Міністерство освіти і науки України

Український державний університет науки і технологій

Факультет Комп'ютерні технології та системи
Кафедра Комп'ютерні інформаційні технології

Пояснювальна записка
до кваліфікаційної роботи
магістра

на тему: «Дослідження ефективності роботи різних СУБД при роботі з розподіленими нереляційними базами даних»

за освітньою програмою **Інженерія програмного забезпечення**
зі спеціальності: **121 Інженерія програмного забезпечення**

Виконав: студент групи «ПЗ2326»

_____ /Сергій ДУДАР/

Керівник: _____ /доц. Олександр ІВАНОВ/

Нормоконтролер: _____ /Світлана ВОЛКОВА/

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

(підпис)

Дніпро – 2025 рік

Ministry of Education and Science of Ukraine

Ukrainian State University of Science and Technologies

Faculty Computer technologies and systems
Department Computer information technology

Explanatory Note
to Master's Thesis

on the topic: «Research of efficiency of work of various DBMS at work with the distributed nonrelational databases »

according to educational curriculum **12 software engineering**
in the Speciality: **121 software engineering**

Done by the student of the group PZ2326: /Serhii DUDAR/

Scientific Supervisor: /Oleksandr IVANOV/

Normative controller: /Svitlana VOLKOVA/

Dnipro – 2025

Міністерство освіти і науки України
Український державний університет науки і технологій

Факультет: Комп'ютерних технологій і систем
Кафедра: Комп’ютерні інформаційні технології
Рівень вищої освіти: магістр
Освітня програма: Інженерія програмного забезпечення
Спеціальність: Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ
Завідувач кафедри ____КІТ
_____ Вадим ГОРЯЧКІН
_____ грудня 2024 р.

ЗАВДАННЯ

На кваліфікаційну роботу ____Магістр____
студенту Дударю Сергію Олеговичу____.

1. Тема дипломної роботи: «Дослідження ефективності роботи різних СУБД при роботі з розподіленими нереляційними базами даних».

Керівник роботи: Іванов Олександр Петрович
затверджені наказом
1196 ст від 05.12.2022 року

2. Строк подання студентом роботи ___.01.2025 року
3. Вихідні дані до дипломної роботи:
_____.
4. Зміст пояснювальної записки (перелік питань до розробки):
   4.1. Аналітична частина: Аналіз сучасного стану дослідження за науковими літературними джерелами;
   4.2. Основна частина: Дослідження ефективності роботи СУБД;
5. Перелік демонстраційного матеріалу:
   5.1. презентація;
   5.2. демонстраційне відео.

КАЛЕНДАРНИЙ ПЛАН

№ п/п	Назва розділів дипломного проекту	Термін виконання розділів проекту (роботи)	Примітка
1	Вступ	01.09.2024 – 10.09.2024	
2	Огляд літератури	10.09.2024 – 15.09.2024	
3	Постановка задачі, технічне завдання	15.09.2024 – 30.09.2024	
4	Створення тестової програми	01.10.2024 – 21.10.2024	
5	Перші тестування	21.10.2024 – 30.10.2024	30%
6	Аналіз результатів	30.10.2024 – 11.11.2024	
7	Охорона праці	11.11.2024 – 14.11.2024	
8	Розрахунок економічних показників	14.11.2024 – 18.11.2024	60%
9	Оформлення пояснювальної записки	18.11.2024 – 26.12.2024	
10	Демонстраційні матеріали	26.12.2024 – 16.01.2025	100%

Керівник дипломного проекту _____ (підпис) Іванов О.П. (ПІБ)

Завдання прийняв до виконання _____ (підпис) Дудар С.О. (ПІБ)

РЕФЕРАТ

Об’єктом даного дослідження є системи управління розподіленими не реляційними базами даних.

Предметом дослідження є ефективність роботи систем управління розподіленими не реляційними базами даних.

Метою даної роботи є визначення найбільш ефективної та продуктивної системи управління базами даних.

Методи дослідження: проведення експериментів, збір даних експериментів та аналіз зібраних даних.

Результати та їх новизна: Результати роботи можуть бути використані при виборі СУБД для проекту, в якому буде необхідність використання великих об’ємів даних та необхідності їх зручного масштабування.

Пояснювальна записка складається зі вступу, 4 розділів, висновків, бібліографічного списку та 3 додатків:

- у вступі описується сутність роботи та її актуальність (3 сторінки);
- у першому розділі висвітлено аналіз сучасного дослідження проблеми, а також проаналізовано наявні документ-орієнтовані СУБД (12 сторінок);
- у другому розділі надано обґрунтування напрямку дослідження (6 сторінок);
- у третьому розділі описано процес проектування і розробки інструментарію для дослідження часової ефективності (22 сторінки);
- у четвертому розділі описано виконані дослідження часової ефективності алгоритмів (14 сторінок);
- додатки – технічне завдання, текст програми, керівництво користувача.

Таблиць – 13, рисунків – 13, бібліографія – 31 джерело.

ЗМІСТ

ВСТУП

У сфері інформаційних технологій дуже часто виникає необхідність використовувати бази даних для вирішення різноманітних задач пов’язаних зі зберіганням інформації. Часто доводиться зберігати її у різних місцях програмної системи або комп’ютерної мережі, тобто треба мати справу з розподіленими базами даних[1]. Але водночас з цим постає питання, як правильно представити та зберігати великі об’єми даних. Тобто яку СУБД краще використовувати: SQL або NoSQL?

Щоби відповісти на ці питання необхідно для початку з’ясувати переваги нереляційних баз даних над реляційними [2], та який тип краще використовувати для конкретних потреб. Приведемо невелике порівняння даних двох типів зберігання інформації:

- noSQL легко справляється з дуже великими обсягами даних;
- як правило, нереляційні СУБД швидше виконують операції запису (швидкість операцій читання залежить від типу бази даних NoSQL і типу запитуваних даних);
- noSQL дуже гнучка (в порівнянні з реляційними СУБД, яким для початку необхідна структура);
- noSQL пропонує автоматизовану реплікацію і масштабування. Сьогодні нереляційні СУБД активно розвиваються і усувають загальні проблеми в роботі з даними, серед яких реплікація і масштабування – одні з найважливіших;
- noSQL легко масштабується і працює в кластері;
- також NoSQL має широкий вибір програм і моделей для роботи з різними типами даних.

В останнє десятиріччя все більше стають популярними та широко застосованими нереляційні бази даних, а саме СУБД для роботи з документ-орієнтованими базами даних[3]. Тому в контексті даної роботи необхідно дослідити доцільність використання такого типу нереляційних баз даних.

Актуальність роботи. Для початку потрібно розібратись зі сферою використання такого типу нереляційних БД. Документна база даних – гарний вибір для додатків управління контентом, таких як платформи для блогів і розміщення відео. При використанні документної бази даних кожна сутність, яка відстежується додатком, може зберігатися як окремий документ. Документна база даних дозволяє розробнику з зручністю оновлювати додаток при зміні вимог. Крім того, якщо необхідно змінити модель даних, то потрібно оновлення тільки тих документів, яких стосується дана зміна. Також такі БД підходять для зберігання каталожної інформації. Тобто їх дуже доцільно використовувати для інтернет-комерції.

І тут постає питання: яку СУБД використовувати для реалізації своїх потреб? Для відповіді на це питання необхідно порівняти продуктивність роботи документ-орієнтованих СУБД. Зазвичай це робиться методом заміру часу виконання основних операцій при роботі з базами даних. Операції можуть бути такими:

- вставка даних;
- вибірка даних;
- оновлення;
- пошук;
- фільтрація;
- групування.

Для великих проектів необхідна велика кількість інформації, тому об’єми баз даних будуть достатньо великими. З цього слідує те, що час виконання основних операцій обробки даних дуже важливий для продуктивності роботи.

Об’єкт дослідження. Системи управління розподіленими нереляційними документ-орієнтованими базами даних.

Предмет дослідження. Ефективність роботи систем управління розподіленими не реляційними базами даних.

Мета і завдання дослідження. Метою даної роботи є визначення найбільш ефективної та продуктивної системи управління базами даних.

Поставлена мета зумовлює необхідність вирішення наступного ряду завдань:

- розробка програмного інструментарію, за допомогою якого можна буде збирати інформацію з проведених експериментів у зручному для аналізу вигляді;
- дослідження, аналіз отриманих результатів та порівняння.

Методи дослідження. Проведення експериментів, збір даних експериментів та аналіз зібраних даних.

Наукова новизна. Додана можливість обчислення очікуваного часу роботи кожного запиту до бази даних за допомогою різних нереляційних СУБД та обрати найефективнішу для поточних потреб.

Практичне значення. Результати дослідження можуть бути використані іншими при розробці нових проектів з використанням нереляційних документ-орієнтованих баз даних та полегшення вибору СУБД спираюсь на свої потреба та вимоги.

1 АНАЛІЗ ПРОБЛЕМИ ЕФЕКТИВНОСТІ СУБД ПРИ РОБОТІ З НЕРЕЛЯЦІЙНИМИ БАЗАМИ ДАНИХ

1.1 Призначення та сфера застосування

Розподілена база даних – це набір відношень, що зберігаються в різних вузлах комп'ютерної мережі та логічно пов'язані таким чином, щоб складати єдину сукупність даних. Розподілена база даних передбачає зберігання даних на кількох вузлах мережі, обробку даних та їх передачу між цими вузлами у процесі виконання запитів. Розбиття даних у розподіленій базі даних може досягатися шляхом зберігання різних таблиць на різних комп'ютерах або зберігання різних фрагментів однієї таблиці на різних комп'ютерах. Для користувача (або програмного додатку) не повинно мати значення, як розподілені дані між комп'ютерами. Робота з розподіленою базою даних повинна здійснюватися так само, як і з централізованою.

Розподілені бази даних використовуються в тих випадках, коли є дуже великий обсяг даних і їх вже важко тримати в одному місці. Тому і використовують такі бази даних, щоб розподілити їх на декілька частин.

1.2 Постановка задачі

Необхідно розробити програмний продукт, який можна буде застосувати для проведення експериментів з підключенням до розподілених кластерів у різних СУБД. Програма повинна давати можливість користувачеві обирати необхідну СУБД, обирати тип запиту, який він хоче заміряти.

В якості результату програма має повертати час виконання операції, в залежності від того, що запросив користувач.

Програма має бути простою у використанні, не містити дуже складного та громіздкого інтерфейсу. Призначення всіх полів для вводу повинно бути простим та зрозумілим.

1.3 Огляд документ-орієнтованих СУБД

Існує декілька СУБД такого типу:

- MongoDb;
- CouchDb.

1.3.1 Порівняння MongoDb та CouchDb

CouchDB і MongoDB є документно-орієнтованими базами даних NoSQL, але вони суттєво відрізняються у своїх реалізаціях та мають різний функціонал.

CouchDB використовує напівструктурований формат JSON для зберігання даних. Запити до бази даних CouchDB виконуються через RESTful HTTP API, використовуючи HTTP або JavaScript. MongoDB використовує BSON, варіант JSON, який зберігає дані у двійковому форматі. MongoDB використовує власну мову запитів, на відміну від SQL, хоча вони мають певну схожість.

В інших областях CouchDB і MongoDB мають кілька спільних функцій. Документ є основною одиницею даних в обох цих базах даних. Вони також не мають схеми, що означає, що можливо зберігати документи в базі без попереднього визначення схеми або структури для цього документа. Ця функція надає більшу гнучкість щодо даних, що зберігаються в CouchDB і MongoDB.

Теорема CAP[4] стверджує, що будь-яка розподілена база даних може мати максимум дві з трьох бажаних якостей: узгодженість, доступність і толерантність до розділів. Також важливо сказати про ще декілька важливих моментів, такі як:

- Послідовність: усі клієнти завжди мають однаковий погляд на дані.
- Доступність: усі клієнти можуть записувати в базу і читати з бази даних в будь-який момент часу.
- Допуск до розділу: кластер бази даних може продовжувати свою працездатність, не дивлячись на те, що зв’язки між вузлами пропадають.

CouchDB і MongoDB відрізняються за своїм підходом до теореми CAP. CouchDB віддає перевагу доступності та толерантності до розділів, в той час як MongoDB надає перевагу узгодженості та толерантності до розділів. CouchDB використовує кінцеву узгодженість. Клієнти можуть писати в один вузол бази даних, і ця інформація гарантовано поширюється на іншу частину бази даних. MongoDB в той же час використовує сувору послідовність. База даних використовує набір реплік для забезпечення резервного копіювання, але за ціною доступності.

MongoDB зазвичай має перевагу над CouchDB, коли ви працюєте з величезними наборами даних, високопродуктивними вимогами, включаючи більш

високу швидкість читання. Ця база даних також краще підходить для розгортань та масштабованості, де не можна відразу сказати до яких розмірів зросте ваша база даних.

CouchDB пропонує реплікацію як "головний-головний", так і "головний-підпорядкований", тоді як MongoDB охоплює лише конфігурації "головний-підпорядкований ". У реплікації "головний-майстер", також відомий як мульти-майстер, будь-який вузол в кластері може діяти як головний і приймати запити на читання та запис.

Чотири найважливіші відмінності між CouchDB і MongoDB:

- CouchDB приймає запити через RESTful HTTP API, тоді як MongoDB використовує власну мову запитів;
- CouchDB надає пріоритет доступності, а MongoDB до послідовності;
- MongoDB має набагато більшу базу користувачів і популярність, ніж CouchDB, що полегшує пошук підтримки та найму співробітників саме для цього рішення;
- обидві бази даних є безкоштовними проектами з відкритим кодом, але потребують платної вкерсії для розгортання великих робочих проектів.

1.3.2 Огляд особливостей MongoDB

MongoDB — це доступна з вихідним кодом кросплатформена програма баз даних, орієнтована на документи. Класифікована як програма баз даних NoSQL, MongoDB використовує документи, подібні до JSON.

Основні можливості та особливості даної СУБД:

1. Запити.

MongoDB підтримує функціонал для виконання пошуку за полями, діапазонами значень та регулярними виразами. Можна налаштувати запити на повернення лише конкретних полів документа або застосовувати користувацькі функції JavaScript. Також є можливість формувати запити, які повертають випадкову вибірку результатів заданого розміру.

2. Індексація.

Для документів у MongoDB доступна індексація полів, яка може бути первинною, вторинною або комбінованою. Це дозволяє підвищити швидкість виконання запитів.

3. Реплікація.

MongoDB підтримує високий рівень доступності завдяки наборам реплік, які включають щонайменше дві копії даних. Члени набору реплік можуть виступати в ролі основної чи вторинної репліки залежно від ситуації. За замовчуванням запис даних і виконання запитів відбувається через основну репліку, тоді як вторинні репліки синхронізуються з основною. У разі збою основної репліки MongoDB автоматично виконує процес вибору нової основної. Вторинні репліки також можуть обробляти запити на читання, але дані можуть бути лише частково синхронізованими. Для реплікованої конфігурації з одним вторинним сервером рекомендується використовувати арбітр — спеціальний процес, який бере участь лише у виборах основної репліки. Оптимально розподілена система MongoDB включає як мінімум три сервери: одну основну репліку, одну вторинну та арбітр.

4. Балансування навантаження.

MongoDB дозволяє масштабувати систему горизонтально завдяки шардингу. Використовуючи ключ сегментації, дані у колекції розподіляються по різних сегментах (шардах). Ключ може бути хешованим для рівномірного розподілу навантаження. Шард є основною одиницею зберігання даних і може складатися з однієї або декількох реплік. MongoDB забезпечує балансування навантаження між серверами та дублювання даних, щоб система залишалася працездатною навіть у разі апаратного збою.

5. Зберігання файлів.

MongoDB підтримує використання GridFS як файлової системи. Ця технологія дозволяє розподіляти файли на блоки, кожен із яких зберігається у вигляді окремого документа. GridFS має функції балансування навантаження та реплікації даних між різними серверами. Цей функціонал інтегровано в драйвери MongoDB. Файли можна обробляти за допомогою утиліти mongofiles або відповідних плагінів для Nginx і lighttpd.

6. Агрегація.

MongoDB пропонує три способи виконання агрегації даних: конвеєр агрегації, функцію Map-Reduce і цільові методи агрегації. Map-Reduce використовується для пакетної обробки даних, однак конвеєр агрегації зазвичай забезпечує кращу продуктивність. Конвеєр дозволяє виконувати агрегації, аналогічні оператору GROUP BY у SQL, із можливістю комбінування операторів. Серед функцій агрегації доступні оператори для об'єднання документів із кількох колекцій ($lookup) і статистичні функції, такі як обчислення стандартного відхилення.

1.3.3 Огляд особливостей CouchDB

Apache CouchDB — це документно-орієнтована база даних NoSQL з відкритим вихідним кодом, реалізована на Erlang[5].

CouchDB використовує кілька форматів і протоколів для зберігання, передачі та обробки своїх даних. Він використовує JSON для зберігання даних, JavaScript як мову запитів, використовуючи MapReduce, і HTTP для API.

CouchDB був вперше випущений у 2005 році, а пізніше став проектом Apache Software Foundation у 2008 році.

На відміну від реляційної бази даних CouchDB не зберігає дані та зв’язки в таблицях. Натомість кожна база даних являє собою сукупність незалежних документів. Кожен документ має власні дані та автономну схему. Програма може отримати доступ до кількох баз даних, наприклад, однієї, що зберігається на мобільному телефоні користувача, а іншої на сервері. Метадані документа містять інформацію про перегляд, що дає змогу об’єднати будь-які відмінності, які могли виникнути під час від’єднання баз даних.

CouchDB реалізує форму багатоверсійного контролю паралельності (MVCC), тому він не блокує файл бази даних під час запису. Конфлікти залишаються на вирішення програми. Вирішення конфлікту зазвичай передбачає спочатку об’єднання даних в один із документів, а потім видалення застарілого.

Інші функції включають семантику ACID на рівні документа з кінцевою узгодженістю, (інкрементальну) MapReduce та (інкрементальну) реплікацію. Одним з відмінних атрибутів CouchDB є реплікація з кількома основними вузлами, що

дозволяє йому масштабуватися на різних машинах для створення високопродуктивних систем. Вбудований веб-додаток під назвою Fauxton (раніше Futon) допомагає з адмініструванням.

Дана система управління нереляційними базами даних має такі особливості:

- ACID семантика.

CouchDB забезпечує семантику ACID. Це робиться за допомогою реалізації форми Multi-version Concurrency Control, що означає, що CouchDB може обробляти велику кількість одночасних читачів і записувачів без конфліктів.

- Створено для офлайн використання.

CouchDB може реплікуватися на пристрої (наприклад, смартфони), які можуть виходити в автономний режим і виконувати синхронізацію даних за користувача, коли пристрій знову в мережі.

- Розподілена архітектура з реплікацією.

CouchDB був розроблений з урахуванням двонаправленої реплікації (або синхронізації) та роботи в автономному режимі. Це означає, що кілька реплік можуть мати власні копії тих самих даних, змінювати їх, а потім синхронізувати ці зміни пізніше.

- Зберігання документів.

CouchDB зберігає дані як «документи», як одну або кілька пар «поле/значення», виражені як JSON. Значення полів можуть бути простими речами, такими як рядки, числа або дати; але також можна використовувати впорядковані списки та асоціативні масиви. Кожен документ у базі даних CouchDB має унікальний ідентифікатор і немає необхідної схеми документа.

- Кінцева послідовність.

CouchDB гарантує остаточну узгодженість, щоб забезпечити як доступність, так і толерантність до розділів.

- Карта/Зменшення переглядів та індексів.

Збережені дані структуруються за допомогою переглядів. У CouchDB кожне представлення створюється за допомогою функції JavaScript, яка діє як половина карти карти/зменшення операції. Функція приймає документ і перетворює його в

єдине значення, яке вона повертає. CouchDB може індексувати подання та оновлювати ці індекси в міру додавання, видалення або оновлення документів.

- HTTP API

Усі елементи мають унікальний URI, який відкривається через HTTP. Він використовує методи HTTP POST, GET, PUT і DELETE для чотирьох основних операцій CRUD (Create, Read, Update, Delete) на всіх ресурсах.

CouchDB також пропонує вбудований інтерфейс адміністрування, доступний через Інтернет під назвою Futon.

1.4 Огляд літератури

1.4.1 «Шардінг»

«Шардінг» – це метод розподілу даних між кількома машинами. MongoDB використовує шардінг для підтримки розгортань з дуже великими наборами даних і високошвидкісними операціями.

Системи баз даних з великими обсягами даних або продуктами з високою пропускною здатністю можуть дуже сильно виснажити потужності одного сервера. Наприклад, висока частота запитів може виснажити потужність ЦП сервера. Розміри робочих наборів, що перевищують оперативну пам’ять системи, впливають на ємність вводу/виводу пристроїв [6].

Існує два способи вирішення проблеми зростання системи: вертикальне і горизонтальне масштабування.

Вертикальне масштабування – це збільшення ресурсів одного сервера. Під ресурсами розуміється використання потужнішого ЦП, додавання більшого об’єму оперативної пам’яті або збільшення пам’яті диску. Обмеження доступної технології можуть обмежити можливість однієї машини бути достатньо потужною для даного робочого навантаження. Крім того, хмарні провайдери мають достатньо сильні обмеження на основі доступних конфігурацій обладнання. Тому в результаті ми отримуємо, що існує практичний максимум для вертикального масштабування.

Горизонтальне масштабування передбачає поділ системного набору даних і навантаження на кілька серверів, додавання додаткових серверів для збільшення ємності за потреби. Хоча загальна швидкість або потужність окремої машини може

бути невисокою, кожна машина обробляє частину загального робочого навантаження, що потенційно може забезпечити кращу ефективність, ніж один високошвидкісний сервер високої ємності. Розширення потужності розгортання вимагає лише додавання додаткових серверів за потреби, що може бути нижчою загальною вартістю, ніж обладнання високого класу для однієї машини. Компромісом є збільшення складності інфраструктури та обслуговування для розгортання.

1.4.2 «Шардований» кластер

Розглянемо детально, що таке «шардований» кластер [7] на прикладі MongoDB.

Шардований кластер MongoDB складається з таких компонентів:

- шард: кожен сегмент містить підмножину розділених даних. Кожен фрагмент можна розгорнути як набір реплік;
- mongos: mongos діє як маршрутизатор запитів, тобто виступає інтерфейсом між клієнтськими програмами та розподіленим кластером;
- конфігураційні сервери: сервери, що зберігають метадані та налаштування конфігурації для кластера.

Наступний графік описує взаємодію компонентів у сегментованому кластері:

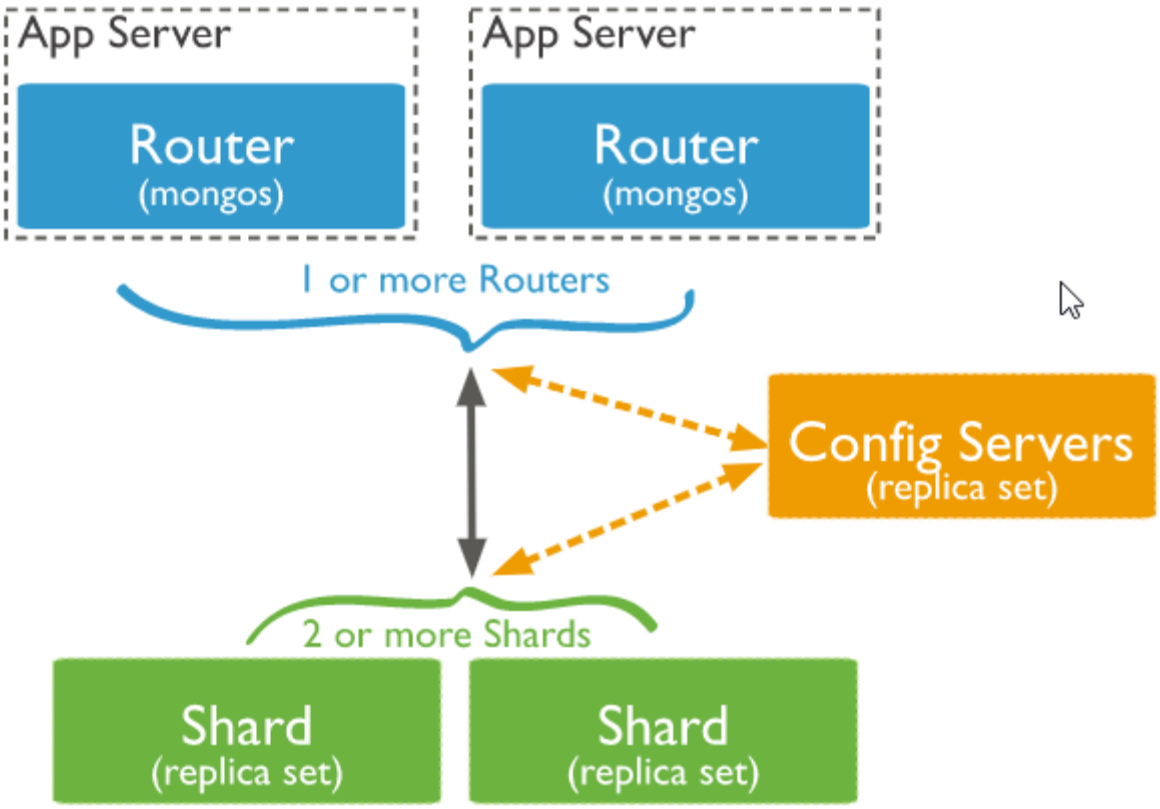

Рисунок 1.1 – Компоненти шардованого кластеру

MongoDB розподіляє дані на рівні колекції, розподіляючи дані колекції між сегментами в кластері.

Програма спілкується з маршрутизаторами (монго) щодо запиту, який потрібно виконати. Примірник mongos звертається до серверів конфігурації, щоб перевірити, який сегмент містить потрібний набір даних для відправки запиту до цього сегмента. Нарешті, результат запиту буде повернуто до програми.

1.4.3 Шард-ключ

При розподілі колекції MongoDB ключ «шарда» створюється як один із початкових кроків. «Ключ шарда» використовується для розподілу документів колекції MongoDB по всіх сегментах. Ключ складається з одного або кількох полів у кожному документі. Розділений ключ є незмінним і не може бути змінений після сегментування. Розділена колекція містить лише один ключ сегмента [8].

При розподілі заповненої колекції колекція повинна мати індекс, який починається з ключа сегмента. Для порожніх колекцій, які не мають відповідного індексу, MongoDB створить індекс для вказаного ключа сегмента.

Ключ шарда може безпосередньо впливати на продуктивність кластера. Це може призвести до вузьких місць у програмах, пов’язаних з кластером. Щоб пом’якшити це, перед розподілом колекції необхідно створити ключ шарда на основі:

- схема набору даних;
- як запитується набір даних.

1.4.4 «Шматки»

Частини — це підмножини спільних даних. MongoDB розділяє розділені дані на фрагменти, які розподіляються між сегментами спільного кластера. Кожен фрагмент має включний нижній і ексклюзивний верхній діапазон на основі ключа шарда. Балансувальник, специфічний для кожного кластера, обробляє розподіл фрагментів [9].

Балансувальник працює як фонове завдання і розподіляє шматки за потребою, щоб досягти рівномірного балансу шматків по всіх сегментах. Цей процес називається рівномірним розподілом патрона.

Нижче представлена схема взаємодії компонентів шардованого кластеру:

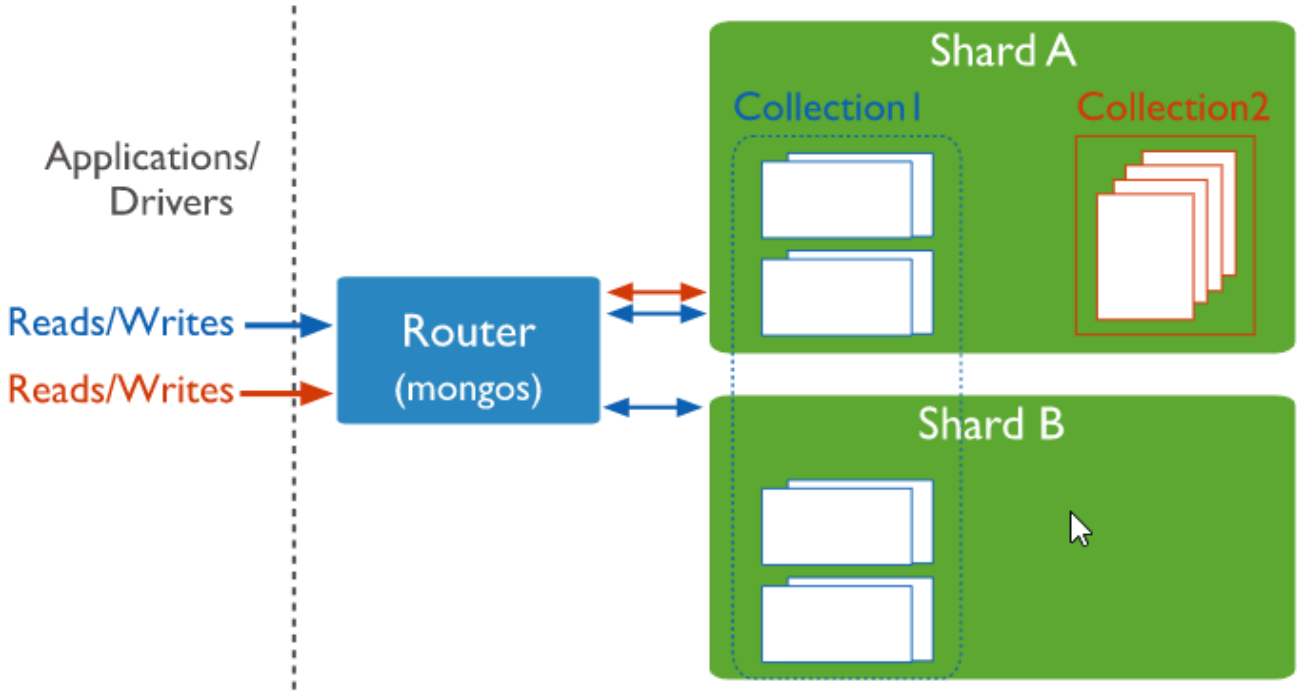

Рисунок 1.2 – Схема взаємодії компонентів кластеру

1.4.5 Переваги та обмеження шардингу

У традиційних сценаріях реплікації основний вузол обробляє основну частину операцій запису, тоді як вторинні сервери обмежені операціями лише для читання або підтримкою резервної копії набору даних. Однак, оскільки для шардингу використовуються сегменти з наборами реплік, усі запити розподіляються між усіма вузлами в кластері.

Оскільки кожен шард складається з підмножини повного набору даних, просте додавання додаткових фрагментів збільшить ємність сховища кластера без необхідності проводити складну апаратну реструктуризацію.

Реплікація вимагає вертикального масштабування при обробці великих наборів даних. Ця вимога може призвести до апаратних обмежень і непомірних витрат у порівнянні з підходом горизонтального масштабування. Але оскільки MongoDB використовує горизонтальне масштабування, робоче навантаження розподіляється. Коли виникає потреба, до кластера можна додати додаткові сервери.

У шардингу продуктивність як читання, так і запису безпосередньо корелює з кількістю серверних вузлів у кластері. Цей процес забезпечує швидкий метод підвищення продуктивності кластера шляхом простого додавання додаткових вузлів.

Розділений кластер може продовжувати працювати, навіть якщо один або кілька сегментів недоступні. Хоча дані про ці сегменти недоступні, клієнтська програма все ще може отримати доступ до всіх інших доступних сегментів у кластері без простою. У виробничих середовищах усі окремі фрагменти розгортаються як набори реплік, що ще більше підвищує доступність кластера.

Для підтримки розподіленого кластера необхідно ретельно планувати й підтримувати шардування — через його складність.

Коли ви розподіляєте колекцію MongoDB, неможливо видалити розділену колекцію.

Ключ сегмента безпосередньо впливає на загальну продуктивність базового кластера, оскільки він використовується для ідентифікації всіх документів у колекціях.

Існують деякі операційні обмеження в сегментованому середовищі MongoDB. У випадку, коли ключ або префікс складного ключа сегмента відсутні, Mongo виконає операцію широкомовної передачі, яка запитує всі сегменти в кластері, що може призвести до довготривалих завдань запиту.

Висновки до розділу 1

В першому розділі були розглянуті основні відмінності між двома нереляційними СУБД. В ході аналізу було виявлено недоліки та переваги кожної з них. Було з'ясовано, що як саме необхідно розбивати нереляційні бази даних на кластери та як саме необхідно з ними працювати. Також було розглянуто більш детально принципи зберігання і розподілення даних всередині розглянутих СУБД.

На основі всієї наявної інформації було обрано наступні операції з базами даних:

- вставка;
- оновлення;
- вибірка;
- пошук;
- фільтрація;
- групування.

2 ОБҐРУНТУВАННЯ ЕКСПЕРИМЕНТАЛЬНОГО НАПРЯМКУ ДОСЛІДЖЕННЯ ЧАСОВОЇ ЕФЕКТИВНОСТІ СУБД

Порівняння продуктивності різних NoSQL (нереляційних систем управління базами даних) важливе з кількох причин, кожна з яких сприяє загальній ефективності та результативності систем баз даних. Ось ключові аспекти, що наголошують на важливості порівняння продуктивності різних баз даних NoSQL.

1. Вибір відповідної бази даних для завдання.

Порівняння продуктивності допомагає визначити, яка база даних NoSQL найкраще підходить для конкретних випадків використання. Наприклад, деякі бази даних можуть чудово працювати у сценаріях з інтенсивним читанням, тоді як інші працюють краще з робочими навантаженнями з інтенсивними операціями запису.

Також розуміння характеристик продуктивності кожної бази даних NoSQL дозволяє розробникам та архітекторам узгоджувати вибір бази даних із функціональними вимогами програми.

2. Оптимізація використання ресурсів.

Порівняння показників продуктивності допомагає організаціям оптимізувати використання ресурсів, гарантуючи, що вони ефективно інвестують ресурси без надмірного виділення чи недостатнього використання устаткування.

Тести продуктивності допоможуть визначити можливість масштабувати базу даних, тим самим допомогти компаніям планувати потенційне зростання і гарантуючи, що обрана база даних може масштабуватися горизонтально чи вертикально за необхідності.

3. Підвищення швидкості відгуку додатків.

Продуктивність безпосередньо впливає на досвід користувача. Вибираючи базу даних NoSQL, оптимізовану для конкретних операцій, програмні додатки можуть забезпечити швидший час відгуку, що призведе до підвищення задоволеності користувачів.

Порівняння показників продуктивності допомагає виявити бази даних з меншою затримкою, що вкрай важливо для додатків, які потребують обробки даних в режимі реального часу або майже в реальному часі.

4. Дотримання угод про рівень обслуговування та бізнес-цілей.

Організації часто визначають угоди про рівень обслуговування продуктивності додатків. Порівняння баз даних NoSQL гарантує, що обрана база даних відповідає угодам про рівень обслуговування, відповідає бізнес-цілям та підтримує якість обслуговування.

Тести продуктивності допомагають оцінити надійність та стабільність баз даних у різних умовах, допомагаючи гарантувати, що обрана база даних може постійно відповідати очікуванням щодо продуктивності.

5. Управління робочим навантаженням даних.

Різні бази даних можуть по-різному обробляти робочі навантаження з інтенсивним читанням або записом. Порівняння продуктивності допомагає організаціям ефективно розподіляти робоче навантаження за даними, запобігаючи вузьким місцям та забезпечуючи збалансованість операцій.

Розуміння впливу на продуктивність стратегій моделювання даних та індексування для кожної бази даних допомагає розробляти ефективні схеми баз даних та оптимізувати продуктивність запитів.

6. Перспективність та адаптованість.

Бази даних NoSQL з часом розвиваються, з'являються оновлення та нові функції. Регулярні оцінки продуктивності допомагають організаціям бути в курсі останніх покращень та адаптувати вибір бази даних до нових технологій.

У міру зміни вимог додатків чітке розуміння характеристик продуктивності різних баз даних NoSQL дозволяє організаціям адаптувати свій вибір баз даних до потреб.

7. Усунення несправностей та налагодження.

Порівняння продуктивності допомагає виявити потенційні проблеми, вузькі місця або області для покращення. Ця інформація корисна для усунення пошкоджень та налагодження програм.

2.1 Поняття часової складності

Часова складність – це міра часу, який алгоритм займає в залежності від розміру вхідних даних. Це спосіб опису ефективності алгоритму з погляду швидкості зростання часу його роботи зі збільшенням розміру вхідних даних. Часова складність часто виражається з використанням позначення великого, яке забезпечує верхню межу швидкості зростання алгоритму.

Велике О. Позначення:

O(1) – постійний час:

Описує алгоритми, які займають однакову кількість часу, незалежно від розміру вхідних даних. Прикладом є доступ до елемента масиву індексу.

O(log n) – логарифмічний час:

Звичайне явище для алгоритмів, які поділяють проблему більш дрібні підзадачи. Прикладом може бути двійковий пошук у відсортованому масиві.

O(n) – лінійний час:

Час роботи прямо пропорційний розміру вхідних даних. Прикладом є перебір масиву для пошуку елемента.

O(n log n) – лінійний час:

Часто асоціюється з алгоритмами «розділяй і владарюй», такими як сортування злиттям та швидке сортування.

O(n^2) – квадратичний час:

Зазвичай використовується для алгоритмів із вкладеними ітераціями за вхідними даними. Прикладами є бульбашкова сортування та сортування вставками.

O(n^k) – поліноміальний час:

Загальна форма для поліноміальної тимчасової складності, де k – константа. У цю категорію потрапляють алгоритми із вкладеними циклами.

O(2^n) – експонентний час:

Описує алгоритми, де час виконання подвоюється з кожним додатковим елементом на вході. Рекурсивні алгоритми, які вирішують завдання розміру n, часто є експонентними.

Звідси можна виділити ключові моменти:

Найкращий, середній та найгірший випадки – часова складність часто враховує найгірший сценарій, але іноді аналізуються середні або найкращі сценарії.

В нашому випадку на результати експерименту будуть впливати наступні чинники:

- кількість об`єктів в базі даних (n);
- спосіб зберігання даних в конкретній СУБД;
- особливості використання індексів.

2.2 Використання довірчого інтервалу

Довірчий інтервал – це середнє значення вашої оцінки плюс та мінус варіація цієї оцінки. Це діапазон значень, між якими, як ви очікуєте, перебуватиме ваша оцінка, якщо ви повторите тест, у межах певного рівня достовірності.

Впевненість у статистиці – це ще один спосіб опису ймовірності. Наприклад, якщо ви будуєте довірчий інтервал з рівнем достовірності 75 %, ви впевнені, що у 75 випадках зі 100 оцінка перебуватиме між верхнім та нижнім значеннями, заданими довірчим інтервалом.

Бажаний рівень достовірності зазвичай дорівнює одиниці мінус значення альфа (α), яке використовується у статистичному тесті:

Рівень впевненості = 1 − a

Таким чином, якщо ви використовуєте значення альфа p <0,05 для статистичної значущості, тоді рівень достовірності буде 1–0,05 = 0,95, або 95%. [25].

Під час дослідження може відбутися така ситуація, коли деякі тестові реузльтати будуть суттєво відрізнятися від середніх. Це може бути викликано тим, що пристрій, на якому проводиться дослідження, був занятий деяким службовим сервісом, який виконується на фоні. Такі результати варто виключати з підрахунків, аби вони не впливали на загальну статистику, на основі якої будуть робитися деякі висновки.

Для цього й потрібні довірчі інтервали, аби виключати аномальні результати, які знаходяться за межею деякого заздалегідь визначеного довірчого інтервалу (наприклад, 90%).

2.3 Обчислення часової ефективності операції

Часова ефективність кожного з досліджуваних алгоритмів буде обчислюватися за результатами експериментів. Дуже важливо виконати експеримент декілька сотень разів, щоб більш точно встановити середній показник часу. Середній показник часу для одного експерименту буде обчислюватися за такою формулою:

$$T_{avg} = \frac{\sum_{i=0}^{n} t_i}{n} \qquad (2.3)$$

де T_{avg} – середній час роботи алгоритму при заданих вхідних даних,

n – кількість експериментів,

t_i – час роботи алгоритму в і-тому експерименті.

Після того, як будуть знайдені показники середнього часу для декількох різних експериментів, по цим даним буде побудований графік та лінія тренду до нього. За допомогою засобів програми Excel можна буде знайти функцію побудованої лінії тренду, наприклад, такого виду: y = 0.7*x + 53.2, якщо залежність буде лінійною. Підставляючи деяку характеристику (наприклад, роздільну здатність екрану) під x, можна буде обчислити приблизний час роботи алгоритму.

На основі отриманого часу можна буде робити припущення яку СУБД краще використовувати для певного набору операцій та для задовільнення певних потреб бізнесу (зазвичай це і береться за основу і головну мету).

2.4 Вибір операцій для дослідження

1. Вибірка даних.

Вибір даних включає запит до бази даних MongoDB для отримання конкретних документів на основі заданих критеріїв.

Варіанти використання: отримання профілю користувача, отримання інформації про продукт або отримання певних записів залежно від умов.

Оптимізація запитів: Використовуйте відповідні індекси для полів, що беруть участь у запитах, для підвищення ефективності пошуку.

Рекомендації щодо сегментування: переконайтеся, що вибраний ключ сегментування відповідає загальним шаблонам запитів для оптимізації розподілу даних.

2. Вставка даних.

Вставка даних передбачає впорядкування результатів запиту у порядку на основі одного чи кількох полів.

Варіанти використання: вставка нового документу чи поля в базу даних.

3. Фільтрація даних.

Опис операції:

Фільтрування даних включає запит до бази даних для отримання документів, відповідних певним умовам.

Варіанти використання: фільтрація замовлень щодо статусу, пошук користувачів із певними характеристиками.

Складність запиту. Складність умов фільтра може спричинити час виконання; оптимізувати запити підвищення ефективності.

Індексування: поля індексу, які використовуються у фільтрах для прискорення пошуку даних.

4. Групування даних.

Опис операції:

Групування даних передбачає агрегування документів з урахуванням зазначених полів щоб одержати зведеної статистики чи аналітичної інформації.

Варіанти використання: угруповання даних про продаж по регіонах, агрегування активності користувачів по місяцях.

Індексування. Переконайтеся, що поля, які використовуються для групування, проіндексовані відповідно.

Розмір даних. Великі набори даних можуть вплинути на продуктивність під час угруповання. Потрібно відстежувати та оптимізувати використання ресурсів.

5. Оновлення даних.

Опис операції:

Оновлення даних передбачає зміну наявних документів у базі даних.

Варіанти використання: оновлення цін на продукти, зміна переваг користувачів.

Складність оновлення: Складність операцій оновлення, включаючи модифікатори та оновлення, може вплинути на час виконання.

6. Пошук даних.

Опис операції:

Дана операція передбачає пошук конкретного документу за ключем.

Висновки до розділу 2

В другому розділі було освітлено та обґрунтовано експериментальний напрямок дослідження часової ефективності операцій вибірки, вствки, групування, фільтрування, оновлення даних та пошуку у різних системах управління базами даних. Було згадано, що потенційно мож вплинути на результати вимірювань. Також в цьому розділі було з’ясовано, що часова ефективність досліджуваних операцій та запитів буде оцінюватися наступним чином:

- проведення експерименту з конкретною операцією певну кількість разів;
- виключення аномальних результатів, які знаходяться за межею деякого довірчого інтервалу;
- повторити шаг 1 і 2, але з іншими запитами;
- побудувати порівняльні таблиці отриманих результатів;
- побудувати діаграми з отриманими результатами.

3 ПРОЕКТУВАННЯ ТА РОЗРОБКА ПРОГРАМНОГО ІНСТРУМЕНТАРІЮ ДЛЯ ДОСЛІДЖЕННЯ ЧАСОВОЇ ЕФЕКТИВНОСТІ ДОКУМЕНТООРІЄНТОВАНИХ СУБД

3.1 Зовнішнє проектування

3.1.1 Вхідні данні

Вхідними даними програми є:

- тип запита до бази даних (текст);
- кількість тестів (ціле число).

3.1.2 Вихідні дані

Результатом роботи програми є наступні вихідні дані:

- візуальне відображення часу виконання операції;
- файл формату .CSV, який містить результати експерименту та має такі дані:
  a. час роботи операції;
  b. використаний тип операції;

3.1.3 Формалізація задач

Формалізація задачі на рівні зовнішнього проектування представлена у вигляді діаграми варіантів використання.

В уніфікованій мові моделювання (UML) діаграма варіантів використання може узагальнити ролі користувачів вашої системи (також відомих як актори) та їх взаємодію з системою. Ефективна діаграма варіантів використання може допомогти вашій команді обговорити та представити:

- сценарії, у яких ваша система або програма взаємодіє з людьми, організаціями або зовнішніми системами;
- цілі, яких ваша система або програма допомагає цим об’єктам (відомим як актори) досягти;
- область вашої системи.

Користувач може виконувати наступні варіанти використання:

- встановити необхідні параметри експерименту;
- запустити тестування типу операції;

- повернутися до вікна встановлення параметрів експерименту;
- зберегти файл з результатами експерименту.

Схема варіантів використання (Use-case diagram) наведена на рис. 3.1:

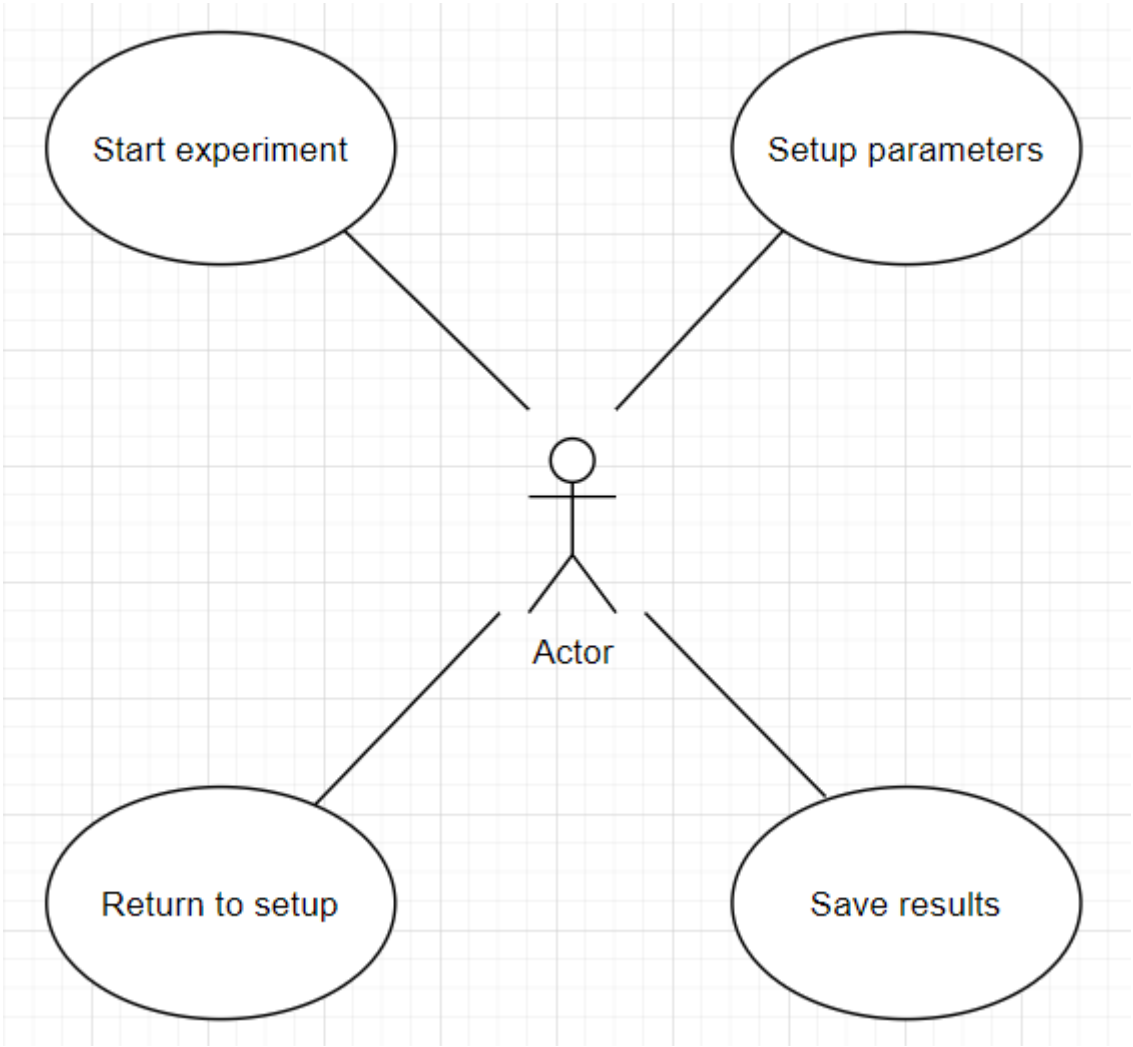

Рисунок 3.1 – діаграма варіантів використання

3.1.4 Проектування динаміки системи

На початку потрібно визначити, які можуть бути сценарії використання системи, що розроблюється. Для цього можна побудувати UML діаграму послідовності.

Діаграми послідовності – це діаграми взаємодії, які докладно описують виконання операцій. Вони фіксують взаємодію між об'єктами у певній колаборації. Діаграми послідовності орієнтовані тимчасово і візуально показують порядок

взаємодії, використовуючи вертикальну вісь діаграми для часу, які повідомлення відправляються і коли.

Діаграми послідовності фіксують:

- взаємодію, що відбувається у співпраці, яка або реалізує варіант використання, або операцію (діаграми екземплярів або загальні діаграми);
- взаємодію високого рівня між користувачем системи та системою, між системою та іншими системами або між підсистемами (іноді відомі як діаграми послідовності систем).

На рис. 3.2 зображено розроблену діаграму послідовності для одного основного сценарію роботи користувача з системою. Актор, тобто користувач, є ініціатором послідовності дій.

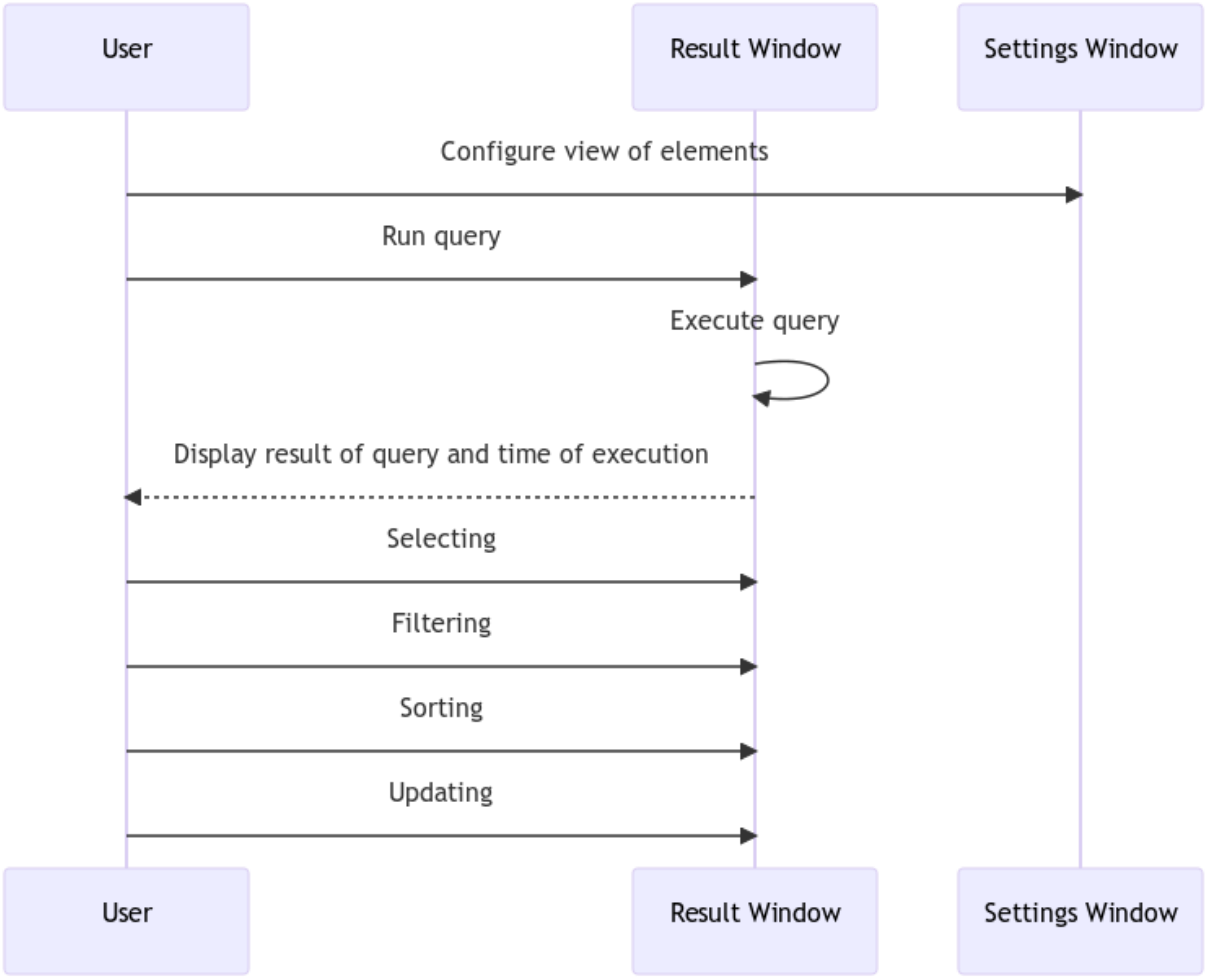

Рисунок 3.2 – Діаграма послідовності

3.1.5 Проектування інтерфейсу користувача

Дизайн інтерфейсу користувача — це процес створення візуального макета та зовнішнього вигляду програмного забезпечення, програм, веб-сайтів або будь-якого цифрового інтерфейсу, з яким взаємодіє користувач. Приклади компонентів інтерфейсу користувача включають розташування кнопок на сторінці, копію, яка направляє вас через кроки для створення облікового запису з іменем користувача та паролем, і значки, які направляють вас під час покупки. Дизайн інтерфейсу користувача спрямований на створення простих, естетично привабливих і функціональних інтерфейсів, які допомагають користувачам виконувати завдання.

Розробники інтерфейсу користувача зазвичай тісно співпрацюють із розробниками користувацького досвіду, і хоча UI та UX тісно взаємопов’язані, це окремі дисципліни. Якщо дизайн інтерфейсу користувача зосереджується на візуальних та інтерактиваних елементах продукту, як макет, типографія, кольори, піктограми та кнопки. А UX-дизайн, в свою чергу, стосується загального досвіду користувача, включаючи такі високі концепції, як загальне призначення продукту та те, як користувач відчуває це. І те, і інше має вирішальне значення для успішного результату: надійний дизайн інтерфейсу користувача підтримує базову стратегію UX.

Під час розробки інтерфейсу потрібно дотримуватись базових принципів, які допоможуть зробити інтерфейс користувача більш інтуітивним та простим. Наступні шість принципів демонструють яким чином краще побудувати інтерфейс користувача.

1. Зробити роботи користувача зручною.

Користувач, у цьому випадку, є вашим клієнтом. Щоб покращити їхній досвід перегляду, важливо створити на вашому веб-сайті або додатку приємне середовище. Ось як зробити навігацію інтуїтивно зрозумілішою та зручнішою:

- менше значить більше. Незалежно від вашого смаку в мистецтві, мінімалізм — це назва гри в дизайні інтерфейсу користувача. Хороший дизайн інтерфейсу користувача вимагає видалення зі сторінки всього, що не

відповідає вашим потребам. Кричущі шрифти, непотрібна інформація та надто багатослівний текст є основними винуватцями.

- тримайте чітку візуальну ієрархію. Закон Фітта стверджує, що більші цілі легше та швидше клацати, ніж менші, віддалені цілі. Іншими словами, переконайтеся, що такі важливі елементи, як кнопки «Купити», помітні та помітні, а ключові фотографії та інформація природно привертають увагу.
- не ускладнювати. Чудовий інтерфейс користувача використовує просту мову, читабельні шрифти та скорочену колірну схему. Негативний простір (або пробіл) на сторінці також бажаний.
- тримайте це послідовно. Елементи дизайну інтерфейсу користувача, такі як кнопки, макет сторінки та колірні схеми, повинні залишатися однаковими на всьому сайті. Дотримуйтесь загальних умов дизайну, наприклад як розміщення меню у верхній частині сторінки.
- уникайте жаргону. Нехай ваші відвідувачі почуваються комфортно на вашому сайті, використовуючи формулювання, які не лякають і не містять непотрібних технічних термінів.

Іншими словами, ваша мета - чистий, привабливий дизайн із цілеспрямованими візуальними елементами, які надають користувачам необхідну інформацію, коли це необхідно.

2. Дозвольте користувачеві контролювати досвід.

Дизайн інтерфейсу користувача – це також можливість пізнати свою аудиторію. За допомогою маркетингових досліджень або навіть демографічної інформації, доступної через облікові записи соціальних мереж, ви можете заглибитися в їхні потреби та інтереси. Отримайте розуміння, співчуваючи користувачам і визначаючи їхні мотиви для відвідування вашого сайту. Які кнопки та візуальна інформація потрібна користувачам заздалегідь? Що досвідченіші користувачі можуть захотіти глибше? Розставляючи пріоритети для своєї аудиторії, ви дозволяєте її потребам визначати інтерфейс.

Ще один спосіб дозволити користувачам контролювати процес — це дозволити їм скасувати дію без негативних наслідків. Це робить вивчення сайту більш

доступним і безпечним, знаючи, що вони завжди можуть повернутися назад і повернутися пізніше, якщо вони натраплять на нудні послідовності введення даних. Так звані навігаційні стежки, які показують користувачам, де вони були та куди йдуть, допомагають їм скласти розумову карту сайту та легше знаходити інформацію.

3. Зробіть це інтуїтивно зрозумілим

Хороший користувальницький інтерфейс інтуїтивно зрозумілий, усе саме там, де очікує користувач. Навіть найбільш «проривні» бренди не порушують очікувань користувачів у своєму дизайні інтерфейсу користувача. Натомість метою має бути бездоганна взаємодія з користувачем.

Знайомі рисунки, шаблони, вибір шрифтів і макети сторінок зменшують когнітивне навантаження вашого відвідувача. Це також означає, що вам не потрібно заново винаходити велосипед або пояснювати кожне дизайнерське рішення. Маленькі трирядкові меню у верхній частині кожної програми працюють, по причині того, шо всі знають, яким чином вони працюють.

4. Бути доступним

Кольоровою сліпотою страждає близько 5% дорослого чоловічого населення світу та 0,5% жіночого населення. Всесвітня організація охорони здоров’я повідомляє, що 253 мільйони людей мають ту чи іншу форму сліпоти або порушення зору, 466 мільйонів мають глухоту та втрату слуху, а ще 200 мільйонів мають інтелектуальну недостатність.

Створити хороший веб-сайт означає створити хороший веб-сайт для всіх. WCAG Web Accessibility Initiative надає розробникам інтерфейсу користувача докладні вказівки. Одне емпіричне правило полягає в тому, щоб не покладатися на один елемент для здійснення всього спілкування. Дві кнопки з написами «Так» і «Ні» також можуть мати галочку та X відповідно. Зробивши їх зеленими та червоними, можна додати ще один рівень спілкування, гарантуючи, що дальтонізм або навички читання не завадять користувачеві переміщатися по сайту.

Тут також допомагає попередній принцип — «Зробіть це інтуїтивно зрозумілим». Стандартні символи та фрази тримають усіх на твердій землі.

Піктограми, такі як піктограма кошика для покупок, знак долара та галочки, широко відомі та дозволяють користувачам усіх типів легко орієнтуватися.

5. Повідомляти користувачам, коли все працює.

Користувачі хочуть знати, коли щось працює. Спробуйте додати легку анімацію під час натискання кнопки для підтвердження дії, сторінку подяки, щоб закрити цикл після успішної транзакції чи заповнення форми, індикатор виконання, якщо функція потребує часу для завантаження, або ряд вимог, які поступово змінюються зелений під час спроби створити пароль. Усе це може викликати у ваших відвідувачів знайоме відчуття завершеності.

3.1.6 Створення ескізів інтерфейсу

На етапі проектування інтерфейсу користувача зручно користуватися “мокапами” - іншими словами ескізами нашого інтерфейсу, першому баченню інтерфейсу. Тому про створені “мокапів” було акцентовано всю увагу на узгодженості всіх елементів інтерфейсу користувача, щоб всі дані компоненти були інтуїтивно зрозумілі майбутнім юзерам даного додатку і їх було доволі легко навчити користуванню. Тобто всі елементі робились в узгодженому стилі та із зрозумілими назвами кнопок.

Інтерфейс користувача складається з наступних елементів:

1) Рядок заголовка вікна: угорі є рядок заголовка вікна з назвою "Time_Comparer", а також стандартні значки керування вікном для мінімізації, розгортання/відновлення та закриття вікна.
2) Параметри бази даних: ліворуч є панель із позначкою «MongoDb», яка свідчить про те, що програма може порівнювати продуктивність операцій MongoDB з іншою базою даних. На цій панелі є три параметри з перемикачами:
   - вибірка;
   - групування;
   - оновлення;
   - фільтрація;
   - пошук;

- вставка.

Ці параметри відповідають різним типам запитів до бази даних або операцій, які користувач може перевірити на час виконання.

3) Індикатор часу виконання: у центрі над великою сірою рамкою розташована позначка «Час виконання», яка вказує на те, що програма здатна відображати час, потрібний для виконання вибраної операції.
4) Поле для відображення запитів: у центрі інтерфейсу створена прямокутна область. Це простір, призначений для відображення результатів запитів і з чого складаються дані з різних запитів.
5) Протилежні параметри бази даних: праворуч є панель, подібна до панелі ліворуч із позначкою «CouchDb». Тут користувач може вибрати подібні операції для другої бази даних, щоб провести порівняння. Доступні параметри тут такі ж самі, що і зліва:
   - вибірка;
   - групування;
   - оновлення;
   - фільтрація;
   - пошук;
   - вставка.
6) Кнопки керування: під цими опціями є дві кнопки:
   - «показати все»: цю кнопку можна використовувати для відображення всіх даних або результатів;
   - «виконати запит»: це кнопка, яку користувач натисне, щоб виконати вибрані операції та виміряти час їх виконання.

Далі можна скріншот головної форми додатку, який містить всі елементи для роботи з програмою:

Рисунок 3.3 – Ескіз екрану додатку

3.2 Внутрішнє проектування

3.2.1 Технологічна платформа

В якості технологічної платформи для розробки програмного додатку було вибрано технологію WPF від компанії Microsoft.

WPF або Windows Presentation Foundation - це структура інтерфейсу користувача (інтерфейс користувача), яка створює клієнтські додатки для робочого столу (desktop). Платформа розробки WPF підтримує широкий набір функцій розробки додатків, включаючи модель додатків, ресурси, елементи керування, графіку, макет, зв’язування даних, документи та захист.

WPF є частиною .NET і дуже схожий на ASP.NET або Windows Forms WPF використовує Extensible Application Markup Language XAML для надання декларативної моделі для програмування додатків.

В основі WPF лежить такий архітектурний патерн, як MVVM (Model View ViewModel). Він складається з наступних концептів:

- структура побудована таким чином, що представлення отримують свій стан із моделей представлень замість використання «коду з коробки»;
- означає, що команди використовуються замість подій для обробки дій користувача;
- дані передаються в представлення за допомогою прив'язки даних. Також можна передавати дані з представлення до серверної частини за допомогою двостороннього зв'язування даних;
- програма WPF використовує N-рівневу архітектуру MVVM, як показано на малюнку нижче.

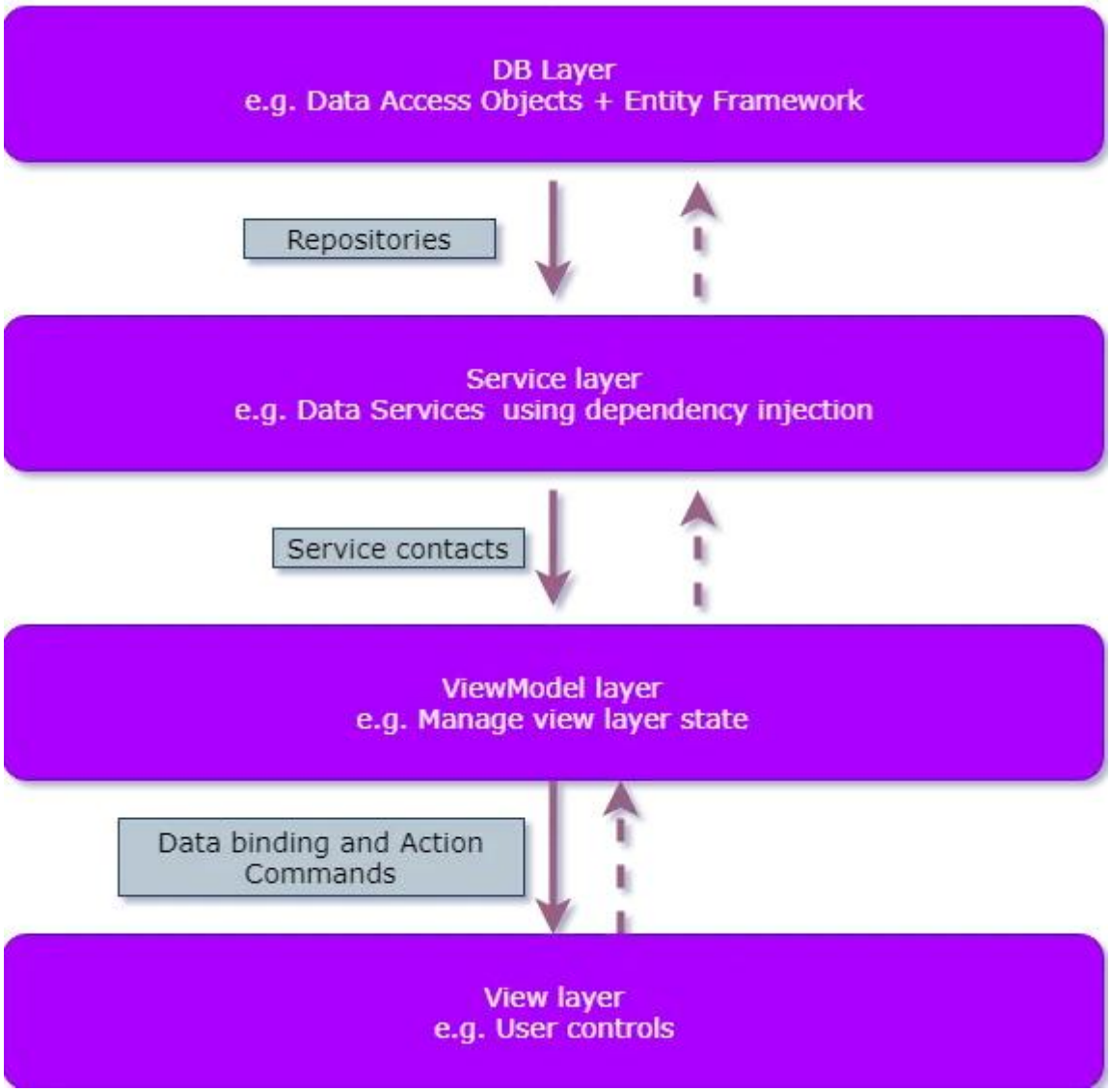

Рисунок 3.4 – Принцип роботи WPF

В якості інструменту для розробки було вибрано WPF спираючись на наступні переваги:

- можливість створювати власні елементи керування інтерфейсом користувача з великим набором можливостей та гнучкістю;
- у налаштуваннях із різною роздільною здатністю перевагою є масштабованість;
- дуже корисна особливість, яка допомагає створювати більш підтримуванні додатки - повторне використання коду;
- надає можливість взаємодіяти з елементами керування, які, в свою чергу, мають високоефективні та потужні можливостями зв’язування даних;
- це продукт компанії Microsoft, тому можна знайти доволі багато документації по цьому напрямку і також отримувати оновлення з новим функціоналом;
- можливість додавати анімацію та інші спеціальні ефекти до контролерів і компонентів інтерфейсу користувача на екрані.

3.2.2 Вибір мови програмування

Для розробки інструменту для проведення подальших досліджень було обрано мову C#.

C# – сучасна об'єктно-орієнтована і типобезпекова мова програмування [33]. C # відноситься до широко відомого сімейства мов C, і буде добре знайомою кожному, хто працював з C, C ++, Java або JavaScript.

Основною перевагою C# є поняття типобезпековості, що означає, що типиданих у C# можуть взаємодіяти лише за допомогою протоколів, визначених самим типом. Простіше кажучи, у C# розробники не можуть розглядати рядковий тип як цілий тип, і навпаки. Це основна функція певних мов програмування, яка допомагає запобігти помилкам, які виникають, коли один тип даних (неправильно) використовується так, ніби це інший.

Далі розглянемо основні переваги даної мови програмування, які забезпечують надійність та масштабованість додатків.

C# є об'єктно-орієнтованим. В ООП можна легко визначити структуру (і тип) даних і згрупувати їх в об’єкти, які створюються з класів (думайте про класи як про плани, а про об’єкти як про речі, які ви робите з цих планів). Таке групування даних

полегшує розробку додатків, підтримку додатків і повторне використання коду. Це також робить код легшим для виправлення та зменшує ймовірність знаходження помилок.

C# є частиною сімейства мов C, тому одна з ключових особливостей C# відома як сумісність, що означає, що він підтримує інші функції мов сімейства C (тобто ми можемо використовувати їх у поєднанні одна з одною). Сімейство мов програмування C включає такі важкі мови, як C, C++ і Java.

Управління пам'яттю у C# знімає тягар з розробника у формі вбудованого збирача сміття, також відомого як GC. Збирач сміття керує пам’яттю, відстежуючи невикористовувані об’єкти, і автоматично звільняє пам’ять, коли вона більше не потрібна, тому розробникам не потрібно про це турбуватися. Загалом, мета збирача сміття полягає в тому, щоб керувати невикористаними об’єктами, очищати пам’ять і розподіляти пам’ять для нових об’єктів, коли старі об’єкти більше не потребують її.

C# є кросплатформним, що є ще однією перевагою C# є те, що мова програмування є кросплатформною. Це дивовижний спосіб сказати, що якщо ви створюєте програму на C#, вона може працювати на будь-якій операційній системі чи платформі, включаючи Apple, iOS, Windows, Android або в хмарі.

C# — це структурована мова програмування. Окрім того, що C# є об’єктно-орієнтованою та безпечною для типів мовою програмування, C# також є структурованою мовою програмування. Як випливає з назви, це означає, що програми, написані на C#, написані логічно, структуровано – розбиті на невеликі модулі, відомі як процедури та функції. Таке структурування коду полегшує його читання, розуміння, підтримку, налагодження та більш ефективну роботу.

3.2.3 Вибір системи контролю версіями

Контроль версій, також відомий як контроль джерела, — це практика відстеження та керування змінами програмного коду.

Системи контролю версій — це програмні інструменти, які допомагають командам розробників керувати змінами вихідного коду з часом.

Особливо такі системи є корисними при спільній роботі над проектом в команді. Наприклад, один розробник може працювати над новим функціоналом, а

інший працювати над кодом в іншій частині проекту. Зміни, внесені в одну частину програмного забезпечення можуть бути несумісними зі змінами, внесеними іншим розробником. Система керування версіями відстежує зміни, внесені кожним учасником команди та допомагає виявляти конфлікти при паралельній роботі.

В якості системи контролю версій було обрано Git. Git дозволяє мати кілька локальних гілок, які можуть бути повністю незалежними одна від одної [42].

Git — це безкоштовна система керування версіями з відкритим вихідним кодом, спочатку створена Лінусом Торвальдсом у 2005 році. На відміну від старих централізованих систем контролю версій, таких як SVN і CVS, Git є розподіленим: кожен розробник має повну історію свого сховища коду локально. Це робить початковий клон сховища повільнішим, але подальші операції, такі як фіксація змін, порівняння зроблених змін, злиття та журнал, значно пришвидшуються.

Git також чудово підтримує розгалуження, злиття та переписування історії сховищ, що призвело до багатьох інноваційних та потужних робочих процесів та інструментів. Pull-запити є одним із таких популярних інструментів, який дозволяє командам співпрацювати в гілках Git і ефективно переглядати код один одного. Git є найпоширенішою системою контролю версій у світі на сьогоднішній день і вважається сучасним стандартом розробки програмного забезпечення.

3.2.4 Взаємодія класів розроблюваної системи

На початку потрібно визначитись з основним функціоналом додатку, що буде розроблятись:

- можливість вибору різних типів запитів до різних СУБД (MongoDb, CouchDb);
- візуальне відображення контексту запитів до розподілених баз даних в спеціальній формі;
- виконання “перформанс” експериментів з заміром часу на кожну операцію до баз даних;
- збереження результатів у файл для подальшого порівняння у інших інструментах.

Якщо подумати про архітектуру систему на абстрактному рівні, то попередньо можна визначити систему класів програмного додатку. Оскільки в нас буде два провайдери даних (MongoDb, CouchDb), то відповідно нам потрібно створити клас репозиторій, який буде абстракцією для декількох контекстів баз даних. Від нього будуть створені конкретні нащадки для кожного підключення до баз даних і відповідно методу з виконання запитів до них. Також потрібен окремий клас для роботи з заміром часу виконання потрібних нам операцій. Крім того, треба мати декілька класів-моделей для представлення наших даних (в даному випадку документів, так як ми працюємо з документоорієнтовними базами даних). І в кінці потрібно клас, який буде відповідальним за збереження результатів експериментів до файлу і стандартний клас для WPF – View, який відповідає за рендеринг нашого інтерфейсу користувача.

Схема взаємодії класів зображена на рис. 3.4:

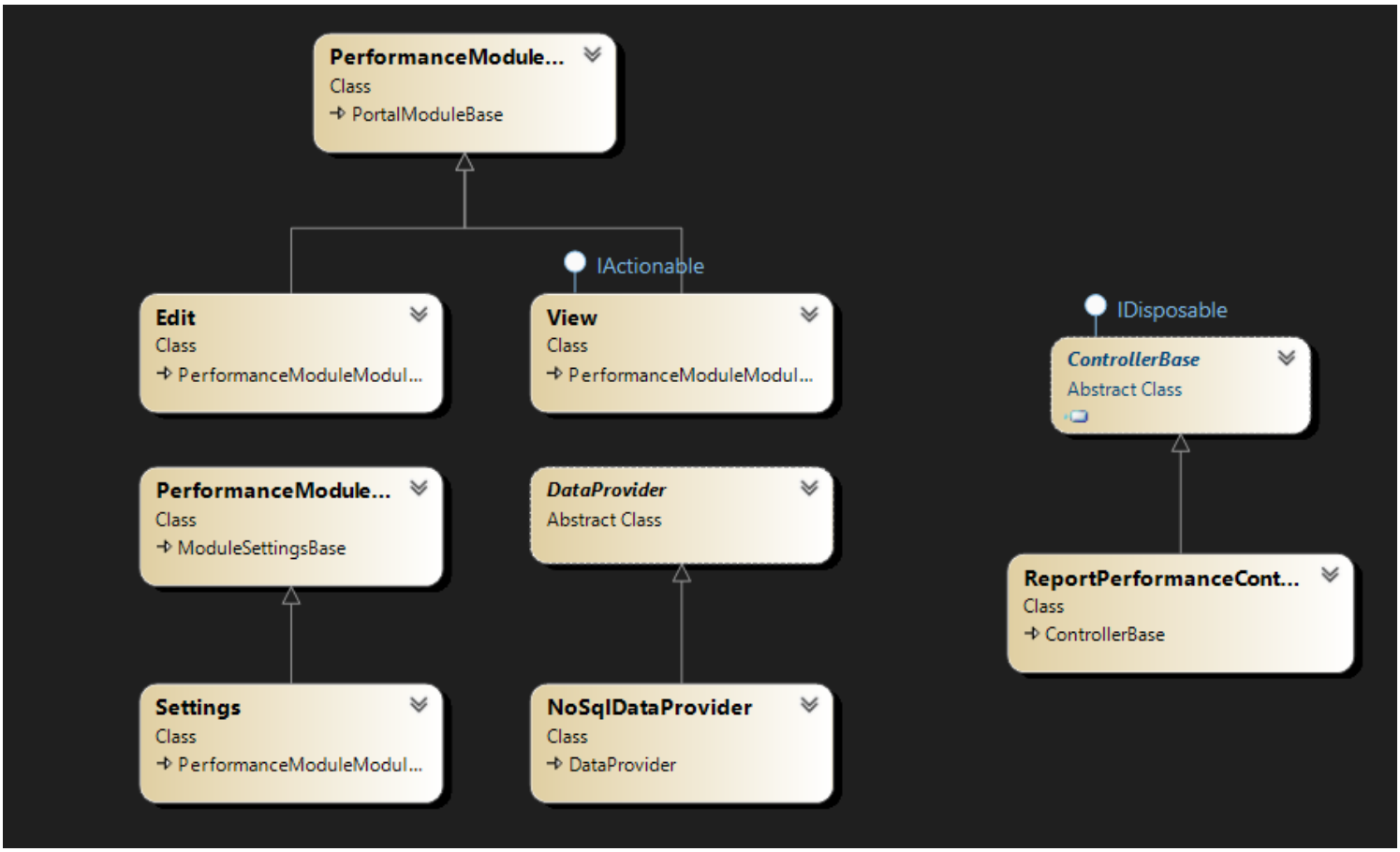

Рисунок 3.4 – Схема взаємодії класів

PerformanceModuleBase – це абстрактний клас через назву, що означає, що його не можна створити самостійно. Він забезпечує базову структуру для класів модуля продуктивності. Він не має прямих зв’язків з іншими показаними класами. Цей клас потрібен для того, щоб зробити обмеження створення неправильних типів з певною поведінкою.

PortalModuleBase – цей клас з’єднаний із «PerformanceModuleBase» пунктирною стрілкою з незафарбованим наконечником стрілки, що зазвичай представляє зв’язок залежності, що означає, що клас «PortalModuleBase» може використовувати деякі функції «PerformanceModuleBase».

Edit – це конкретний клас, який, здається, походить від "PerformanceModuleSettingsBase", як показано суцільною лінією з замкнутою стрілкою, яка представляє успадкування. Клас "Edit" успадковував би властивості та методи свого суперкласу.

PerformanceModuleSettingsBase – це, імовірно, абстрактний клас (запропонований назвою, виділеною курсивом), який служить основою для класів із налаштуваннями, пов’язаними з модулями продуктивності. Він служить суперкласом для класу "Редагувати", як зазначено у відносинах успадкування.

View та Edit – обидва ці класи є конкретними. "View" успадковує "PerformanceModuleSettingsBase", подібно до класу "Edit". Клас «Edit» з’єднаний із «View» пунктирною стрілкою, яка вказує на «ICacheable», що може означати, що «View» залежить від інтерфейсу «ICacheable» або використовує його.

ControllerBase – абстрактний клас, на що вказує назва, виділена курсивом, і слово «абстрактний клас», написане під ним. Абстрактні класи не можуть бути створені, і вони зазвичай надають шаблон для підкласів.

ReportControllerBase – цей клас успадковує «ControllerBase», що означає, що це більш спеціалізований контролер, який, імовірно, містить додаткову логіку, специфічну для звітів.

DataProvider і NoSqlDataProvider – «DataProvider» — це абстрактний клас, який використовується для визначення спільного інтерфейсу або базової функції для всіх постачальників даних. "NoSqlDataProvider" успадковує "DataProvider", вказуючи на те, що "NoSqlDataProvider" є певним типом постачальника даних, призначеного для роботи з базами даних NoSQL.

3.2.5 Використані принципи проектування

Під час проектування системи були використані наступні принципи проектування: SOLID, YAGNI та DRY. Далі зупинимось більш детально на кожному з них.

Принцип YAGNI («Це вам не знадобиться») — це практика розробки програмного забезпечення, згідно з якою функції слід додавати лише за потреби. Як частина філософії екстремального програмування (XP), YAGNI усуває надлишки та неефективність у розробці, щоб сприяти покращеній частоті випуску релізів.

Цей принцип допомагає розробникам уникнути марних зусиль на функціях, які, як передбачається, знадобляться в певний момент. Ідея полягає в тому, що це припущення часто виявляється невірним. Навіть якщо функція зрештою є бажаною,

все одно може виявитися, що реалізація не потрібна. Аргумент полягає в тому, щоб розробники не витрачали час на створення сторонніх елементів, які можуть бути непотрібними та можуть перешкоджати або уповільнювати процес розробки.

Оскільки YAGNI допомагає уникнути витрачання часу на функції, які можуть не використовуватися, основні функції програми краще розроблені, і на кожен випуск витрачається менше загального часу.

Принцип «Don’t Repeat Yourself» (DRY) стверджує, що «Кожна частинка знання повинна мати єдине, однозначне і надійне представлення в системі». Отже, основна задача принципу — позбутися дублювання логіки, що сильно спростить майбутню підтримку коду, адже за необхідності щось змінити достатньо буде це зробити лише в одному місці. І тут найважливішим є поняття «частинка знання». Воно стосується не символів в коді, а, скоріше, бізнес-логіки.

Принципи SOLID — це набір із п’яти принципів проектування, які допомагають розробці програмного забезпечення для створення більш зручного, масштабованого та гнучкого програмного забезпечення. Ці принципи були введені Робертом К. Мартіном і широко використовуються в об'єктно-орієнтованому програмуванні. Акронім SOLID означає:

1) Принцип єдиної відповідальності (SRP):
   - клас повинен мати лише одну причину для зміни, тобто він повинен мати лише одну відповідальність або роботу;
   - цей принцип підтримує ідею про те, що клас повинен інкапсулювати одну концепцію або функціональність.
2) Принцип відкритості/закритості (OCP):
   - програмні сутності (класи, модулі, функції тощо) мають бути відкритими для розширення, але закритими для модифікації;
   - він заохочує розробників розширювати поведінку модуля, не змінюючи його вихідний код, сприяючи використанню інтерфейсів і абстрактних класів.
3) Принцип підстановки Лісков (LSP):

- об'єкти суперкласу повинні бути замінними на об'єкти підкласу без впливу на коректність програми;
- підтипи повинні бути замінними для своїх базових типів без зміни коректності програми.

4) Принцип поділу інтерфейсів (ISP):
   - клас не повинен бути змушений реалізовувати інтерфейси, які він не використовує;
   - цей принцип підштовхує до створення конкретних інтерфейсів, орієнтованих на клієнта, а не створення великих інтерфейсів загального призначення, які можуть змусити клієнтів реалізовувати методи, які їм не потрібні.
5) Принцип інверсії залежностей (DIP):
   - модулі високого рівня не повинні залежати від модулів низького рівня; обидва повинні залежати від абстракцій;
   - абстракції не повинні залежати від деталей; деталі повинні залежати від абстракцій;
   - цей принцип підкреслює використання абстракцій (наприклад, інтерфейсів або абстрактних класів) для відокремлення модулів високого рівня від модулів низького рівня, сприяючи гнучкості та легкості змін.

Застосування цих принципів SOLID у розробці програмного забезпечення допомагає створювати модульний код, зручний для обслуговування та менш схильний до помилок.

3.3 Стратегія тестування

Тестування програмного забезпечення охоплює широкий спектр завдань, що є подібними до етапів розробки програмного забезпечення. До них належать визначення цілей тестування, проектування, створення тестових сценаріїв, перевірка коректності тестів та безпосереднє виконання тестових сценаріїв із подальшим аналізом отриманих результатів. Особливу роль у цьому процесі відіграє проектування тестів. Для створення стратегії тестування існує широкий спектр підходів [47].

Основні питання, на які має відповісти стратегія тестування:

- що необхідно протестувати;
- які методи тестування будуть використані.

Методи тестування поділяються на дві основні категорії: тестування «чорної скриньки» (за вхідними даними) [27] та тестування «білої скриньки» (логіки програми) [28].

Методи тестування за принципом «чорної скриньки» є ефективними для програм, що мають лінійну структуру. Щоб визначити найоптимальніший метод для такої структури, кожен із методів аналізується окремо. Для складних структур або програм, які обробляють дані зі складною організацією, доцільно поєднувати два методи тестування: поділ на еквівалентні класи (метод «чорної скриньки») та підхід, орієнтований на вхідні та вихідні дані, що враховує залежності між ними. Такий підхід дозволяє виявити не лише помилки, що охоплюються специфікацією, але й ті, що залишаються поза межами стандартних тестів за допомогою методу «білої скриньки».

Було прийнято рішення протестувати програму методом припущень про помилки, який належить до категорії тестування «чорної скриньки».

Цей метод базується на досвіді тестувальника, знаннях про типові помилки в аналогічних системах, а також розумінні потенційно вразливих місць програми. У рамках цього підходу тестувальник формує гіпотези щодо можливих дефектів, після чого створюються тестові сценарії для перевірки цих гіпотез. Метод припущення про помилки передбачає генерацію тестових випадків на основі ймовірних проблем, таких як некоректна обробка граничних значень, помилки в алгоритмах обробки даних чи недоліки в користувацькому інтерфейсі (наприклад, незручне розташування елементів або помилки у відображенні).

3.3.1 Тестування програми методом припущення про помилку

Оскільки метод припущення про помилку відноситься до методів «чорного ящика», то треба допустити, що ми не маємо доступу до вихідного коду.

Для проведення деякого експерименту користувач спочатку повинен ввести параметри експерименту. Він повинен вибрати тип операції, яку хоче заміряти. Інші елементи інтерфейсу передбачають, що буде обраний лише один з запропонованих варіантів (під час вибору операції та файлу моделі), а це означає, що користувач не зможе зробити тут щось непередбачуване.

Тому, можна зробити припущення, що ввівши некоректну інформацію в вищезгаданих полях, можна викликати дефекти в програмі. На табл. 3.1 зображено результати тестування.

Таблиця 3.1 – Тестування методом припущення про помилку

Сценарій	Очікуваний результат	Актуальний результат
1	2	3
Жодна опція дропдауну не була вибрана	Повідомлення про те, що потрібно вибрати хоча б одну опію.	Повідомлення про те, що потрібно вибрати хоча б одну опію
Була вибрана опція дропдауну з певною операцією	Відбувається процесинг операції та замір часу виконання з відображенням у відповідному полі	Відбувається процесинг операції та замір часу виконання з відображенням у відповідному полі

3.3.2 Налагодження програми

Оскільки програмний додаток має багато функціональних частин, таких як запити до бази даних та логічні операції до них, то налагодження програми було проведено за методом індукції та просування від місця виникнення помилки.

Для налагодження програми були використані можливості редактору «Visual Studio». Ця IDE дозволяє переглянути стек викликів [29], за допомогою якого можна визначити послідовність виклику методів і відповідно метод, в якому потенційно може бути помилка. Ниже зображено приклад стеку викликів з редактору коду «Visual Studio»

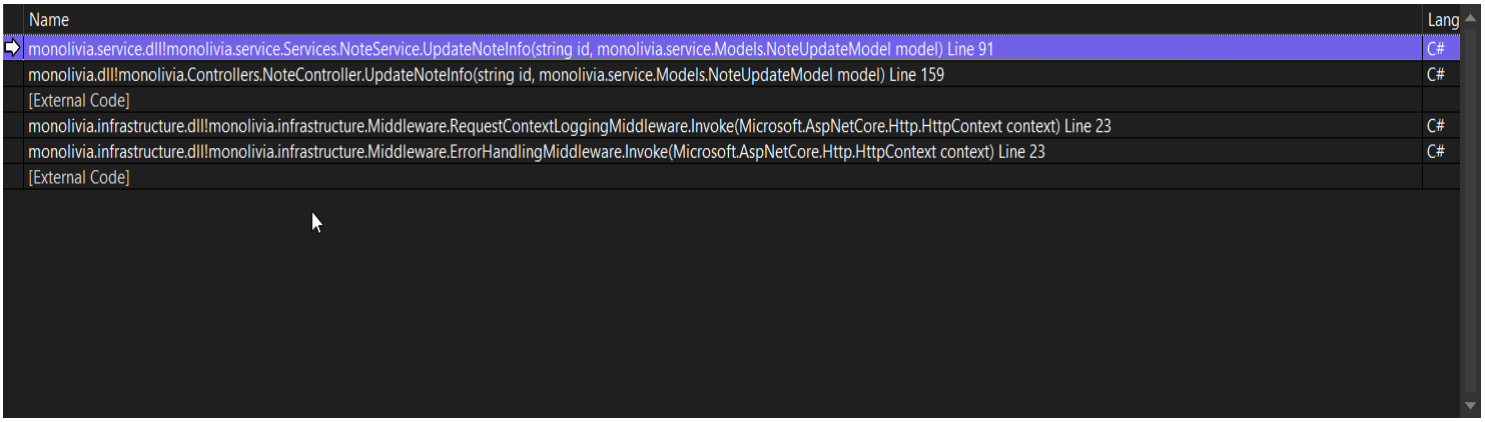

Рисунок 3.5 – Стек викликів

Також редактор має таку корисну функцію, як встановлення точок зупину («breakpoint») [30]. Краще за все ставити їх за декілька кроків до можливої помилки, як на рис. 3.7. Це дозволить краще зрозуміти, що посприяло появі помилки, а потім виправити її та протестувати цю ділянку коду ще раз. Для налагодження ділянок коду з великою кількістю циклів або з масивними «LINQ» запитами, доволі корисно використовувати умовні точки зупину [30]. Їх використання дозволяє зупиняти роботу програми лише при певних умовах, а не кожної ітерації. Це буде дуже корисно при налагодженні ділянок коду всередені циклів з 10 та більше ітерацій, щоб не зупинятися на кожній з них.

Рисунок 3.6 – Встановлення точки зупину

В ході налагодження програми було виявлено декілька помилок у запитах до бази даних та вимірюванні ефективності. Всі виявлені помилки були виправлені.

Висновки до розділу 3

У третьому розділі було детально розглянуто аспекти зовнішнього та внутрішнього проектування.

Під час зовнішнього проектування було формалізовано задачу, визначені вхідні та вихідні параметри та розроблена схема способів використання. Також було розглянуто систему у динаміці за допомогою розробленої діаграми послідовності. Крім цього був розроблений простий інтерфейс користувача після визначення основних положень щодо розробки інтуїтивно зрозумілого користувацького інтерфейсу.

При внутрішньому проектуванні було обрано технологічну платформу, мову програмування та систему контролю версій. Було розроблено ієрархію класів та їх взаємодію згідно з принципами парадигми ООП.

4 ДОСЛІДЖЕННЯ ЧАСОВОЇ ЕФЕКТИВНОСТІ ОПЕРАЦІЙ РОБОТИ З ДАНИМИ У РІЗНИХ СУБД

4.1 Підготовка до експерименту

4.1.1 Опис використаного програмно-апаратного середовища

Дослідження проводилося з використанням персонального комп’ютеру (ноутбуку) на базі операційної системи Windows. Нижче наведено основні характеристики використаного девайсу.

Процесор:

- модель: Intel Core i7-10550H;
- тактова частота процесора: від 1.8 до 3.7 ГГц;
- кількість ядер: 4;
- кількість потоків: 8;
- об'єм кеш-пам'яті: 6 МБ.

Оперативна пам'ять:

- модель: Corsair Vengeance LPX DDR4-3200 16384MB PC4-25600;
- тип пам’яті: DIMM DDR4;
- частота пам’яті: 3200 МГц;
- пропускна здатність: 25600 МБ/с;
- ємність одного модуля оперативної пам'яті: 16 ГБ;
- кількість модулів оперативної пам'яті: 2;
- загальна ємність оперативної пам'яті: 32 ГБ.

Пам’ять диску:

- модель: Samsung 990 Pro 1TB;
- об’єм пам’яті: 512 ГБ;
- інтерфейс підключення: PCIe NVMe 4.0;
- швидкість читання: до 7450 МБ/с;
- швидкість запису: до 6900 МБ/с.

Відеокарта:

- модель: Intel HD Graphics 630;
- тип пам’яті: Shared (спільно з оперативною пам’яттю);

- інтерфейс: вбудована в процесор;
- обсяг пам’яті: 4 ГБ;
- частота роботи GPU: 300-1150 МГц.

4.1.2 Опис підходу для визначення часу роботи операцій взаємодіїї з базою даних

Оцінка часу виконання операцій у базах даних є ключовою складовою в даному етапі дослідження. Ми будемо заміряти час саме операцій до тестової бази даних, яка розподілена по ключу по трьом кластерам.

Перш за все, необхідно визначити основні метрики, які будуть використовуватися для оцінки продуктивності:

- час виконання запиту: час, що проходить від моменту надсилання запиту до отримання відповіді;
- пропускна здатність: кількість операцій, виконаних за одиницю часу;
- затримка: середній час очікування відповіді на операцію.

В даному досліді не враховується ще одна дуже важлива характеристика, як конкурентність – можливість обробляти декілка запитів одночасно. Для чистоти експерименту ми оминяємо цю метрику і робимо тільки один запит до бази в одиницю часу.

Щоб проаналізувати продуктивність різних СУБД, в нашому випадку це MongoDb та CouchDb, поділимо досліджувані операції на наступні типи та опишимо в загальному:

- операції вставки (Insert): додавання нових записів до бази.
- операції оновлення (Update): модифікація існуючих записів.
- операції читання (Read): вибірка, пошук, фільтрація та агрегація даних.

Далі змоделюємо тестові сценарії:

- сценарій вставки: послідовне додавання записів у базу з різними обсягами даних;
- сценарій оновлення: вибір конкретного запису за унікальним ідентифікатором і його модифікація;
- сценарій вибірки: вибірка по всім документам;

- сценарій пошуку: пошук за індексованими та неіндексованими полями;
- сценарій групування: агрегація записів за певними критеріями.

Для заміру часу виконання вищезгаданих операцій будемо використовувати вбудовані методи мови C#. Саме в ношому випадку будемо використовувати клас Stopwatch, який є частиною простору імен System.Diagnostics і використовується для точного вимірювання проміжків часу. Він забезпечує високоточний спосіб вимірювання тривалості виконання операцій або затримок. Він має основні методи, які і будемо реалізовувати для вимірювання часу:

- Start(): запускає або продовжує вимірювання часового інтервалу;
- Stop(): зупиняє вимірювання часу.

Цей клас було обрано тому, що він має високоточний таймер і мінімізує похибку в ході вимірювання.

4.2 Проведення експерименту

Далі проведемо дослідження часу виконання вищезгаданих операцій до тестових баз даних розміром сто тисяч документів та мільйон документів. Слід зазначити, що заміри часу виконувались для двох кейсів:

- без додавання індексів;
- з додавання індексів.

4.2.1 Проведення експерименту без додавання індексів

Спочатку проведемо експеримент з визначенням середнього часу виконання операцій зі 100 тисячами документами в MongoDb. Для мінімізації похибки було взято середній час виконання за 100 запусків. Для наступних експериментів будуть використовуватись аналогічні умови. Результати даного експерименту представлені в таблиці 4.1.

Таблиця 4.1 – Результати визначення середнього часу роботи операцій для 100 тисяч документів в MongoDb

Операція	Середній час роботи (мс)
Вставка (insert)	1388.29
Оновлення (update)	345.5
Фільтрація з декількома умовами (filter)	678.5
Пошук (find)	117.6
Групування (group)	1423.2
Вибірка (select all)	283.4

Далі проведемо експеримент з визначенням середнього часу виконання операцій зі 100 тисячами документами в CouchDb. Результати даного експерименту представлені в таблиці 4.2.

Таблиця 4.2 – Результати визначення середнього часу роботи операцій для 100 тисяч документів в CouchDb

Операція	Середній час роботи (мс)
Вставка (insert)	1412.3
Оновлення (update)	213.19
Фільтрація з декількома умовами (filter)	589.34
Пошук (find)	145.48
Групування (group)	127.76
Вибірка (select all)	443.6

Для наочності результатів зробимо одну порівняльну таблицю, де будуть результати для обох СУБД. Порівняння результатів наведено в таблиці 4.3.

Таблиця 4.3 – Порівняння результатів визначення середнього часу роботи операцій для 100 тисяч документів в CouchDb і MongoDb.

Операція	Середній час роботи (мс)	
	MongoDb	CouchDb
Вставка (insert)	1388.29	1412.3
Оновлення (update)	345.5	213.19
Фільтрація з декількома умовами (filter)	678.5	589.34
Пошук (find)	117.6	145.48
Групування (group)	1423.2	127.76
Вибірка (select all)	283.4	443.6

Наступним кроком буде проведення експерименту з визначення середнього часу виконання операцій до бази даних з мільйоном документів. Спочатку проведемо дослід з MongoDb. Результати даного експерименту представлені в таблиці 4.4.

Таблиця 4.4 – Результати визначення середнього часу роботи операцій для мільйона документів в MongoDb

Операція	Середній час роботи (мс)
Вставка (insert)	4380.7
Оновлення (update)	615.54
Фільтрація з декількома умовами (filter)	997.45
Пошук (find)	453.32
Групування (group)	1746.35
Вибірка (select all)	1262.47

Далі проведемо експеримент з визначенням середнього часу виконання операцій з мільйоном документів в CouchDb. Результати даного експерименту представлені в таблиці 4.5.

Таблиця 4.5 – Результати визначення середнього часу роботи операцій для мільйона документів в CouchDb

Операція	Середній час роботи (мс)
Вставка (insert)	4734.8
Оновлення (update)	357.39
Фільтрація з декількома умовами (filter)	887.12
Пошук (find)	430.04
Групування (group)	1535.56
Вибірка (select all)	1689.56

Для даного експерименту також наведемо порівняльну таблицю результатів середнього часу виконання операцій для обох СУБД. Результати наведені в таблиці 4.6.

Таблиця 4.6 – Порівняння результатів визначення середнього часу роботи операцій для мільйона документів в CouchDb і MongoDb.

Операція	Середній час роботи (мс)	
	MongoDb	CouchDb
Вставка (insert)	4380.7	4734.8
Оновлення (update)	615.54	357.39
Фільтрація з декількома умовами (filter)	997.45	887.12
Пошук (find)	453.32	430.04
Групування (group)	1746.35	1535.56
Вибірка (select all)	1262.47	1689.56

4.2.2 Проведення експерименту з додаванням індексів

Далі проведемо низку дослідів для визначення впливу індексів на результат виконання операцій. Спочатку проведемо експеримент з визначенням середнього часу виконання операцій зі 100 тисячами документами в MongoDb. Результати експерименту представлені в таблиці 4.7.

Таблиця 4.7 – Результати визначення середнього часу роботи операцій для 100 тисяч документів з використання індексів в MongoDb

Операція	Середній час роботи (мс)
Вставка (insert)	535.70
Оновлення (update)	411.23
Фільтрація з декількома умовами (filter)	925.81
Пошук (find)	18.68
Групування (group)	163.58
Вибірка (select all)	345.67

Наступним проводимо аналогічний експеримент для CouchDb. Результати експерименту наведені в таблиці 4.8.

Таблиця 4.8 – Результати визначення середнього часу роботи операцій для 100 тисяч документів з використання індексів в CouchDb

Операція	Середній час роботи (мс)
Вставка (insert)	472.25
Оновлення (update)	348.67
Фільтрація з декількома умовами (filter)	771.23
Пошук (find)	19.11
Групування (group)	137.34
Вибірка (select all)	409.56

Для даного експерименту знову ж таки наведемо порівняльну таблицю результатів середнього часу виконання операцій для обох СУБД. Результати наведені в таблиці 4.9.

Таблиця 4.9 – Порівняння результатів визначення середнього часу роботи операцій для 100 тисяч документів з використанням індексів в CouchDb і MongoDb

Операція	Середній час роботи (мс)	
	MongoDb	CouchDb
Вставка (insert)	535.70	472.25
Оновлення (update)	411.23	348.67
Фільтрація з декількома умовами (filter)	925.81	771.23
Пошук (find)	18.68	19.11
Групування (group)	163.58	137.34
Вибірка (select all)	345.67	409.56

Тепер виконаємо аналогічні дії для мільйона документів з використанням індексів. Результати для MongoDb представлені в таблиці 4.10.

Таблиця 4.10 – Результати визначення середнього часу роботи операцій для мільйона документів з використання індексів в MongoDb

Операція	Середній час роботи (мс)
Вставка (insert)	4676.1
Оновлення (update)	384.46
Фільтрація з декількома умовами (filter)	857.52
Пошук (find)	31.21
Групування (group)	1450.74
Вибірка (select all)	2440.36

Наступним проведемо експеримент з визначенням середнього часу виконання операцій з мільйоном документів з використанням індексів в CouchDb. Результати даного експерименту представлені в таблиці 4.11.

Таблиця 4.11 – Результати визначення середнього часу роботи операцій для мільйона документів з використання індексів в CouchDb

Операція	Середній час роботи (мс)
Вставка (insert)	4486.04
Оновлення (update)	539.19
Фільтрація з декількома умовами (filter)	959.84
Пошук (find)	16.23
Групування (group)	1603.23
Вибірка (select all)	2544.77

Для поточного етапу експерименту представимо порівняльну таблицю результатів середнього часу виконання операцій для обох СУБД. Результати наведені в таблиці 4.12.

Таблиця 4.12 – Порівняння результатів визначення середнього часу роботи операцій для мільйона документів з використанням індексів в CouchDb і MongoDb

Операція	Середній час роботи (мс)	
	MongoDb	CouchDb
Вставка (insert)	4676.1	4486.04
Оновлення (update)	384.46	539.19
Фільтрація з декількома умовами (filter)	857.52	959.84
Пошук (find)	31.21	16.23
Групування (group)	1450.74	1603.23
Вибірка (select all)	2440.36	2544.77

4.3 Результати експерименту

Тепер проаналізуємо результати, отримані у всих дослідах. Спочатку побудуємо порівняльні діаграми на основі порівняльних таблиць для кейсів без використання індексів.

Діаграми замірів часу операцій для 100 тисяч та мільйона документів представлено на рисунках 4.1 та 4.2 відповідно.

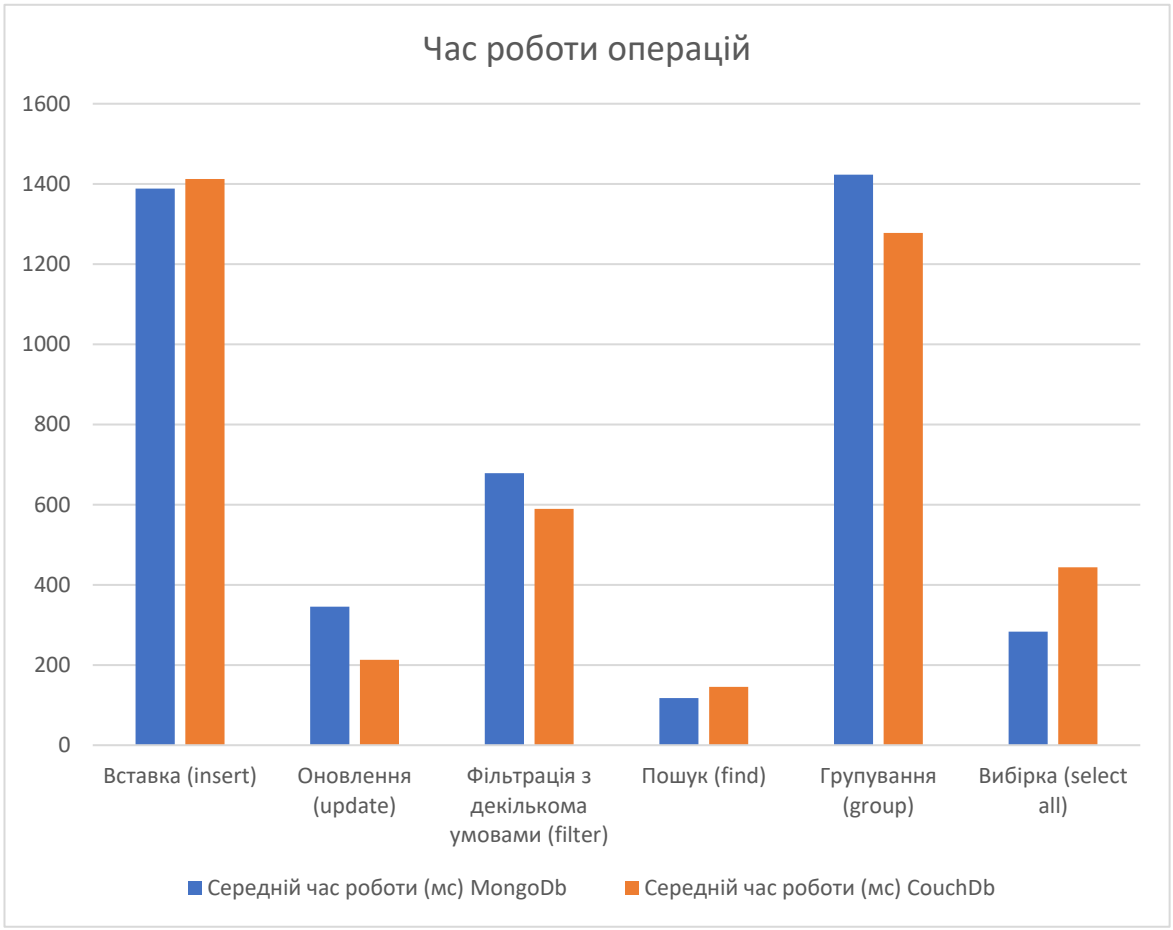

Рисунок 4.1 – Графік порівняння середнього часу роботи операцій для 100 тисяч документів

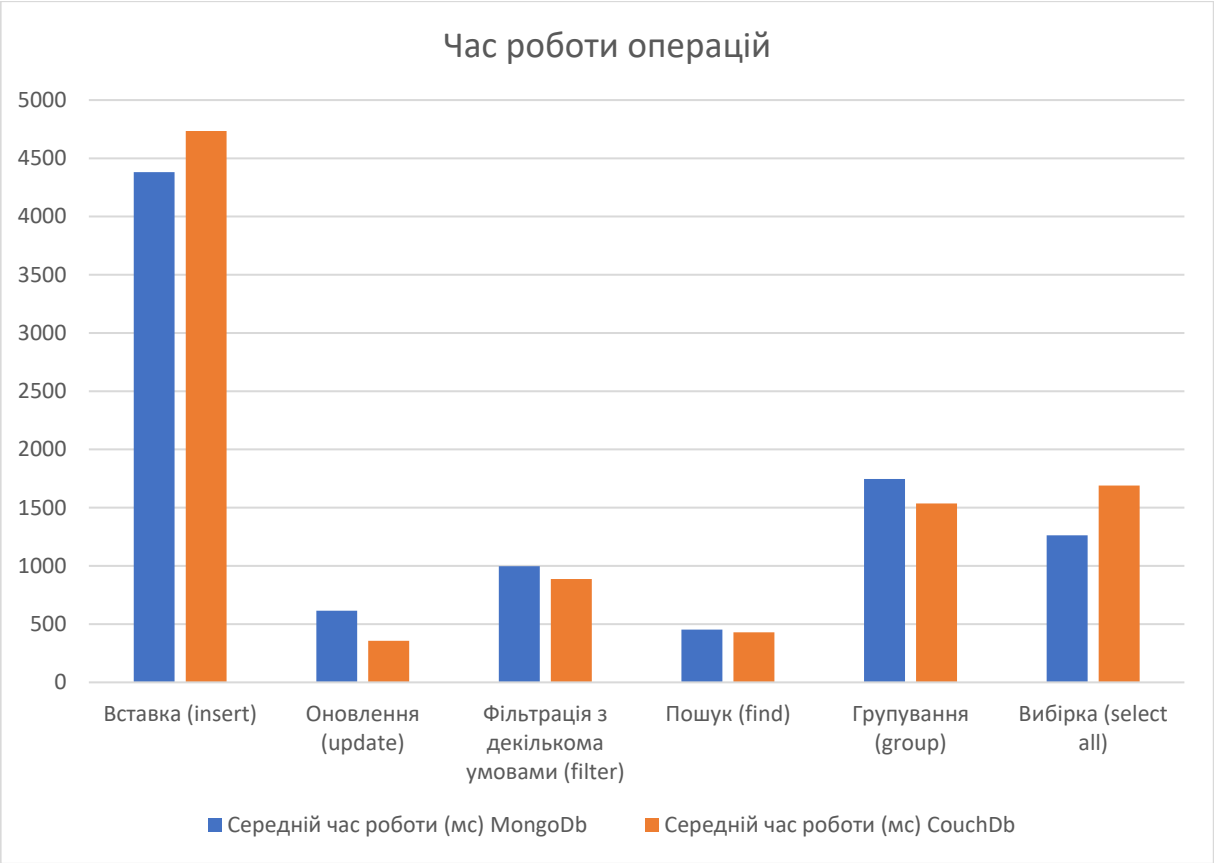

Рисунок 4.2 – Графік порівняння середнього часу роботи операцій для мільйона документів

З першого графіку видно, що MongoDb і CouchDb мають майже однаковий час вставки. Це свідчить про схожу ефективність обох баз даних у додаванні документів при середньому обсязі даних.

З приводу оновлення даних CouchDb демонструє значно швидший час (~200 мс) порівняно з MongoDb (~300 мс). Це є прямим наслідком механізму ревізій у CouchDb, який спрощує процес оновлення.

Також MongoDb потребує більше часу на фільтрацію за декількома умовами. CouchDb ефективніше обробляє фільтрацію завдяки попередньо створеним переглядам.

У пошуку також CouchDb трохи швидша, тому що механізм переглядів забезпечує стабільну продуктивність.

Групування відбувається довше у MongoDb (~1400 мс), ніж в CouchDb (~1200 мс). Це скоріше за все пов’язано з механізмом агрегації MongoDb, який вимагає значно більше ресурсів.

Вибірка всіх документів у MongoDb швидша, що може вказувати на ефективніше використання кешування.

З другого графіку маємо доволі схожу картину, порівняно з першим графіком. Операція вставки так само трохи швидша у MongoDb. Тобто можемо сказати, що данна СУБД краще масштабується для великих обсягів даних завдяки механізму «шардінгу».

Інші операції, такі як оновлення, фільтрація, групування та вибірка всіх документів залишаються без змін і для більшого набору даних. Тоді як пошук на великому обсягі даних став працювати швидше в MongoDb. Другим етапом проаналізуємо результати для випадку з використанням індексів. Для цього так само побудуємо порівняльні діаграми на основі порівняльних таблиць.

Другим етапом проаналізуємо результати для випадку з використанням індексів. Для цього так само побудуємо порівняльні діаграми на основі порівняльних таблиць. Діаграми замірів часу операцій
для 100 тисяч та мільйона документів представлено на рисунках 4.3 та 4.4 відповідно.

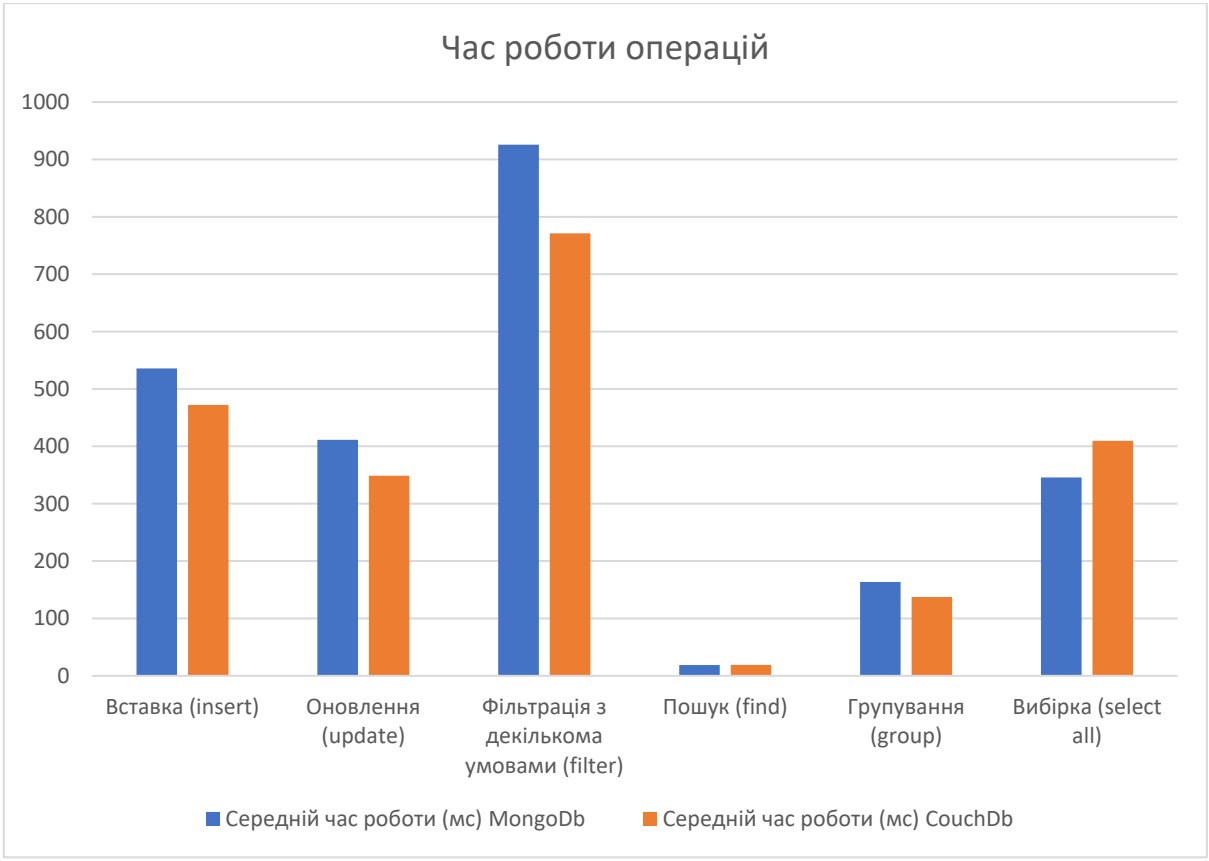

Рисунок 4.3 – Графік порівняння середнього часу роботи операцій для 100 тисяч документів

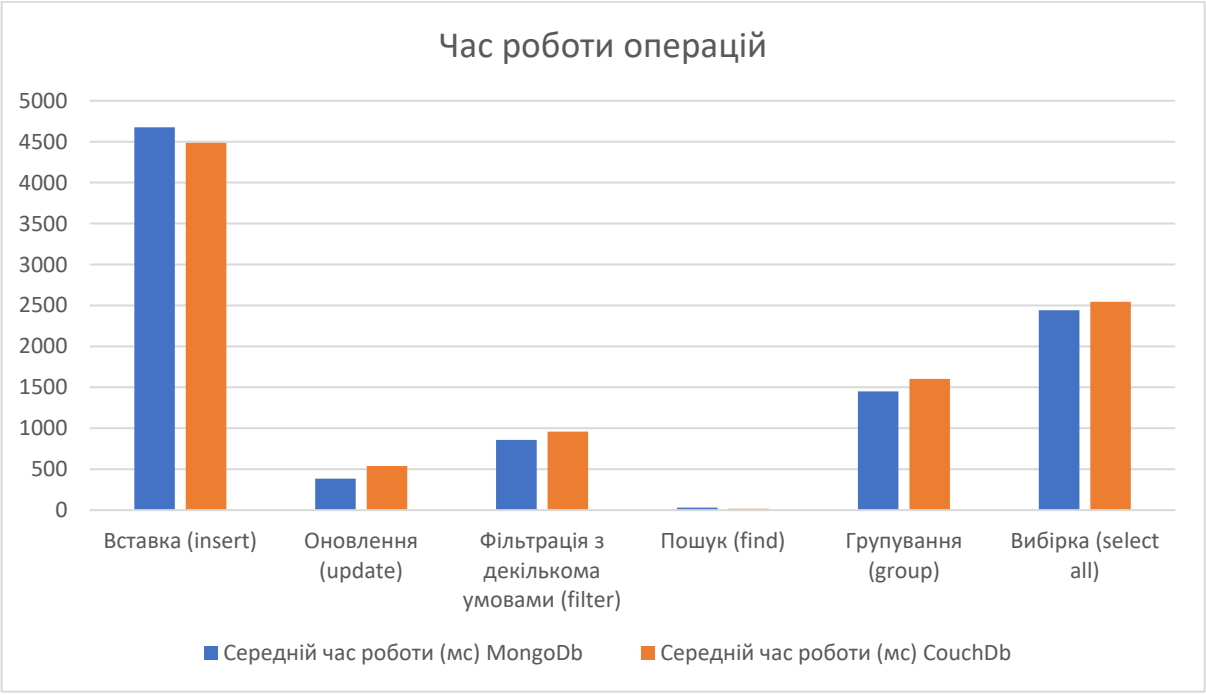

Рисунок 4.4 – Графік порівняння середнього часу роботи операцій для мільйона документів

З першого графіку, який показує нам ефективність роботи операцій до бази даних з використанням індексів, можна побачити, що MongoDb та CouchDb демонструють близький час виконання (близько 500 мс). Це свідчить про те, що додавання документів із підтримкою індексів впливає на продуктивність обох баз даних майже однаково.

Операція оновлення, в свою чергу, виконується швидше у CouchDb. Причиною цього є механізм ревізій, який спрощує оновлення існуючих записів.

З фільтрацією схожа картина і MongoDb поступається CouchDb. Це обумовлено тим, що у CouchDb використовуються попередньо створені перегляди, які зменшують час обробки.

Пошук на менших обсягах даних працює майже однаково для обох баз даних.

Для операції групування MongoDb демонструє більший час (~500 мс), ніж CouchDb (~400 мс). Це тому, що друга краще справляється з групуванням завдяки оптимізації механізму MapReduce.

MongoDb працює швидше, ніж CouchDb. Тут так, як і в попередньому випадку, без використання індексів, MongoDb має ефективніший механізм кешування для повної вибірки.

З другого графіку можемо бачити трохи іншу картину. З операцією вставки MongoDb впорується краще, ніж CouchDb. З цього можна зробити висновок, що перша краще масштабується на великих обсягах даних завдяки своєму механізму шардингу.

На операції оновлення CouchDb залишаєтья швидшою. Це підтверджує стабільність механізму ревізій.

Всі інші операції працюють швидше в MongoDb. Це свідчить про те, що вона є більш адаптованою на великих обсягах даних та те, що індекси дають вагому перевагу при великій кількості даних.

Висновки до розділу 4

У четвертому розділі було проведено низку експериментів з різними вхідними даними (з використанням індексів та без). Головною метою було зрозуміти

ефективність операцій до баз даних, а саме часову продуктивність. Тому було виконано заміри середнього часу виконання операцій зі ста запусків.

Після проведення експериментів та аналізу отриманої інформації було виокремлено наступні переваги та недоліки двох документоорієнтованих СУБД порівняно один з одним.

MongoDb краще масштабується для великих обсягів даних (1,000,000 документів) та ефективно обробляє фільтрацію, групування та вибірку всіх документів. Але тим часом повільніше обробляє оновлення порівняно з CouchDb.

CouchDb краще обробляє оновлення завдяки механізму ревізій. Швидше виконує пошук і групування на середніх обсягах даних (100,000 документів). Але гірше масштабується для великих наборів даних, особливо у вибірці всіх документів.

Тобто на основі отриманих даних можна зробити висновок, що MongoDb краще підходить для програмних систем, що працюють з великим набором даних, де потрібна масштабованість і ефективне сканування великих обсягів даних. А CouchDb краще для сценаріїв, де потрібно часто оновлювати дані та виконувати групування на середніх обсягах даних.

ЗАГАЛЬНИЙ ВИСНОВОК

Результатом данної дипломної роботи є визначення більш продуктивної документ-орієнтованої СУБД при роботі з розподіленими базами даними. Для дослідження було обрано MongoDb та CouchDb.

В ході виконання роботи було досліджено фунцкціональні можливості представлених баз даних. Було розглянуто основні поняття, які є базисними для документ-орієнтованих баз даних та було визначено основні відмінності між ними.

Також після проведення аналізу предметної області було обрано операції для дослідження часової ефективності, а саме:

- операція вставки нових документів в базу;
- операція оновлення документів в базі;
- операція вибірки всіх документів в базі;
- операція фільтрації документів з декількома умовами;
- операція пошуку документа в базі;
- операція групування документів в базі.

Було розглянуто зовнішнє і внутрішнє проектування програмного додатку для збору різних даних та проведення досліджень. Визначені основні аспекти розробки інтерфейсу користувача та розроблено інтуїтивно зрозумілий інтерфейс.

Для аналізу часової ефективності операцій до баз даних було розроблено програмний додаток, який дає змогу користувачам працювати з базою даних та виконувати заміри часу виконання відповідних операцій.

За допомогою створеного додатку було проведено порівняльний аналіз виконання часу виконання операцій з документ-орієнтованими базами даних. Заміри проводились в двох основних тестових сценаріях:

- без використання індексів;
- з використанням індексів.

На основі цих сценаріїв було проведено порівняння двох СУБД між собою та з’ясовано, яку з них краще використувати в конкретній ситуації, що має суттєво допомгти при виборі. Було зроблено висновок, що MongoDb краще підходить для

програмних систем, що працюють з великим набором даних, де потрібна масштабованість і ефективне сканування великих обсягів даних. А CouchDb краще для сценаріїв, де потрібно часто оновлювати дані та виконувати групування на середніх обсягах даних.

Також було розглянуто джерела інформації, що використовувались під час виконання роботи.

БІБЛІОГРАФІЧНИЙ СПИСОК

1. «Розподілена база даних» [Ел. ресурс]. Available: https://uk.wikipedia.org/wiki/. [Дата звернення: 01.12.2024].
2. «Реляційна база даних» [Ел. ресурс]. Available: https://uk.wikipedia.org/wiki/. [Дата звернення: 02.12.2024].
3. «Документ-орієнтована система керування базою даних» [Ел. ресурс]. Available: https://uk.wikipedia.org/wiki [Дата звернення: 02.12.2024].
4. «Теорема CAP» [Ел. ресурс]. Available: https://uk.wikipedia.org/wiki/%D0%A2%D0%B5%D0%BE%D1%80%D0%B5%D0%BC%D0%B0_CAP. [Дата звернення: 02.12.2024].
5. «Apache CouchDb» [Ел. ресурс]. Available: https://habr.com/ru/sandbox/12967/. [Дата звернення: 02.12.2024].
6. «MongoDb sharding» [Ел. ресурс]. Available: https://www.mongodb.com/docs/manual/sharding/. [Дата звернення: 02.12.2024].
7. «MongoDb sharded cluster» [Ел. ресурс]. Available: https://www.mongodb.com/docs/manual/sharding/#sharded-cluster. [Дата звернення: 03.12.2024].
8. «Mongodb shard keys» [Ел. ресурс]. Available: https://www.mongodb.com/docs/manual/sharding/#shard-keys. [Дата звернення: 03.12.2024].
9. «Природність інтерфейсу» [Ел. ресурс]. Available: https://wiki.cuspu.edu.ua/index.php/Природність_інтерфейсу. [Дата звернення: 03.12.2024].
10. «Проектування користувацького інтерфейсу» [Ел. ресурс]. Available: http://moodle.ipo.kpi.ua/moodle/mod/resource/view.php?id=39243. [Дата звернення: 03.12.2024].
11. «Проектирование графического интерфейса пользователя» [Ел. ресурс]. Available: https://habr.com/ru/post/208966/. [Дата звернення: 03.12.2024].

12. «Sequence diagram» [Ел. ресурс]. Available: https://en.wikipedia.org/wiki/Sequence_diagram. [Дата звернення: 03.12.2024].
13. «Краткий обзор C#» [Ел. ресурс]. Available: https://docs.microsoft.com/ru-ru/dotnet/csharp/tour-of-csharp/. [Дата звернення: 03.12.2024].
14. «Сборка мусора» [Ел. ресурс]. Available: https://docs.microsoft.com/ru-ru/dotnet/standard/garbage-collection/. [Дата звернення: 03.12.2024].
15. «Исключения и обработка исключений» [Ел. ресурс]. Available: https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/exceptions/. [Дата звернення: 03.12.2024].
16. «Лямбда-выражения» [Ел. ресурс]. Available: https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/lambda-expressions. [Дата звернення: 03.12.2024].
17. «Синтаксис LINQ» [Ел. ресурс]. Available: https://docs.microsoft.com/ru-ru/dotnet/csharp/linq/. [Дата звернення: 03.12.2024].
18. «Асинхронное программирование с использованием ключевых слов async и await» [Ел. ресурс]. Available: https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/concepts/async/. [Дата звернення: 03.12.2024].
19. «Сопоставление шаблонов» [Ел. ресурс]. Available: https://docs.microsoft.com/ru-ru/dotnet/csharp/pattern-matching/. [Дата звернення: 03.12.2024].
20. «Типы» [Ел. ресурс]. Available: https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/types/. [Дата звернення: 03.12.2024].
21. «Система управления версиями» [Ел. ресурс]. Available: https://ru.wikipedia.org/wiki/Система_управления_версиями. [Дата звернення: 03.12.2024].
22. «About: Branching and Merging» [Ел. ресурс]. Available: https://git-scm.com/about/branching-and-merging. [Дата звернення: 03.12.2024].
23. «About: Small and Fast» [Ел. ресурс]. Available: https://git-scm.com/about/small-and-fast. [Дата звернення: 03.12.2024].

24. «YAGNI principle» [Ел. ресурс]. Available: https://whatis.techtarget.com/definition/You-arent-gonna-need-it. [Дата звернення: 04.12.2024].
25. «Don’t repeat yourself» [Ел. ресурс]. Available: https://deviq.com/don-t-repeat-yourself/. [Дата звернення: 04.12.2024].
26. «DRY or WET and Why?» [Ел. ресурс]. Available: https://medium.com/@nrk25693/dry-or-wet-and-why-867ac3096483. [Дата звернення: 04.12.2024].
27. «Концепция: Стратегия тестирования» [Ел. ресурс]. Available: http://dit.isuct.ru/Publish_RUP/core.base_rup/guidances/concepts/test_strategy_9981F03E.html. [Дата звернення: 04.12.2024].
28. «Тестирование по стратегии чёрного ящика» [Ел. ресурс]. Available: https://ru.wikipedia.org/wiki/Тестирование_по_стратегии_чёрного_ящика. [Дата звернення: 04.12.2024].
29. «Тестирование по белого ящика» [Ел. ресурс]. Available: https://ru.wikipedia.org/wiki/Тестирование_белого_ящика. [Дата звернення: 04.12.2024].
30. «View the call stack and use the Call Stack window in the debugger» [Ел. ресурс]. Available: https://docs.microsoft.com/en-us/visualstudio/debugger/how-to-use-the-call-stack-window?view=vs-2019. [Дата звернення: 04.12.2024].
31. «Use breakpoints in the Visual Studio debugger» [Ел. ресурс]. Available: https://docs.microsoft.com/en-us/visualstudio/debugger/using-breakpoints?view=vs-2019. [Дата звернення: 04.12.2024].
32. «Stopwatch Class» [Ел. ресурс]. Available: https://docs.microsoft.com/en-us/dotnet/api/system.diagnostics.stopwatch?view=net-5.0. [Дата звернення: 05.12.2024].

ДОДАТОК А
Технічне завдання

ЗАТВЕРДЖУЮ
Проректор Українського державного
університету науки і технологій
Анатолій РАДКЕВИЧ

СИСТЕМА ДЛЯ ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ РОБОТИ РІЗНИХ СУБД ПРИ РОБОТІ З РОЗПОДІЛЕНИМИ НЕРЕЛЯЦІЙНИМИ БАЗАМИ ДАНИХ

Технічне завдання
1116130.01193-01

Завідувач кафедри КІТ
_____Вадим ГОРЯЧКІН
Керівник розробки
_____Олександр ІВАНОВ
Виконавець
_____Сергій ДУДАР
Нормоконтролер
_____Світлана ВОЛКОВА

2025

ЗАТВЕРДЖЕНО
1116130.01193-01

СИСТЕМА ДЛЯ ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ РОБОТИ РІЗНИХ СУБД ПРИ РОБОТІ З РОЗПОДІЛЕНИМИ НЕРЕЛЯЦІЙНИМИ БАЗАМИ ДАНИХ

Технічне завдання
1116130.01193-01

Листів 13

2025

АНОТАЦІЯ

Документ 1116130.01193-01 «Система для дослідження ефективності роботи різних СУБД при роботі з розподіленими базами даних» Технічне завдання входить до складу програмної документації до дипломного проекту.

У даному документі представлене призначення та область застосування програмного продукту, основні вимоги, стадії та строки виконання проекту, техніко-економічні показники, що пред’являються до програмного продукту.

ЗМІСТ

ВСТУП

У сфері інформаційних технологій дуже часто виникає необхідність використовувати бази даних для вирішення різноманітних задач пов’язаних зі зберіганням інформації. Часто доводиться зберігати її у різних місцях програмної системи або комп’ютерної мережі, тобто треба мати справу з розподіленими базами даних. Але водночас з цим постає питання, як правильно представити та зберігати великі об’єми даних. Тобто яку СУБД краще використовувати: SQL або NoSQL?

Щоби відповісти на ці питання необхідно для початку з’ясувати переваги нереляційних баз даних над реляційними, та який тип краще використовувати для конкретних потреб. Приведемо невелике порівняння даних двох типів зберігання інформації:

- NoSQL легко справляється з дуже великими обсягами даних;
- Як правило, нереляційні СУБД швидше виконують операції запису (швидкість операцій читання залежить від типу бази даних NoSQL і типу запитуваних даних);
- NoSQL дуже гнучка (в порівнянні з реляційними СУБД, яким для початку необхідна структура);
- NoSQL пропонує автоматизовану реплікацію і масштабування. Сьогодні нереляційні СУБД активно розвиваються і усувають загальні проблеми в роботі з даними, серед яких реплікація і масштабування – одні з найважливіших;
- NoSQL легко масштабується і працює в кластері;
- Також NoSQL має широкий вибір програм і моделей для роботи з різними типами даних.

В останнє десятиріччя все більше стають популярними та широко застосованими нереляційні бази даних, а саме СУБД для роботи з документорієнтованими базами даних. Тому в контексті даної роботи необхідно дослідити доцільність використання такого типу нереляційних баз даних.

Для початку потрібно розібратись зі сферою використання такого типу нереляційних БД. Документна база даних – гарний вибір для додатків управління

контентом, таких як платформи для блогів і розміщення відео. При використанні документної бази даних кожна сутність, яка відстежується додатком, може зберігатися як окремий документ. Документна база даних дозволяє розробнику з зручністю оновлювати додаток при зміні вимог. Крім того, якщо необхідно змінити модель даних, то потрібно оновлення тільки тих документів, яких стосується дана зміна. Також такі БД підходять для зберігання каталожної інформації. Тобто їх дуже доцільно використовувати для інтернет-комерції.

І тут постає питання: яку СУБД використовувати для реалізації своїх потреб? Для відповіді на це питання необхідно порівняти продуктивність роботи документних СУБД. Зазвичай це робиться методом заміру часу виконання основних операцій при роботі з базами даних. Операції можуть бути такими:

- вставка даних;
- вибірка даних;
- пошук;
- оновлення;
- фільтрація за декількома умовами;
- групування.

Для великих проектів необхідна велика кількість інформації, тому об’єми баз даних будуть достатньо великими. З цього слідує те, що час виконання основних операцій обробки даних дуже важливий для продуктивності роботи.

1 ПІДСТАВА ДО РОЗРОБКИ

Підставою для розробки даного програмного продукту є наказ №779 ст. «Про призначення керівників та затвердження тем магістерських робіт» від 10.10.2019 р., затверджений ректором Дніпровського національного університету залізничного транспорту імені академіка В. Лазаряна.

Тема: Система для дослідження ефективності роботи різних СУБД при роботі з розподіленими базами даних.

Керівник дипломного проекту – доц. Іванов О.П.

2 ПРИЗНАЧЕННЯ РОЗРОБКИ

2.1 Функціональне призначення

За допомогою даного продукту користувач матиме змогу проводити експерименти для оцінки часової ефективності таких операцій роботи з даними: вставка, вибірка, пошук, сортування, групування. Використовуючи різні комбінації вхідних даних, він зможе збирати результати експериментів для подальшого аналізу.

2.2Експлуатаційне призначення

Програмний продукт призначений для забезпечення швидкого збору даних щодо часу роботи вищезгаданих операцій до баз даних в різноманітних умовах.

3 ВИМОГИ ДО ПРОГРАМИ

3.1 Вимоги до функціональних характеристик

Програма повинна:

- мати можливість зчитувати та записувати дані в базу даних;
- можливість роботи запити вибірки та фільтрації з різними умовами;
- мати можливість працювати з індексами: дослідження можливостей створення та використання індексів для оптимізації роботи з даними та запитами;
- надавати можливість ввести параметри експерименту (тип операції по варіанту бази даних).

Вхідні дані:

- тип операції (текст);
- кількість тестів (ціле число);

Вихідні дані:

- візуальне відображення часу виконання операції;
- файл формату .CSV, який містить результати експерименту та має такі дані:
    a) Час роботи операції;
    b) Використаний тип операції.

3.2 Вимоги до надійності

Одним із критеріїв правильного функціонування програмного продукту є забезпечення надійності роботи програмного продукту. Вимоги до надійності програмного продукту повинні відповідати наступним вимогам:

- програма не повинна допускати невимушену втрату та пошкодження даних;
- кількість відмов системи не повинна перевищувати однієї відмови на 2000 запусків системи (під відмовою слід вважати непрацездатність системи після її запуску, тобто необхідність запустити систему повторно).

3.3 Умови експлуатації

Для нормального функціонування програмного продукту необхідно виконання наступних вимог:

- ЕОМ повинні відповідати вимогам чинних в Україні стандартів, нормативних актів з охорони праці;
- програмний комплекс повинен використовуватись в приміщеннях, призначених для роботи ЕОМ з наступними кліматичними умовами температура – 21-25 °С, відносна вологість повітря 40-60%;
- користувач повинен бути ознайомлений з керівництвом користувача.

3.4 Вимоги до складу та параметрів технічних засобів

Для коректного функціонування програмного продукту вимагається наявність ЕОМ, що задовольняє нормальну роботу:

Мінімальна конфігурація комп’ютера для забезпечення роботи програмного продукту:

- ОС Windows 10;
- ОЗП не менш ніж 2048 МБ;
- вільний простір пам’яті не менше 300 МБ.

3.5 Вимоги до інформаційної та програмної сумісності

Для функціонування програмного продукту необхідна ОС Windows 10 або вище.

4 ВИМОГИ ДО ПРОГРАМНОЇ ДОКУМЕНТАЦІЇ

До складу документації мають входити:

- текст програми;
- керівництво користувача для користування мобільним додатком.

Вся документація програмних додатків повинна задовольняти вимоги до програмної документації.

5 СТАДІЇ ТА ЕТАПИ РОЗРОБКИ

Стадії та етапи розробки програми представлені у табл. 5.1.

Таблиця 5.1 – Стадії та етапи розробки

Стадії розробки	Етапи розробки	Терміни виконання
1.Технічне завдання (ТЗ)	Постановка задачі	12.10.2024 – 13.10.2024
	Огляд літератури та аналіз аналогів	14.10.2024 – 17.10.2024
	Розробка структур вхідних і вихідних даних	19.10.2024 – 23.10.2024
	Визначення вимог до програми. Вибір та обґрунтування мови програмування	26.10.2024 – 30.10.2024
	Узгодження та затвердження ТЗ	02.11.2024 – 09.11.2024
2. Робочий проект	Розробка та програмування логіки програми	09.11.2024 – 25.11.2024
	Розробка і реалізація інтерфейсу користувача	25.11.2024 – 28.11.2024
	Відлагодження програми	28.11.2024 – 30.11.2024
	Розробка, узгодження та затвердження програмної документації	05.12.2024 – 12.12.2024
3. Впровадження	Підготовка і передача програми та програмної документації замовнику	15.12.2025 – 04.01.2025

6 ПОРЯДОК КОНТРОЛЮ ТА ПРИЙМАННЯ

Контроль здійснюється за допомогою виконання набору тестів з метою знаходження помилок в програмі та його специфікації. Контроль виконання роботи забезпечується керівником розробки.

Прийом програми здійснюється уповноваженою комісією.

7 ТЕХНІЧНО-ЕКОНОМІЧНІ ПОКАЗНИКИ

Показники та їх розрахунок не були описані у документах бо розробка програмного продукту несе в собі навчальний характер, а не комерційний.

ДОДАТОК Б

Текст програми

ЗАТВЕРДЖУЮ
Проректор Українського державного
університету науки і технологій
Анатолій РАДКЕВИЧ

СИСТЕМА ДЛЯ ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ РОБОТИ РІЗНИХ СУБД ПРИ РОБОТІ З РОЗПОДІЛЕНИМИ НЕРЕЛЯЦІЙНИМИ БАЗАМИ ДАНИХ

Текст програми
1116130.01193-01 12 01

Завідувач кафедри КІТ
_____Вадим ГОРЯЧКІН
Керівник розробки
_____Олександр ІВАНОВ
Виконавець
_____Сергій ДУДАР
Нормоконтролер
_____Світлана ВОЛКОВА

2025

ЗАТВЕРДЖЕНО
1116130.01193-01 12 01

СИСТЕМА ДЛЯ ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ РОБОТИ РІЗНИХ СУБД ПРИ РОБОТІ З РОЗПОДІЛЕНИМИ НЕРЕЛЯЦІЙНИМИ БАЗАМИ ДАНИХ

Текст програми
1116130.01193-01 12 01
Листів 11

2025

АНОТАЦІЯ

Документ 1116130.01193-01 12 01 «Система для дослідження ефективності роботи різних СУБД при роботі з розподіленими нереляційними базами даних.» Текст програми» входить до складу програмної документації до дипломного проекту. Даний документ містить структуру програми та текст програмного коду.

ЗМІСТ

1 ТЕКСТ ПРОГРАМИ

```
using System.Data;
using System;
using DotNetNuke.Common.Utilities;
using DotNetNuke.Framework.Providers;
namespace Christoc.Modules.CompanyInformationModule.Data
{

    /// -----------------------------------------------------------------------
    /// <summary>
    /// An abstract class for the data access layer
    ///
    /// The abstract data provider provides the methods that a control data provider (sqldataprovider)
    /// must implement. You'll find two commented out examples in the Abstract methods region below.
    /// </summary>
    /// -----------------------------------------------------------------------
    public abstract class DataProvider
    {

        #region Shared/Static Methods

        private static DataProvider provider;

        // return the provider
        public static DataProvider Instance()
        {
            if (provider == null)
            {
                const string assembly =
"Christoc.Modules.CompanyInformationModule.Data.SqlDataprovider,CompanyInformationModule";
                Type objectType = Type.GetType(assembly, true, true);

                provider = (DataProvider)Activator.CreateInstance(objectType);
                DataCache.SetCache(objectType.FullName, provider);
            }

            return provider;
        }

        [System.Diagnostics.CodeAnalysis.SuppressMessage("Microsoft.Design", "CA1024:UsePropertiesWhereAppropriate",
Justification = "Not returning class state information")]
        public static IDbConnection GetConnection()
        {
            const string providerType = "data";
            ProviderConfiguration _providerConfiguration = ProviderConfiguration.GetProviderConfiguration(providerType);

            Provider objProvider = ((Provider)_providerConfiguration.Providers[_providerConfiguration.DefaultProvider]);
            string _connectionString;
            if (!String.IsNullOrEmpty(objProvider.Attributes["connectionStringName"]) &&
!String.IsNullOrEmpty(System.Configuration.ConfigurationManager.AppSettings[objProvider.Attributes["connectionStringN
ame"]]))
            {
                _connectionString =
System.Configuration.ConfigurationManager.AppSettings[objProvider.Attributes["connectionStringName"]];
            }
            else
            {
                _connectionString = objProvider.Attributes["connectionString"];
            }
```

```
            IDbConnection newConnection = new System.Data.SqlClient.SqlConnection();
            newConnection.ConnectionString = _connectionString.ToString();
            newConnection.Open();
            return newConnection;
        }

        #endregion

        #region Abstract methods

        //public abstract IDataReader GetItems(int userId, int portalId);

        //public abstract IDataReader GetItem(int itemId);


        #endregion

    }

}

using DotNetNuke.Entities.Users;
using SL.Controller;
using SL.Controller.Associations;
using SL.Controller.eCommerce;
using SL.Domain;
using SL.Domain.Associations;
using SL.Domain.eCommerce.PaymentGateway;
using SL.Domain.Enums;
using SL.Services.Logic;
using SL.Utility;
using SL.ViewModels;
using SL.ViewModels.Association;
using System;
using System.Collections.Generic;
using System.Linq;

namespace Christoc.Modules.CompanyInformationModule.Components
{
    public class CompanyInformationController : ControllerBase
    {
        private readonly CompanyController companyController;
        private readonly UserCompanyController userCompanyController;
        private readonly SessionController sessionController;
        private readonly AssociationController associationController;
        private readonly AuthorizeDotNetServiceController authDotNetController;
        private readonly ProvisionController provisionController;
        private readonly OrderProcessController orderController;
        private readonly BillingInfoController billingController;
        private readonly ARBController arbController;
        private readonly CancelledSLIQUsersController cancelledSLIQUsersController;

        public CompanyInformationController()
        {
            base.EnsureAllContexts();
            companyController = new CompanyController(base.SLContext, base.OtasContext);
            userCompanyController = new UserCompanyController(base.SLContext, base.OtasContext);
            sessionController = new SessionController();
            associationController = new AssociationController(base.SLContext);
            authDotNetController = new AuthorizeDotNetServiceController();
            provisionController = new ProvisionController(base.SLContext, base.OtasContext);
            orderController = new OrderProcessController(base.SLContext, base.OtasContext);
```

```
    billingController = new BillingInfoController(base.SLContext);
    arbController = new ARBController(base.SLContext);
    cancelledSLIQUsersController = new CancelledSLIQUsersController(base.SLContext);
}

public void UpdateCompany(Company company)
{
    companyController.UpdateCompany(company);
}

public Company GetCompanyByUserId(int userId)
{
    return userCompanyController.GetCompanyByUserId(userId);
}

public Company GetCompanyById(long companyId)
{
    return companyController.GetCompanyById(companyId);
}

public UserCompany GetUserCompanyByUserId(int userId, long companyId)
{
    return userCompanyController.GetUserCompany(userId, companyId);
}

public Company GetCompany(int userId)
{
    return userCompanyController.GetCompanyByUserId(userId);
}

public bool IsUserCompanyAdmin(int userId, long companyId)
{
    return userCompanyController.IsUserCompanyAdmin(userId, companyId);
}

public bool IsSLAdmin(UserInfo user)
{
    return user.IsSLAdmin();
}

public List<KeyValuePair<string, int>> GetAssociationsToAdd(long companyId)
{
    return associationController.GetAssociationsToAdd(companyId);
}

public List<CompanyAssociationVM> GetAssociationsByCompany(long companyId)
{
    return associationController.GetCompanyAssociationsByCompany(companyId);
}

public void AddCompanyAssociation(int associationId, long companyId)
{
    associationController.AddCompanyAssociation(companyId, associationId, false);
}

public void DeleteCompanyAssociation(int companyAssociationId)
{
    associationController.DeleteCompanyAssociation(companyAssociationId);
}

public void UpdateCompanyAssociationGroupNumber(List<KeyValuePair<int, string>> updates)
{
```

```
        associationController.UpdateCompanyAssociationGroupNumber(updates);
    }

    public void UpdateCompanyAssociationGroupNumber(int companyAssociationId, string newGroupNumber)
    {
        associationController.UpdateCompanyAssociationGroupNumber(companyAssociationId, newGroupNumber);
    }

    public List<KeyValuePair<int, string>> GetAssociationGroups(int associationId)
    {
        return associationController.GetAssociationGroups(associationId);
    }

    public CompanyAssociation GetCompanyAssociation(int companyAssociationId)
    {
        return associationController.GetCompanyAssociation(companyAssociationId);
    }

    public List<CompanySubscriptionViewModel> GetCompanySubscriptions(long companyId)
    {
        var subscriptionViewModels = new List<CompanySubscriptionViewModel>();
        var companyProvisions = provisionController.GetCompanyProvisions(companyId);
        var arbSubscriptions = companyProvisions.SelectMany(p => p.Subscriptions).ToList();
        var orders = companyProvisions.SelectMany(p => p.Orders).ToList();


        arbSubscriptions.ForEach(subscription =>
        {
            var order = orders.FirstOrDefault(p => p.OrderNumber == subscription.OrderNumber);
            var vm = new CompanySubscriptionViewModel();
            vm.OrderId = subscription.OrderNumber;
            vm.SubscriptionId = subscription.SubscriptionId;
            if (order != null)
            {
                vm.OrderDate = order.OrderDate;
                vm.ProductId = order.ProductId;
                vm.ProductName = order.ProductName;
            }
            vm.CanCancel = subscription.CompanyProvisionAccess.ProvisionAccessEnum.IsActive();
            vm.CanViewBilling = subscription.Status != ARBStatus.Cancelled && subscription.Status !=
ARBStatus.Terminated && subscription.Status != ARBStatus.Expired;
            vm.NextBillDate = subscription.NextChargeDate;
            vm.PendingCancellationDate = subscription.PendingCancellationDate;
            vm.ProvisionAccess = subscription.CompanyProvisionAccess.ProvisionAccessEnum;
            vm.AuthorizeStatus = subscription.Status;
            vm.CompanyId = companyId;
            vm.ProvisionFeature = subscription.CompanyProvisionAccess.ProvisionFeature;
            vm.ProvisionAccess = subscription.CompanyProvisionAccess.ProvisionAccessEnum;
            if (vm.ProvisionFeature == ProvisionFeature.SLIQ)
            {
                vm.ProvisionAccessDisplay = subscription.CompanyProvisionAccess.ProvisionAccessEnum.IsActive()
                    ? String.Format("{0} ({1})", subscription.CompanyProvisionAccess.ProvisionAccessEnum.ToDescription()
                    , subscription.CompanyProvisionAccess.Company.MaxSLIQUsers)
                    : subscription.CompanyProvisionAccess.ProvisionAccessEnum.ToDescription();
            }
            else if (vm.ProvisionFeature == ProvisionFeature.FiveUserPack)
            {
                vm.ProvisionAccessDisplay = subscription.CompanyProvisionAccess.ProvisionAccessEnum.ToDescription();
            }
            else
            {
                vm.ProvisionAccessDisplay = subscription.CompanyProvisionAccess.ProvisionAccessEnum.ToDescription();
```

```
            }

            vm.CardType = subscription.ActivePaymentInformation.CreditCardType;
            vm.CardNumber = subscription.ActivePaymentInformation.LastFourDigits;
            vm.ExpirationNoticeDate = subscription.ActivePaymentInformation.ExpirationNotificationDate;
            vm.BillingAddress = String.Format("{0}, {1}, {2}, {3}", subscription.ActivePaymentInformation.Address,
subscription.ActivePaymentInformation.City
                , subscription.ActivePaymentInformation.State, subscription.ActivePaymentInformation.Zip,
subscription.ActivePaymentInformation.Country);
            vm.BillingName = String.Format("{0} {1}", subscription.ActivePaymentInformation.FirstName,
subscription.ActivePaymentInformation.LastName);
            vm.LastUpdatedDate = subscription.ActivePaymentInformation.LastModifiedOn;

            var billingInfo = billingController.GetCurrentBillingInfo(vm.SubscriptionId);

            vm.BillingEmail = billingInfo.Email;

            subscriptionViewModels.Add(vm);
        });
        return subscriptionViewModels;
    }
    internal void MarkSubscriptionforCancellation(string[] hdnAccepts, Company company, ARBSubscription sub)
    {
        cancelledSLIQUsersController.MarkSubscriptionforCancellation(hdnAccepts, company, sub);
    }
    public void UpdateSubscription(ARBSubscription sub)
    {
        arbController.UpdateARBSubscription(sub);
    }

    public ARBSubscription GetCompanyProvision(int subscriptionId)
    {
        return arbController.GetArbSubscription(subscriptionId);
    }

    public void CancelSubscription(int orderId, Company company)
    {
        orderController.ProcessOrderCancellation(orderId, company);
    }

    public void ChangeSubscriptionPendingCancellationDate(int subscriptionId, UserInfo userInfo)
    {
        try
        {
            using (var logic = new SubscriptionServiceLogic())
            {
                logic.ChangeSubscriptionPendingCancellationDate(subscriptionId, userInfo);
            }
        }
        catch (Exception ex)
        {
            throw new Exception(ex.Message);
        }
    }

    public List<CompanyOrderViewModel> GetCompanyOrders(long companyId)
    {
        var orders = orderController.GetOrders(companyId);


        return orders;
    }
```

```
        public void UpdateCanceledSLIQUsers(List<CancelledSLIQUsers> cancelledSLIQAccesUsers) {
            cancelledSLIQUsersController.UpdateCancelledSLIQUsers(cancelledSLIQAccesUsers);
        }

        public List<CancelledSLIQUsers> GetCanceledSLIQUsersByCompanyId(long companyId)
        {
            return cancelledSLIQUsersController.GetCancelledSLIQUsersByCompanyId(companyId);
        }
        internal int GetNoofUsersToCancell(Company company)
        {
            return cancelledSLIQUsersController.GetUsersCountToMarkCancel(company);
        }
    }
}

using System;
using System.Collections.Generic;
using System.Data;
using System.Linq;
using System.Text;
using System.Text.RegularExpressions;
using System.Web.UI;
using System.Web.UI.HtmlControls;
using System.Web.UI.WebControls;

namespace Christoc.Modules.CompanyInformationModule
{

    public partial class View : CompanyInformationModuleModuleBase, IActionable
    {
        public Company Company { get; set; }
        public bool IsCompanyAdmin { get; set; }
        public bool IsSLAdmin { get; set; }

        protected void Page_Load(object sender, EventArgs e)
        {
            try
            {
                using (var controller = new CompanyInformationController())
                {
                    LoadCompanyAndPermissions(controller);
                    if (Company != null)
                    {
                        BindNavigateURLs();
                        string qs = Request.QueryString[Constants.QueryString.Page] != null ?
Request.QueryString[Constants.QueryString.Page].ToString() : string.Empty;
                        BindSelectedTabs(controller, qs);

                        if (!IsPostBack && Request.QueryString[Constants.QueryString.Success] != null &&
Request.QueryString[Constants.QueryString.Success].ToString() == "1")
                        {

                            if (Request.QueryString[Constants.QueryString.EmailOnly] != null &&
Request.QueryString[Constants.QueryString.EmailOnly].ToString() == "1")
                            {
                                new FlashMessage().Success(this, "Successfully updated your email address.");
                            }
                            else
                            {
                                var message = Request.QueryString[Constants.QueryString.Message] != null ?
Request.QueryString[Constants.QueryString.Message] : String.Empty;
```

```
                        new FlashMessage().Success(this, "Successfully updated your billing information. " + message);
                    }
                }
            }
            else
            {
                CompanyInformationContainer.Visible = false;
                new FlashMessage().Info(this, "You are not currently assigned to a company.");
            }
        }
    }
    catch (Exception exc) //Module failed to load
    {
        Exceptions.ProcessModuleLoadException(this, exc);
    }
}

public void BindNavigateURLs()
{
    var urlHelper = new UrlHelper();
    tbInformation.NavigateUrl = String.Format("{0}?{1}={2}&{3}={4}",
urlHelper.BuildPageUrl(Constants.Pages.CompanyInformationPage)
        , Constants.QueryString.CompanyId, Company.Id, Constants.QueryString.Page,
CompanyInformationPageEnum.Information.ToString());
    tbSubscriptions.NavigateUrl = String.Format("{0}?{1}={2}&{3}={4}",
urlHelper.BuildPageUrl(Constants.Pages.CompanyInformationPage)
        , Constants.QueryString.CompanyId, Company.Id, Constants.QueryString.Page,
CompanyInformationPageEnum.Billing.ToString());
    tbAssociations.NavigateUrl = String.Format("{0}?{1}={2}&{3}={4}",
urlHelper.BuildPageUrl(Constants.Pages.CompanyInformationPage)
        , Constants.QueryString.CompanyId, Company.Id, Constants.QueryString.Page,
CompanyInformationPageEnum.Associations.ToString());
}

public void BindSelectedTabs(CompanyInformationController controller, string page)
{
    if (page == CompanyInformationPageEnum.Billing.ToString())
    {
        if (!IsCompanyAdmin && !IsSLAdmin) return;
        CheckForCancel(controller);
        LoadSubscriptions(controller);
        BindSLIQUsersToCancel(controller);
        LoadOrders(controller);
        tbSubscriptions.Selected = true;
        SubscriptionsPageView.Selected = true;
    }
    else if (page == CompanyInformationPageEnum.Associations.ToString())
    {
        if (!IsSLAdmin) return;
        LoadAssociations(controller);
        tbAssociations.Selected = true;
        AssociationsPageView.Selected = true;
    }
    else
    {
        LoadCompanyInfo(controller);
        tbInformation.Selected = true;
        InformationPageView.Selected = true;
    }
}
```

```
private long SelectDocuments()
{
    var reviewsCollection = db.GetCollection<BsonDocument>("Reviews");
    var sw = new Stopwatch();

    sw.Start();
    var allDocuments = reviewsCollection.Find(new BsonDocument()).ToList();
    sw.Stop();
    var executionTime = sw.ElapsedMilliseconds;
    return executionTime;
}

private long GroupingDocuments()
{
    var reviewsCollection = db.GetCollection<BsonDocument>("Reviews");
    var sw = new Stopwatch();

    sw.Start();
    var group = reviewsCollection.Aggregate().Group(new BsonDocument { { "_id", "accommodates" } }).ToList();
    sw.Stop();
    var executionTime = sw.ElapsedMilliseconds;
    return executionTime;
}

private long FilteringDocuments()
{
    var reviewsCollection = db.GetCollection<BsonDocument>("Reviews");
    var sw = new Stopwatch();

    sw.Start();
    var filter = new BsonDocument("price", new BsonDocument("$gt", 100));
    var result = reviewsCollection.Find(filter).ToList();
    sw.Stop();
    var executionTime = sw.ElapsedMilliseconds;
    return executionTime;
}

private long SortingDocuments()
{
    var reviewsCollection = db.GetCollection<BsonDocument>("Reviews");
    var sw = new Stopwatch();

    sw.Start();
    var sortedCollection = reviewsCollection.Find(new BsonDocument()).Sort("{price:1}").ToList();
    sw.Stop();
    var executionTime = sw.ElapsedMilliseconds;
    return executionTime;
```

ДОДАТОК В
Керівництво користувача

ЗАТВЕРДЖУЮ
Проректор Українського державного
університету науки і технологій
Анатолій РАДКЕВИЧ

СИСТЕМА ДЛЯ ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ РОБОТИ РІЗНИХ СУБД ПРИ РОБОТІ З РОЗПОДІЛЕНИМИ НЕРЕЛЯЦІЙНИМИ БАЗАМИ ДАНИХ

Керівництво користувача
1116130.01193-01 І3 01

Завідувач кафедри КІТ
_____Вадим ГОРЯЧКІН
Керівник розробки
_____Олександр ІВАНОВ
Виконавець
_____Сергій ДУДАР
Нормоконтролер
_____Світлана ВОЛКОВА

2025

ЗАТВЕРДЖЕНО
1116130.01193-01 І3 01

СИСТЕМА ДЛЯ ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ РОБОТИ РІЗНИХ СУБД ПРИ РОБОТІ З РОЗПОДІЛЕНИМИ НЕРЕЛЯЦІЙНИМИ БАЗАМИ ДАНИХ

Керівництво користувача.
1116130.01193-01 І3 01

Листів 10

2025

АНОТАЦІЯ

Документ 1116130.01193-01 І3 01 «Система для дослідження ефективності роботи різних СУБД при роботі з розподіленими нереляційними базами даних.»

Керівництво користувача. Керівництво по роботі з системою для дослідження часової ефективності операцій до різних документоорієнтованих баз даних входить до складу програмної документації до дипломного проекту. Даний документ містить опис призначення та умов застосування програми, опис операцій.

ЗМІСТ

ВСТУП

Програмний продукт «Система для дослідження ефективності роботи різних СУБД при роботі з розподіленими нереляційними базами даних» призначений для перевірки та швидкого збору даних щодо ефективності різних операцій (вставка, фільтрація, групування, пошук) до бази даних. Ефективність алгоритмів оцінюється часом роботи відносно одне одного для кожної СУБД окремо, а саме – MongoDb та CouchDb.

На сьогоднішній день існує безліч провайдерів нереляційних баз даних призначених для різних цілей і які підходять для тих чи інших операцій та умов. Тому завжди буде актуальним питання, яку нерелдяційну СУБД використати в тих чи інших умовах та яка х них буде найефективнішою в роботі з розподіленими базами даних.

Основною задачею даного програмного продукту є визначення часу роботи вищезгаданих операцій в залежності від вхідних параметрів та швидкий збір цих даних в зручному вигляді для подальшого аналізу.

Застосування такої програмної системи корисне в тому, щоб збирати дані щодо часу роботи операцій до різних нереляційних провайдерів баз даних для подальшого розуміння, яку саме систему використовувати для подальших потреб.

1 ПРИЗНАЧЕННЯ ТА УМОВИ ЗАСТОСУВАННЯ

1.1 Функціонал програми

Програма повинна:

- мати можливість зчитувати та записувати дані в базу даних;
- можливість роботи запити вибірки та фільтрації з різними умовами;
- мати можливість працювати з індексами: дослідження можливостей створення та використання індексів дял оптимізації роботи з даними та запитами;
- надавати можливість ввести параметри експерименту (тип операції по варіанту бази даних).

Вхідні дані:

- тип операції (текст);
- кількість тестів (ціле число);

Вихідні дані:

- візуальне відображення часу виконання операції;
- файл формату .CSV, який містить результати експерименту та має такі дані:
    a) Час роботи операції;
    b) Використаний тип операції.

1.2 Вимоги до складу і параметрів технічних засобів

Для коректної роботи програма повинна виконуватися на Windows-сумісних комп’ютерах, що мають такі мінімальні характеристики:

- 2-ядерний процесор з тактовою частотою не менше 1,6 ГГц;
- 8 ГБ доступної оперативної пам’яті;
- жорсткий диск на 128 ГБ;
- пристрій виводу зображення з роздільною здатністю екрану 1920x1080.

1.3 Вимоги до інформаційної і програмної сумісності

Вимога до інформаційної і програмної сумісності лише одна – операційна система Windows 10 має бути версії 20H2 і пізніше.

2 ПІДГОТОВКА ДО РОБОТИ

Для роботи з даним програмним продуктом користувачу необхідно мати:

- персональний комп’ютер;
- встановлену операційну систему Windows 10;
- девайси для взаємодії з персональним комп’ютером – клавіатуру, комп’ютерну мишу та екран.

2.1 Запуск програмного додатку

Для запуску програмного додатку потрібно виконати наступні команди у визначеному порядку:

1) Перейти до папки, яка містить виконуваний файл з назвою Sharded_Cluster.exe;
2) Двічі натиснути лівою кнопкою миші по іконці виконуваному файлу.

3 ОПИС ОПЕРАЦІЙ

Запустіть програмний додаток та перейдіть до екрану налаштування експерименту. На рис. 3.1 представлений вигляд екрану налаштування експерименту. Далі виберіть тип операції, яку ви хочете порівняти, для MongoDB, встановивши відповідну радіокнопку в секції MongoDB. Аналогічно виберіть тип операції для CouchDB в секції CouchDB. Натисніть кнопку "Run query". Зачекайте, поки програма виконає запити. Потім перегляньте час виконання операцій в полі "Execution time". В кінці перегляньте результати виконання запитів та додаткову інформацію у великій сірій області.

Після запуску користувач одразу побачить екран налаштування експерименту (рис. 3.1). Інтерфейс користувача складається з наступних елементів:

Рядок заголовка вікна: угорі є рядок заголовка вікна з назвою "Time_Comparer", а також стандартні значки керування вікном для мінімізації, розгортання/відновлення та закриття вікна.

Параметри бази даних: ліворуч є панель із позначкою «MongoDb», яка свідчить про те, що програма може порівнювати продуктивність операцій MongoDB з іншою базою даних. На цій панелі є три параметри з перемикачами:

- вибір;
- групування;
- сортування;
- фільтрування;

Ці параметри відповідають різним типам запитів до бази даних або операцій, які користувач може перевірити на час виконання.

Індикатор часу виконання: у центрі над великою сірою рамкою розташована позначка «Час виконання», яка вказує на те, що програма здатна відображати час, потрібний для виконання вибраної операції.

Поле для відображення запитів: у центрі інтерфейсу створена прямокутна область. Це простір, призначений для відображення результатів запитів і з чого складаються дані з різних запитів.

Протилежні параметри бази даних: праворуч є панель, подібна до панелі ліворуч із позначкою «CouchDb». Тут користувач може вибрати подібні операції для другої бази даних, щоб провести порівняння. Доступні параметри тут такі ж самі, що і зліва:

- вибір;
- групування;
- сортування;
- фільтрування.

Кнопки керування: під цими опціями є дві кнопки:

- «показати все»: цю кнопку можна використовувати для відображення всіх даних або результатів;
- «виконати запит»: це кнопка, яку користувач натисне,щоб виконати вибрані операції та виміряти час їх виконання.

Рисунок 3.1 – Екран налаштування експерименту

4 АВАРІЙНІ СИТУАЦІЇ

Якщо програмний додаток буде запускатися з пристрою де немає інтернет з’єднання, додаток виведе на екран повідомлення про помилку та завершить роботу.

Якщо програмний засіб під час роботи буде поводити некоректно чи із помилкою, необхідно перезапустити мобільним додаток для продовження роботи.

5 РЕКОМЕНДАЦІЇ ЩОДО ЗАСТОСУВАННЯ

Після встановлення додатку на пристрій його необхідно запустити через виконуючий файл.

Після запуску додатку перед користувачем відкриється головний екран з усім необхідним функціоналом.

Після ознайомлення із функціями додатку, його роботу можна звершити натиснувши апаратну кнопку вимкнення.