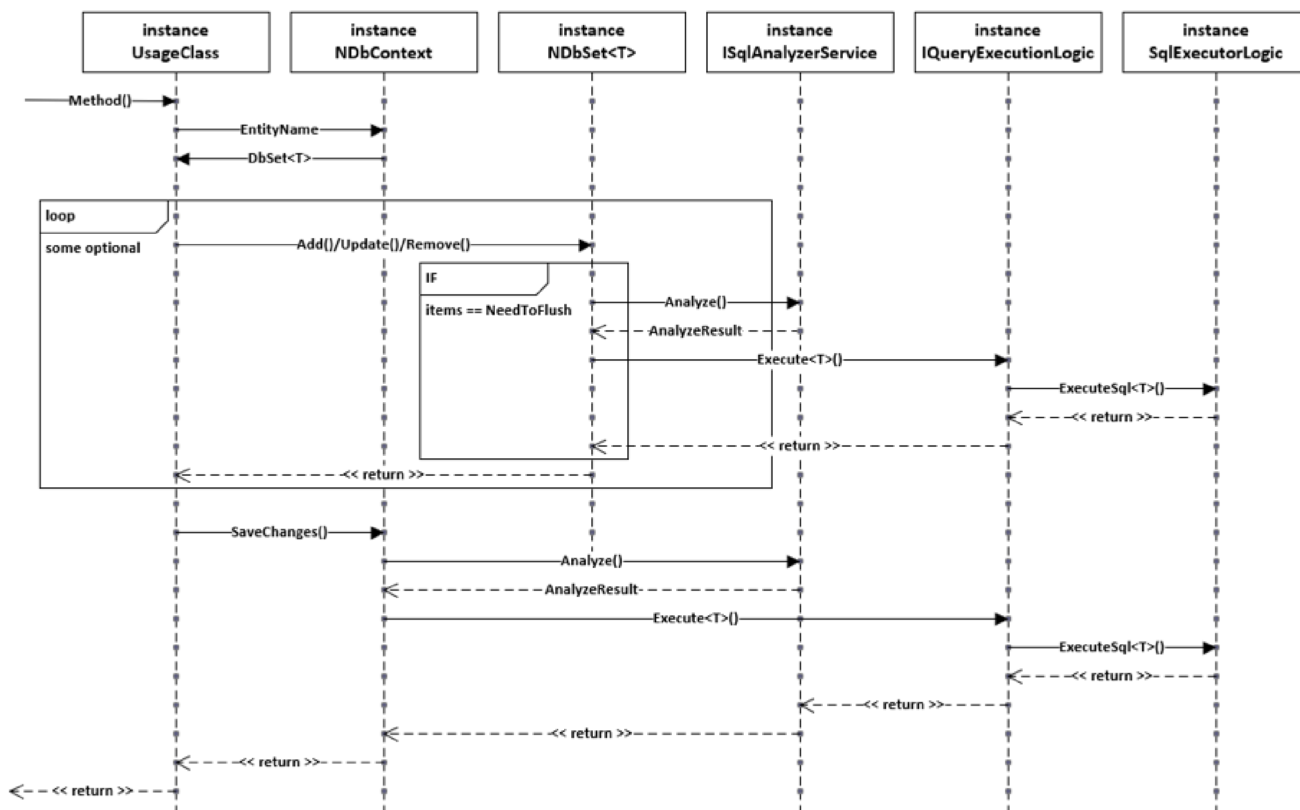
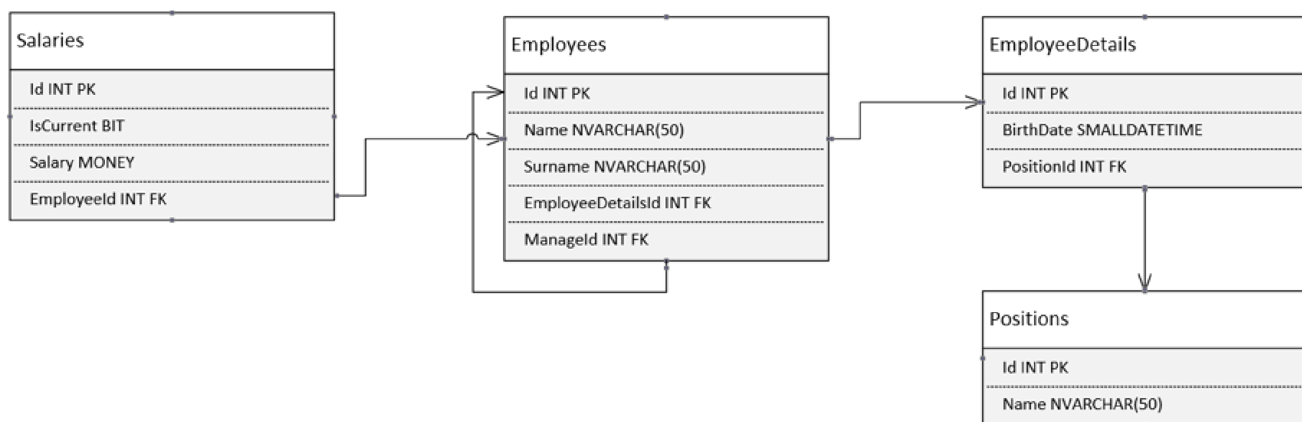
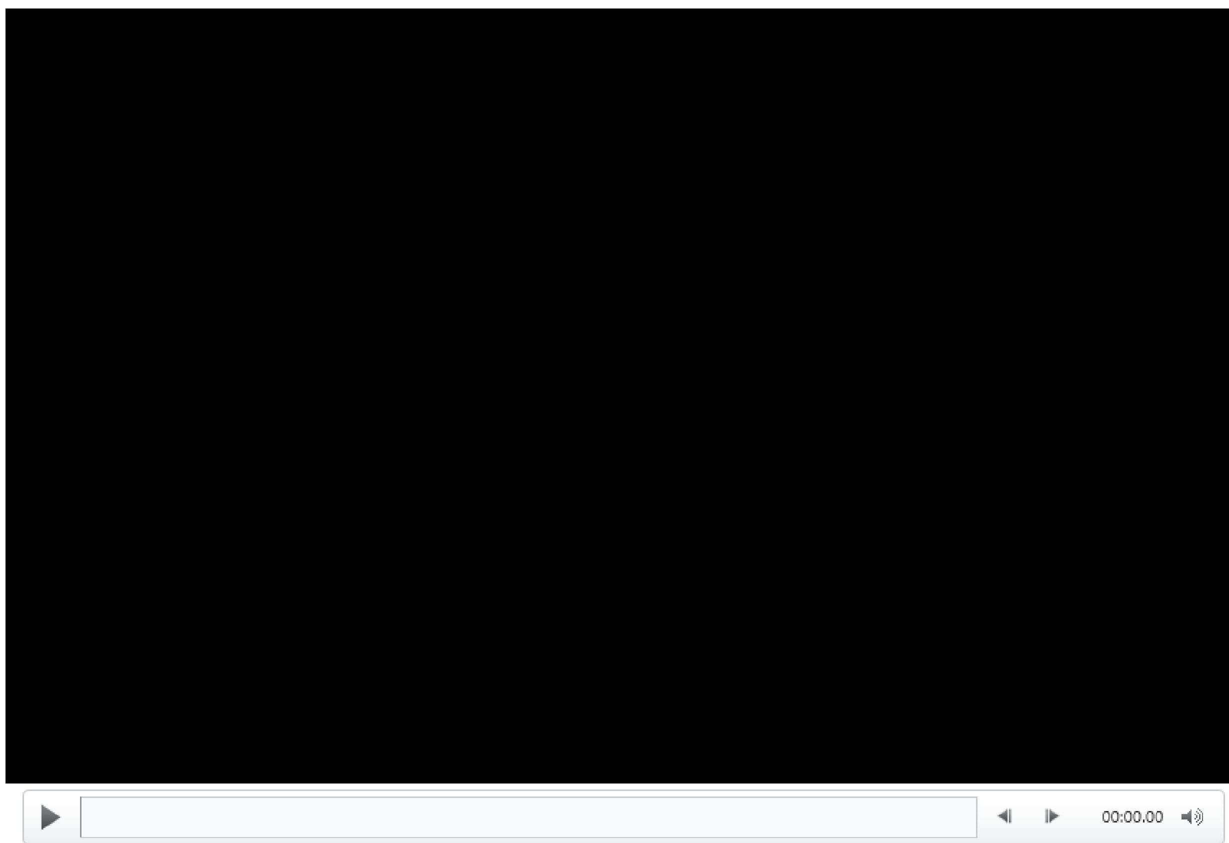


СХЕМА ТЕСТОВОЇ БАЗИ ДАНИХ



ДЕМОНСТРАЦІЯ З ТЕСТУВАННЯМ



ВИСНОВКИ

1. Створення алгоритмів для аналізу виконуваних SQL запитів з метою їх подальшого повторного використання суттєво може прискорити результат при виконанні складних запитів.
2. Адаптація моделей реліаційної алгебри для модифікації операцій з'єднання прискорює роботу складних запитів та запитів вибірки із з’єднаннями.
3. Розробка алгоритму для пакетної передачі даних з інтеграцією в ORM технологію у вигляді розширень дозволяє пришвидшити виконання операцій оновлення, видалення та додавання даних таблиць.

ПУБЛІКАЦІЇ ТА АПРОБАЦІЯ

Статті:

1. Гнатюк В.І. Оптимізація ORM технологій Entity Framework Core за допомогою розширення T-SQL // Зв'язок. №4 (146), 2021. Подана до розгляду.

Тези доповідей:

1. Гнатюк В.І. Оптимізація ORM технологій Entity Framework Core за допомогою розширення T-SQL. // XIII Науково-технічна конференція «Сучасні інфокомунікаційні технології», 10 грудня 2021 року, Державний університет телекомунікації, Київ, Україна.

ДЯКУЮ ЗА УВАГУ!