КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ТАРАСА ШЕВЧЕНКА
Факультет комп'ютерних наук та кібернетики
Кафедра математичної інформатики

Кваліфікаційна робота
на здобуття ступеня бакалавра
за спеціальністю 122 Комп'ютерні науки
на тему:

ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ NOSQL СУБД: ЗБІЛЬШЕННЯ ПРОДУКТИВНОСТІ ТА МАСШТАБОВАНОСТІ

Виконав студент 4 курсу
ТАРАНЮК Дмитро

_____
(підпис)

Науковий керівник:
асистент, кандидат технічних наук
ФЕДОРУС Олексій

_____
(підпис)

Засвідчую, що в цій роботі немає запозичень з праць інших авторів без відповід
Студент

_____
(підпис)

Роботу розглянуто й допущено до захисту на засіданні кафедри математичної інформатики
«____» _____
Завідувач кафедри
ТЕРЕЩЕНКО Василь

_____
(підпис)

Київ - 2023

ЗМІСТ

РЕФЕРАТ

Обсяг роботи: 40 сторінок, 4 рисунка, 4 таблиці, 5 джерел посилань.

Метою цього дослідження було оцінити та порівняти продуктивність чотирьох провідних систем управління базами даних (СУБД): MongoDB, Cassandra, Redis та Memcached. Намір полягав у тому, щоб виявити їхні відносні сильні та слабкі сторони в різних контекстах застосування, таких як розмір даних, складність запитів та рівень паралелізму.

Для досягнення цієї мети кожна СУБД була налаштована і піддана різноманітним тестам, спеціально розробленим для імітації потенційних реальних сценаріїв. Ці тести сприяли збору емпіричних даних про те, як кожна СУБД може працювати в конкретних умовах.

Первинні результати показали, що MongoDB і Cassandra перевершили Redis і Memcached при роботі зі складними запитами, продемонструвавши їх надійну обробку складних взаємодій з даними. На противагу цьому, Redis і Memcached продемонстрували чудову продуктивність у високошвидкісних ситуаціях з високим рівнем паралелізму, що підкреслює їх придатність для додатків, де швидкість має першочергове значення.

По суті, це дослідження показало, що вибір СУБД повинен сильно залежати від конкретних потреб і вимог конкретного додатку. Таким чином, воно розвіює поняття "універсального рішення", коли мова йде про системи управління базами даних, і підкреслює важливість цілеспрямованого вибору і тестування.

ВСТУП

У швидкоплинну цифрову епоху важливість баз даних та їхньої продуктивності вийшла на перший план. В наш час є дуже багатий вибір баз даних, що пропонує безліч варіантів, кожен з яких має свої унікальні атрибути, переваги та обмеження. Серед безлічі варіантів MongoDB, Cassandra, Redis та Memcached виділяються своїми можливостями та доведеною ефективністю в різних сценаріях. Однак, всебічне розуміння продуктивності цих баз даних в різних умовах експлуатації залишається складною і недослідженою сферою, що вимагає ретельного вивчення.

Актуальність цього дослідження підкреслюється зростаючою залежністю від цих баз даних для цілого ряду застосувань, включаючи, але не обмежуючись ними, веб-розробку, аналітику в реальному часі та обробку великих даних. Вибір бази даних суттєво впливає на продуктивність, масштабованість та ефективність продукту. Тому глибоке розуміння поведінки цих баз даних при різних робочих навантаженнях та умовах експлуатації має вирішальне значення як для академічних кіл, так і для промисловості.

Основною метою цього дослідження є проведення ретельного бенчмаркінгового аналізу баз даних MongoDB, Cassandra, Redis та Memcached. Це дослідження спрямоване на розробку емпіричного розуміння їхніх характеристик продуктивності, тим самим надаючи дані, які можуть допомогти у виборі відповідної бази даних для конкретних застосувань.

Об'єктами дослідження є бази даних MongoDB, Cassandra, Redis та Memcached. Для вивчення динаміки їхньої продуктивності використовується подвійна методологія дослідження. По-перше, бази даних піддаються ідентичному набору навантажень, щоб зрозуміти їхню операційну

ефективність за схожих умов. По-друге, вони аналізуються за різних навантажень та умов роботи, щоб простежити, як змінюється їхня продуктивність за різних сценаріїв. Цей подвійний підхід забезпечує комплексну та порівняльну перспективу продуктивності цих баз даних, що дозволяє приймати більш обґрунтовані рішення щодо вибору та використання баз даних.

РОЗДІЛ 1. ОГЛЯД СУБД, ЩО ДОСЛІДЖУЮТЬСЯ

1.1. Redis

Redis, відомий також як "віддалений сервер словників", представляє собою високопродуктивне сховище структур даних у пам'яті, написане з використанням відкритого вихідного коду. Ця інноваційна технологія знайшла широке застосування у різних областях, включаючи бази даних, кешування даних та брокерів повідомлень, пропонуючи користувачам гнучкість та високу швидкість обробки.

Однією з ключових особливостей Redis є його унікальна модель даних "ключ-значення", що значно відрізняється від традиційних реляційних баз даних. Ця модель дозволяє зберігати дані в пам'яті за допомогою пар ключ-значення, що робить процес зберігання та пошуку даних надзвичайно швидким та ефективним.

Окрім цього, Redis підтримує різноманітні типи структур даних. Зокрема, він дозволяє користувачам використовувати рядки, хеш-таблиці, списки, множини та інші формати даних, що значно розширює границі його можливого застосування. Таке багатогранне використання даних робить Redis незамінним інструментом для різних бізнес-задач.

Redis вирізняється своєю винятковою продуктивністю та швидкістю завдяки тому, що він працює в оперативній пам'яті. Це робить його ідеальним вибором для таких задач, як кешування, управління сесіями, аналітика в реальному часі та інші схожі застосування, які вимагають моментального відгуку.

Однак, попри всі ці переваги, Redis має одну потенційну проблему: його пам'ять нестабільна. Це означає, що при відновленні системи або втраті

електропостачання дані можуть бути втрачені. Щоб подолати цю проблему, Redis пропонує два рішення: періодичне створення знімків пам'яті або додавання кожної команди до журналу. Обидва підходи допомагають забезпечити довговічність даних, які зберігаються в Redis, заходячи вперед до можливих втрат.

1.2. MongoDB

MongoDB – це відкрита і вільно доступна NoSQL база даних, зорієнтована на роботу з документами. Вона здобула широке визнання завдяки своїй вражаючій масштабованості та гнучкості, що дає їй можливість легко пристосовуватися до широкого спектру бізнес-вимог та викликів, що ставляться перед сучасними базами даних.

Основним відмінним атрибутом MongoDB є її використання JSON-подібних документів з можливістю додавання гнучких схем. Ця альтернатива традиційному табличному формату реляційних баз даних надає більше свободи у представленні даних. Документна модель MongoDB забезпечує гнучкість, дозволяючи користувачам зберігати дані у форматах, які більш точно відображають структуру їх застосувань.

Завдяки цій унікальній структурі, MongoDB виявила свою велику корисність у ряді сучасних застосувань. Вона стала особливо цінною у розробці систем управління контентом, рішеннях для Інтернету речей, проведенні аналітики в реальному часі та багатьох інших областях, де потрібна швидкість, продуктивність та масштабованість.

Але не дивлячись на всі ці переваги, MongoDB має свої виклики та обмеження. Ключовим з них є те, що вона може бути менш ефективною в обробці складних транзакцій або агрегацій даних порівняно з традиційними

SQL базами даних. Це є важливим чинником, який слід враховувати при плануванні використання MongoDB, особливо в контекстах, де складні транзакції або агрегації є ключовими аспектами бізнес-логіки.

Для розробників і архітекторів баз даних це означає, що вони повинні тщательно обмірковувати свій вибір технології бази даних, уважно вивчаючи преимущества та обмеження MongoDB. Важливо розуміти, що попри всі її сильні сторони, MongoDB, як і будь-яка інша технологія, має свої обмеження і не завжди може бути найкращим рішенням для всіх сценаріїв.

1.3. Memcached

Memcached представляє собою високопродуктивну розподілену систему кешування об'єктів в оперативній пам'яті, яка використовується для значного прискорення роботи динамічних веб-додатків. Це досягається за рахунок зменшення навантаження на базу даних, шляхом зберігання та швидкого отримання в пам'яті даних, що найчастіше використовуються.Memcached відомий своєю простотою та ефективністю. Він використовує простий протокол на основі "ключ-значення", що спрощує процес кешування даних, знижуючи витрати на обробку інтенсивних запитів до баз даних. Це робить Memcached відмінним вибором для веб-додатків, що потребують високої швидкості відгуку.

Втім, на відміну від деяких інших систем кешування, таких як Redis, Memcached не надає підтримку зберігання даних або використання багатоманітних структур даних. Це може бути обмеженням для деяких сценаріїв, але для багатьох веб-застосунків, основна потреба - це швидкість та ефективність, а не розмаїтість структур даних.

Проте Memcached має свої обмеження, і одним з них є відсутність підтримки реплікації та журналів транзакцій. Ці функції можуть бути критичними для деяких сценаріїв, особливо коли потрібна висока надійність або персистентність даних.

Не дивлячись на ці обмеження, Memcached є одним з найпопулярніших інструментів кешування для веб-додатків. Його легкість використання, висока продуктивність та можливість значно знизити навантаження на базу даних роблять його привабливим вибором для багатьох розробників.

Попри все це, важливо усвідомлювати, що обираючи інструмент кешування, слід ретельно розглядати вимоги конкретного додатку. Випадки використання Memcached часто включають кешування результатів бази даних, кешування сесій або об'єктів, що не часто змінюються. Втім, у ситуаціях, коли вам потрібні більш складні структури даних, або якщо ви шукаєте функції, такі як реплікація, Memcached може не бути найкращим вибором.

1.4. Cassandra

Apache Cassandra представляє собою потужну розподілену NoSQL систему баз даних з відкритим вихідним кодом, спроектовану з огляду на обробку величезних обсягів даних. Ця система даних розроблена таким чином, що вона здатна ефективно працювати на великому числі комодитних (стандартних) серверів, забезпечуючи при цьому високу доступність без єдиної точки відмови. Це дозволяє забезпечити надійність та доступність даних, навіть у випадку відмови окремих вузлів.

Ключовою перевагою Apache Cassandra є її виняткова масштабованість та відмовостійкість, які роблять її ідеальним вибором для додатків, що потребують обробки великих обсягів даних. Вона широко використовується у

веб-додатках великого масштабу, системах обміну повідомленнями, системах реального часу для аналізу даних з датчиків та інших сценаріях, які вимагають високої доступності та масштабування.

Однак, незважаючи на величезні переваги Cassandra в плані масштабованості та відмовостійкості, її архітектура може призвести до компромісів у деяких інших областях. Наприклад, акцент на масштабованість може викликати складнощі при обробці більш складних запитів або транзакцій, які в традиційних реляційних базах даних можуть бути оброблені ефективніше.

Крім того, попри те, що Apache Cassandra є високо відмовостійкою, це не означає, що вона вільна від проблем. Бази даних такого масштабу вимагають значних зусиль для налаштування, управління та обслуговування, а це може бути викликом для команд, що не мають достатнього досвіду. Окрім того, певні види запитів та транзакцій можуть виявитися менш ефективними, ніж у традиційних SQL-системах, оскільки Apache Cassandra оптимізована для великих обсягів даних, а не для складних запитів.

Таким чином, при виборі Apache Cassandra як основної системи управління базами даних для вашого проекту, важливо ретельно вивчити її можливості та обмеження. Попри високу масштабованість та відмовостійкість, вона може не бути оптимальним вибором для сценаріїв, які вимагають складних запитів або транзакцій.

РОЗДІЛ 2. ТЕОРІЯ, ЩО ЛЕЖИТЬ В ОСНОВІ БЕНЧМАРКІНГУ БАЗ ДАНИХ

Бенчмаркінг баз даних є усталеною практикою, що забезпечує кількісну оцінку продуктивності бази даних за різних умов. Численні дослідження вивчали різні методології бенчмаркінгу, від синтетичних тестів до симуляцій реальних додатків. У цьому підрозділі ми заглибимося в цю літературу, досліджуючи різні методи бенчмаркінгу, їхні сильні та слабкі сторони, а також їхню актуальність для цього дослідження.

2.1. Що таке бенчмаркінг?

Бенчмаркінг - це метод порівняння продуктивності різних систем або інструментів між собою за допомогою стандартизованого набору метрик. У контексті баз даних бенчмаркінг передбачає виконання конкретних завдань або операцій, які повинна виконувати база даних, і вимірювання часу, необхідного для їх виконання, або обсягу транзакцій, які вона може обробити за певний період. Мета бенчмаркінгу - зрозуміти сильні і слабкі сторони різних систем і визначити області для поліпшення.[6]

2.2. Важливість бенчмаркінгу в базах даних

Порівняльний аналіз є критично важливим для баз даних з кількох причин:

- Оцінка продуктивності: Допомагає оцінити продуктивність різних систем управління базами даних (СУБД), надаючи уявлення про їхню операційну ефективність, швидкість і надійність.
- Вибір системи: Результати бенчмаркінгу допомагають організаціям у виборі найбільш підходящої СУБД для конкретного випадку використання або потреб додатків.
- Оптимізація: Допомагає виявити вузькі місця в роботі СУБД і області для поліпшення, що дозволяє розробникам і адміністраторам баз даних оптимізувати систему для підвищення її продуктивності.
- Порівняння: Надає платформу для порівняння різних СУБД в однакових умовах, забезпечуючи об'єктивність порівняння.[7]

2.3. Фактори, що впливають на продуктивність бази даних

На продуктивність бази даних можуть впливати кілька факторів. Ці фактори можна умовно поділити на апаратні, програмні та пов'язані з робочим навантаженням:

- Апаратне забезпечення: включає такі фактори, як швидкість і кількість процесорів, обсяг і тип оперативної пам'яті, тип і конфігурація пристроїв зберігання даних і пропускна здатність мережі.

- Програмне забезпечення: включає тип СУБД (реляційна, NoSQL тощо), ефективність схеми бази даних, наявність індексів та використання оптимізованих запитів.
- Робоче навантаження: Характер робочого навантаження суттєво впливає на продуктивність бази даних. Наприклад, робоче навантаження, яке є важким для читання, матиме інші характеристики продуктивності порівняно з робочим навантаженням, яке є важким для запису.

2.4. Загальні показники продуктивності

Наступні метрики зазвичай використовуються для вимірювання продуктивності бази даних:

- Пропускна здатність: Кількість операцій, які база даних може виконати за одиницю часу. Вища пропускна здатність свідчить про кращу продуктивність.
- Затримка: Час, необхідний для завершення однієї операції. Менша затримка свідчить про кращу продуктивність.
- Паралельність: Кількість операцій, які можуть виконуватися одночасно без зниження продуктивності.
- Масштабованість: Здатність бази даних підтримувати рівень продуктивності при збільшенні робочого навантаження або попиту.[8]

2.5. Огляд конкретних тестів, що використовуються в бенчмарках

Тести бенчмаркінгу, що використовуються в цьому дослідженні, призначені для вимірювання різних аспектів продуктивності бази даних. Ці тести включають:

- Створення записів: Цей тест вимірює швидкість, з якою нові записи можуть бути додані до бази даних, як масово, так і індивідуально.
- Читання записів: Цей тест вимірює швидкість, з якою записи можуть бути отримані з бази даних, як масово, так і окремо.
- Оновлення записів: Цей тест вимірює швидкість, з якою можна змінювати існуючі записи, як масово, так і окремо.
- Видалення записів: Цей тест вимірює швидкість видалення записів з бази даних, як групових, так і індивідуальних.
- Складний запит: Цей тест вимірює швидкість, з якою база даних може виконувати складні запити, що включають кілька таблиць і умов.
- Високий паралелізм: Цей тест вимірює здатність бази даних обробляти кілька одночасних операцій без значного падіння продуктивності.
- Великий обсяг: Цей тест вимірює продуктивність бази даних при обробці великих обсягів даних.

РОЗДІЛ 3. ТЕХНІЧНІ ХАРАКТЕРИСТИКИ АПАРАТНОГО ТА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1. Апаратні характеристики

Тестування проводилося на комп'ютері з наступними характеристиками:

- Процесор: AMD Ryzen 5 4600H з графікою Radeon, що працює на частоті 3 ГГц з 6 фізичними ядрами та 12 логічними процесорами.
- Встановлена фізична пам'ять (оперативна пам'ять): 16 ГБ
- Сховище даних: Локальний фіксований диск об'ємом 470,45 ГБ, що використовує файлову систему NTFS, з 93,50 ГБ вільного місця, доступного на початку тестування. Розмір файлу підкачки становив 11,0 ГБ.

3.2. Технічні характеристики програмного забезпечення

Програмне середовище для проведення порівняльних тестів було наступним:

- Операційна система: Windows 10
- Версія Docker: 20.10.12, збірка e91ed57

Версії баз даних, що використовуються в контейнерах Docker:

- Redis: 6.0
- MongoDB: 4.4
- Memcached: 1.6
- Cassandra: 4.0

3.3. Встановлення та налаштування бази даних

Кожна СУБД була встановлена у своїй останній стабільній версії, доступній на момент тестування, з використанням контейнерів Docker. Контейнери були налаштовані таким чином, щоб виділити оптимальну частку системних ресурсів для кожного з них, що забезпечило справедливе та точне бенчмаркове тестування.

3.4. Програмне забезпечення для бенчмаркінгу

Тестування проводилося за допомогою спеціального інструменту бенчмаркінгу, створеного для цього дослідження, написаного на мові C# з використанням середовища виконання .NET Core 7.0. Версії відповідних клієнтів/драйверів баз даних, що використовувалися в інструменті бенчмаркінгу, були наступними:

- StackExchange.Redis[3]: 2.6.111
- MongoDB.Driver[1]: 2.19.1
- EnyimMemcachedCore[4]: 2.16.0
- CassandraCSharpDriver[2]: 3.19.2

3.4.1. Деталі реалізації

Архітектура цього засобу для бенчмаркінгу демонструє ефективні принципи проектування та організована у вигляді окремих компонентів: BenchmarkRunner, IDbService, BenchmarkResults та різноманітні класи Scenario. Розділення завдань відображає модульний підхід, забезпечуючи систему, яку

можна гнучко і легко розширювати за допомогою нових сценаріїв або для підтримки додаткових баз даних.

```
1 usage
public async Task<BenchmarkResults> RunAsync()
{
    var benchmarkResults = new BenchmarkResults();

    for (int i = 0; i < RunCount; ++i)
    {
        Console.WriteLine($"Starting {i}th run for {DbService.GetType().Name}");
        foreach (var scenario in Scenarios)
        {
            Console.WriteLine($"Starting {scenario.GetType().Name}");
            try
            {
                var scenarioResults = await scenario.ExecuteScenarioAsync();
                benchmarkResults.ScenarioResults.Add(scenarioResults);
            }
            catch (Exception e)
            {
                Console.WriteLine($"An error occurred while executing" +
                                  $" {scenario.GetType().Name} on run {i+1}: {e.Message}");
                benchmarkResults.ScenarioResults.Add(item: new ScenarioResults()
                {
                    ScenarioName = scenario.GetType().Name,
                    SuccessRate = 0
                });
            }
        }
    }

    return benchmarkResults;
}
```

Рисунок 3.1 - Метод RunAsync класу BenchmarkRunner

BenchmarkRunner слугує головним оркестром системи, налаштовуючи, виконуючи та очищаючи систему після кожного сценарію бенчмаркінгу. Цей центральний контроль гарантує, що кожен тест ізольований і виконується в ідентичних умовах, забезпечуючи справедливе порівняння між різними системами управління базами даних (СУБД). Набір також використовує

рефлексію для динамічного завантаження сценаріїв і служб баз даних, що додає ще один рівень гнучкості. Такий підхід полегшує додавання нових тестів або СУБД простим створенням нового класу.

```
21 usages  5 inheritors  1 exposing API
3  public interface IDbService
4  {
       2 usages  5 implementations
5      Task SetupAsync();
       1 usage  5 implementations
6      Task CleanupAsync<T>();
       6 usages  5 implementations
7      Task CreateRecordAsync<T>(string key, T data);
       3 usages  5 implementations
8      Task<T> ReadRecordAsync<T>(string key);
       2 usages  5 implementations
9      Task UpdateRecordAsync<T>(string key, T data);
       3 usages  5 implementations
10     Task DeleteRecordAsync<T>(string key);
       5 usages  5 implementations
11     Task CreateRecordsAsync<T>(IDictionary<string, T> records);
       4 usages  5 implementations
12     Task<IList<T>> ReadRecordsAsync<T>(IEnumerable<string> keys);
       3 usages  5 implementations
13     Task UpdateRecordsAsync<T>(IDictionary<string, T> records);
       3 usages  5 implementations
14     Task DeleteRecordsAsync<T>(IEnumerable<string> keys);
       1 usage  5 implementations
15     Task<IList<T>> ComplexQueryAsync<T>(string fieldName, object value, int limit);
16 }
```

Рисунок 3.2 - Інтерфейс IDbService

Інтерфейс IDbService абстрагується від специфіки взаємодії з різними СУБД. Доступ до кожної СУБД здійснюється через службовий клас, який реалізує цей інтерфейс, що означає, що BenchmarkRunner може взаємодіяти з будь-якою СУБД у послідовний спосіб. Це забезпечує справедливість, оскільки під час тестування всі СУБД розглядаються однаково. Можливість додавання

нових СУБД до системи шляхом створення нового сервісного класу, що реалізує інтерфейс IDbService, також демонструє адаптивність системи.

```
19      public virtual async Task<ScenarioResults> ExecuteScenarioAsync()
20      {
21          await SetupAsync();
22
23          ScenarioResults results;
24          try
25          {
26              results = await RunAsync();
27          }
28          finally
29          {
30              await CleanupAsync();
31          }
32
33          return results;
34      }
35  }
```

Рисунок 3.3 - Метод ExecuteScenarioAsync базового класу Scenario

Класи Scenario втілюють логіку кожного окремого тесту бенчмарку. Інкапсуляція кожного тесту у власний клас спрощує процес додавання, видалення або модифікації тестів без впливу на решту системи. Такий модульний дизайн не тільки забезпечує масштабованість і гнучкість, але й допомагає гарантувати справедливість, ізолюючи кожен тест. Використання абстрактного базового класу для сценаріїв пропонує чіткий шаблон для створення нових тестів, а включення асинхронних методів дозволяє тестувати при високому навантаженні та паралельності.

```
1 usage
public string ToCsv()
{
    StringBuilder csv = new StringBuilder();
    csv.AppendLine("ScenarioName,Duration,Passed,ErrorMessage");

    foreach (var scenarioResult :ScenarioResults in ScenarioResults)
    {
        csv.AppendLine($"{scenarioResult.ScenarioName}," +
                       $"{scenarioResult.GetTotalDuration()}," +
                       $"{scenarioResult.GetSuccessRate()}");
    }

    return csv.ToString();
}
```

Рисунок 3.4 - Метод ToCSV класу BenchmarkResults

Клас BenchmarkResults є уніфікованим механізмом для запису та звітування про результати еталонних тестів. Використання формату CSV спрощує аналіз результатів за допомогою різних інструментів, а включення обробки помилок гарантує, що будь-які проблеми, які виникають під час тестів, будуть задокументовані для подальшого дослідження.

Система була побудована з використанням .NET Core та C#, використовуючи потужні можливості мови та середовища виконання, такі як async/await для паралельних операцій та рефлексія для сценаріїв та сервісів, що динамічно завантажуються. Крос-платформна підтримка .NET Core означає, що система може працювати на широкому спектрі апаратного забезпечення та операційних систем, що підвищує її гнучкість та масштабованість.

Система включає як загальні, так і специфічні сценарії, що забезпечує ретельну оцінку кожної СУБД. Загальні сценарії тестують загальні операції,

які повинна виконувати будь-яка СУБД, в той час як специфічні сценарії розроблені для перевірки сильних і слабких сторін кожної конкретної СУБД. Така комбінація забезпечує всебічну оцінку кожної СУБД, сприяючи об'єктивності тестів.

Таким чином, архітектура цієї системи цілеспрямовано розроблена для забезпечення гнучкості, масштабованості та об'єктивності еталонних тестів. Вона забезпечує надійну основу для оцінки та порівняння продуктивності різних СУБД, що робить її безцінним інструментом для будь-якої організації, яка прагне приймати обґрунтовані рішення щодо своєї інфраструктури баз даних.

3.4.2. Опис кожного тестового сценарію

Всього налічується 11 різних сценаріїв, кожен з яких призначений для перевірки певного аспекту продуктивності кожної бази даних. Вони перелічені нижче, зі стислими описами:

- CreateRecordsBulk: Цей тест має на меті оцінити продуктивність створення декількох записів в масі. Він тестує продуктивність бази даних при обробці великої кількості записів одночасно.
- CreateRecordsSingle: Цей тест схожий на попередній, але він вставляє записи окремо. Цей сценарій дає уявлення про продуктивність бази даних під час створення одного запису.
- ReadRecordsBulk: Цей тест зчитує велику кількість записів одночасно. Він слугує для порівняння продуктивності бази даних при роботі з великими обсягами даних.

- ReadRecordsSingle: Цей сценарій зчитує окремі записи з бази даних, імітуючи поширений варіант використання для багатьох додатків.
- UpdateRecordsBulk: Цей тест змінює велику кількість записів одночасно. Він дає уявлення про продуктивність оновлення бази даних при роботі з великими обсягами даних.
- UpdateRecordsSingle: Цей сценарій оновлює окремі записи, що є поширеною операцією в багатьох додатках.
- DeleteRecordsBulk: Цей тест видаляє велику кількість записів одночасно. Він дає уявлення про продуктивність бази даних при видаленні необроблених даних.
- DeleteRecordsSingle: Цей сценарій видаляє окремі записи. Це звичайна операція, яка може дати уявлення про те, як кожна база даних обробляє видалення записів.
- ComplexQuery: Цей тест виконує складний запит, який включає фільтрацію і сортування. Він допомагає зрозуміти продуктивність бази даних при виконанні складних операцій, які часто зустрічаються в реальних додатках.
- HighConcurrency: Цей тест виконує велику кількість операцій одночасно, щоб оцінити продуктивність бази даних під високим навантаженням.
- LargeVolume: Цей тест призначений для оцінки продуктивності бази даних при роботі з великим обсягом даних.

Кожен з вищезгаданих тестів був реалізований як окремий сценарій в нашому засобі для бенчмаркінгу. Особливості виконання кожного тесту, включаючи кількість використовуваних записів, рівень паралелізму та конкретні операції, визначалися на основі загальних шаблонів використання, що спостерігаються в реальних додатках.

3.5. Очікувані результати

Очікувалося, що кожен сценарій дасть уявлення про характеристики продуктивності кожної бази даних за певних умов. Наприклад, тести масових операцій дадуть розуміння здатності бази даних обробляти великі обсяги операцій одночасно, в той час як тести одиночних операцій підкреслять продуктивність в більш детальних, транзакційних операціях. Тест складних запитів продемонструє, як кожна база даних обробляє складні обчислення і фільтрацію даних, тоді як тести високого рівня паралелізму і великих обсягів даних покажуть продуктивність в умовах високого навантаження або великого обсягу даних.

РОЗДІЛ 4. РЕЗУЛЬТАТИ БЕНЧМАРКІНГУ ТА ЇХ АНАЛІЗ

У цьому розділі представлені результати бенчмаркінгових тестів і наданий аналіз того, що ці результати означають з точки зору характеристик продуктивності кожної бази даних. Результати відображені в табличному форматі, що дозволить легко порівняти різні бази даних для кожного тестового сценарію.

Подальший аналіз включатиме обговорення того, чому певні бази даних працювали краще в конкретних сценаріях. Фактори, які будуть розглянуті, включатимуть архітектурні відмінності між базами даних, те, як вони обробляють різні типи операцій, структуру даних, можливості індексування та інші аспекти, які можуть вплинути на їхню продуктивність.

Кожен сценарій буде проаналізовано окремо, а потім порівняно для виявлення сильних і слабких сторін кожної бази даних, що дасть змогу отримати комплексний огляд їхніх характеристик продуктивності. Це дозволить надати обґрунтовані рекомендації щодо того, яку базу даних використовувати в різних ситуаціях, на основі їхньої перевіреної продуктивності в кожному з тестових сценаріїв.

4.1. Результати

Нижче будуть приведені конкретні результати виконання сценаріїв з наступними налаштуваннями:

```
var crudSingleScenarioParams = new ScenarioParameters()
{
    ItemCount = 10000,
    Mode = CrudMode.Single
};
var crudBulkScenarioParams = new ScenarioParameters()
{
    ItemCount = 10000,
    Mode = CrudMode.Bulk
};
var complexQueryScenarioParams = new ScenarioParameters()
{
    ItemCount = 10000,
    SortField = 5,
    Limit = 100
};
var highConcurrencyScenarioParams = new ScenarioParameters()
{
    ItemCount = 10000
};
var largeVolumeScenarioParams = new ScenarioParameters()
{
    ItemCount = 100000
};
```

Рисунок 4.1 - Налаштування сценаріїв в класі BenchmarkRunner

Як видно на рис. 1, для окремих CRUD(Create, Read, Update, Delete) операцій кількість елементів була обрана в обсязі 10,000, так само, як і для тих

самих запитів «масово» (Bulk), для сценарію «складний запит» кількість елементів така сама, з яких треба знайти 100 максимальних, порівнюючи по полю номер 5 (чи залишку при діленні 5 на к-сть полів, якщо їх менше 6), для сценарію з високою многопотоковістю ті самі 10000 елементів, проте операції з ними виконуються не «масово», а паралельно, і нарешті, 100,000 елементів для тесту великого об’єму даних. Нижче наведені результати, отримані для кожної СУБД. Успішність показує відсоток успішних виконань сценарію протягом тесту, від 0 до 1, успішність x означає x*100 % успішних виконань. Час виконання приводиться середній по всім тестовим запускам, в форматі mm:ss.000 (хвилини:секунди, де секунди приведені з точністю до 3 цифр після крапки).

4.1.1. Cassandra

Таблиця 4.1 - Результати бенчмаркінгу Cassandra

Назва сценарію	Час виконання (сек)	Успішність
CreateRecordsBulk	00:00.115	1
CreateRecordsSingle	00:12.115	1
ReadRecordsBulk	00:00.313	1
ReadRecordsSingle	00:12.721	1
UpdateRecordsBulk	00:00.115	1
UpdateRecordsSingle	00:12.246	1
DeleteRecordsBulk	00:00.074	1
DeleteRecordsSingle	00:11.684	1
ComplexQuery	00:00.026	1
HighConcurrency	00:00.823	1
LargeVolume	00:05.072	0.8

Cassandra продемонструвала винятково хороші результати при виконанні масових операцій: всі операції створення, читання, оновлення та видалення були виконані менш ніж за секунду. Операції з окремими записами займали більше часу, середня тривалість яких становила близько 12 секунд. Операції з високим рівнем паралелізму оброблялися добре, середній час завершення становив приблизно 0,82 секунди. Операція LargeVolume мала змішані результати: один екземпляр не впорався, але інші успішно завершилися із середнім часом близько 6 секунд. Операція ComplexQuery була оброблена з винятковою швидкістю, що свідчить про те, що Cassandra добре підходить для сценаріїв, де складні запити є поширеним явищем.

4.1.2. Memcached

Таблиця 4.2 - Результати бенчмаркінгу Memcached

Назва сценарію	Час виконання (сек)	Успішність
CreateRecordsBulk	00:09.234	1
CreateRecordsSingle	00:09.213	1
ReadRecordsBulk	00:09.320	1
ReadRecordsSingle	00:09.244	1
UpdateRecordsBulk	00:09.531	1
UpdateRecordsSingle	00:09.578	1
DeleteRecordsBulk	00:08.851	1
DeleteRecordsSingle	00:08.881	1
ComplexQuery	00:00.000	0
HighConcurrency	00:02.057	1
LargeVolume	06:06.921	1

Порівняно з Cassandra, операції Memcached виконувалися значно довше. Масові операції та операції з окремими записами зайняли близько 9 секунд. Слід зазначити, що Memcached не підтримує сценарій ComplexQuery, що може бути обмеженням для деяких сценаріїв використання. Операції HighConcurrency оброблялися досить швидко, в середньому близько 2 секунд. Memcached добре впоралася зі сценарієм LargeVolume, пройшовши всі тести, хоча це зайняло більше 6 хвилин.

4.1.3. Mongo

Таблиця 4.3 - Результати бенчмаркінгу MongoDB

Назва сценарію	Час виконання (сек)	Успішність
CreateRecordsBulk	00:00.296	1
CreateRecordsSingle	00:14.255	1
ReadRecordsBulk	00:00.119	1
ReadRecordsSingle	00:14.493	1
UpdateRecordsBulk	00:01.157	1
UpdateRecordsSingle	00:14.935	1
DeleteRecordsBulk	00:00.018	1
DeleteRecordsSingle	00:13.464	1
ComplexQuery	00:00.012	1
HighConcurrency	00:03.773	1
LargeVolume	00:14.553	1

Mongo показала широкий діапазон продуктивності. Для масових операцій операції створення та читання були виконані дуже швидко, менше ніж за 0,2 секунди. Операції оновлення зайняли більше часу, в середньому близько 1,2 секунди. Операції з окремими записами виконувалися повільніше, із середнім часом завершення близько 14 секунд. Операція ComplexQuery була оброблена дуже швидко, а операції HighConcurrency зайняли в середньому близько 3,8 секунди. Mongo добре впоралася зі сценарієм LargeVolume із середнім часом виконання близько 14,5 секунд.

4.1.4. Redis

Таблиця 4.4 - Результати бенчмаркінгу Redis

Назва сценарію	Час виконання (сек)	Успішність
CreateRecordsBulk	00:00.044	1
CreateRecordsSingle	00:07.585	1
ReadRecordsBulk	00:00.046	1
ReadRecordsSingle	00:07.574	1
UpdateRecordsBulk	00:00.040	1
UpdateRecordsSingle	00:07.656	1
DeleteRecordsBulk	00:00.017	1
DeleteRecordsSingle	00:07.417	1
ComplexQuery	00:00.000	0
HighConcurrency	00:00.526	1
LargeVolume	00:01.200	1

Redis продемонстрував найшвидші операції серед чотирьох баз даних. Всі операції створення, читання та оновлення, як масові, так і поодинокі, були виконані менш ніж за 8 секунд. Як і Memcached, Redis не підтримує сценарій ComplexQuery. Операції HighConcurrency були найшвидшими серед баз даних, з середнім часом близько 0,5 секунди. Операція LargeVolume також оброблялася швидко, з середнім часом близько 1,2 секунди.

4.2. Аналіз

З бенчмаркінгових тестів видно, що кожна база даних має свої сильні та слабкі сторони.

Кассандра виділяється своєю здатністю обробляти складні запити, а також виконувати масові операції і працювати під високим навантаженням паралельними запитами. Однак, її продуктивність з операціями з одним записом і великими об'ємами даних була неоднозначною.

Memcached показала стабільну продуктивність для всіх операцій, але була повільнішою порівняно з Redis при виконанні одиничних операцій, і не показала покращення швидкості при виконанні масових операцій порівняно з одиничними. Її нездатність обробляти складні запити також може бути недоліком в залежності від конкретного випадку використання. Також час виконання тесту з великим об’ємом даних найгірший з великим відривом, що може пояснюватись менш ефективним використанням ресурсів оперативної пам’яті, ніж в Redis, в якого проблем з цим тестом не виникло.

Mongo добре впоралася з масовими операціями і мала другу найкращу продуктивність при роботі з високим рівнем паралелізму. Однак операції з одним записом були повільнішими.

Redis перевершив інші за швидкістю виконання більшості операцій і був єдиним, хто завершив усі сценарії менш ніж за 10 секунд. Однак, як і Memcached, він не підтримує складні запити.

Таким чином, вибір бази даних залежить від конкретного сценарію використання та вимог програми. Для додатків, де швидкість має першорядне значення, а складні запити не потрібні, Redis буде чудовим вибором. Для додатків, які часто використовують складні запити, краще підійде Cassandra. Для загального використання Mongo і Memcached можуть забезпечити

стабільну продуктивність, причому Mongo пропонує кращу підтримку складних запитів, а Memcached забезпечує більш стабільну продуктивність при виконанні всіх типів операцій.

Якщо висока паралельність є важливим аспектом програми, Cassandra і Redis стають сильними претендентами завдяки своїй відмінній продуктивності в сценарії HighConcurrency. З іншого боку, якщо додаток буде мати справу з великим обсягом даних, всі бази даних, крім Cassandra, яка показала змішані результати, та Memcached, якому знадобилося неймовірно багато часу на їх обробку(6 хвилин) є непоганими варіантами.

Memcached і Redis можуть бути менш придатними для додатків, які сильно покладаються на складні запити, через відсутність підтримки в цій області. Однак вони обидва відмінно підходять для ситуацій, які вимагають швидкого відгуку, і добре справляються з високим рівнем паралелізму, що робить їх хорошим вибором для додатків з високим трафіком, таких як системи чату або аналітика в реальному часі.

MongoDB, завдяки своїй здатності обробляти складні запити і великі обсяги даних у поєднанні з хорошою продуктивністю в умовах високої паралельності, може стати кращою базою даних для таких додатків, як системи управління контентом або блог-платформи, де операції читання і запису відбуваються часто, а цілісність даних має першочергове значення.

Нарешті, Cassandra, з її відмінною обробкою складних запитів і високим рівнем паралелізму операцій, може добре підійти для додатків, де потрібно швидко обробляти великі обсяги даних, наприклад, в аналітиці великих даних або системах моніторингу в реальному часі.

Вибір бази даних також залежатиме від інших факторів, таких як досвід команди розробників у роботі з конкретною базою даних, складність моделі даних, вимоги до масштабованості та середовище розгортання. Отже, бажано

провести ретельний аналіз, враховуючи всі ці фактори, перш ніж приймати рішення щодо конкретної системи баз даних.

РОЗДІЛ 5. ВИСНОВКИ, ЗРОБЛЕНІ В РЕЗУЛЬТАТІ ДОСЛІДЖЕННЯ

Дослідження продемонструвало, що не існує універсальної системи баз даних, і вибір бази даних повинен бути адаптований до унікальних вимог кожної програми. Було помічено, що різні бази даних працюють по-різному за різних обставин, таких як розмір даних, складність запитів та рівень паралелізму.

5.1. Як результати узгоджуються з теоретичними очікуваннями

Теоретично зрозуміло, що різні СУБД розробляються з урахуванням різних сценаріїв використання, і це знайшло відображення в наших результатах.

Наприклад, очікувалося, що Memcached і Redis, які є сховищами даних в пам'яті, будуть чудово працювати у високошвидкісних, високопаралельних сценаріях завдяки своїй архітектурі в пам'яті, і це підтвердилося в дослідженні. На противагу цьому, очікувалося, що вони не будуть добре обробляти складні запити, що також спостерігалося в результатах.

Аналогічно, здатність MongoDB ефективно обробляти складні запити завдяки своїй гнучкій, документно-орієнтованій природі була підтверджена дослідженням. Той факт, що Cassandra добре показала себе в сценаріях з високим рівнем паралелізму, відповідає теоретичним очікуванням, враховуючи її архітектуру, яка акцентує увагу на швидкості запису та горизонтальному масштабуванні.

5.2. Наслідки для реальних додатків

Це дослідження підкреслює важливість вибору правильної СУБД для ваших конкретних потреб. Для додатків реального часу, які потребують високої швидкості і можуть обробляти прості дані, сховище даних в пам'яті, таке як Redis або Memcached, може бути дуже корисним. Для додатків, які потребують обробки складних запитів, більше підійдуть бази даних на кшталт MongoDB або Cassandra.

Дослідження також показує, що хоча деякі бази даних добре працюють у певних сценаріях, вони можуть працювати не так добре за інших умов. Отже, при виборі СУБД розробники повинні враховувати весь життєвий цикл додатку та його даних.

5.3. Рекомендації щодо ефективного використання БД

На основі висновків, зроблених у цьому дослідженні, ми пропонуємо кілька рекомендацій щодо ефективного використання баз даних:

- Зрозумійте природу вашого додатку: Характер вашої програми - розмір даних, рівень паралелізму та складність запитів - суттєво впливає на вибір СУБД.
- Враховуйте масштабованість: Якщо очікується, що ваш додаток буде швидко зростати, вам потрібна СУБД, яка може ефективно масштабуватися. У цьому дослідженні було виявлено, що MongoDB та Cassandra добре справляються з масштабуванням.

- Враховуйте досвід команди розробників: Знайомство вашої команди з конкретною СКБД також може вплинути на вибір. Вибір СКБД, яка відповідає навичкам команди, може пришвидшити розробку та усунення несправностей.
- Сплануйте моделювання даних: Залежно від вашої програми, вам може знадобитися СУБД, яка підтримує реляційне моделювання даних (бази даних SQL), документно-орієнтоване моделювання (наприклад, MongoDB) або пари "ключ-значення" (наприклад, Redis).
- Тестуйте, перш ніж впроваджувати: Результати цього дослідження є загальними спостереженнями. Найкраще провести аналогічні тести з вашим конкретним додатком і робочим навантаженням, щоб знайти найбільш підходящу СУБД.

ВИСНОВКИ

Загальний висновок, зроблений з цього дослідження, показує, що не існує єдино оптимальної системи управління базами даних (СУБД), придатної для всіх типів додатків. Відносна ефективність та результативність чотирьох широко використовуваних СУБД, досліджених тут - MongoDB, Cassandra, Redis та Memcached - показала значну залежність від унікальних та специфічних вимог, що висуваються середовищем застосування.

У випадку MongoDB і Cassandra обидві СУБД продемонстрували вражаючу здатність обробляти складні запити. Вони продемонстрували потужні можливості маніпулювання даними, що робить їх корисними для додатків, які вимагають складних взаємодій з даними. Ці СУБД демонструють неабияку корисність у контекстах, де модель даних є складною і вимагає широкого спектру можливостей для запитів. Ефективність роботи та гнучкість запитів роблять їх оптимальним вибором для таких випадків використання.

І навпаки, при розгляді додатків, де швидкість, висока паралельність і час відгуку мають першочергове значення, перевага Redis і Memcached стала очевидною. Ці СУБД чудово показали себе у високошвидкісних середовищах, зарекомендувавши себе як потужні інструменти для додатків, що вимагають швидких транзакцій з даними. Їх майстерність у забезпеченні високої продуктивності в умовах дефіциту часу зміцнює їхню позицію як придатних систем для таких додатків.

Це дослідження дає чітке розуміння критичної необхідності розуміння різних можливостей кожної СУБД і важливості узгодження цих можливостей з конкретними вимогами конкретної програми. Він рішуче відкидає

універсальний підхід і натомість наголошує на необхідності ретельного вивчення, ретельного тестування і тонкого налаштування різних СУБД у конкретному контексті застосування. Мета полягає в тому, щоб досягти оптимального узгодження між вимогами програми і сильними сторонами СУБД, тим самим максимізуючи продуктивність і ефективність системи.

Крім того, це дослідження також закладає основу для подальших досліджень. Існує відкритий шлях для вивчення інших важливих факторів, таких як масштабованість, стійкість та потужність цих СУБД.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Офіційна документація MongoDB C# Driver: https://mongodb.github.io/mongo-csharp-driver/2.19/apidocs/html/R_Project_CSharpDriverDocs.htm
2. Офіційна документація Cassandra C# Driver: https://docs.datastax.com/en/latest-csharp-driver-api/api/Cassandra.html
3. Офіційна документація StackExchange.Redis: https://stackexchange.github.io/StackExchange.Redis/
4. Гітхаб EnyimMemcached: https://github.com/enyim/EnyimMemcached
5. Стаття «NoSQL explained»: https://www.mongodb.com/nosql-explained
6. Lewis, Byron C.; Crews, Albert E. (1985). The Evolution of Benchmarking as a Computer Performance Evaluation Technique
7. Стаття «Значущий процес бенчмаркінгу»: https://www.ifixit.com/Wiki/Guide_to_perform_meaningful_computer_benchmarking_test_for_productivity
8. Стаття «Науковий бенчмаркінг паралельних комп’ютерних систем»: https://spcl.inf.ethz.ch/Teaching/2020-dphpc/hoefler-scientific-benchmarking.pdf

ДОДАТКИ

```
public class BenchmarkRunner
{
    3 usages
    private List<IDbScenario> Scenarios { get;  set; }
    9 usages
    private IDbService DbService { get; }

    private const int RunCount = 5;

    1 usage
    public BenchmarkRunner(IDbService dbService)
    {
        Scenarios = new List<IDbScenario>();
        DbService = dbService;
    }

    1 usage
    public async Task<BenchmarkResults> RunAsync()
    {
        var benchmarkResults = new BenchmarkResults();

        for (int i = 0; i < RunCount; ++i)
        {
            Console.WriteLine($"Starting {i}th run for {DbService.GetType().Name}");
            foreach (var scenario in Scenarios)
            {
                Console.WriteLine($"Starting {scenario.GetType().Name}");
                try
                {
                    var scenarioResults = await scenario.ExecuteScenarioAsync();
                    benchmarkResults.ScenarioResults.Add(scenarioResults);
                }
                catch (Exception e)
                {
                    Console.WriteLine($"An error occurred while executing" +
                                      $" {scenario.GetType().Name} on run {i+1}: {e.Message}");
                    benchmarkResults.ScenarioResults.Add( item: new ScenarioResults()
                    {
                        ScenarioName = scenario.GetType().Name,
                        SuccessRate = 0
                    });
                }
            }
        }

        return benchmarkResults;
    }
```

Додаток 1

```
53      public void SetupScenarios(Dictionary<Type, IDataFactory> factories)
54      {
55          var scenarios = new List<IDbScenario>();
56          var crudSingleScenarioParams = new ScenarioParameters()
57          {
58              ItemCount = 10000,
59              Mode = CrudMode.Single
60          };
61          var crudBulkScenarioParams = new ScenarioParameters()
62          {
63              ItemCount = 10000,
64              Mode = CrudMode.Bulk
65          };
66          var complexQueryScenarioParams = new ScenarioParameters()
67          {
68              ItemCount = 10000,
69              SortField = 5,
70              Limit = 100
71          };
72          var highConcurrencyScenarioParams = new ScenarioParameters()
73          {
74              ItemCount = 10000
75          };
76          var largeVolumeScenarioParams = new ScenarioParameters()
77          {
78              ItemCount = 100000
79          };
80
81          foreach (var (type, factory) in factories)
82          {
83              CreateScenarios(scenarios, typeof(CreateRecordsScenario<>), type, factory, DbService,
84                  crudSingleScenarioParams,
85                  crudBulkScenarioParams);
86              CreateScenarios(scenarios, typeof(ReadRecordsScenario<>), type, factory, DbService,
87                  crudSingleScenarioParams,
88                  crudBulkScenarioParams);
89              CreateScenarios(scenarios, typeof(UpdateRecordsScenario<>), type, factory, DbService,
90                  crudSingleScenarioParams,
91                  crudBulkScenarioParams);
92              CreateScenarios(scenarios, typeof(DeleteRecordsScenario<>), type, factory, DbService,
93                  crudSingleScenarioParams,
94                  crudBulkScenarioParams);
95              CreateScenarios(scenarios, typeof(ComplexQueryScenario<>), type, factory, DbService,
96                  complexQueryScenarioParams);
97              CreateScenarios(scenarios, typeof(HighConcurrencyTestScenario<>), type, factory, DbService,
98                  highConcurrencyScenarioParams);
99              CreateScenarios(scenarios, typeof(LargeVolumeScenario<>), type, factory, DbService,
100                 largeVolumeScenarioParams);
101         }
102
103         Scenarios = scenarios;
104     }
```

Додаток 2

```
    7 usages
106     private static void CreateScenarios(List<IDbScenario> scenarios, Type scenarioType, Type type,
107         IDataFactory dataFactory,
108         IDbService dbService, ScenarioParameters singleParams, ScenarioParameters? bulkParams = null)
109     {
110         var runtimeType = scenarioType.MakeGenericType(type);
111         if (bulkParams != null)
112         {
113             var bulkScenario = (IDbScenario?)Activator.CreateInstance(
114                 runtimeType, params args: dataFactory, dbService, bulkParams);
115             if (bulkScenario is null)
116                 throw new NullReferenceException( message: "Bulk scenario is null");
117             scenarios.Add(bulkScenario);
118         }
119
120         var singleScenario = (IDbScenario?)Activator.CreateInstance(
121             runtimeType, params args: dataFactory, dbService, singleParams);
122         if (singleScenario is null)
123             throw new NullReferenceException( message: "Single scenario is null");
124         scenarios.Add(singleScenario);
125     }
126 }
127
128
    2 usages
129 public class BenchmarkResults
130 {
        3 usages
131     public List<ScenarioResults> ScenarioResults { get; } = new List<ScenarioResults>();
132
        1 usage
133     public string ToCsv()
134     {
135         StringBuilder csv = new StringBuilder();
136         csv.AppendLine("ScenarioName,Duration,Passed,ErrorMessage");
137
138         foreach (var scenarioResult :ScenarioResults in ScenarioResults)
139         {
140             csv.AppendLine($"{scenarioResult.ScenarioName}," +
141                            $"{scenarioResult.GetTotalDuration()}," +
142                            $"{scenarioResult.GetSuccessRate()}");
143         }
144
145         return csv.ToString();
146     }
147 }
148
```

Додаток 3

```
namespace DbBenchmark.DbServices;

21 usages  5 inheritors  1 exposing API
public interface IDbService
{
    2 usages  5 implementations
    Task SetupAsync();
    1 usage  5 implementations
    Task CleanupAsync<T>();
    6 usages  5 implementations
    Task CreateRecordAsync<T>(string key, T data);
    3 usages  5 implementations
    Task<T> ReadRecordAsync<T>(string key);
    2 usages  5 implementations
    Task UpdateRecordAsync<T>(string key, T data);
    3 usages  5 implementations
    Task DeleteRecordAsync<T>(string key);
    5 usages  5 implementations
    Task CreateRecordsAsync<T>(IDictionary<string, T> records);
    4 usages  5 implementations
    Task<IList<T>> ReadRecordsAsync<T>(IEnumerable<string> keys);
    3 usages  5 implementations
    Task UpdateRecordsAsync<T>(IDictionary<string, T> records);
    3 usages  5 implementations
    Task DeleteRecordsAsync<T>(IEnumerable<string> keys);
    1 usage  5 implementations
    Task<IList<T>> ComplexQueryAsync<T>(string fieldName, object value, int limit);
}
```

Додаток 4

```
using DbBenchmark.DataClass;

namespace DbBenchmark.Scenarios.DataFactory;

11 usages  2 inheritors  1 exposing API
public interface IDataFactory
{
    4 usages  1 implementation
    Data Create(int i);
}

5 usages  1 inheritor
public interface IDataFactory<T> : IDataFactory where T:Data
{
    1 implementation
    new T Create(int i);
}
1 usage
public class PersonFactory : IDataFactory<Person>
{
    1 usage
    public Person Create(int i)
    {
        return new Person
        {
            FirstName = $"fn{i}",
            LastName = $"ln{i}",
            Age = i
        };
    }

    0+4 usages
    Data IDataFactory.Create(int i)
    {
        return Create(i);
    }
}
```

Додаток 5

```
1 usage  9 inheritors
public abstract class Scenario
{
    11 usages
    public ScenarioParameters ScenarioParams { get; set; }

    protected Scenario()
    {
        ScenarioParams = new ScenarioParameters();
    }

    1 usage
    protected Scenario(ScenarioParameters scenarioParameters)
    {
        ScenarioParams = scenarioParameters;
    }
    1 usage  5 overrides
    protected abstract Task SetupAsync();
    1 usage  6 overrides
    protected abstract Task<ScenarioResults> RunAsync();
    1 usage  1 override
    protected abstract Task CleanupAsync();

    public virtual async Task<ScenarioResults> ExecuteScenarioAsync()
    {
        await SetupAsync();

        ScenarioResults results;
        try
        {
            results = await RunAsync();
        }
        finally
        {
            await CleanupAsync();
        }

        return results;
    }
}
```

Додаток 6

```
System.Net.ServicePointManager.DefaultConnectionLimit = Constants.MaxConnectionPoolSize;

var factories = new Dictionary<Type, IDataFactory>();
factories.Add(typeof(Person), new PersonFactory());


async Task<string> RunBenchmark(IDbService service)
{
    var benchmarkRunner = new BenchmarkRunner(service);
    benchmarkRunner.SetupScenarios(factories);
    var benchmarkResults = await benchmarkRunner.RunAsync();
    // Write the results to a CSV file
    var csvData :string = benchmarkResults.ToCsv();
    return csvData;
}

//init DB services
var services = new List<IDbService>();

var mongo = new MongoService( host: "localhost",  port: 27017);
await TestService(mongo);
services.Add(mongo);

var cassandra = new CassandraService( hostName: "localhost",  port: 9042);
await cassandra.SetupAsync();
await TestService(cassandra);
services.Add(cassandra);

var redis = new RedisService();
await TestService(redis);
services.Add(redis);

var memCached = new MemcachedService();
await TestService(memCached);
services.Add(memCached);

//run benchmarks and dump results
var folderPath = "C:\\Users\\Dima\\RiderProjects\\DBench\\DbBenchmark\\results";
foreach (var service in services)
{
    var csv :string = await RunBenchmark(service);
    var name :string = service.GetType().Name;
    File.WriteAllText( path: $"{folderPath}\\{name}Results.csv",  contents: csv);
}
```

Додаток 7