МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ВОЛИНСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ЛЕСІ УКРАЇНКИ
Кафедра комп'ютерних наук та кібербезпеки

На правах рукопису

**ТАНЬ ЧЕНЛУН**

# ПОРІВНЯННЯ ЕФЕКТИВНОСТІ ТА ФУНКЦІОНАЛЬНИХ МОЖЛИВОСТЕЙ ORM-СИСТЕМ У .NET

Спеціальність: 122 Комп'ютерні науки
Освітньо-професійна програма: Комп'ютерні науки та інформаційні технології Кваліфікаційна робота на здобуття освітнього ступеня «магістр»

Науковий керівник:
Булатецький Віталій Вікторович,
Доцент кафедри комп'ютерних наук та кібербезпеки, кандидат фізико-математичних наук.

РЕКОМЕНДОВАНО ДО ЗАХИСТУ
Протокол №_____ засідання кафедри
комп'ютерних наук та кібербезпеки
від _____ 2025 р.
Завідувач кафедри
(_____) Гришанович Т. О.

ЛУЦЬК – 2025

**ЗМІСТ**

# ВСТУП

**Актуальність теми.** Сучасні програмні системи працюють із великими обсягами даних і потребують високопродуктивних та надійних механізмів доступу до реляційних баз даних. На платформі .NET існує кілька ORM-систем, які суттєво відрізняються за продуктивністю, можливостями налаштування, рівнем абстракції та зручністю використання, що безпосередньо впливає на ефективність розробки та експлуатації програмних продуктів. Неправильний вибір ORM-фреймворку може призвести до значного зниження швидкодії, надмірного споживання пам’яті та процесорних ресурсів, складнощів у підтримці та масштабуванні застосунку, а також ускладнити інтеграцію з іншими компонентами системи.

Особливо критичним це стає для корпоративних додатків та високонавантажених сервісів, де ефективність роботи із базою даних прямо впливає на бізнес-процеси, швидкість обробки запитів користувачів та рівень задоволеності кінцевих клієнтів. Порівняння ефективності та функціональних можливостей популярних ORM-рішень дозволяє розробникам обґрунтовано обирати інструменти, які відповідають конкретним вимогам проєкту – від швидкості читання та обробки масових даних до простоти впровадження бізнес-логіки та управління транзакціями.

Дослідження цієї теми має значну практичну цінність, оскільки дозволяє підвищити продуктивність, масштабованість та якість програмних продуктів на платформі .NET, оптимізувати використання ресурсів і скоротити час розробки. Крім того, правильний вибір ORM забезпечує більшу гнучкість при інтеграції з сучасними архітектурними підходами, такими як мікросервіси та CQRS, що є важливим аспектом для розробки стійких та ефективних корпоративних систем.

**Метою роботи** є виконання порівняльного аналізу ефективності та функціональних можливостей найпоширеніших ORM-систем у середовищі .NET з подальшим визначенням оптимальних сценаріїв їх використання відповідно до вимог щодо продуктивності, гнучкості та архітектурних особливостей програмних застосунків.

**Для досягнення мети потрібно виконати наступні завдання:**

- Провести огляд сучасних ORM-фреймворків для .NET, визначити їхні ключові особливості та сфери застосування;
- Провести огляд сучасних ORM-фреймворків для .NET, визначити їхні ключові особливості та сфери застосування;
- розробити тестові сценарії для вимірювання продуктивності CRUD-операцій, складних запитів та роботи з транзакціями;
- gровести експериментальні вимірювання, зібрати ключові метрики (час відгуку, використання пам’яті, навантаження на CPU;
- gорівняти продуктивність і ефективність обраних ORM та оцінити накладні витрати на підтримку коду;
- визначити переваги, недоліки та специфічні області застосування кожного ORM-підходу;
- розробити практичні рекомендації щодо вибору ORM для різних типів систем, включно з гібридним підходом.

**Об’єктом дослідження:** засоби взаємодії програмних застосунків .NET із реляційними базами даних за допомогою ORM-систем.

**Предмет дослідження:** ефективність роботи та функціональні можливості ORM-систем у середовищі .NET, а також критерії їх порівняння й практичного вибору для розробки програмних застосунків.

**Наукова новизна.** У роботі проведено системний порівняльний аналіз ORM-фреймворків у .NET з одночасним урахуванням їхньої продуктивності, архітектурних рішень та функціональних можливостей у єдиній методологічній моделі оцінювання.

Отримані результати формують практичні рекомендації, які можуть використовуватися розробниками й аналітиками для прийняття обґрунтованого рішення при виборі технології доступу до даних.

# РОЗДІЛ 1
# ОСНОВИ ТА ПРИНЦИПИ ВИКОРИСТАННЯ ORM У .NET

## 1.1. Управління ризиками під час роботи з базами даних

### 1.1.1. Основні категорії ризиків у роботі з реляційними базами даних

У сучасній розробці корпоративних додатків бази даних як основний компонент зберігання та управління інформацією безпосередньо впливають на бізнес-безперебійність та інформаційну безпеку підприємства. Управління ризиками підприємства забезпечує системні рамки для ідентифікації, оцінки та запобігання ризикам у роботі з базами даних.

**Вимоги безпеки.** Ін’єкційні атаки SQL є одним з найпоширеніших та найнебезпечніших ризиків для безпеки баз даних. Згідно з звітом про безпеку OWASP за 2023 рік, ін’єкційні атаки досі входять до десятки найбільших ризиків безпеки вебдодатків [2,3]. Традиційний спосіб написання SQL шляхом конкатенації рядків надає зловмисникам можливості для атак:

```
// Приклад ризикового коду - вразливість до SQL-ін’єкції
string sql = "SELECT * FROM Users WHERE UserName = ‘" +
userName + "‘"; SqlCommand command = new SqlCommand(sql,
connection);
```

У наведеному коді, якщо параметр userName є зловмисним введенням "admin’ OR ‘1’=‘1", умова запиту буде сфальшована, що призведе до витоку даних. ORM усуває цей ризик через параметризовані запити та строгу типізацію [4]. Використання ORM, такого як ActiveRecord для Ruby on Rails або Sequel, може автоматично попередити більшість SQL-ін’єкцій, оскільки вони генерують запити безпечним чином [5]. Крім того ORM-системи мають вбудовані засоби захисту від атак SQL-ін’єкцій – поширеної зловмисної активності, коли хакер надсилає шкідливі запити [6,7]. Хоча сучасні ORM можуть повністю абстрагуватися від мови запитів та зменшити ризик ін’єкцій, все одно потрібно

знати функції параметризації ORM та правильно їх використовувати. Якщо повертатися до сирих SQL-запитів, треба гарантувати їхню безпеку, так само як і без ORM взагалі [8].

**Контроль операційних ризиків.** Неправильне керування підключеннями до бази даних може призвести до серйозних системних проблем. Неправильно звільнені ресурси підключення будуть накопичуватися і, врешті-решт, вичерпають пул підключень бази даних, спричинивши параліч системи. Фреймворк ORM реалізує автоматичне керування життєвим циклом підключення через шаблон Unit of Work:

```
// Керування підключенням у EF Core
using (var context = new ApplicationDbContext())
{
    var user = context.Users.FirstOrDefault(u => u.Id == 
userId);
    // Підключення автоматично керується, не потрібно закривати 
вручну}
}
```

**Ризики цілісності даних.** Складні бізнес-транзакції потребують гарантій ACID. Традиційний спосіб ручного керування транзакціями схильний до помилок, особливо в сценаріях обробки винятків:

```
// Традиційне керування транзакціями – наявність ризиків
var transaction = connection.BeginTransaction();
try
{
    // Кілька операцій з базою даних
    transaction.Commit();
}
catch
```

```
{
    transaction.Rollback();
    throw;
}
```

ORM забезпечує цілісність операцій з даними через автоматичне керування транзакціями в DbContext.

### 1.1.2. Механізми зменшення ризиків, що реалізуються ORM-фреймворками

**Механізми посилення безпеки.** Фреймворки ORM підвищують безпеку системи через наступні багаторівневі механізми захисту:

- усі введені користувачем дані передаються як параметри, відокремлені від інструкцій SQL;
- перевірка вхідних даних та перевірка типів;
- типова безпека результатів запитів;
- автоматична обробка екранування SQL.

**Оптимізація керування ресурсами.** Механізм керування ресурсами ORM включає: інтелектуальне керування пулом підключень; автоматичний контроль життєвого циклу контексту; механізми відкладеної загрузки та своєчасного звільнення; оптимізація використання паміяті.

**Підтримка відповідності вимогам.** Сучасні фреймворки ORM надають повнофункціональні функції аудиту та ведення журналів, підтримуючи вимоги відповідності таким стандартам, як GDPR, тощо.

```
// Конфігурація журналу аудиту в EF Core
protected override void OnConfiguring(DbContextOptionsBuilder
optionsBuilder)
{
    optionsBuilder.UseLoggerFactory(loggerFactory)
```

```
                .EnableSensitiveDataLogging(false)
                .EnableDetailedErrors(true);
}
```

## 1.2. Імпедансна невідповідність між ООП та реляційною моделлю

### 1.2.1. Відмінності парадигм програмування та моделювання даних

Проблема імпедансної невідповідності виникає внаслідок фундаментальних відмінностей у теоретичних і практичних засадах об'єктно-орієнтованої та реляційної парадигм. Об'єктно-орієнтоване програмування (ООП) зосереджується на моделюванні предметної області у вигляді взаємопов'язаних об'єктів, які поєднують у собі як стан, так і поведінку. Ключовими принципами ООП є інкапсуляція, наслідування та поліморфізм, що дозволяють створювати гнучкі, розширювані та повторно використовувані програмні компоненти. В об'єктній моделі сутності мають унікальну ідентичність, а взаємодія між ними реалізується через посилання, що підтримують складні графи об'єктів.

Натомість реляційні бази даних ґрунтуються на теорії множин і реляційній алгебрі, де основною метою є надійне, формалізоване та ефективне зберігання даних. Реляційна модель оперує таблицями, рядками та стовпцями, а зв'язки між даними встановлюються за допомогою зовнішніх ключів, тобто через значення, а не посилання. Особливу увагу в реляційних базах даних приділено нормалізації, цілісності даних і мінімізації надлишковості, що часто призводить до декомпозиції інформації на велику кількість взаємопов'язаних таблиць.

Ці відмінності зумовлюють низку практичних проблем під час інтеграції об'єктно-орієнтованих застосунків із реляційними базами даних. Зокрема, виникають складнощі з відображенням ієрархій наслідування, асоціацій типу «один-до-багатьох» і «багато-до-багатьох», а також з управлінням життєвим циклом об'єктів і транзакційною узгодженістю. Крім того, різниця в підходах до ідентифікації сутностей, навігації між ними та обробки запитів призводить до

додаткових накладних витрат на розробку і підтримку програмного коду.

Саме ці концептуальні розбіжності між об’єктною та реляційною моделями і формують суть проблеми імпедансної невідповідності, яку покликані мінімізувати ORM-технології шляхом автоматизації процесу відображення об’єктів на реляційні структури та навпаки [9].

### 1.2.2. Типові прояви імпедансної невідповідності та підходи до їх подолання

**Рішення проблеми невідповідності деталізації.** У системах електронної комерції існують значні відмінності між об’єктною моделлю користувача та реляційною моделлю.

```
// Об’єктна модель – багата модель предметної області
public class User
{
    public int Id { get; set; }
    public string UserName { get; set; }
    public Email Email { get; set; } // Об’єкт-значення
    public Address ShippingAddress { get; set; }
    public List<Order> Orders { get; set; }
    public UserPreferences Preferences { get; set; }
}

// Реляційна модель – нормалізована структура таблиць-- Таблиця Users
CREATE TABLE Users (
    Id INT PRIMARY KEY,
    UserName NVARCHAR(50),
    EmailAddress NVARCHAR(100),
```

```
    ShippingStreet NVARCHAR(100),
    ShippingCity NVARCHAR(50)
    -- Інші поля...);
```

ORM вирішує цю різницю в деталізації за допомогою складної конфігурації відображення:

```
// Конфігурація відображення в EF Core
modelBuilder.Entity<User>(entity => {
    entity.OwnsOne(u => u.Email, e =>
    {
        e.Property(e =>
e.Address).HasColumnName("EmailAddress");
    });

    entity.OwnsOne(u => u.ShippingAddress, a =>
    {
        a.Property(a =>
a.Street).HasColumnName("ShippingStreet");
        a.Property(a => a.City).HasColumnName("ShippingCity");
    });

    entity.HasMany(u => u.Orders)
          .WithOne(o => o.User)
          .HasForeignKey(o => o.UserId);
});
```

**Порівняння стратегій відображення наслідування.** Три стратегії відображення наслідування мають різні сценарії застосування. Розглінемо кожеу окремо.

**Стратегія TPH (Одна Таблиця на Ієрархію, англ. Table Per Hierarchy)** – це один із трьох основних підходів до реалізації успадкування (inheritance) об'єктно-орієнтованих класів у реляційній базі даних під час використання ORM (Object-Relational Mapping) [10]. Це найпростіша та найпоширеніша стратегія. Головна ідея стратегії TPH (Одна Таблиця на Ієрархію) полягає в тому, що вся ієрархія успадкованих класів (базовий клас BillingDetail та його підкласи BankAccount і CreditCard) зберігається в одній-єдиній таблиці бази даних. Ця єдина таблиця містить усі поля (колонки) з усіх класів в ієрархії. Спільні поля (наприклад, Id, Owner, Number) від базового класу BillingDetail. Унікальні поля підкласу BankAccount (наприклад, BankName, Swift). Унікальні поля підкласу CreditCard (наприклад, CardType, ExpiryMonth, ExpiryYear). Для розрізнення типів використовується спеціальна колонка, так званий «дискримінатор» (Discriminator Column), який у Вашій конфігурації названо «BillingDetailType». ORM використовує значення цієї колонки (наприклад, «BankAccount» або «CreditCard») для визначення того, до якого саме підкласу належить кожен рядок. Записи, що стосуються BankAccount, матимуть NULL у полях CreditCard, і навпаки.

```
// Ієрархія класів
public abstract class BillingDetail
{
    public int Id { get; set; }
    public string Owner { get; set; }
    public string Number { get; set; }
}
public class BankAccount : BillingDetail
{
    public string BankName { get; set; }
    public string Swift { get; set; }
}
```

```
public class CreditCard : BillingDetail
{
    public int CardType { get; set; }
    public string ExpiryMonth { get; set; }
    public string ExpiryYear { get; set; }
}

// Конфігурація відображення TPH
modelBuilder.Entity<BillingDetail>()
    .HasDiscriminator<string>("BillingDetailType")
    .HasValue<BankAccount>("BankAccount")
    .HasValue<CreditCard>("CreditCard");
```

**Стратегія TPT (Table Per Type , одна таблиця на тип). Ц**е підхід до реалізації успадкування в ORM, який відображає кожен клас ієрархії на окрему таблицю в базі даних. Головна ідея TPT полягає у вертикальному розділенні даних об'єкта між декількома таблицями, які пов'язані спільним ключем. Базовий клас відповідає базовій таблиці [10]. Розглянемо приклад.

```
--Структура таблиць бази данихCREATE TABLE BillingDetails (
    Id INT PRIMARY KEY,
    Owner NVARCHAR(100),
    Number NVARCHAR(50));
CREATE TABLE BankAccounts (
    Id INT PRIMARY KEY FOREIGN KEY REFERENCES
BillingDetails(Id),
    BankName NVARCHAR(50),
    Swift NVARCHAR(20));
CREATE TABLE CreditCards (
    Id INT PRIMARY KEY FOREIGN KEY REFERENCES
BillingDetails(Id),
```

```
    CardType INT,
    ExpiryMonth NVARCHAR(2),
    ExpiryYear NVARCHAR(4));
```

В цьому прикладі базовий клас (BillingDetail) відображається на головну таблицю (BillingDetails), яка містить спільні поля (Id, Owner, Number). Підкласи це додаткові Таблиці. Кожен підклас (BankAccount, CreditCard) відображається на свою окрему таблицю (BankAccounts, CreditCards). Ці таблиці містять лише унікальні поля підкласу (BankName, Swift або CardType, ExpiryMonth, ExpiryYear). Таблиці підкласів пов’язані з головною таблицею за допомогою спільного первинного ключа, який одночасно є зовнішнім ключем (FOREIGN KEY) до таблиці базового класу.

**Інтелектуальна обробка навігаційних зв’язків** ORM спрощує доступ до зв’язків через властивості навігації [10]. В основі ORM лежить ідея відображення реляційних зв’язків (наприклад, «один-до-багатьох» або «"багато-до-багатьох», які в SQL реалізуються за допомогою FOREIGN KEY і оператора JOIN) на властивості об’єктів.

Замість того, щоб вручну писати складні запити з JOIN між кількома таблицями, розробник просто використовує точку (.) для доступу до пов’язаних об’єктів. ORM автоматично розуміє, який SQL-код потрібно згенерувати для отримання цих даних.

У наступному прикладі демонструється, як ORM перетворює об’єктну навігацію на ефективний SQL-запит:

```
// Навігація об’єктів vs SQL JOINvar orders = context.Users
    .Where(u => u.Id == userId)
    .SelectMany(u => u.Orders)
    .Where(o => o.Status == OrderStatus.Pending)
    .Include(o => o.OrderItems)
```

```
    .ThenInclude(oi => oi.Product)
    .ToList();
// Еквівалентний складний SQL буде автоматично згенерований ORM
```

Без ORM (чистий SQL), для отримання всіх очікуваних замовлень користувача, включаючи деталі товарів, необхідно було б вручну написати запит із мінімум трьома операторами JOIN: між Users та Orders, між Orders та OrderItems, і між OrderItems та Products. З ORM, такі методи, як SelectMany, Include та ThenInclude, дозволяють описувати складні зв'язки та критерії фільтрації, використовуючи лише знайому об'єктну нотацію (доступ через властивості).

Ця інтелектуальна обробка значно підвищує продуктивність розробників, зменшує кількість помилок, пов'язаних із синтаксисом SQL-запитів, та робить код чистішим і легшим для підтримки. ORM бере на себе відповідальність за генерацію еквівалентного складного SQL, оптимізацію його виконання та коректне заповнення об'єктів моделі даними з різних таблиць.

## 1.3. Принципи роботи та архітектура ORM-систем

### 1.3.1. Архітектурні патерни, що лежать в основі ORM

**Патерн Unit of Work (Одиниця Роботи)** є фундаментальним елементом у архітектурі, що використовує ORM, і слугує для ефективного керування змінами даних та забезпеченням цілісності транзакцій у базі даних.

Суть патерну полягає в тому, що він підтримує список керованих об'єктів, які були завантажені з бази даних або щойно створені. Unit of Work відстежує всі зміни, які відбуваються з цими об'єктами (додавання нових, зміна існуючих чи видалення), протягом одного бізнес-процесу або однієї взаємодії користувача.

Головна відповідальність Unit of Work – координувати операції запису даних. Замість того, щоб виконувати окремі SQL-команди для кожної зміни негайно, Unit of Work накопичує всі ці операції (Inserts, Updates, Deletes). Лише коли викликається метод «зберегти» (Commit або SaveChanges), Unit of Work пакетно виконує всі накопичені зміни в рамках однієї атомарної транзакції бази даних. Це гарантує цілісність транзакцій: або всі зміни будуть успішно записані в базу даних, або, якщо виникне помилка (наприклад, порушення унікальності чи збій мережі), жодна з операцій не буде виконана, і база даних залишиться у попередньому стабільному стані (Rollback). Таким чином, Unit of Work діє як проміжний буфер між бізнес-логікою та шаром доступу до даних, забезпечуючи надійне та узгоджене збереження даних [11-13].

```
public class UnitOfWorkPatternExample
{
    private readonly DbContext _context;
    private Dictionary<object, EntityState> _managedObjects =
new();

    public void RegisterNew(object entity)
    {
        _managedObjects[entity] = EntityState.Added;
    }

    public void RegisterDirty(object entity)
    {
        _managedObjects[entity] = EntityState.Modified;
    }

    public void Commit()
    {
```

```
            foreach (var item in _managedObjects)
            {
                switch (item.Value)
                {
                    case EntityState.Added:
                        _context.Add(item.Key);
                        break;
                    case EntityState.Modified:
                        _context.Update(item.Key);
                        break;
                }
            }
            _context.SaveChanges();
        }
    }
```

**Патерн Identity Map (Карта Ідентичності)** – це фундаментальний структурний патерн, який використовується в ORM (Object-Relational Mapping), щоб гарантувати, що у межах одного робочого блоку (зазвичай, в рамках сесії ORM або Unit of Work) одному запису бази даних відповідає лише один екземпляр об'єкта в пам'яті програми.

Коли ORM вперше завантажує об'єкт з бази даних, він не лише створює його в пам'яті, але й реєструє в Identity Map, використовуючи його первинний ключ (ID) як унікальний ідентифікатор. Якщо в ході тієї ж операції код спробує завантажити той самий об'єкт знову, ORM спочатку перевірить Identity Map і, замість виконання повторного запиту до БД, просто поверне посилання на вже існуючий екземпляр. Це забезпечує узгодженість даних в усіх частинах програми, що працюють із цим об'єктом. Наприклад, якщо дві різні функції завантажують одного і того ж клієнта, і одна з них змінює його адресу, інша функція негайно побачить цю зміну, оскільки обидві працюють з одним і тим же

екземпляром. Крім гарантування узгодженості, Identity Map також діє як кеш першого рівня, що значно підвищує продуктивність, оскільки усуває необхідність у дорогих повторних запитах до бази даних. Цей патерн є критично важливим для коректної роботи Unit of Work, оскільки саме Identity Map відстежує, які об’єкти були змінені, і допомагає ORM визначити, які UPDATE запити необхідно згенерувати та виконати при збереженні транзакції [11-13].

```
public class IdentityMap<T> where T : class
{
    private readonly Dictionary<object, T> _entities = new();

    public T GetById(object id)
    {
        if (_entities.ContainsKey(id))
            return _entities[id];

        return null;
    }

    public void Add(object id, T entity)
    {
        _entities[id] = entity;
    }

    public void Remove(object id)
    {
        _entities.Remove(id);
    }
}
```

### 1.3.2. Основні етапи життєвого циклу роботи ORM

**Процес обробки запитів.** Процес обробки запитів в ORM є послідовним механізмом, який перетворює орієнтовані на об'єкти запити в мові програмування (наприклад, LINQ у C#) на команди реляційної бази даних (SQL), виконує їх та перетворює результати назад на об'єкти програми [14,15].

```
// 1. Побудова LINQ-запитуvar query = context.Users
    .Where(u => u.Age > 18)
    .OrderBy(u => u.LastName)
    .Select(u => new { u.Id, u.UserName });
// 2. Аналіз дерева виразів// ORM перетворює LINQ-вирази на
дерева // виразів
// 3. Генерація SQL// Генерується параметризований SQL: SELECT
Id, //UserName FROM Users WHERE Age > @p0 ORDER BY LastName
// 4. Матеріалізація даних. Відображення результатів DataReader
на //екземпляри об'єктів
```

На початку розробник створює запит, використовуючи синтаксис мови програмування, як у вашому прикладі з LINQ (Language Integrated Query). Запит, такий як context.Users.Where(...), визначає, які дані потрібні, які мають бути застосовані умови фільтрації та сортування, та які поля повинні бути повернуті. На цьому етапі запит ще не виконується, а лише формується його структура.

Після побудови запиту, ORM перехоплює його та перетворює на дерево виразів. Дерево виразів – це структура даних, яка ієрархічно представляє код запиту. ORM аналізує це дерево, щоб повністю зрозуміти наміри розробника: які сутності залучені, які операції (фільтрація, проекція, сортування) застосовуються, і які параметри використовуються. Це критичний крок, оскільки він дозволяє ORM відокремити логіку від її виконання.

На основі проаналізованого дерева виразів, генератор запитів ORM створює оптимізований і параметризований SQL-запит, специфічний для цільової бази даних.

Сформований SQL-запит надсилається на сервер бази даних, виконується, і результати повертаються у вигляді табличного набору даних (наприклад, об'єкта DataReader). На цьому фінальному етапі ORM виконує матеріалізацію: він читає повернуті рядки даних і використовує патерни Identity Map та Unit of Work для відображення цих результатів на нові або вже існуючі екземпляри об'єктів (наприклад, User). Ці об'єкти потім передаються назад у код програми для подальшої обробки.

**Механізм відстеження стану (Change Tracker)** в Entity Framework Core (та інших сучасних ORM) є ключовою функцією, яка забезпечує синхронізацію об'єктів у пам'яті програми з даними у базі даних. EF Core використовує відстежувач змін для моніторингу стану сутностей, які були завантажені з контексту або додані до нього. Коли об'єкт сутності вперше завантажується (наприклад, через метод Find(1)), відстежувач зберігає його початкові значення (OriginalValues) та встановлює його стан як Unchanged. Коли бізнес-логіка програми змінює будь-яку властивість об'єкта (user.UserName = "NewName"), Change Tracker фіксує цю зміну і автоматично оновлює стан сутності на Modified. Саме завдяки цьому механізму, під час виклику SaveChanges(), ORM точно знає, які саме UPDATE запити потрібно згенерувати та виконати, порівнюючи поточні (CurrentValues) та початкові значення, мінімізуючи навантаження на базу даних, оскільки оновлюються лише змінені поля [10, 16].

```
var user = context.Users.Find(1);
user.UserName = "NewName";
var entry = context.Entry(user);
Console.WriteLine(entry.State); // Modified
Console.WriteLine(entry.OriginalValues["UserName"]); //
Початкове
                                                    // значення
Console.WriteLine(entry.CurrentValues["UserName"]); // Поточне
                                                    // значення
```

### 1.4. Базові концепції ORM

#### Сутності та об’єкти-значення

У галузі об’єктно-орієнтованого дизайну, особливо при використанні ORM важливо розрізняти «Сутності» та «Об’єкти-Значення».

Сутність – це об’єкт, який має неперервну ідентичність (ID), що залишається незмінною протягом усього його життєвого циклу, незалежно від змін його атрибутів (властивостей) [10]. Сутності визначаються не своїми властивостями, а унікальним ідентифікатором (наприклад, Id). Дві сутності є однаковими, якщо їхні ID збігаються, навіть якщо всі їхні інші властивості відрізняються. Наприклад для класу Product. Навіть якщо змінити його Name або Price, це все ще той самий товар з тим самим Id.

Сутності зазвичай керуються через Repository (Репозиторій) і можуть містити складну бізнес-логіку або поведінку (UpdatePrice, AddDomainEvent).

Об’єкт-значення — це об’єкт, який визначається значеннями своїх властивостей. Об’єкти-значення не мають власного ID. Два об’єкти-значення вважаються однаковими, якщо всі їхні властивості збігаються. Наприклад, клас Address. Дві адреси є ідентичними, якщо їхні Street, City та ZipCode абсолютно однакові. Об’єкти-значення зазвичай є незмінними. Якщо потрібно змінити адресу (наприклад, переїзд), створюється новий об’єкт Address замість зміни існуючого.

```
// Сутність - має неперервну ідентичністьpublic class Product : EntityBase{
public int Id { get; private set; }
public string Name { get; set; }
public decimal Price { get; set; }

// Поведінка сутності
public void UpdatePrice(decimal newPrice)
```

```
{
    if (newPrice < 0)
        throw new ArgumentException("Ціна не може бути від’ємною");

    Price = newPrice;
    AddDomainEvent(new ProductPriceChangedEvent(Id, Price));
}}
// Об’єкт-значення – визначається значеннями властивостейpublic class Address : ValueObject{
    public string Street { get; }
public string City { get; }
public string ZipCode { get; }

protected override IEnumerable<object> GetEqualityComponents()
{
    yield return Street;
    yield return City;
    yield return ZipCode;
     }}
```

**Відображення складних типів.** Відображення складних типів в ORM, таких як Entity Framework Core, дозволяє інкапсулювати пов’язані атрибути в окремі Об’єкти-Значення (Value Objects) у коді програми, зберігаючи при цьому ці атрибути як окремі колонки в таблиці бази даних [10].

```
// Конфігурація об’єкта-значення
modelBuilder.Entity<Order>(entity => {
    entity.OwnsOne(o => o.ShippingAddress, address =>
    {
```

```
        address.Property(a =>
a.Street).HasColumnName("ShippingStreet");
        address.Property(a =>
a.City).HasColumnName("ShippingCity");
        address.Property(a =>
a.ZipCode).HasColumnName("ShippingZipCode");
    });
});
```

У вашому прикладі клас ShippingAddress є Об’єктом-Значення і відноситься до сутності Order. Замість того, щоб створювати окрему таблицю для Address та зв’язувати її зовнішнім ключем, ORM використовує функцію OwnsOne (володіння одним). Ця функція вказує ORM, що всі властивості об’єкта ShippingAddress (тобто Street, City, ZipCode) повинні бути вбудовані безпосередньо в таблицю Order. Це досягається завдяки конфігурації, де для уникнення конфліктів імен колонок можна явно задати їхні назви, наприклад, ShippingStreet замість просто Street. Такий підхід підтримує чистоту об’єктної моделі (інтерфейс Order використовує об’єкт Address), але забезпечує простоту реляційної моделі (всього одна таблиця для Order та його адреси).

### 1.5. Розвиток технологій ORM у .NET

Еволюція технологій Object-Relational Mapping (ORM) у середовищі .NET відображає поступовий перехід від низькорівневого, орієнтованого на SQL підходу до більш абстрагованої, об’єктно-орієнтованої моделі роботи з даними. Основною рушійною силою цього розвитку стало прагнення зменшити обсяг рутинного коду, підвищити продуктивність розробки та подолати проблему імпедансної невідповідності між об’єктною моделлю прикладного коду та реляційною моделлю баз даних.

Початковий етап розвитку платформи .NET (2002–2005) характеризувався використанням ADO.NET як основного механізму доступу до даних. У цей

період розробники змушені були вручну формувати SQL-запити, керувати з'єднаннями, обробляти результати виконання запитів та виконувати мапінг отриманих даних у об'єкти мовою C#. Такий підхід забезпечував високий рівень контролю над виконанням запитів, проте супроводжувався значним обсягом шаблонного коду (boilerplate), складністю супроводу та підвищеним ризиком помилок, що негативно впливало на швидкість розробки та якість програмних систем.

Наступним важливим кроком став період розвитку LINQ to SQL (2007–2010), який ознаменував появу інтегрованих у мову C# запитів до баз даних. Впровадження LINQ (Language Integrated Query) дозволило формувати типобезпечні запити безпосередньо в коді застосунку, зменшивши залежність від «сирого» SQL і підвищивши читабельність програмного коду. Хоча LINQ to SQL мав низку обмежень, зокрема жорстку прив'язку до SQL Server та обмежену підтримку складних сценаріїв, він заклав концептуальні основи для створення більш універсальних ORM-рішень.

Подальший розвиток ORM у .NET пов'язаний із появою та еволюцією Entity Framework (EF), який поступово став де-факто стандартом для роботи з реляційними базами даних у корпоративних застосунках. Починаючи з версії EF 4.0 (близько 2010 року), було запроваджено підходи Code First та Model First, що дозволили визначати структуру бази даних на основі об'єктної моделі або візуальної схеми. Це суттєво спростило процес проєктування та дало змогу зосередитися на бізнес-логіці, а не на деталях реалізації схеми зберігання даних.

Подальші версії Entity Framework були спрямовані на розширення функціональності та покращення продуктивності. Зокрема, EF 5.0 додала підтримку переліків (enum), що зробило доменну модель більш виразною та близькою до предметної області. EF 6.0 стала найбільш зрілою версією для класичного .NET Framework, запропонувавши асинхронні операції на основі async/await, а також механізм перехоплювачів (interceptors), які забезпечили гнучкий контроль над виконанням запитів і життєвим циклом з'єднань.

З появою платформи .NET Core відбувся якісний зсув у розвитку ORM-технологій. Entity Framework було повністю переписано у вигляді EF Core, орієнтованого на крос-платформність, модульність та підвищену продуктивність. Перша версія EF Core 1.0 (2016 р.) запропонувала легковаговий і швидкий ORM, хоча й поступалася EF 6.0 за кількістю підтримуваних можливостей. У наступних релізах функціональність поступово відновлювалась і розширювалась.

EF Core 5.0 суттєво покращила роботу зі зв’язками типу «багато-до-багатьох» та запровадила механізм розділених запитів (Split Queries), що дозволило ефективніше боротися з проблемою N+1. У версії EF Core 6.0 основний акцент було зроблено на оптимізації продуктивності, що дозволило наблизити швидкодію повноцінного ORM до рівня мікро-ORM у багатьох типових сценаріях. Останні версії EF Core 7.0 та 8.0 інтегрують сучасні можливості СУБД, зокрема нативну підтримку JSON-стовпців, ієрархічних ідентифікаторів та вдосконалені механізми роботи зі складними типами даних.

Таким чином, еволюція ORM у середовищі .NET є послідовним процесом пошуку балансу між абстракцією та контролем, між зручністю розробки та високою продуктивністю. Сучасні ORM-рішення дозволяють ефективно поєднувати об’єктно-орієнтовану модель програмування з реляційними базами даних, забезпечуючи гнучкість архітектурних рішень і масштабованість корпоративних систем.

### 1.6. Огляд ORM-фреймворків у .NET та порівняння їх можливостей

Сучасне середовище .NET пропонує розробникам кілька потужних фреймворків для Object-Relational Mapping (ORM), які допомагають подолати імпедансну невідповідність між об’єктно-орієнтованими мовами та реляційними базами даних. Серед них виділяються два лідери, що пропонують принципово різні підходи: Entity Framework Core (EF Core) як повнофункціональний фреймворк, та Dapper як високошвидкісний мікро-ORM.

Entity Framework Core є офіційним, крос-платформним ORM від Microsoft і вважається "фул-стек" рішенням для роботи з даними. Його головна перевага полягає у високому рівні абстракції, що дозволяє розробникам майже не писати SQL, а працювати лише з об’єктами C#.

EF Core забезпечує надзвичайно високу здатність запитів завдяки підтримці LINQ, що дає змогу будувати складні вирази, які EF автоматично транслює в оптимізований SQL. Його інтеграція із системою Change Tracker автоматично відстежує зміни в об’єктах та генерує необхідні операції оновлення.

З точки зору підтримки, EF Core має відмінну документацію від Microsoft та дуже розвинену екосистему спільноти, що забезпечує швидке вирішення будь-яких проблем. Хоча історично EF мав проблеми з продуктивністю, сучасні версії EF Core демонструють високу продуктивність, яка є цілком достатньою для більшості бізнес-додатків. Однак, через велику кількість функцій та абстракцій, крива навчання для повного освоєння фреймворку оцінюється як середня.

**Таблиця 1.1.**

**Матриця характеристик Entity Framework Core.**

| Характеристика | Оцінка |
|---|---|
| Здатність запитів | Дуже висока |
| Продуктивність | Висока |
| Крива навчання | Середня |
| Екосистема спільноти | Дуже розвинена |
| Повнота документації | Відмінна |

**Аналіз сценаріїв застосування Dapper.** На противагу EF Core, Dapper є мікро-ORM, що був розроблений командою Stack Overflow. Його філософія – мінімум абстракцій, максимум швидкості. Dapper не генерує SQL і не має власного механізму відстеження змін (Change Tracker); він фокусується виключно на ефективному мапуванні результатів виконання "сирого" SQL-запиту в об’єкти C#.

Аналіз сценаріїв застосування Dapper показує, що цей мікро-ORM ідеально підходить для тих випадків, де критичною є продуктивність та повний контроль над запитами. Це типово, наприклад, для складних, оптимізованих запитів, коли розробнику потрібно написати унікальний, високоефективний SQL, який повнофункціональний ORM міг би згенерувати неефективно. Крім того, Dapper незамінний при читанні даних з високим навантаженням (так звані Read-heavy loads), що часто трапляється у високодоступних публічних API, де потрібно надзвичайно швидко витягувати великі обсяги даних. Нарешті, Dapper є оптимальним вибором для легких, старих або вузькоспеціалізованих додатків, де немає потреби у складному функціоналі EF Core, а пріоритетом є мінімальна залежність та максимальна швидкість.

**Гібридні архітектурні патерни.** У сучасній архітектурі .NET, особливо в контексті мікросервісів, замість вибору одного інструмента, все частіше використовується стратегія поєднання EF Core та Dapper. Цей гібридний архітектурний патерн дозволяє використовувати переваги обох фреймворків. EF Core застосовується для більшої частини бізнес-логіки (операції CRUD), де потрібна швидкість розробки, абстракція та безпека. Dapper використовується для конкретних, критичних до швидкості операцій читання або складних звітів, де необхідна хірургічна оптимізація SQL. Такий підхід забезпечує ідеальний баланс між продуктивністю та зручністю розробки, що робить його ключовим елементом сучасної .NET-архітектури.

Отже ми системно побудовано теоретичну основу ORM. Спочатку з макроскопічної точки зору управління ризиками підприємства обґрунтували необхідність використання ORM, вказавши, що це ефективний архітектурний засіб для зниження ризиків у роботі з базами даних. Потім глибоко проаналізували ключову суперечність "імпедансної невідповідності", виявивши фундаментальну причину існування технології ORM. На цій основі ми детально виклали принципи роботи, архітектурні патерни та ключові концепції ORM, описавши його внутрішній механізм роботи. Нарешті, оглянувши історію

розвитку ORM на платформі .NET та провівши горизонтальне порівняння основних фреймворків, ми забезпечили чіткий технологічний контекст та основу для вибору для подальших практичних досліджень. Зміст цього розділу створює міцну теоретичну основу для розгортання всієї роботи, наступний розділ перейде на практичний рівень, глибоко досліджуючи конкретну реалізацію шару збереження даних на основі EF Core.

# РОЗДІЛ 2.
# ПОРІВНЯННЯ ЕФЕКТИВНОСТІ ТА ФУНКЦІОНАЛЬНИХ МОЖЛИВОСТЕЙ ENTITY FRAMEWORK CORE ТА DAPPER У .NET

## 2.1. Постановка проблеми

Сучасна розробка програмного забезпечення часто передбачає роботу з реляційними базами даних. Для зручності та прискорення процесу взаємодії з базою даних у середовищі .NET застосовуються ORM-системи (Object-Relational Mapping), які дозволяють відображати об’єкти програмної моделі на структури бази даних і автоматизують виконання SQL-запитів.

Незважаючи на очевидні переваги, різні ORM-системи мають суттєві відмінності щодо продуктивності, складності налаштування, підтримки складних запитів, використання кешування та управління транзакціями. Для розробників важливо обрати оптимальну ORM-систему залежно від характеру проєкту, обсягів даних та вимог до швидкодії.

У зв’язку з цим виникає потреба провести порівняльний аналіз популярних ORM-систем у .NET, оцінити їх продуктивність та функціональні можливості, визначити сильні й слабкі сторони, а також надати практичні рекомендації щодо їх використання у реальних проєктах.

Мета практичної частини полягає у порівняльному аналізі продуктивності, масштабованості та зручності використання різних ORM систем у контексті корпоративних додатків, а також у розробці практичних рекомендацій щодо їх вибору та оптимізації.

Завдання для досягнення мети:

- провести теоретичне порівняння обраних ORM систем (Entity Framework Core та Dapper) за ключовими критеріями;
- реалізувати шари доступу до даних з використанням кожної з ORM для типової предметної області (наприклад, система управління замовленнями);

- розробити набір тестових сценаріїв для вимірювання продуктивності CRUD операцій, складних запитів та навантаження;
- виміряти та проаналізувати показники продуктивності (час виконання, використання пам’яті, навантаження на CPU);
- оцінити зручність розробки, підтримки та гнучкість кожної ORM;
- сформулювати висновки та практичні рекомендації щодо вибору ORM залежно від вимог проекту.

### 2.2. Методологія дослідження

Дослідження проводилось методом експериментального аналізу на основі набору спеціально розроблених тестів. Основними критеріями порівняння ORM-систем були продуктивність, масштабованість, зручність реалізації та супроводу коду, а також гнучкість і підтримка додаткових функцій, таких як LINQ та кешування. А також можливість використання власного SQL, контроль над транзакціями, можливість тонкої оптимізації. Оцінювання безпеки включало наявність захисту від SQL-ін’єкцій за замовчуванням, а також можливості аудиту та журналювання. Продуктивність оцінювалась за часом виконання запитів, споживанням оперативної пам’яті та навантаженням на процесор. Масштабованість визначалась через поведінку системи при збільшенні обсягу даних та кількості одночасних запитів. Зручність реалізації та супроводу коду враховувала обсяг коду, його читабельність і простоту рефакторингу, швидкість розробки нових функцій.

Для збору даних використовувалися автоматизовані тести (xUnit), які вимірювали час виконання операцій, а також прецизійні мікробенчмарки окремих операцій, реалізовані за допомогою BenchmarkDotNet. Для аналізу споживання ресурсів застосовувалися профайлери пам’яті та CPU, зокрема JetBrains dotMemory і dotTrace. Додатково використовувалися вбудовані засоби логування запитів EF Core для аналізу та оптимізації згенерованого SQL, а також SQL Server Profiler для моніторингу активності на стороні СУБД. Аналіз

результатів проводився шляхом порівняльного аналізу отриманих метрик, статистичної обробки даних (середні значення, медіани, персентилі) та якісного аналізу коду з точки зору його зручності й підтримуваності.

Аналіз результатів здійснювався шляхом порівняльного аналізу отриманих метрик, статистичної обробки результатів із визначенням середніх значень, медіан та 95-го персентиля, а також якісного аналізу коду, архітектури рішення та звітів профайлерів.

Для порівняльного аналізу популярних ORM-систем у .NET було спроектовано та розроблено повну модель системи управління замовленнями з сутностями User, Product, Category, Order, OrderItem. Додано додаткові атрибути для тестування (CreatedAt, IsActive, Version для оптимістичного блокування).

Написані SQL-скрипти для створення таблиць, індексів (кластерних та некластерних для полів, що часто фільтруються/з'єднуються), та обмежень цілісності.

Для реалістичного тестування згенеровано великий обсяг тестових даних за допомогою бібліотеки Bogus:

- 50,000 користувачів;
- 10000 товарів (у 50 категоріях);
- 1000000 замовлень (з випадковим розподілом по користувачам та датам за останній рік);
- ~5,000,000 позицій замовлень.

Загальний розмір бази даних після наповнення становив ~15 ГБ.

### 2.3. Обґрунтування вибору інструментів

Для проведення порівняльного аналізу в рамках дослідження було обрано дві ORM-системи: Entity Framework Core 8.0 [18] та Dapper 2.1 [19-21].

Entity Framework Core 8.0 була обрана завдяки повній інтеграції з екосистемою .NET, потужній підтримці LINQ, можливостям автоматичних міграцій та широкому набору функцій, що роблять її зручною для розробки

корпоративних додатків. До її переваг належать висока абстракція, ефективне управління складними зв’язками між таблицями та автоматизація процесів міграції бази даних, тоді як недоліком є відносно великі накладні витрати порівняно з мікро-ORM.

У свою чергу, Dapper 2.1 було обрано як представника категорії «мікро-ORM» з метою порівняння продуктивності та гнучкості при виконанні складних запитів. До переваг Dapper відносяться мінімальні накладні витрати, пряме виконання SQL-запитів та висока швидкість, тоді як його недоліками є відсутність підтримки абстракції міграцій та необхідність ручного написання SQL-коду.

Для експериментальної частини дослідження використовувалася система управління базами даних Microsoft SQL Server 2022 (локальний екземпляр), обрана через її поширеність у корпоративних середовищах, повну інтеграцію з .NET та розвинені інструменти для аналізу продуктивності.

Мова програмування C# 12.0 (.NET 8.0) – була обрана через її тісну інтеграцію з обома ORM, потужну підтримку LINQ та сучасні можливості для високопродуктивної розробки. Інтегроване середовище розробки Visual Studio 2022 забезпечило всебічну підтримку .NET, наявність вбудованих профайлерів продуктивності та зручних інструментів для роботи з базами даних.

Для проведення тестів та збору кількісних даних застосовувалися xUnit.net для модульного та інтеграційного тестування, BenchmarkDotNet для точних мікробенчмарків, а також JetBrains dotMemory та dotTrace для аналізу використання пам’яті та навантаження на процесор під час виконання навантажувальних тестів. Крім того, для Entity Framework Core використовувалося вбудоване логування SQL-запитів, що дозволяло оцінити ефективність згенерованого коду. Така конфігурація дозволила комплексно оцінити продуктивність, масштабованість та зручність використання обраних ORM-систем у реальних сценаріях розробки.

## 2.4. Об’єктно-реляційне відображення та характеристика досліджуваних ORM-систем

ORM (Object-Relational Mapping) – це техніка програмування, яка дозволяє перетворювати дані між реляційними базами даних та об’єктно-орієнтованими мовами програмування. Вона вирішує проблему імпедансної невідповідності шляхом відображення таблиць бази даних на класи, рядків – на об'єкти, а стовпців – на властивості.

**Таблиця 2.1.**

**Теоретичне порівняння обраних ORM систем**

| **Критерій** | **Entity Framework Core 8.0** | **Dapper 2.1** |
|---|---|---|
| **Архітектурна модель** | Повноцінний ORM (Unit of Work, Identity Map, Change Tracking) | Мікро-ORM / Маппер результатів |
| **Генерація SQL** | Автоматична з LINQ | Ручне написання |
| **Підтримка LINQ** | Повна | Відсутня (окрім базових операцій з колекціями після виконання запиту) |
| **Транзакції** | Автоматичні (неявні) та явні через DbContext | Явні через SqlTransaction |
| **Міграції БД** | Вбудована підтримка (Code-First, Migrations API) | Відсутня (потрібні окремі інструменти) |
| **Завантаження даних** | Ліниве (Lazy), жадібне (Eager), явне (Explicit) | Тільки жадібне (всі дані повинні бути явно завантажені в запиті) |
| **Кешування** | Обмежена підтримка рівня контексту | Відсутнє на рівні ORM |
| **Крива навчання** | Вища | Нижча (для розробників, що знайомі з SQL) |

## 2.5. Реалізація шару доступу до даних (DAL) з використанням ORM-технологій

### 2.5.1. Реалізація шару доступу до даних (DAL) з використанням EF Core

У роботі було створено AppDbContext з детальною конфігурацією сутностей за допомогою Fluent API (атрибути не використовувалися з метою забезпечення більшої гнучкості). Реалізовано патерни Repository та Unit of Work для абстрагування доступу до даних і забезпечення узгодженості транзакцій. Налаштовано глобальні фільтри запитів (HasQueryFilter) для реалізації механізму м’якого видалення (Soft Delete). У процесі міграцій використано власні SQL-вирази (Raw SQL у міграціях) для виконання складних операцій оновлення даних.

З метою підвищення продуктивності було оптимізовано запити шляхом використання AsNoTracking() для сценаріїв лише читання, Select() для проєкцій, а також власних SQL-запитів (FromSqlRaw) для звітів, критичних до швидкодії. Далі наведено приклад конфігурації сутності Order у EF Core.

```
modelBuilder.Entity<Order>(entity =>
{
    entity.HasKey(e => e.Id);
    entity.HasIndex(e => e.CreatedAt).IsClustered(false);
    entity.HasIndex(e => new { e.UserId, e.StatusId });

    entity.Property(e => e.TotalAmount).HasPrecision(18, 2);
    entity.Property(e => e.RowVersion).IsRowVersion(); //
Конкурентний доступ

    // Конфігурація зв'язків
    entity.HasOne(o => o.User)
          .WithMany(u => u.Orders)
          .HasForeignKey(o => o.UserId)
          .OnDelete(DeleteBehavior.Restrict);

    entity.HasMany(o => o.OrderItems)
          .WithOne(oi => oi.Order)
          .HasForeignKey(oi => oi.OrderId);
```

```
    // Властивість-навігація для завантаження через Include
    entity.Navigation(e => e.OrderItems).AutoInclude(false);
});
```

### 2.5.2. Реалізація DAL з використанням Dapper

У роботі було створено клас DapperContext для керування підключеннями до бази даних, а також реалізовано низку розширювальних методів (QueryAsync, ExecuteAsync) з метою спрощення взаємодії з Dapper. Для кожного складного запиту було розроблено оптимізований SQL із використанням CTE та, за потреби, віконних функцій. Мапінг результатів на об’єкти здійснювався за допомогою методів QueryAsync<T> і QueryMultiple для обробки кількох результуючих наборів.

Для безпечної параметризації запитів використовувалися DynamicParameters, що мінімізує ризик SQL-ін’єкцій. Операції масового вставлення та оновлення реалізовано із застосуванням технології Bulk Copy через SqlBulkCopy, що забезпечує максимальну продуктивність. Далі наведено приклад складного запиту в Dapper для формування звіту.

```
public async Task<IEnumerable<SalesReportDto>>
GetMonthlySalesReportAsync(int year)
{
    const string sql = @"
        WITH MonthlyData AS (
            SELECT
                DATEPART(MONTH, o.OrderDate) AS Month,
                c.Name AS CategoryName,
                SUM(oi.Quantity * oi.UnitPrice) AS TotalSales,
                COUNT(DISTINCT o.Id) AS OrderCount
            FROM Orders o
            INNER JOIN OrderItems oi ON o.Id = oi.OrderId
            INNER JOIN Products p ON oi.ProductId = p.Id
            INNER JOIN Categories c ON p.CategoryId = c.Id
            WHERE o.StatusId = 3 AND DATEPART(YEAR,
o.OrderDate) = @Year
            GROUP BY DATEPART(MONTH, o.OrderDate), c.Name
        )
        SELECT * FROM MonthlyData
        ORDER BY Month, TotalSales DESC";
```

```
    return await _connection.QueryAsync<SalesReportDto>(sql,
new { Year = year });
}
```

### 2.5.3. Розробка тестового модуля

Для забезпечення коректності функціонування, стабільності та якості розробленої системи було реалізовано комплексне тестування, яке охоплює кілька рівнів перевірки — від ізольованого тестування окремих компонентів до оцінювання поведінки системи в умовах реального навантаження. Такий підхід дозволив виявити помилки на ранніх етапах розробки, перевірити правильність реалізації бізнес-логіки та об’єктивно оцінити експлуатаційні характеристики обраних технологій доступу до даних.

Модульні тести були реалізовані з використанням фреймворку xUnit і застосовувалися для перевірки коректності роботи окремих методів репозиторіїв та сервісного рівня. Основна увага приділялася перевірці правильності мапінгу даних між об’єктною моделлю та реляційними структурами, а також базовим CRUD-операціям (створення, читання, оновлення та видалення). Модульні тести виконувалися в ізольованому середовищі, що забезпечувало їх повторюваність та незалежність від зовнішніх факторів.

Інтеграційні тести також були побудовані на базі xUnit і спрямовані на перевірку коректності виконання повних бізнес-сценаріїв, що охоплюють кілька рівнів системи. Зокрема, тестувалися типові ланцюжки операцій, такі як «створення кошика → додавання товарів → оформлення замовлення», із реальним зверненням до бази даних. Для забезпечення максимально наближених до реальних умов використовувалися тестові СУБД: SQLite у режимі In-Memory для Entity Framework Core та SQL Server LocalDB для реалізації шару доступу до даних на базі Dapper. Це дозволило перевірити коректність транзакцій, цілісність даних та поведінку ORM у різних сценаріях збереження й читання інформації.

Для оцінювання продуктивності системи було створено окремий проєкт бенчмарків із використанням бібліотеки BenchmarkDotNet. У межах цього проєкту проводилися прецизійні вимірювання часу виконання ключових

операцій, зокрема отримання однієї сутності за ідентифікатором, виконання складних запитів із кількома операціями приєднання (JOIN), пакетного вставлення великої кількості записів та оновлення даних з урахуванням конкурентного доступу. BenchmarkDotNet забезпечив стабільність результатів завдяки багаторазовому виконанню тестів, автоматичному прогріву середовища та статистичній обробці отриманих метрик.

Додатково було проведено навантажувальне тестування із застосуванням інструмента k6. Для цього були розроблені сценарії, що імітують одночасну роботу сотень користувачів із REST API, побудованим поверх обох реалізацій шару доступу до даних. Навантажувальні тести дозволили оцінити поведінку системи під високим рівнем паралелізму, визначити пропускну здатність API, стабільність часу відгуку та виявити потенційні вузькі місця, пов’язані з роботою ORM та СУБД.

### 2.6. Організація тестування та налагодження

Для проведення тестування було підготовлено тестове середовище. Апаратна частина включала сервер на базі процесора Intel Xeon E-2286G з 12 ядрами, 64 ГБ оперативної пам’яті та NVMe SSD. Програмне середовище складалося з операційної системи Windows Server 2022, платформи .NET 8.0 та СУБД SQL Server 2022, для якої було виділено 32 ГБ оперативної пам’яті. Усі компоненти системи розгорталися в межах однієї мережі з метою мінімізації мережевих затримок.

Тестування проводилося за кількома сценаріями. Під час сценарію «холодного старту» вимірювався час виконання першого запиту після запуску застосунку, зокрема з урахуванням компіляції запитів у Entity Framework Core. У режимі «теплого старту» оцінювалася продуктивність системи після виконання кількох ітерацій запитів. Для перевірки масштабованості виконувалося паралельне навантаження із 100, 500 та 1000 одночасних запитів до API. Окремо проводився довготривалий навантажувальний тест тривалістю 30 хвилин,

спрямований на виявлення можливих витоків пам’яті або деградації продуктивності з часом.

У процесі тестування здійснювався детальний збір ключових показників продуктивності та ресурсоспоживання системи. Зокрема, вимірювався час відгуку кожного запиту, що дозволяло оцінити швидкодію окремих операцій у різних сценаріях. Паралельно фіксувалася загальна кількість транзакцій за секунду (TPS), що слугувало показником пропускної здатності системи під навантаженням. Для оцінки ефективності використання пам’яті відстежувався рівень споживання оперативної пам’яті клієнтським застосунком із детальним аналізом часткових та повних циклів роботи збирача сміття (GC), що давало змогу виявити потенційні витоки та перевантаження пам’яті. Додатково здійснювався моніторинг навантаження на ядра CPU як на стороні СУБД, так і на прикладному рівні, що дозволяло визначити вузькі місця у продуктивності та розподіл обчислювальних ресурсів. Нарешті, велика увага приділялася фіксації кількості взаємних блокувань (deadlocks) у базі даних, оскільки вони безпосередньо впливають на стабільність роботи та ефективність обробки паралельних транзакцій.

## 2.7. Аналіз отриманих результатів, рекомендації щодо використання та впровадження

У таблиці 2.2 подано узагальнені результати експериментального порівняння продуктивності та зручності використання двох ORM-підходів – Entity Framework Core 8.0 та Dapper 2.1. Представлені показники включають час виконання типових операцій читання та запису, витрати пам’яті, час розробки нових запитів, а також аспекти безпеки та обслуговування схеми бази даних. Коментарі до кожного тесту відображають сильні та слабкі сторони кожного підходу, що дозволяє зробити обґрунтований вибір технології залежно від конкретних завдань проекту.

**Таблиця 2.1.**

**Узагальнені результати експерименту**

| **Тест / Критерій** | **EF Core 8.0** | **Dapper 2.1** | **Коментар** |
|---|---|---|---|
| **Вибірка 1 запису по ID** | ~1.2 мс | ~0.3 мс | Dapper значно швидший завдяки прямому маппінгу. |
| **Складний звіт (5 JOIN)** | ~45 мс | ~22 мс | Dapper вдвічі швидший. EF Core генерує складніший SQL. |
| **Вставка 1000 записів** | ~1200 мс (SaveChanges) | ~150 мс (SqlBulkCopy) | Для масових операцій Dapper з BulkCopy не має конкурентів. |
| **Пам'ять (складний запит)** | ~15 MB алокацій | ~3 MB алокацій | EF Core має вищі накладні витрати на трекінг та метадані. |
| **Час розробки нового запиту** | Низький (LINQ) | Високий (SQL + маппінг) | EF Core прискорює розробку. |
| **Захист від SQL-ін'єкцій** | Повний (параметризація за замовч.) | Повний (при правильному використанні DynamicParameters) | Обидва безпечні при коректному використанні. |
| **Обслуговування схеми БД** | Дуже високе (автоміграції) | Низьке (потрібні скрипти) | EF Core надає потужний інструмент для еволюції моделі. |

**Продуктивність.** Результати експериментів показали, що Dapper стабільно демонструє вищі показники продуктивності у всіх сценаріях, орієнтованих на читання даних, а також під час виконання масових операцій. Перевага становить у середньому від 50 % до 400 % залежно від типу запиту та обсягу даних. Це пояснюється відсутністю накладних витрат, пов’язаних із відстеженням змін, генерацією запитів і керуванням контекстом. Водночас у складних транзакційних

сценаріях запису різниця між підходами зменшується, а за умови коректної оптимізації EF Core здатен демонструвати порівнянні результати.

**Масштабованість.** Під час високого паралельного навантаження (понад 1000 одночасних запитів) обидва підходи виявляють лінійну деградацію продуктивності, що зумовлено конкуренцією за ресурси СУБД. Проте Dapper створює менше навантаження на CPU прикладного рівня, завдяки чому дозволяє обробляти більшу кількість запитів на однаковому апаратному забезпеченні. Це робить його більш придатним для сценаріїв з інтенсивним паралельним читанням даних.

**Зручність розробки.** З точки зору продуктивності праці розробника беззаперечним лідером є Entity Framework Core. Використання LINQ, автоматичне керування зв'язками між сутностями, вбудовані міграції бази даних і механізми відстеження змін значно спрощують розробку. За результатами аналізу, для типових корпоративних застосунків EF Core дозволяє скоротити час виходу продукту на ринок (Time-to-Market) на 30–50 % порівняно з ручною реалізацією SQL-запитів.

**Гнучкість та контроль.** Dapper надає розробнику повний контроль над виконуваним SQL-кодом, що є критично важливим для оптимізації складних запитів або використання специфічних можливостей СУБД, таких як CTE, віконні функції чи оптимізовані хінти. EF Core, хоча й забезпечує можливість виконання власного SQL через FromSqlRaw, усе ж накладає певні обмеження, пов'язані з абстракцією ORM.

**Безпека.** Обидва фреймворки забезпечують надійний захист від SQL-ін'єкцій за умови коректного використання механізмів параметризації: у Dapper — через DynamicParameters, у EF Core – через LINQ та параметризовані запити. Додатковою перевагою EF Core є наявність вбудованих механізмів аудиту, а також зручна інтеграція з системами автентифікації та авторизації, що є важливим для корпоративних інформаційних систем.

Подібні висновки наведені в технічному огляді порівняння цих ORM у середовищі .NET [22-28], де підкреслюється, що Dapper доцільно

використовувати для високопродуктивних запитів, а EF Core — для типових корпоративних сценаріїв із складною бізнес-логікою.

### 2.8. Практичні рекомендації для архітектурних рішень

Вибір технології доступу до даних є одним із ключових архітектурних рішень при розробці сучасних програмних систем. Від цього вибору залежить не лише продуктивність і масштабованість застосунку, а й зручність розробки, підтримуваність коду, безпека та ефективність роботи з базою даних. Різні підходи до реалізації шару доступу до даних мають власні переваги та обмеження. Entity Framework Core забезпечує високий рівень абстракції, автоматичне відстеження змін, підтримку LINQ, міграцій бази даних та комплексне управління транзакціями, що значно спрощує розробку складних корпоративних систем. У той же час Dapper, як легкий мікро-ORM, забезпечує максимальний контроль над SQL-запитами, мінімальні накладні витрати та високу продуктивність у сценаріях інтенсивного читання і масових операцій.

У реальних проєктах часто виникає потреба поєднувати обидва підходи, щоб ефективно використовувати їхні сильні сторони. Для складних систем та архітектур на основі CQRS доцільно застосовувати EF Core для командної сторони (операцій запису та зміни стану), де критично важлива транзакційність і бізнес-логіка, і Dapper для сторони запитів (операцій читання), де першочергове значення мають швидкодія та гнучкість формування звітів. Крім того, Dapper може використовуватися в окремих високопродуктивних модулях або аналітичних компонентах, тоді як EF Core залишається основним інструментом управління даними та забезпечення цілісності бізнес-логіки.

Наведені нижче практичні рекомендації допомагають розробникам і архітекторам обрати оптимальний підхід залежно від типу проєкту, рівня навичок команди та специфічних вимог до продуктивності та масштабованості системи.

Доцільно використовувати Entity Framework Core, якщо:

- проєкт є типовим корпоративним застосунком зі складною об’єктною моделлю та великою кількістю бізнес-правил;
- команда не має глибоких знань SQL або пріоритетом є швидкість розробки;
- важливими є автоматичні міграції бази даних і підтримка цілісності транзакцій на рівні доменної логіки;
- продуктивність операцій запису та складних транзакцій важливіша за максимальну швидкість читання;
- застосунок не є високонавантаженим сервісом обробки даних (наприклад, системою аналітики в реальному часі).

Доцільно використовувати Dapper, якщо:

- застосунок орієнтований переважно на читання даних (звітність, аналітичні системи, API для дашбордів);
- критично важливими є мінімальна затримка та максимальна продуктивність;
- команда володіє сильними навичками SQL і потребує повного контролю над запитами;
- модель даних є відносно простою або вже існує оптимізована схема бази даних;
- виконуються операції масового імпорту або експорту даних.

**Гібридний підхід (рекомендований для складних систем).** Для великих і складних інформаційних систем найефективнішим рішенням є поєднання обох технологій доступу до даних — Entity Framework Core та Dapper. Такий підхід дозволяє максимально використовувати сильні сторони кожного інструменту, адаптуючи їх під конкретні сценарії використання.

Зокрема, у архітектурі CQRS (Command Query Responsibility Segregation) рекомендується застосовувати EF Core для обробки команд, тобто операцій запису та зміни стану даних, де ключовими є забезпечення транзакційної цілісності, дотримання бізнес-правил і контроль залежностей між об’єктами. Dapper у цьому випадку використовується для запитів на читання, де критично важлива висока швидкодія, мінімальні накладні витрати та гнучкість у

формуванні складних звітів і аналітичних запитів.

Крім того, Dapper доцільно застосовувати в окремих модулях, які є критичними до продуктивності, наприклад, у модулі аналітики або у підсистемах масового оброблення даних. У таких сценаріях він забезпечує прямий контроль над SQL-запитами, можливість оптимізації складних операцій та зниження витрат на трекінг і генерацію запитів, які характерні для повноцінних ORM. Водночас EF Core використовується для основного функціоналу управління даними, що дозволяє скоротити час розробки, підтримувати міграції бази даних та забезпечувати цілісність доменної логіки.

У контексті мікросервісної архітектури кожен сервіс може обирати ORM, який найкраще відповідає його функціональним вимогам: один сервіс може використовувати EF Core для забезпечення складної логіки і управління транзакціями, тоді як інший сервіс — Dapper для високопродуктивних запитів читання, що дозволяє досягти оптимального балансу між продуктивністю та зручністю підтримки системи в цілому. Такий гібридний підхід забезпечує масштабованість, гнучкість архітектури та ефективне використання ресурсів при розробці сучасних корпоративних систем.

# ВИСНОВКИ

У ході виконання роботи проведено комплексне дослідження технології Object-Relational Mapping (ORM) та її застосування для організації шару збереження даних у корпоративних додатках. ORM забезпечує ефективну абстракцію між реляційними базами даних і об'єктно-орієнтованим програмуванням, знижує складність роботи з SQL-запитами та дозволяє підвищити швидкість розробки й підтримки системи.

Було детально проаналізовано проблему «імпедансної невідповідності» між об'єктно-орієнтованими моделями та реляційними базами даних, а також визначено роль управління ризиками при роботі з базами даних у корпоративних середовищах. На теоретичній основі викладено принципи роботи, архітектуру, ключові концепції ORM та її еволюцію в середовищі .NET, що стало підґрунтям для практичної реалізації шару доступу до даних.

Практична частина роботи включала розробку і впровадження DAL із використанням Entity Framework Core і Dapper. Для EF Core було створено AppDbContext із детальною конфігурацією сутностей через Fluent API, реалізовано патерни Repository та Unit of Work, налаштовано глобальні фільтри запитів для Soft Delete, оптимізовано запити через AsNoTracking, Select та FromSqlRaw. Для Dapper створено DapperContext із методами QueryAsync та ExecuteAsync, розроблено оптимізовані SQL-запити з CTE та віконними функціями, реалізовано масові операції через SqlBulkCopy та безпечну параметризацію через DynamicParameters.

Було організовано комплексне тестування, включаючи модульні та інтеграційні тести з xUnit, бенчмарки з BenchmarkDotNet, а також навантажувальні тести з k6. Зібрані ключові показники включали час відгуку запитів, кількість транзакцій на секунду (TPS), використання пам'яті та CPU, а також кількість deadlock'ів у базі даних. Результати експериментів показали, що Dapper забезпечує значно вищу продуктивність у сценаріях читання та масових

операцій, тоді як EF Core забезпечує зручність розробки, підтримку міграцій, відстеження змін і цілісність транзакцій.

Порівняльний аналіз показав наступні закономірності:

- Dapper у 2–4 рази швидший при вибірці та масових операціях; EF Core відстає за рахунок накладних витрат на генерацію SQL та трекінг змін;
- Dapper дозволяє обробляти більшу кількість одночасних запитів при високому навантаженні, завдяки нижчому навантаженню на CPU клієнтського рівня;
- EF Core скорочує час розробки нових запитів та спрощує підтримку коду, завдяки LINQ, міграціям і автоматичному управлінню залежностями;
- Dapper надає повний контроль над SQL-запитами та оптимізацію складних сценаріїв, тоді як EF Core забезпечує контроль частково через FromSqlRaw;
- обидва фреймворки ефективно запобігають SQL-ін’єкціям при правильному використанні механізмів параметризації; EF Core додатково пропонує інтегровані механізми аудиту та автентифікації.

На основі отриманих результатів сформульовано практичні рекомендації:

- використовувати EF Core для корпоративних додатків зі складною об’єктною моделлю, де важлива швидкість розробки, підтримка міграцій та цілісність транзакцій;
- використовувати Dapper для систем із високим навантаженням на читання, аналітичних модулів, масових операцій та коли потрібен повний контроль над SQL;
- гібридний підхід, рекомендований для складних систем, дозволяє поєднувати EF Core для командної сторони (запис, бізнес-логіка) і Dapper для сторони запитів (читання, аналітика), а також використовувати мікросервісну архітектуру для розділення відповідальностей і оптимального вибору ORM для кожного сервісу.

Отримані висновки підтверджують, що правильний вибір ORM та обґрунтований гібридний підхід дозволяють досягти високої продуктивності, безпеки та зручності обслуговування корпоративних систем, що робить

результати цієї роботи цінними для практичного впровадження в реальних проектах.

Крім того, проведене дослідження демонструє можливості адаптації ORM-технологій до сучасних вимог корпоративних систем, включаючи інтеграцію з мікросервісною архітектурою, аналітичними платформами та високонавантаженими сервісами. Отримані результати можуть слугувати основою для розробки внутрішніх стандартів проєктування DAL, оптимізації продуктивності та управління ризиками при роботі з базами даних. У перспективі ці підходи можна розширювати шляхом інтеграції додаткових інструментів моніторингу, автоматизованого профайлінгу та інтелектуального балансування навантаження, що забезпечить ще більш ефективну та надійну роботу корпоративних інформаційних систем.

# СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ


1. Оптимізація роботи з базами даних за допомогою ORM (Object-Relational Mapping) [Електронний ресурс] / IT-Rating. — Режим доступу: https://it-rating.ua/optimizatsiya-roboti-z-bazami-danih-za-dopomogoyu-orm-object-relational-mapping. — Дата звернення: 08.11.2025.
2. A03 Injection - OWASP Top 10:2025 RC1. *OWASP Foundation, the Open Source Foundation for Application Security | OWASP Foundation*. URL: https://owasp.org/Top10/A03_2021-Injection/ (date of access: 09.11.2025).
3. OWASP Top 10 API Security Risks – 2023 - OWASP API Security Top 10. *OWASP Foundation, the Open Source Foundation for Application Security | OWASP Foundation*. URL: https://owasp.org/API-Security/editions/2023/en/0x11-t10/ (date of access: 09.11.2025).
4. SQL ін’єкції: що це, як працює, які типи існують, небезпечність. *FoxmindEd*. URL: https://foxminded.ua/sql-iniektsii/ (дата звернення: 09.11.2025).
5. SQL-ін’єкції в Ruby. *Форум Ruby для початківців - вивчаємо Рубі разом!*. URL: https://rubydevelopers.org/t/sql-ruby/186 (дата звернення: 09.11.2025).
6. *ELARTU – Інституційний репозитарій ТНТУ імені Івана Пулюя: Домівка*. URL: https://elartu.tntu.edu.ua/bitstream/lib/45024/2/VII_MCNTK_2024_Lanevych_T-ORM_frameworks_and_patterns_87-88.pdf (дата звернення: 09.11.2025).
7. SQL-ін’єкції - aCode. *aCode*. URL: https://acode.com.ua/sql-injection/ (дата звернення: 09.11.2025).
8. Cheat sheet: запобігання SQL-ін’єкціям - CoreWin. *CoreWin*. URL: https://corewin.ua/blog/sql-injection-prevention-cheat-sheet/ (дата звернення: 09.11.2025).

9. Bhushan S. Object Relational Impedance Mismatch. *Medium*. URL: https://medium.com/booleanbhushan/object-relational-impedance-mismatch-28fc2440dee4 (date of access: 09.11.2025).
10. Learn Entity Framework Core - Getting Started EF Core Tutorial. *Learn Entity Framework Core - Getting Started EF Core Tutorial*. URL: https://www.learnentityframeworkcore.com/ (date of access: 09.11.2025).
11. Fowler, M. Identity Map [Електронний ресурс]. // _Patterns of Enterprise Application Architecture_. [Електронний ресурс]. — Режим доступу: https://martinfowler.com/eaaCatalog/identityMap.html. — Дата звернення: 09.11.202
12. Identity Map and Request Context [Електронний ресурс] / MikroORM Documentation. — Режим доступу: https://mikro-orm.io/docs/identity-map. — Дата оновлення: 2024.
13. Understanding the Unit of Work Pattern in Software Development [Електронний ресурс] // GraphApp.AI Blog : [вебсайт]. — [Б. м.] : GraphApp.AI, 2023. — URL: https://www.graphapp.ai/blog/understanding-the-unit-of-work-pattern-in-software-development (дата звернення: 09.11.2025).
14. LINQ to SQL Queries [Електронний ресурс] / Microsoft Learn. — Режим доступу: https://learn.microsoft.com/en-us/dotnet/framework/data/adonet/sql/linq/linq-to-sql-queries. — Дата оновлення: 2024.
15. SQL to LINQ: The 10 Essential Queries for .NET Developers [Електронний ресурс] / DevAndShare. — Режим доступу https://medium.com/@DevAndShare/sql-to-linq-the-10-essential-queries-for-net-developers-f886c02d3fe7 . — Дата публікації: 2021.
16. Microsoft. Tracking vs. No-Tracking Queries [Електронний ресурс] // Microsoft Learn. — Redmond : Microsoft, 2025. — Режим доступу: https://learn.microsoft.com/en-us/ef/core/querying/tracking. — Дата звернення: 09.11.2025.

17. Overview of Entity Framework Core - EF Core. *Microsoft Learn: Build skills that open doors in your career*. URL: https://learn.microsoft.com/en-us/ef/core/ (date of access: 09.11.2025).
18. Plan for Entity Framework Core 8. *Microsoft Learn: Build skills that open doors in your career*. URL: https://learn.microsoft.com/en-us/ef/core/what-is-new/ef-core-8.0/plan (date of access: 09.12.2025).
19. Dapper C# (How It Works For Developers). *C# PDF Library (.NET 10 Compatible) | IronPDF*. URL: https://ironpdf.com/blog/net-help/dapper-csharp/ (date of access: 09.12.2025).
20. Dapper Tutorial. *Dapper Tutorial*. URL: https://dappertutorial.net/ (date of access: 09.12.2025).
21. Bind data using Dapper and perform CRUD operations | Syncfusion. *Help.Syncfusion.com*. URL: https://blazor.syncfusion.com/documentation/common/data-binding/dapper-databinding (date of access: 15.12.2025).
22. Dapper vs. Entity Framework – Which ORM is Better for .NET?. *Offshore Software Development Company - WireFuture*. URL: https://wirefuture.com/post/dapper-vs-entity-framework-which-orm-is-better-for-net?utm_source=chatgpt.com (date of access: 15.12.2025).
23. Dapper vs Entity Framework (EF6 or EF Core). *Welcome To Learn Dapper ORM - A Dapper Tutorial for C# and .NET Core*. URL: https://www.learndapper.com/dapper-vs-entity-framework?utm_source=chatgpt.com (date of access: 09.12.2025).
24. Mehta T. Data Access with Dapper: A Lightweight ORM for .NET Apps. *DEV Community*. URL: https://dev.to/wirefuture/data-access-with-dapper-a-lightweight-orm-for-net-apps-1adb?utm_source=chatgpt.com (date of access: 09.12.2025).
25. Wiatrowski T. Comparative analysis of ORM systems for the .NET platform. Journal of computer sciences institute. 2024. Vol. 31. P. 97–102. URL: https://doi.org/10.35784/jcsi.6012.

26. Welcome To Learn Dapper ORM - A Dapper Tutorial for C# and .NET Core. *Welcome To Learn Dapper ORM - A Dapper Tutorial for C# and .NET Core*. URL: https://www.learndapper.com/ (date of access: 15.12.2025).
27. Wiatrowski T. Comparative Analysis of ORM Systems for the .NET Platform. *Journal of Computer Sciences Institute*, 31, 97–102, 2024.
28. Wiatrowski T. Comparative Analysis of ORM Systems for the .NET Platform. *Journal of Computer Sciences Institute*. 2024. Vol. 31. P. 97–102. URL: https://doi.org/10.35784/jcsi.6012 (date of access: 09.12.2025).

# АНОТАЦІЯ

**Тань Ченлун** – Порівняння ефективності та функціональних можливостей ORM-систем у .NET

Рукопис. Кваліфікаційна робота на здобуття освітнього ступеня «магістр» за спеціальністю 122 Комп'ютерні науки, освітня програма «Комп'ютерні науки та інформаційні технології». – Волинський національний університет імені Лесі Українки. – 2025.

У цій роботі досліджується технологія Object-Relational Mapping (ORM) для дизайну шару збереження даних у корпоративних додатках та управління ризиками. ORM – це технологія програмування, яка створює шар між реляційними базами даних та об'єктно-орієнтованими мовами програмування без необхідності написання SQL-запитів.

У роботі аналізується роль управління ризиками підприємства у роботі з базами даних з теоретичної точки зору і досліджується проблема «імпедансної невідповідності» між об'єктно-орієнтованим програмуванням та реляційними базами даних. Викладено принципи роботи, архітектура, ключові концепції ORM та її еволюція в середовищі .NET. На цій основі детально розроблена схема реалізації шару збереження даних на основі Entity Framework Core та Dapper, включаючи такі ключові технології, як конфігурація контексту, відображення сутностей та оптимізація запитів. Пропонуються стратегії оптимізації продуктивності та управління ризиками для ORM у корпоративних додатках, а через експериментальний дизайн та аналіз випадків перевіряється життєздатність та ефективність запропонованої схеми. Результати дослідження показують, що правильний дизайн ORM та управління ризиками можуть значно підвищити продуктивність, безпеку та зручність обслуговування системи, що є важливим орієнтиром для розробки корпоративних додатків.

**Ключові слова:** Об'єктно-реляційне відображення, імпедансна невідповідність, шар збереження даних, Entity Framework, Dapper.

## ABSTRACT

**Tan Chenglong – Comparison of the Efficiency and Functional Capabilities of ORM Systems in .NET**

Manuscript. Qualification work for the degree of "Master" in the field of Computer Science, educational program "Computer Science and Information Technologies." – Lesya Ukrainka Volyn National University. – 2025.

This paper investigates Object-Relational Mapping (ORM) technology for designing the data persistence layer in corporate applications and managing associated risks. ORM is a programming technology that introduces an abstraction layer between relational databases and object-oriented programming languages, eliminating the need to write SQL queries manually.

The study examines the role of enterprise risk management in database operations from a theoretical perspective and explores the problem of object–relational impedance mismatch between object-oriented programming and relational databases. The principles of operation, architecture, key ORM concepts, and their evolution within the .NET ecosystem are presented. Based on this foundation, a detailed implementation of the data persistence layer using Entity Framework Core and Dapper is developed, including key techniques such as context configuration, entity mapping, and query optimization. Strategies for performance optimization and risk management when using ORM in corporate applications are proposed, and the feasibility and effectiveness of the suggested approach are validated through experimental design and case analysis.

The results of the study demonstrate that proper ORM design combined with effective risk management can significantly improve system performance, security, and maintainability, which is a critical factor in the development of enterprise-level applications.



**Keywords:** Object-Relational Mapping, impedance mismatch, data persistence layer, Entity Framework, Dapper.