Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет комп'ютерних наук (або центр післядипломної освіти, або навчально-науковий центр заочної форми навчання)
(повна назва)

Кафедра програмної інженерії
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти другий (магістерський)

Дослідження та оптимізація продуктивності баз даних у розподілених обчислювальних мережах
(тема)

Виконав:
студент (ка) 2 курсу, групи ІПЗздм-22-1

Константинов О.Р.
(прізвище, ініціали)

Спеціальність 121 – Інженерія програмного забезпечення
(код і повна назва спеціальності)
Тип програми освітньо-наукова

Керівник доц. Вечур О.В
(посада, прізвище, ініціали)

Допускається до захисту
Зав. кафедри _____ З.В.Дудар
(підпис) (прізвище, ініціали)

2024 р.

Харківський національний університет радіоелектроніки

Факультет комп'ютерних наук (або центр післядипломної освіти, або навчально-науковий центр заочної форми навчання)
Кафедра програмної інженерії
Рівень вищої освіти другий (магістерський)
Спеціальність 121 – Інженерія програмного забезпечення
Тип програми освітньо-наукова програма
Освітня програма Інженерія програмного забезпечення
(шифр і назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«___» _____ 2024 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

студентові Константинову Олександру Романовичу
(прізвище, ім'я, по батькові)

1. Тема роботи «Дослідження та оптимізація продуктивності баз даних у розподілених обчислювальних мережах»
Затверджена наказом по університету від «22» квітня 2024 р. № 60 Стз
2. Термін подання студентом роботи до екзаменаційної комісії 14.06.2024
3. Вихідні дані до роботи опис досліджуваних баз даних, теорія баз даних у розподілених обчислювальних мережах для проведення досліджень за обраною предметною областю, принципи розробки програмного забезпечення
4. Перелік питань, що потрібно опрацювати в роботі
реферат, вступ, аналіз предметної галузі, аналіз та порівняння існуючих реляційних на нереляційних баз даних, дослідження існуючих алгоритмів розробки, проектування архітектури віртуальної моделі, написання програмних рішень, проведення експериментів та аналіз отриманих результатів, висновки, перелік джерел посилань.

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1	Аналіз предметної галузі та постановка задачі	01.03 – 15.03.24	*виконано*
2	Аналіз та вибір баз даних для дослідження	15.03 – 31.03.24	*виконано*
3	Аналіз та моделювання предметної області	05.05 – 15.05.24	*виконано*
4	Планування експериментів	01.04 – 15.04.24	*виконано*
5	Реалізація програмних модулів	10.04 – 30.04.24	*виконано*
6	Експериментальні дослідження	01.05 – 10.05.24	*виконано*
7	Аналіз результатів експериментальних досліджень та розробка рекомендацій	10.05 – 20.05.24	*виконано*
8	Написання та оформлення статті та тез доповіді	20.05 – 28.05.24	*виконано*
9	Підготовка пояснювальної записки	01.05 – 02.06.24	*виконано*
10	Підготовка презентації та доповіді	02.06 – 08.06.24	*виконано*
11	Нормоконтроль	10.06 – 12.06.24	*виконано*
12	Рецензування	08.06 – 12.06.24	*виконано*
13	Занесення диплома в електронний архів	12.06.2024	*виконано*
14	Попередній захист	10.06.2024	*виконано*
15	Допуск до захисту у зав. кафедри	14.06.2024	*виконано*

Дата видачі завдання 17 січня 2024р.

Студент (ка) _____ Константинов О.Р.
(підпис)

Керівник роботи _____ доц. Вечур О.В.
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ / ABSTRACT

Пояснювальна записка: 96 ст., 11 рис., 2 табл., 5 додатків, 23 джерела.

АЛГОРИТМИ ПРОГНОЗУВАННЯ, ХМАРНА БАЗА ДАНИХ, AZURE CLOUD, AZURE COSMOS DB, COSMOS DB API FOR NOSQL.

Об’єктом дослідження є види та способи подання знань в розподілених обчислювальних мережах.

Метою роботи є аналіз предметної галузі , збір даних та постановка задачі дослідження цієї галузі.

У методах досліджень використовуються збір, аналіз, порівняння існуючих освітніх програм.

У результаті роботи зроблено комплекс ряд досліджень та підведені висновки.

PREDICTION ALGORITHMS, CLOUD DATABASE, AZURE CLOUD, AZURE COSMOS DB, COSMOS DB API FOR NOSQL.

The object of research is the types and methods of presenting knowledge in hypermedia systems.

The purpose of the work is the analysis of the subject area, data collection and formulation of the research task of this area.

Collection, analysis, and comparison of existing educational programs are used in research methods.

As a result of the work, a complex of studies was made and conclusions were summarized.

Я, Константинов Олександр Романович, студент(ка) гр. ІПЗздм-22-1, здобувач вищої освіти на другому (магістерському) рівні кафедри «Програмна

інженерія», заявляю: моя кваліфікаційна робота на тему «Дослідження та оптимізація продуктивності баз даних у розподілених обчислювальних мережах», що буде представлена в екзаменаційну комісію для публічного захисту, виконана самостійно, в ній не містяться елементи плагіату і вона може бути опублікована в електронному архіві відкритого доступу ElArKhNURE. Всі запозичення з друкованих та електронних джерел мають відповідні посилання.

Я ознайомлений(на) з діючим положенням «Про протидію академічному плагіату в ХНУРЕ», згідно з яким виявлення плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту та застосування дисциплінарних заходів.

ЗМІСТ

Вступ……....9
1 Аналіз предметної галузі……...11
1.1 Оптимізація базових структур та алгоритмів…….....12
1.2 Оптимізація бази даних……12
1.3 Мікросервіси…….....13
1.4 Кешування…….....14
1.5 Масштабування…….....15
1.6 Оптимізація, заснована на правилах……16
1.7 Оптимізація запитів в розподілених базах даних……..17
1.8 Постановка задачі…….....22
2 Опис та аналіз різних типів баз даних та їх застосування у реальному житті….24
3 Дослідження та порівняння різних баз даних……..31
3.1 PostgreSQL…….....35
3.1.1 Переваги PostgreSQL…….....36
3.1.2 Недоліки PostgreSQL…….....36
3.2 MySQL……37
3.2.1 Переваги MySQL……..37
3.2.2 Недоліки MySQL……..38
3.3 MongoDB……38
3.3.1 Переваги MongoDB……..39
3.3.2 Недоліки MongoDB……..40
3.4 Висновки до розділу…….....40
4 Аналіз часової складності методів оптимізації структури баз даних NoSQL….....42
4.1 Методи оптимізації структури нерозподілених баз даних NoSQL……..42
4.1.1 Метод визначення структури колекцій для нерозподілених баз даних типу «ключ-документ»……43
4.1.2 Методика визначення колекцій в базах даних типу «ключ-документ» в залежності від вихідних даних …….....43

4.1.3 Метод визначення структури вкладених документів в БД типу «ключ-документ»…….44
4.2 Методи оптимізації структури розподілених баз даних NoSQL…….....45
4.2.1 Метод побудови інформаційного графа запиту для розподіленої реляційної бази даних……46
4.2.2 Метод побудови схеми розподіленої бази даних NoSQL типу «ключ-документ»…….46
4.2.3 Метод оптимізації доступу до даних в розподіленій базі даних NoSQL типу «ключ-документ»……..47
4.3 Висновок…….48
5 Перетворення запитів із структури реляційної бази даних у формат бази NoSQL типу “ключ-документ”……...50
5.1 Мови запитів до баз даних……51
5.2 Трансляція запитів з формату PostgreSQL у формат MongoDB……53
5.2.1 Парсинг запиту в форматі PostgreSQL…….....56
5.2.2 Формування словника запитів MongoDB……..60
5.2.3 Переробка представлень словника з урахуванням структури бази даних MongoDB……..63
5.2.4 Формування вихідного запиту з елементів словника до бази даних MongoDB……..65
5.3 Висновки…….67
6 Розробка та реалізація програмних модулів……....68
6.1 Опис програмних модулів мовою Java, що розділені на кроки……..68
6.2 Тестування програмних частин……72
Висновки……76
Перелік джерел посилання……...77
Додаток А Перелік джерел посилання за науковими напрямами керівника та науковців кафедри програмної інженерії……79
Додаток Б Звіт результатів перевірки на унікальність тексту в базі ХНУРЕ……..80
Додаток В Слайди презентації…….81

Додаток Г Апробація результатів роботи……...90
Додаток Д Експертний висновок результатів перевірки кваліфікаційної роботи на відповідність оформлення вимогам ДСТУ 3008: 2015……96

ВСТУП

У сучасному цифровому світі обробка великих обсягів даних стала не тільки нормою, але й ключовим елементом успіху для багатьох організацій та підприємств різних галузей. Підтримка високої продуктивності баз даних у розподілених обчислювальних мережах важлива для забезпечення швидкого доступу до інформації та ефективного виконання операцій.

Магістерська робота спрямована на дослідження та оптимізацію продуктивності баз даних у розподілених обчислювальних мережах. Ця тема стала актуальною через стрімкий розвиток технологій, що призвів до зростання обсягів даних, а також через поширення використання розподілених обчислювальних середовищ у великих корпораціях та малому бізнесі.

Розподілені обчислювальні мережі використовуються для обробки даних у великих обсягах, розподілення завдань обробки між різними вузлами та забезпечення доступності даних в різних місцях одночасно. Проте, ефективність роботи баз даних у таких середовищах залежить від ряду факторів, включаючи алгоритми реплікації даних, стратегії маршрутизації запитів та механізми синхронізації даних.

Метою даної магістерської роботи є проведення комплексного дослідження продуктивності баз даних у розподілених обчислювальних мережах та розробка оптимальних методів їх оптимізації. Для досягнення цієї мети передбачається:

- аналіз сучасного стану технологій баз даних у розподілених середовищах на основі огляду існуючих методів реплікації, розподілення та доступ до даних у розподілених системах, виявлення їх переваг та недоліків;
- експериментальне дослідження продуктивності на основі виконання серії експериментів для оцінки продуктивності різних алгоритмів та стратегій роботи баз даних у розподілених обчислювальних мережах, а також для виявлення чинників, що впливають на їх ефективність;
- розробка оптимальних підходів до оптимізації на основі отриманих

результатів розроблення нові стратегії та методи оптимізації роботи баз даних у розподілених середовищах з урахуванням специфіки завдань та вимог користувачів.

Ця робота має важливе теоретичне та практичне значення. Результати дослідження можуть бути використані для покращення продуктивності баз даних у розподілених обчислювальних мережах, що сприятиме підвищенню ефективності обробки даних та забезпеченню більш високої якості обслуговування користувачів. Враховуючи складність та масштабність проблеми, дослідження, яке виконано в рамках цієї роботи, є кроком до подальших досягнень в області оптимізації обробки даних у розподілених обчислювальних мережах.

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ

У сучасному світі, зі зростанням обсягів даних та розширенням обчислювальних можливостей, значно підвищується потреба в ефективному управлінні базами даних, особливо у контексті розподілених обчислювальних мереж. Розподілені бази даних, які характеризуються тим, що їх компоненти розміщені на різних фізично віддалених вузлах, вимагають особливого підходу до синхронізації, забезпечення консистентності даних, високої доступності та зниження латентності обробки запитів.

Для підвищення продуктивності розподілених баз даних, вчені та інженери використовують різні техніки, зокрема реплікацію даних для забезпечення швидкого доступу та відмовостійкості, партитування для оптимального розподілу даних між вузлами та зменшення навантаження на окремі вузли, а також кешування для мінімізації звернень до центральної бази даних. Також важливою є оптимізація запитів, що дозволяє знизити навантаження на систему і покращити час відгуку.

На ринку існує багато сучасних технологій, таких як Apache Cassandra, Amazon DynamoDB та Google BigTable, що пропонують гнучкі і ефективні рішення для управління розподіленими даними з високим рівнем доступності та масштабованості. Ці платформи допомагають вирішувати проблеми зі збереженням консистентності даних, високою доступністю та зниженням затримок.

Однак, незважаючи на значний прогрес у розробці технологій для розподілених баз даних, існують виклики, такі як забезпечення даних у безпеці, ефективне управління ресурсами та інтеграція з новітніми технологіями, зокрема зі штучним інтелектом та машинним навчанням, які вимагають подальших досліджень та інновацій [1].

Завдяки постійному вдосконаленню методологій і технологій, можна досягти значного підвищення продуктивності, доступності та безпеки у розподілених базах даних, що є ключем до успішної реалізації глобальних інформаційних систем. Вирішення цих викликів відкриває нові можливості для

подальшого розвитку та оптимізації розподілених обчислювальних мереж.

1.1 Оптимізація базових структур та алгоритмів

Оцінка складності алгоритмів в програмному забезпеченні - це ключовий аспект у керуванні його продуктивністю. Основні алгоритми обробки великих об'ємів даних, які використовуються в програмах, мають велике значення, оскільки складають основну частину операцій.

Наприклад, коли програмісту потрібно створити набір однорідних даних (структуру даних), він може обрати декілька способів її реалізації:

- лінійний масив з прямим доступом;
- лінійний список з довільним доступом;
- лінійний список з обмеженим доступом (черга або стек);
- множина, де кожен елемент може бути представлений лише один раз.

Кожна з таких структур може бути описана як набір основних операцій, які можна розділити на дві групи: модифікація контейнера (додавання або видалення елементів) та пошук в ньому (вибірка окремих елементів).

Ефективність алгоритмів оцінюється за допомогою оцінок складності, які вказують на те, як швидко алгоритм працює в залежності від його основних параметрів. Для операцій з контейнерами основним параметром є розмір контейнера. Наприклад, складність операції вибірки елементу з лінійного масиву за індексом становить O(1), що означає, що швидкість цієї операції не змінюється з розміром масиву. Однак операції вставки або видалення елемента мають складність O(n), оскільки, в середньому, для виконання таких операцій потрібно переписати весь масив [2].

Таким чином, оцінка ефективності алгоритмів є критично важливою для розробників програмного забезпечення, адже від її результатів залежить продуктивність програми.

1.2 Оптимізація бази даних

Оптимізація баз даних у розподілених обчислювальних мережах є складним

завданням, що вимагає дослідження та розробки ефективних стратегій для забезпечення швидкодії, ефективності використання ресурсів та надійності роботи системи. У цьому пункті дипломної роботи акцент буде спрямований на аналіз різних аспектів оптимізації баз даних у розподілених середовищах та розробку нових методів для підвищення їх продуктивності.

Одним із ключових аспектів оптимізації є покращення швидкодії операцій баз даних. Швидке виконання запитів та операцій є важливим для забезпечення реактивності системи та задоволення потреб користувачів. Для досягнення цього будуть досліджуватись різні методи оптимізації запитів, використання індексів та інші стратегії для підвищення швидкодії операцій.

Додатково, оптимізація баз даних також передбачає зменшення витрат на ресурси. Ефективне використання пам'яті та обчислювальної потужності може допомогти знизити витрати на обслуговування та забезпечити економію ресурсів. Для досягнення цієї мети планується дослідження різних методів кешування даних, реплікації та інші техніки для оптимізації використання ресурсів.

Результати наших досліджень та розроблених методів оптимізації можуть мати значний практичний вплив на розробку та підтримку інформаційних систем у різних галузях. Вони можуть бути використані для покращення продуктивності та ефективності роботи баз даних у розподілених обчислювальних мережах, що в свою чергу сприятиме підвищенню ефективності бізнес-процесів та задоволенню потреб користувачів [3].

Оптимізація баз даних у розподілених обчислювальних мережах є складною та важливою задачею, яка потребує поєднання теоретичних досліджень та практичного досвіду. Наша робота спрямована на знаходження оптимальних рішень для досягнення цієї мети та вирішення викликів, що стоять перед сучасними системами обробки даних.

1.3 Мікросервіси

Мікросервісна архітектура, яка стала вельми популярною в останні роки, сприяє розподілу функцій між окремими частинами програмного забезпечення.

Це дозволяє збільшити гнучкість системи і зробити процес впровадження та супроводження більш передбачуваним. Проте, ця архітектура не лише має свої переваги, але й потенційні недоліки, зокрема, вплив на продуктивність системи.

Основний виклик полягає в тому, що мікросервіси взаємодіють між собою через мережеві виклики. Це може призвести до додаткових затрат на організацію цієї взаємодії. Деякі розробники намагаються зберегти модульність службової частини навіть за цієї вартою, використовуючи найбільш ефективні протоколи взаємодії, наприклад, gRPC. Однак, цей протокол поки що не є загальновизнаним стандартом, що може створювати додаткові виклики для розробників.

У випадку, якщо у нас виникає потреба в оптимізації продуктивності застосунку, може бути корисним розглядати можливість об'єднання окремих мікросервісів у більші одиниці. Це дозволить зменшити кількість мережевих викликів та, отже, підвищити продуктивність системи. Проте, такий підхід, схожий на денормалізацію таблиць бази даних, потребує ретельного аналізу системи з точки зору її ефективності перед впровадженням [4].

1.4 Кешування

Використання кешування є одним з найпростіших та стандартних методів оптимізації програмних систем. Його суть полягає в тому, що результати запитів до сервісу обчислюються лише при першому виклику, а потім зберігаються у спеціальному сховищі. Наступні виклики вже не потребують повторного обчислення, а просто повертають результат із сховища.

Сьогодні всі системи управління базами даних мають вбудовані засоби для кешування SQL-запитів. Оптимізація полягає у правильному налаштуванні параметрів кешування, часу життя збережених результатів, стратегії вилучення застарілих записів тощо.

Кешування викликів службового інтерфейсу є трохи складнішим завданням. Сучасні веб-сервери мають потужні механізми кешування статичних даних, таких як зображення, файли коду та опис сторінок. Збережений у пам'яті вміст підвищує ефективність роботи системи.

Але веб-сервери не можуть кешувати динамічні виклики, оскільки дані є результатом обчислення функцій, а не зберігаються у файлах. Для цього використовуються спеціалізовані системи кешування, наприклад, система Varnish.

Оптимізація за допомогою кешування є простим методом підвищення продуктивності системи, але неправильне застосування або налаштування кешування може призвести до неправильної роботи системи. Наприклад, якщо каталог товарів змінився через завантаження нових прайсів, кеш може видавати застарілу інформацію користувачам, що може призвести до неправильного розуміння асортименту та цін [4].

1.5 Масштабування

Демонструючи можливості масштабування, було вказано на те, що не лише сервер застосунку, але й сховище даних може бути масштабоване. Система управління базами даних може розподіляти дані на різних вузлах мережі, що забезпечує паралельний доступ до даних і прискорює роботу з ними.

Однак основним методом оптимізації тут є розгортання додаткових вузлів застосунку, які можуть розташовуватись на різних мережевих вузлах. Ключову роль в цьому відіграє спеціальний сервіс – диспетчер запитів. Коли приходить запит з прикладної частини системи, диспетчер обирає сервіс, який буде виконувати цей запит. Ці вузли застосунку є точними копіями, тому диспетчеру не потрібно визначатися з функціональним змістом запиту. Все, що він робить, - це визначає вузол, який зараз не працює, і віддає запит на обробку саме йому. Крім того, диспетчер визначає, де і які вузли мережі споживають найменше фізичних ресурсів, і направляє запрос саме туди.

При такому підході виникають дві проблеми:

- весь код застосунку потрібно змінювати і розвивати: виправляються помилки програмного коду, додаються нові можливості, проводиться оптимізація. При цьому всі вузли застосунку повинні бути однаковими, що потребує суворої дисципліни впровадження та розгортання;
- алгоритми роботи диспетчера запитів досить складні.

Першу проблему розв'язують спеціалізовані системи контейнеризації, такі як Docker, яка дозволяє розгортати вузли застосунку в будь-якій конфігурації мережевих вузлів. Другу проблему вирішують спеціалізовані системи балансування навантаження, такі як Kubernetes, яка також виконує необхідну контейнеризацію[4].

1.6 Оптимізація, заснована на правилах

Після досягнення певного прогресу у поліпшенні обробки запитів, були вдосконалені методи доступу до таблиць. Це означає розробку методів доступу, що базуються на використанні індексів та функцій хешування. Однак застосування індексації та хешування збільшує складність обробки запитів. Наприклад, якщо таблиця має індекси за трьома різними колонками, будь-який із них може використовуватися для доступу до таблиці (окрім послідовного доступу до таблиці в фізичному порядку розташування рядків).

Крім того, з'явилося багато нових алгоритмів для виконання з'єднання таблиць. Два найбільш основні алгоритми виконання з'єднання:

З'єднання за допомогою вкладеного циклу (Nested Loop Join): У цьому алгоритмі рядок читається з першої таблиці (зовнішньої), і потім читається кожен рядок другої таблиці (внутрішньої) як кандидат для з'єднання. Цей процес повторюється для кожного рядка першої таблиці.

З'єднання за допомогою об'єднання (Merge Join): Цей метод передбачає відсортовані (або проіндексовані) таблиці, що дозволяє з'єднувати рядки за значеннями колонок. З'єднання завершується за один прохід по кожній таблиці.

Різні шляхи доступу, алгоритми з'єднань і порядок їх виконання можуть значно вплинути на продуктивність. Тому під час виконання з'єднання декількох таблиць важливо враховувати різні можливі комбінації для вибору найефективнішого шляху виконання.

Один із перших підходів до подолання комбінаторної складності виконання з'єднань полягає у встановленні евристичних правил для вибору між шляхами доступу та методами з'єднань, цей підхід називається оптимізацією, заснованою

на правилах (rule-based optimization). При використанні цього методу план виконання складається на основі існуючих шляхів доступу та їх рангів. Усі шляхи доступу ранжуються на основі знань про правила та послідовності їх виконання.

Ранги шляхів доступу для СКБД Oracle визначаються таким чином:

- один рядок за ROWID *;
- один рядок за кластерним з'єднанням;
- один рядок за хеш-кластером з унікальним або первинним ключем;
- один рядок за унікальним або первинним ключем;
- кластерне з'єднання;
- ключ хеш-кластера;
- ключ індексного кластера;
- складовий індекс;
- індекс по одиночному стовпцю (за умовою рівності);
- індексний пошук по закритому інтервалу;
- індексний пошук по відкритому інтервалу;
- сортування-об'єднання;
- MAX і MIN по індексованим стовпцям;
- ORDER BY по індексованим;
- повний перегляд таблиці.

Ранг шляху доступу визначається на основі знань про послідовність його виконання. Метод оптимізації за синтаксисом враховує ранги шляхів доступу. Якщо для будь-якої таблиці існує більше одного шляху доступу, вибирається той шлях, чий ранг вище, оскільки за інших рівних умов він виконується швидше, ніж шлях з більш низьким рангом.

План виконання запиту у методі оптимізації за синтаксисом формується з обраних шляхів доступу з максимальними рангами[5].

1.7 Оптимізація запитів в розподілених базах даних

Розвиток розподілених баз даних став результатом поширення розподілених систем, зокрема інформаційно-пошукових систем, які часто інтегрують численні

бази даних, орієнтовані на різноманітні предметні галузі. Розробка таких баз даних вимагає вирішення складних завдань, що включають не тільки інтеграцію і проектування локальних традиційних баз даних, а й розв’язання нових проблем, характерних для розподілених баз. До таких проблем належать:

- оптимізація розміщення даних у вузлах мережевої структури, на яку сучасні CASE-системи (Computer Aided System Engineering) ще не налаштовані;
- масштабування баз даних в умовах географічного розподілу;
- розробка та використання комунікаційного середовища для розподіленої бази даних;
- оптимізація розміщення даних для забезпечення рівномірного навантаження;
- синхронізація доступу до даних та паралельна обробка запитів для підвищення продуктивності системи;
- підтримка даних у кількох вузлах мережі для мінімізації передач даних при виконанні запитів;
- фрагментація об’єктів бази даних та розширення можливостей пошуку при обробці запитів;
- управління транзакціями для забезпечення коректного завершення розподілених операцій, багаторівневий захист даних і забезпечення доступу до інформації різної структури і змісту з різних джерел.

У зв'язку з цими змінами з'явилися нові вимоги до розподілених баз даних та нові технології для їх проектування. Логічно єдина база даних ділиться на фрагменти, кожен з яких розміщений на окремому комп'ютері, всі комп'ютери з'єднані між собою мережевими лініями. Кожен фрагмент керується своєю системою управління базами даних. До основних функцій СУБД додаються:

- розширені служби для встановлення з'єднань та передачі даних між вузлами;
- розширені можливості каталогізації для збереження інформації про розподіл об'єктів у мережі;

- механізми для обробки розподілених запитів, оптимізації та організації віддаленого доступу;
- функції для забезпечення цілісності та відновлення даних з урахуванням можливих збоїв у інших вузлах мережі чи зв'язку.

Методи підтримки розподілених баз даних включають фрагментацію, реплікацію, розподілені обмеження цілісності, розподілені запити та розподілені транзакції.

Базу даних розділяють фізично на вузли даних шляхом фрагментації та реплікації даних. У випадку наявності схеми реляційної бази даних кожна таблиця розбивається на горизонтальні або вертикальні фрагменти. Горизонтальна фрагментація здійснюється за допомогою операції селекції, яка розміщує кожний рядок таблиці в одному з фрагментів, відповідно до умови фрагментації. Наприклад, таблицю "Employee" можна розділити за місцем розташування робочих місць службовців. У вертикальній фрагментації стовпці таблиці розділяються на фрагменти за допомогою операції проекції. Фрагментація дозволяє наблизити дані до місця їх найбільш інтенсивного використання, що потенційно зменшує витрати на передачу даних та розмір таблиць у запитах користувачів.

Запит, який звертається до двох чи більше вузлів бази даних, але не змінює дані на них, називається розподіленим. Вузол, що виконує запит, повинен виявити, що запит містить посилання на дані на іншому вузлі, виділити підзапит для цього вузла і переслати його на віддалений вузол.

У централізованій системі керування базами даних процес складається з двох кроків: декомпозиції запиту (query decomposition) та оптимізації запиту. Декомпозиція запиту полягає у перетворенні його з мови SQL в вираз реляційної алгебри. Під час декомпозиції запит проходить семантичний аналіз, при якому некоректні запити відкидаються, а коректні спрощуються. Для заданого SQL-запиту існує кілька алгебраїчних представлень, і вибирається найкраще залежно від витрат, необхідних для обчислення. Оптимізація запиту полягає у пошуку оптимального алгебраїчного виразу для запиту.

У розподіленій системі керування базами даних до кроків декомпозиції і оптимізації запиту додаються ще дві дії: локалізація даних (data localization) і глобальна оптимізація запиту (global query optimization). Під час локалізації даних глобальний запит перетворюється на запити до фрагментів баз даних, враховуючи їх розподіл.

Програма локалізації для горизонтально (вертикально) фрагментованої відносини складається з об'єднання (union) (або з'єднання (join)) відповідних фрагментів. Це означає, що на етапі локалізації даних кожне глобальне ставлення запиту замінюється його програмою локалізації, а потім результуючий фрагментний запит спрощується і реструктурується з метою отримання ефективного запиту. Для спрощення та реструктуризації можуть використовуватися ті ж правила, що і на етапі декомпозиції. Проте, як і на етапі декомпозиції, остаточний запит над фрагментами може бути ще далекий від оптимального; даний процес лише виключає погані алгебраїчні запити.

Мета глобальної оптимізації полягає в тому, щоб знайти стратегію виконання запиту, яка близька до оптимальної. Це можна виразити через операції реляційної алгебри і комунікаційні примітиви, які використовуються для передачі даних між вузлами. На попередніх етапах запит вже був у певній мірі оптимізований, зокрема, за рахунок видалення надлишкових виразів.

Проте, проведена оптимізація не залежала від характеристик фрагментів, наприклад, їх потужності. Крім того, на попередніх кроках ще не враховувалися комунікаційні операції. Шляхом зміни порядку операцій всередині одного фрагментного запиту можна отримати багато еквівалентних планів його виконання. Оптимізація запиту полягає в знаходженні найкращого плану з безлічі можливих планів, які досліджує оптимізатор.

Стратегія пошуку визначає спосіб обходу простору пошуку та вибору найкращого плану. Вона визначає порядок вибору та оцінки планів виконання запиту. У розподіленому середовищі функція вартості, часто виражена в одиницях часу, оцінює витрати обчислювальних ресурсів, таких як дисковий простір, кількість обмінів з дисками, час центрального процесора, комунікації і

т.д. Зазвичай це деяка зважена сума витрат на введення-виведення, центральний процесор і комунікацію. У розподілених системах керування базами даних (СКБД) застосовується спрощений підхід, коли найбільший акцент робиться на комунікаційних витратах.

Це особливо важливо для глобальних мереж, де передача даних через лінії зв'язку коштує значно дорожче, ніж при локальній обробці. Для визначення оптимального порядку виконання операцій необхідно оцінювати вартість різних планів виконання з різними порядками операцій. Визначення вартості виконання перед реальним виконанням запиту (статична оптимізація) базується на статистиці фрагментів і формулах для оцінки потужності результатів реляційних операцій.

Важливим аспектом оптимізації запитів є порядок виконання з'єднань, оскільки зміна цього порядку може суттєво збільшити швидкість виконання. Один з базових методів оптимізації послідовності розподілених операцій з'єднання полягає в застосуванні операції напівз'єднання (semijoin). Основна перевага напівз'єднання в розподіленій системі полягає в тому, що воно скорочує обсяги даних, які беруть участь у з'єднанні, та відповідно зменшує комунікаційні витрати.

Результатом роботи глобального оптимізатора є оптимізований алгебраїчний вираз, який включає комунікаційні операції над фрагментами.

Оптимізація, заснована на правилах, забезпечує задовільну продуктивність системи у випадках, коли евристики є точними. Але часто загальновизнані правила не є завжди точними. Для виявлення таких ситуацій оптимізатор запитів повинен аналізувати характеристики даних, такі як:

- кількість рядків у таблиці;
- інтервал та розподіл значень у даному стовпці;
- довжина рядка і, відповідно, кількість рядків на фізичній сторінці диска;
- висота індексу;
- кількість термінальних (leaf) сторінок у індексі.

Ці характеристики даних можуть суттєво впливати на ефективність обробки запиту. Використання таких характеристик призводить до наступного типу оптимізації[6].

1.8 Постановка задачі

Сучасні інформаційні системи вимагають ефективного управління даними, що включає оптимізацію структури та алгоритмів роботи баз даних. Це дозволяє забезпечити високу продуктивність, масштабованість та надійність системи. Основною метою дипломної роботи є розробка методів та інструментів для оптимізації баз даних, які дозволять підвищити ефективність їх використання в інформаційних системах.

Для досягнення цієї мети необхідно вирішити наступні завдання: провести аналіз предметної галузі, зокрема оптимізацію базових структур та алгоритмів, бази даних на рівні архітектури (включаючи мікросервіси та кешування), дослідження методів масштабування та оптимізації, заснованих на правилах, а також оптимізацію запитів у розподілених базах даних. Описати та проаналізувати різні типи баз даних (реляційні, такі як PostgreSQL та MySQL, і NoSQL, такі як MongoDB), порівняти їхні переваги та недоліки, а також області застосування. Провести детальне дослідження PostgreSQL, MySQL та MongoDB, визначити переваги та недоліки кожної бази даних. Розробити та проаналізувати методи оптимізації структури нерозподілених та розподілених баз даних NoSQL, описати методи побудови інформаційного графа запиту та схеми розподіленої бази даних.

Необхідно також дослідити мови запитів до баз даних та розробити методи трансляції запитів з формату PostgreSQL у формат MongoDB, розробити алгоритми парсингу та формування запитів для MongoDB. Розробити програмні модулі мовою Java для реалізації оптимізації та трансляції запитів, провести тестування розроблених програмних модулів для оцінки їх ефективності.

Об'єктом дослідження є бази даних різних типів, зокрема реляційні та NoSQL бази даних. Предметом дослідження виступають методи та інструменти для оптимізації баз даних та трансляції запитів між різними типами баз даних.

Актуальність роботи обумовлена зростаючими вимогами до продуктивності, масштабованості та надійності інформаційних систем, що вимагають ефективного управління даними та оптимізації баз даних. Наукова новизна полягає у розробці нових методів оптимізації структури баз даних та трансляції запитів між різними типами баз даних, що дозволяє підвищити їх продуктивність та ефективність використання в інформаційних системах. Практичне значення роботи полягає у впровадженні розроблених методів та інструментів для оптимізації баз даних у реальних інформаційних системах, що дозволить підвищити їх продуктивність та надійність.

2 ОПИС ТА АНАЛІЗ РІЗНИХ ТИПІВ БАЗ ДАНИХ ТА ЇХ ЗАСТОСУВАННЯ У РЕАЛЬНОМУ ЖИТТІ

У світі баз даних розрізняють реляційні та нереляційні моделі. Нереляційні бази даних можуть використовувати різні підходи, зокрема графи, "ключ-значення", а найчастіше – документно-орієнтовані моделі. Ці системи з'явилися як відповідь на потребу в більшій гнучкості, скороченні часу на обробку даних та спрощенні інтеграції з сучасними мовами програмування. Вони оперують об'єктами, що мають схожу структуру до об'єктів у мовах програмування, зменшуючи необхідність конвертації даних, яка часто потрібна в реляційних системах.

MariaDB та MongoDB виступають яскравими прикладами реляційної та нереляційної моделей баз даних відповідно. MariaDB, що є форком MySQL, пропонує швидший та інноваційний двигун для роботи з великими обсягами даних та складною схемою. Вона підходить для ситуацій, де необхідна чітка консистентність і розподіл даних.

З іншого боку, MongoDB використовується у випадках, коли традиційні реляційні моделі можуть бути занадто обтяжливими для реалізації архітектури. Ця система управління базами даних(далі СУБД) ідеально підходить для систем, що управляють великою кількістю параметрів для користувачів або для створення унікального контенту, такого як сайти або шаблони. Основною перевагою MongoDB є легкість масштабування, оскільки нові дані можна додавати без зайвих зусиль на розширення системи.

Обидві системи відіграють важливу роль у виборі підходящої технології баз даних залежно від специфіки та потреб проекту, проте важливо розуміти їхні особливості та обмеження, особливо при розробці масштабних систем, щоб уникнути перенасичення або недостатності даних.

При виборі між MyISAM і InnoDB, важливо зважати на специфіку вашого проекту. Якщо для вашої системи критичні транзакції і необхідна висока надійність зберігання даних, то кращим варіантом буде InnoDB. Цей двигун

підтримує транзакції, забезпечує цілісність даних на рівні ACID, і має кращі можливості відновлення після збоїв.

Якщо ж ваш проект передбачає більше читання ніж записування, і не критичні транзакційні вимоги, MyISAM може бути вигідним через його ефективність у швидкості читання та простоту. Також, MyISAM підтримує індексування FULLTEXT, що є великою перевагою для систем, які здійснюють багато текстових пошуків.

Крім того, MariaDB пропонує інші типи двигунів, які можуть бути корисними для специфічних випадків. Наприклад, Memory двигун підходить для тимчасових таблиць, які потребують швидкого доступу і не зберігають дані після перезапуску сервера. CSV двигун може бути корисним для систем, де потрібен прямий доступ до даних у форматі CSV.

Окрім технічних характеристик двигунів, при виборі СУБД слід враховувати також такі аспекти, як масштабованість системи, потреби в резервному копіюванні та відновленні даних, а також здатність до горизонтального та вертикального масштабування. MariaDB, з її високим рівнем налаштувань і гнучкості, може надати необхідні інструменти для оптимізації продуктивності та забезпечення високої доступності сервісу.

Таким чином, вибір СУБД та двигуна даних має відбуватися на основі детального аналізу вимог проекту та можливостей різних технологічних рішень, що в кінцевому підсумку дозволить досягти оптимальної продуктивності та ефективності витрат.

Ця частина тексту розглядає особливості СУБД MariaDB та MongoDB.

СУБД MariaDB підтримує всі стандартні механізми зберігання даних, які доступні у MySQL, а також має ряд нових механізмів, таких як ColumnStore, MyRocks (з відмінною компресією даних), Aria (заміна MyISAM з рядом переваг і оновлень), FederatedX, OQGRAPH, SphinxSE, TokuDB, CONNECT, SEQUENCE, Spider, та Cassandra. Всі ці механізми роблять MariaDB мульти-модельною базою даних, що є універсальним інструментом для розробки будь-якої системи.

MariaDB дозволяє зберігати всі існуючі типи даних у таблицях, які можуть бути логічно пов’язані за допомогою ключів. Реляційна модель передбачає наявність зв’язків типів "1-to-1", "1-to-many" або "many-to-many", що дозволяє логічно описати базу даних та таблиці [7].

MariaDB може бути використана як сервіс і розміщена навіть на сторонніх хмарових платформах. Відмінність між базою даних як сервісом (DBaaS) і просто базою полягає в розміщенні, управлінні та побудові бази даних для систем.

Також до бази даних можливо додати тригери та події, які будуть виконуватися за певними правилами, що корисно для підтримки логіки системи, ведення статистики чи аналізу, а також для інших операцій. Більшість таких операцій можливо реалізувати на стороні мови програмування, але вони будуть більш продуктивними на сервері бази даних.

MongoDB, найкраща NoSQL СУБД на даний момент, є документно-орієнтованою СУБД з відкритим вихідним кодом. Вона використовує документну модель зберігання даних, схожу на JSON структуру об’єктів (BSON), що дозволяє легше розуміти та маніпулювати даними за допомогою елементарних запитів. JSON також надає ефективність та надійність у порівнянні з іншими форматами даних.

Текст в порівнянні MongoDB з MariaDB виглядає так, що в базі даних MongoDB формуються колекції, які зберігаються замість таблиць, що характерно для реляційних моделей. Однак, колекції в MongoDB описують самі себе і не потребують додаткової інформації про себе чи типи даних, що вони зберігають. Це робить MongoDB гнучкою і легко масштабованою, зменшуючи вірогідність помилок у процесах обробки даних та підвищуючи стійкість СУБД.

MongoDB використовує формат JSON для збереження даних, що дозволяє серверу бази даних ефективно обробляти дані без необхідності їх перетворення. Вона також використовує динамічні схеми, які усувають необхідність попередньо визначати структуру даних, що дозволяє зберігати ієрархічні відносини та змінювати структуру записів, додаючи або видаляючи поля.

MongoDB підтримує вставку, автоматичний розподіл і вбудовану реплікацію для кращої масштабованості і високої доступності. Вона поєднує в собі вторинні індекси, запити з діапазонами і сортуванням, агрегацією та геопозиційні запити, що робить її потужною та гнучкою базою даних загального призначення.

MongoDB також має підтримку виконання запитів на сервері за допомогою Javascript коду, а також власну мову запитів, яка дозволяє фільтрувати, сортувати та вибирати дані з них. Це робить MongoDB зручною для взаємодії з більшою кількістю мов програмування, порівняно з MariaDB.

Отже, MongoDB і MariaDB мають свої унікальні переваги та особливості, і вибір між ними залежить від конкретних потреб і вимог проекту.

MongoDB дійсно надає широкий спектр інструментів для роботи з даними, роблячи її справжньою платформою для аналізу та обробки даних. Інтегровані інструменти аналітики дозволяють вам збирати статистику, будувати графіки та візуалізувати дані у реальному часі, що дозволяє ефективно взаємодіяти з вашими даними через візуальний інтерфейс.

Використовуючи MongoDB Atlas, ви можете створити систему баз даних як сервіс (DBaaS), яка буде доступна в будь-якій точці світу завдяки хмаровим технологіям. Цей сервіс також дозволяє використовувати мережу доставки контенту (CDN серверів), щоб підвищити надійність, швидкість та спростити роботу з даними.

У MongoDB масштабування бази даних виконується простіше, порівняно з MariaDB, завдяки обмеженню можливостей шардування та реплікації. Однак, реляційна MariaDB має додаткові можливості, такі як реплікація типу Master-Master та можливість шардування, що дозволяє створювати кластери та розподіляти дані за ключами, хешами або списками.

У реляційній моделі MariaDB при створенні розподіленої бази даних інженеру доводиться враховувати ризик розриву зовнішніх ключів та зміну структури таблиць, що може вимагати значних зусиль та часу. У не реляційній

моделі MongoDB, розподіл колекцій по різних зонах можна виконати навіть без спеціальних знань та технічної підготовки, що спрощує процес.

Реплікація баз даних може бути реалізована у двох основних типах: Master-Slave та Master-Master.

У реплікації типу Master-Slave створюється повна копія (Slave) основної бази даних (Master), до якої постійно копіюються всі дані, що записуються або оновлюються в основній базі. Система читання даних відбувається з копії бази даних, тим самим розподіляючи навантаження між операціями. Копія бази даних не виконує жодних операцій та не має жодного відношення до активної бази даних.

У реплікації типу Master-Master обидві бази даних можуть виконувати ті самі операції. Використання такого підходу призводить до збільшення числа операцій між базами даних, що може уповільнити роботу системи. Запити між системою та базами даних можуть дублюватися програмно або на стороні сервера СУБД.

Шардінг дозволяє розподілити базу даних на частини, щоб кожна з них могла бути розміщена на окремому сервері. Розподілення таблиць або колекцій залежить від структури бази даних та виконується у самій системі. Горизонтальний та вертикальний шардінг - це два основних види шардінгу.

У горизонтальному шардінгу таблиця даних розподіляється на окремі сервери, причому на кожному з них зберігається одна й та сама таблиця, а доступ до неї керується архітектурою системи. Наприклад, запити від непарних користувачів можуть оброблятися одним сервером бази даних, а від парних - іншим. У разі відмови серверу може виникнути втрата даних.

У вертикальному шардінгу таблиці або групи таблиць виділяються на окремий сервер для зниження навантаження у випадку великої кількості запитів в одному потоці.

Зазвичай шардінг використовується разом з реплікацією для забезпечення більшої надійності та швидкості обробки запитів.

Так, розподілення бази даних є ключовою складовою архітектури системи при роботі з великими обсягами даних. На етапі проектування системи важливо передбачати зростання кількості запитів та потоків даних і враховувати це при налаштуванні сховища та розгортанні інфраструктури.

Використання хмарних рішень дозволяє автоматизувати процес масштабування кластера серверів баз даних в залежності від навантаження. Автоматичне масштабування дозволяє ефективно використовувати матеріальні ресурси та сервери, забезпечуючи при цьому стійкість системи до різких змін у кількості запитів.

Додатково, автоматичний режим реплікації і шардінгу створює сприятливі умови для зберігання даних та забезпечує надійність системи в цілому. Після стрибків у трафіку сервери баз даних можуть продовжувати ефективно функціонувати, що важливо для забезпечення стійкої та швидкої роботи системи навіть під час пікових навантажень.

Значним відмінностям між цими системами управління базами даних є їх взаємодія з мовами програмування. Для кожного типу СУБД, який розглядається, існує спеціальний драйвер для кожної мови програмування, яка його підтримує. Цей драйвер відповідає за підключення, взаємодію та виконання програмних інтерфейсів. Наприклад, для PHP, яка підтримує обидві СУБД, існують відповідні драйвери. Кожен драйвер реалізує підключення, сесію для використання команд або інтерфейсів взаємодії, такі як кеш, помилки, оновлення даних і т. д. Для MariaDB розроблено значно більше методів, оскільки вона дозволяє управляти не тільки запитами та потоками даних, але і самою СУБД, підключенням та даними.

У останніх версіях мов програмування необхідно мати останні версії MySQL. Для документно-орієнтованої СУБД драйвер значно простіший, не потребує контролю версій і має лише один параметр логу помилок для налаштування. Перетворення програмних команд у команди для СУБД виконується однаково швидко в обох випадках, оскільки програмне забезпечення завантажує всі драйвери при першому запуску [8].

Важливою характеристикою є безпека запитів, що передаються між системою та базою даних. Оскільки типи запитів та дані, які передаються, різні в обох випадках, захист даних також відрізняється. Якщо з'єднання відкрите та стабільне, є можливість виконання шкідливого скрипту. У PHP є можливість захисту від подібних атак шляхом використання програмних інтерфейсів, які перевіряють стан об'єкту, що зберігає дані з бази даних. Для написання безпечного коду слід використовувати певні структури, які можна знайти у документації.

MySQL має більше інтерфейсів для взаємодії програмного коду та бази даних, оскільки з'явилася раніше. Завдяки постійному з'єднанню можливо спроектувати систему таким чином, що вся СУБД буде управлятися програмним забезпеченням, яке розробляється. Розробнику доступні всі можливості СУБД з програмного коду, які застосовуються до самої бази даних. Недоступні опції - налаштування самої бази даних та усіх внутрішніх процесів, включаючи оптимізатор запитів [9].

3 ДОСЛІДЖЕННЯ ТА ПОРІВНЯННЯ РІЗНИХ БАЗ ДАНИХ

Бази даних складаються з даних, організованих за допомогою певної логічної структури, тоді як система управління базами даних (СУБД) - це комп’ютерна програма, що дозволяє взаємодіяти з цими базами. СУБД використовується для зберігання даних, виконання запитів, контролю за доступом та виконання інших завдань, пов'язаних з управлінням даними. Часто терміни "база даних" та "система управління базами даних" вживаються як синоніми, хоча вони не означають одне і те ж. База даних – це набір даних, які можуть походити з різних джерел, а не тільки бути збережені локально на комп’ютері. СУБД – це програмне забезпечення, що дозволяє користувачам взаємодіяти з базами даних.

Кожна СУБД працює за певною структурною моделлю, яка визначає, як мають зберігатися та доступатися дані. Реляційна СУБД, наприклад, використовує реляційну модель, де дані поділяються на таблиці. У цій системі таблиці називаються взаємозв'язками, які складаються з кортежів (рядків), де кожен кортеж має набір атрибутів (стовпців), які він ділить із іншими рядками.

Основне програмне забезпечення системи управління базами даних (СУБД) виступає як посередник між користувачем та базою даних, забезпечуючи управління даними, механізми та системи баз даних для спрощення організації та маніпуляції інформацією. Функціональність СУБД може бути різноманітною, проте типові характеристики універсальної СУБД включають можливості доступу до метаданих, системну бібліотеку управління, захист від небажаного доступу, незалежність даних, інструменти для безпечного зберігання та моніторингу даних, підтримку транзакцій і переказів, авторизований доступ, дистанційне підключення, відновлення після збоїв, та забезпечення відповідності до законодавчих вимог.

При проектуванні баз даних використовуються методи, які підвищують чіткість та організацію даних, зменшуючи дублікацію та залежність між даними. Це досягається шляхом розділення великих таблиць на менші та встановлення взаємозв'язків між ними. Вихідні дані СУБД можуть включати комплексні SQL запити, вбудовані в систему, які дозволяють користувачам візуалізувати,

обробляти дані та відправляти звіти за допомогою вбудованих процедур, пакетів та тригерів. Oracle, наприклад, розробив пакет для передачі файлів, що дозволяє копіювати двійкові файли між базами даних.

Система управління базою даних приймає команди від адміністратора, керуючи завантаженням, модифікацією або отриманням даних згідно з інструкціями. Серед прикладів СУБД можна виділити хмарні бази даних, системи управління базами даних у пам'яті (IMDBMS), системи управління базами даних стовпців (CDBMS) та NoSQL системи.

Реляційна система управління базами даних (СУБД) та мова структурованих запитів (SQL) спільно створюють потужний інструментарій для ефективної роботи з даними в багатьох сучасних застосуваннях. Ось кілька ключових аспектів, які допомагають зрозуміти їхню роль та функціональність:

- реляційна структура даних СУБД використовує табличний формат для організації даних, ці таблиці включають рядки та стовпці, де кожен рядок представляє одиничний запис, а стовпці - атрибути цих записів, реляційна структура дозволяє ефективно зберігати, змінювати та отримувати дані;
- мова SQL є стандартною мовою для управління та маніпулювання даними в реляційних СУБД, вона дозволяє визначати структуру даних (DDL – Data Definition Language), керувати даними (DML – Data Manipulation Language) та керувати доступом до даних (DCL – Data Control Language);
- можливості обробки даних, реляційні СУБД можуть обробляти значно більші обсяги даних порівняно з неструктурованими або ієрархічними системами, це забезпечується завдяки оптимізації запитів та ефективному управлінню ресурсами;
- транзакційна обробка, реляційні СУБД часто використовують модель ACID для гарантії надійності та інтегральності даних в межах транзакцій, атомність, послідовність, ізоляція та довговічність – ключові

характеристики, які забезпечують правильність транзакцій навіть у випадку системних збоїв;

- розподілена обробка, сучасні реляційні СУБД підтримують розподілені архітектури, що дозволяє розміщувати та обробляти дані на декількох серверах одночасно, це підвищує масштабованість та доступність систем;
- статичний та динамічний SQL, статичний SQL використовується для заздалегідь визначених запитів, де структура SQL-запиту відома на момент компіляції програми, динамічний SQL забезпечує гнучкість, дозволяючи конструювати SQL-запити прямо під час виконання програми, що є особливо корисним у складних або інтерактивних застосуваннях.

Ці характеристики реляційних СУБД та SQL створюють сильну основу для багатьох сучасних додатків, від комерційних інформаційних систем до веб-застосунків і систем великих даних.

Статичний та динамічний SQL представляють два різних способи взаємодії з базою даних у програмах. Обидва мають свої переваги та обмеження, залежно від потреб додатку.

Статичний SQL:

- безпечність, статичний SQL перевіряє синтаксис SQL-запитів на етапі компіляції, що забезпечує вищу безпеку та зменшує ризик виконання невалідного чи шкідливого SQL коду;
- продуктивність, оскільки запити вже оптимізовані та компільовані, статичний SQL може пропонувати кращу продуктивність в порівнянні з динамічним SQL, особливо при частому виконанні однотипних запитів;
- складність управління, зміни у структурі бази даних можуть вимагати змін у програмному коді та повторної компіляції, що може бути незручним у великих системах.

Динамічний SQL:

- гнучкість дозволяє генерувати та виконувати SQL-запити динамічно, адаптуючись до змінних умов та вимог користувачів у реальному часі;
- застосування ідеально підходить для сценаріїв, де запити або структура даних не можуть бути повністю визначені заздалегідь, наприклад, у звітних системах чи складних аналітичних застосунках;
- продуктивність може мати нижчу продуктивність порівняно зі статичним SQL через необхідність постійної підготовки запитів та оптимізації в реальному часі.

Розширений динамічний SQL надає гібридний підхід, поєднуючи гнучкість динамічного SQL з можливістю зберігати підготовлені оператори для подальшого використання. Це може бути корисно у сценаріях, де часто використовуються певні складні запити, але також потрібна можливість швидко адаптувати або змінити ці запити без повного переписування коду.

Кожен із цих методів має своє місце в розробці програмного забезпечення, і вибір між ними залежить від конкретних вимог до додатку, його архітектури та цілей оптимізації продуктивності та безпеки.

Інтерфейс рівня виклику (CLI) для DB2 та Open Database Connectivity (ODBC) є ключовими інструментами для розробників, які бажають інтегрувати взаємодію з базами даних у свої додатки, особливо коли вони працюють з динамічним SQL. Ці інтерфейси дозволяють програмам використовувати SQL-оператори в більш гнучкий та ефективний спосіб, оскільки вони відділені від конкретної платформи бази даних.

Ось декілька переваг використання CLI та ODBC:

- універсальність та переносимість це програми, розроблені з використанням CLI та ODBC, можуть працювати на різних базах даних без змін у коді. Це особливо корисно в гетерогенних середовищах, де можуть використовуватися різні СУБД;
- не потребує попередньої компіляції на відміну від вбудованого SQL, програми, розроблені з використанням CLI та ODBC, не вимагають попередньої компіляції, що спрощує розробку та розгортання;

- прямий доступ до функцій SQL надають CLI та ODBC розширені можливості для виконання SQL-запитів, отримання результатів і взаємодії з базою даних з великою гнучкістю та ефективністю;
- збережені процедури та обробка результатів викликані через CLI, можуть повертати набори результатів, що дозволяє додаткам обробляти ці результати як потрібно, збільшуючи тим самим потужність та гнучкість програм;
- підтримка каталогів дозволяє CLI виконувати запити до системних каталогів, зменшуючи необхідність писати спеціалізовані запити для кожного сервера.

Інтерфейс рівня виклику та ODBC є важливими компонентами в арсеналі сучасного розробника програмного забезпечення, що працює з базами даних. Вони забезпечують не тільки гнучкість і широкі можливості взаємодії, але й сприяють підвищенню продуктивності розробника за рахунок спрощення і стандартизації процесів взаємодії з даними [10].

3.1 PostgreSQL

PostgreSQL, також відомий як Postgres, стверджує, що він є найбільш досконалою серед реляційних баз даних. Ця об'єктно-реляційна система управління базами даних (СУБД) була створена з метою сумісності зі стандартами та можливостями розширень. PostgreSQL має численні функції, такі як успадкування таблиць та перевантаження функцій, які часто пов'язані з об'єктно-орієнтованими базами даних.

Однією з найбажаніших функцій PostgreSQL є можливість паралельної обробки, що дозволяє ефективно обробляти кілька завдань одночасно. Це досягається завдяки впровадженню мультиверсійного контролю паралельності (MVCC), який забезпечує відповідність ACID (атомність, послідовність, ізоляція та довговічність) транзакцій.

Незважаючи на те, що PostgreSQL не такий популярний, як MySQL, для нього існує безліч сторонніх інструментів, які полегшують роботу з ним. До прикладів таких інструментів відносяться Postbird та pgAdmin.

Подібно до MySQL, PostgreSQL підтримує числові, дата/часові та рядкові типи даних. Однак він також підтримує інші типи даних, такі як мережеві адреси, геометричні фігури, бітові рядки, записи JSON, повнотекстовий пошук та інші ідіосинкратичні типи даних.

3.1.1 Переваги PostgreSQL

PostgreSQL прагне досягти вищої відповідності SQL порівняно з MySQL або SQLite. Він підтримує 160 з 179 функцій, необхідних для базового стандарту SQL:2011, а також містить багато додаткових функцій, згідно з офіційною документацією. Крім того, PostgreSQL має велику та дуже віддану спільноту розробників, яка активно підтримує та надає безліч Інтернет-ресурсів для кращого використання системи. Це включає різноманітні форуми, офіційну документацію та Wiki-програму PostgreSQL. Завдяки використанню динамічного завантаження та керування каталогом, користувачі можуть програмно розширювати PostgreSQL на льоту. Файл об’єктного коду може бути призначений для завантаження, якщо це потрібно, що додає системі значної розширюваності [11].

3.1.2 Недоліки PostgreSQL

PostgreSQL має деякі недоліки, які впливають на його продуктивність і популярність. По-перше, кожне нове підключення клієнта у PostgreSQL додає новий процес, що займає пам'ять. Це може призвести до значного використання пам'яті в разі багатьох одночасних з'єднань, особливо в середовищах з інтенсивним читанням даних. По-друге, незважаючи на високий рівень використання, PostgreSQL відстає за популярністю від MySQL. Це може бути пов'язано з меншим доступом до сторонніх інструментів та меншою кількістю адміністраторів баз даних із досвідом роботи з PostgreSQL порівняно з MySQL [12].

3.2 MySQL

MySQL є найпопулярнішою реляційною СУБД з відкритим кодом з моменту, коли сайт рейтингу DB-Engines розпочав відстеження популярності баз даних у 2012 році. Він використовується на деяких з найбільших веб-сайтів у світі, таких як Netflix, Spotify, Twitter та Facebook. Існує велика кількість документації та розробників, а також безліччя Інтернет-ресурсів, пов’язаних із MySQL, що полегшує початок роботи з ним.

MySQL був створений з урахуванням надійності та ефективності, але він не завжди повністю відповідає стандартам SQL. Розробники MySQL постійно працюють над поліпшенням відповідності до стандартів SQL, хоча рівень дотримання може варіюватись. Однією з переваг MySQL є можливість отримати доступ до бази даних через окремий запит до серверу, що сприяє кращому контролю з точки зору доступу до бази даних.

MySQL також породив багато сторонніх додатків та інтегрованих бібліотек, які полегшують роботу з ним та розширюють його функціональність. Деякі з найпопулярніших сторонніх інструментів включають HeidiSQL, phpMyAdmin та DBeaver.

Типи даних MySQL розділені на три категорії: дата/час, рядок та числові типи. Це робить MySQL досить гнучкою і зручною для багатьох різних типів проектів та завдань.

3.2.1 Переваги MySQL

Переваги MySQL включають зручність у використанні та широку популярність. Багата кількість Інтернет-матеріалів та друкованої документації сприяє простоті вивчення та використання цієї системи управління базами даних. Крім того, наявність інструментів, таких як phpMyAdmin, спрощує процес роботи з MySQL.

У справі безпеки MySQL відзначається наявністю сценарію під час встановлення, який поліпшує захист бази даних. Цей сценарій допомагає встановити пароль для кореневого користувача, видаляє бази даних, які доступні

всім користувачам за замовчуванням, та встановлює рівні захисту паролем. Крім того, MySQL дозволяє контролювати параметри доступу до прав користувача, що покращує загальний рівень безпеки.

Щодо швидкодії, MySQL відзначається швидкістю роботи, оскільки розробники визначили пріоритети швидкості із застосуванням стратегії не реалізовувати всі функції SQL. Це допомагає MySQL залишатися однією з найшвидших систем управління базами даних.

Нарешті, MySQL підтримує багаторазову реплікацію, що сприяє доступності, надійності та стійкості до несправностей шляхом обміну інформацією між кількома хостами.

3.2.2 Недоліки MySQL

Серед недоліків MySQL можна виділити наступні: однією з проблем є наявність обмежень через відсутність дотримання вимог SQL. Наприклад, MySQL може мати обмеження у підтримці певних операцій, таких як речення FULL JOIN, що може ускладнити або навіть унеможливити виконання деяких складних запитів та операцій з даними. Після придбання MySQL компанією Sun Microsystems у 2008 році та подальшого переходу до корпорації Oracle у 2009 році, користувачі відзначають більш млявий процес розвитку. Спільнота з відкритим кодом, яка раніше активно реагувала на виправлення проблем та розробку нових функцій, тепер може бути менш активною або навіть не реагувати на потреби користувачів, що може призвести до затримок у вирішенні проблем та впровадженні нового функціоналу. Ці недоліки можуть ускладнювати роботу з MySQL у випадках, коли потрібно виконати складні операції з даними або коли користувачі очікують швидкого розвитку та підтримку функціоналу [13].

3.3 MongoDB

MongoDB – це потужна система керування базами даних (СКБД) типу NoSQL, яка відрізняється своєю гнучкістю та можливостями розширення. Основними принципами MongoDB є гнучкість схеми, горизонтальна

масштабованість та висока швидкодія доступу до даних. MongoDB використовує документо-орієнтовану модель даних, де кожен об'єкт бази даних представляється у вигляді документа у форматі BSON (бінарний JSON).

У порівнянні з традиційними реляційними базами даних, MongoDB дозволяє працювати з набагато більшою кількістю даних із різними структурами. Це дає змогу легко масштабувати базу даних від невеликих проектів до великих та високонавантажених додатків.

Однією з ключових особливостей MongoDB є його масштабованість. Він дозволяє розподіляти дані на кілька серверів для підтримки високої доступності та обробки великих обсягів даних. Крім того, MongoDB використовує гнучкий механізм кластеризації для забезпечення стабільності системи навіть під час збоїв.

Ще однією перевагою MongoDB є його простота використання та розробки. Він надає розробникам інтуїтивний API для роботи з даними, а також широкий набір інструментів для адміністрування та моніторингу бази даних.

MongoDB підтримує багато різних типів даних, включаючи текстові, числові, масиви, вбудовані документи та бінарні дані. Ця гнучкість управління даними робить MongoDB потужним інструментом для розробки різноманітних додатків та веб-систем.

3.3.1 Переваги MongoDB

MongoDB – це потужна система керування базами даних (СКБД), яка володіє численними перевагами. MongoDB дозволяє зберігати дані у вигляді документів JSON, не вимагаючи строго заданої схеми, що дозволяє легко змінювати структуру даних без перепроектування бази даних. Він легко масштабується горизонтально, дозволяючи розподіляти дані по кількох серверах для забезпечення високої доступності та швидкодії. MongoDB використовує вбудований кеш та оптимізовані алгоритми для швидкого доступу до даних, що робить його ідеальним для високонавантажених додатків з великою кількістю запитів. Документо-орієнтована модель даних у MongoDB, яка використовує документи JSON для зберігання даних, спрощує роботу з ними, особливо для

розробників, які працюють з мовами програмування, що підтримують JSON. MongoDB надає зручний API та інструменти для роботи з даними, що робить розробку та адміністрування бази даних простішою. Він підтримує різні режими масштабування, від простого стенда до розподіленої архітектури з безліччю серверів. MongoDB надає багато функцій, включаючи операції агрегації, текстовий пошук, геопросторові запити та багато іншого. Крім того, MongoDB має велику та активну спільноту розробників, яка надає підтримку, розробляє розширення та забезпечує навчальні матеріали для новачків.

3.3.2 Недоліки MongoDB

Хоча MongoDB має численні переваги, вона також має кілька недоліків. MongoDB високо вимоглива до ресурсів, таких як пам'ять та оброблювальна потужність, особливо при обробці великих обсягів даних. Вона має меншу стійкість до відмов порівняно з деякими реляційними базами даних, особливо при великому навантаженні або неправильній настройці. Обмежена підтримка транзакцій до версії 4.0 може ускладнювати управління консистентністю даних. Адміністрування MongoDB може бути складним, особливо для тих, хто не має досвіду роботи з нереляційними базами даних. Обмежена підтримка операції JOIN ускладнює об'єднання даних з різних колекцій. Складнощі зі змінами в структурі даних можуть виникати через відсутність жорсткої схеми та необхідність ручного оновлення кожного документа. Крім того, MongoDB має обмеженість у виборі транзакційних операцій порівняно з традиційними реляційними базами даних [14].

3.4 Висновки до розділу

У світі реляційних баз даних існують різні системи управління базами даних (СУБД), кожна з яких має свої переваги і недоліки. Три найпопулярніших варіанти це PostgreSQL, MySQL та MongoDB кожна з яких займає визначне місце в індустрії та має власні характеристики, які слід враховувати при виборі.

MySQL є популярною реляційною базою даних з широким спектром функцій та підтримкою спільноти. Вона надійна, швидка та досить проста у використанні, що робить її відмінним вибором для багатьох веб-додатків та інших застосувань.

PostgreSQL є потужною реляційною базою даних з високою продуктивністю та широким спектром функцій. Вона підтримує паралельну обробку та має високий рівень стандартизації, що робить її ідеальним вибором для складних та вимогливих застосувань.

MongoDB є популярною NoSQL базою даних, яка використовує документоріентовану модель та гнучкий підхід до зберігання та обробки даних. Вона проста у використанні та масштабуванні, що робить її відмінним вибором для проектів, які потребують гнучкої схеми даних та швидкої розробки.

Загалом, вибір між цими базами даних залежить від конкретних потреб та вимог вашого проекту. Якщо вам потрібна стандартна реляційна база даних з широкою підтримкою та добре вивченою технологією, MySQL чи PostgreSQL можуть бути найкращими варіантами. У той же час, якщо ви шукаєте гнучку та просту у використанні базу даних для неструктурованих або напівструктурованих даних, MongoDB може виявитися кращим вибором.

4 АНАЛІЗ ЧАСОВОЇ СКЛАДНОСТІ МЕТОДІВ ОПТИМІЗАЦІЇ СТРУКТУРИ БАЗ ДАНИХ NOSQL

У розділі представлено детальний опис методів оптимізації структури розподілених і нерозподілених баз даних NoSQL типу "ключ-документ" на основі аналізу метаданих про збережені об'єкти та запити. Розглядаються різноманітні підходи до оптимізації баз даних, включаючи методи для баз даних без вкладених документів та з вкладеними документами. Основна увага приділяється визначенню оптимальної структури колекцій, класифікації даних залежно від вихідних параметрів, а також методам аналізу та оптимізації глибини вкладеності документів. Для розподілених баз даних наводяться методи побудови інформаційного графа запиту та створення ефективної схеми розподілених даних, що дозволяють покращити продуктивність та ефективність обробки запитів.

4.1 Методи оптимізації структури нерозподілених баз даних NoSQL

Бази даних NoSQL (Not Only SQL) представляють собою широкий клас систем керування базами даних, які відрізняються від традиційних реляційних баз даних тим, що не використовують стандартний мову запитів SQL. Ці бази даних використовуються для збереження та обробки великих об'ємів даних, що часто зазнають швидкої зміни або не мають фіксованої структури.

Одним з ключових аспектів проектування баз даних NoSQL є оптимізація їх структури для ефективного зберігання та обробки даних. У цьому контексті виникає необхідність у методах оптимізації, які дозволяють покращити продуктивність та ефективність роботи нерозподілених баз даних NoSQL.

У дослідженні розглянемо різні методи оптимізації структури нерозподілених баз даних NoSQL, включаючи методи визначення структури колекцій, класифікації даних, аналізу вкладених документів та інші. Кожен з цих методів спрямований на покращення продуктивності та ефективності бази даних, дозволяючи забезпечити оптимальне використання ресурсів та забезпечити швидкий доступ до даних [15].

4.1.1 Метод визначення структури колекцій для нерозподілених баз даних типу «ключ-документ»

Цей метод призначений для аналізу даних і визначення найефективнішої структури колекцій в нерозподілених базах даних типу "ключ-документ". Під час аналізу враховуються як кількість об'єктів, так і кількість атрибутів (або властивостей) кожного об'єкта [14].

Нехай n позначає кількість об'єктів, m – коефіцієнт складності аналізу, а O вказує на асимптотичну верхню межу часової складності алгоритму у відношенні до розміру вхідних даних (див.рис. 4.1). Тоді часова складність аналізу даних для цього методу визначається як формула 4.1:

$$T_{analysis} = O(m*\log(n)) \quad (4.1)$$

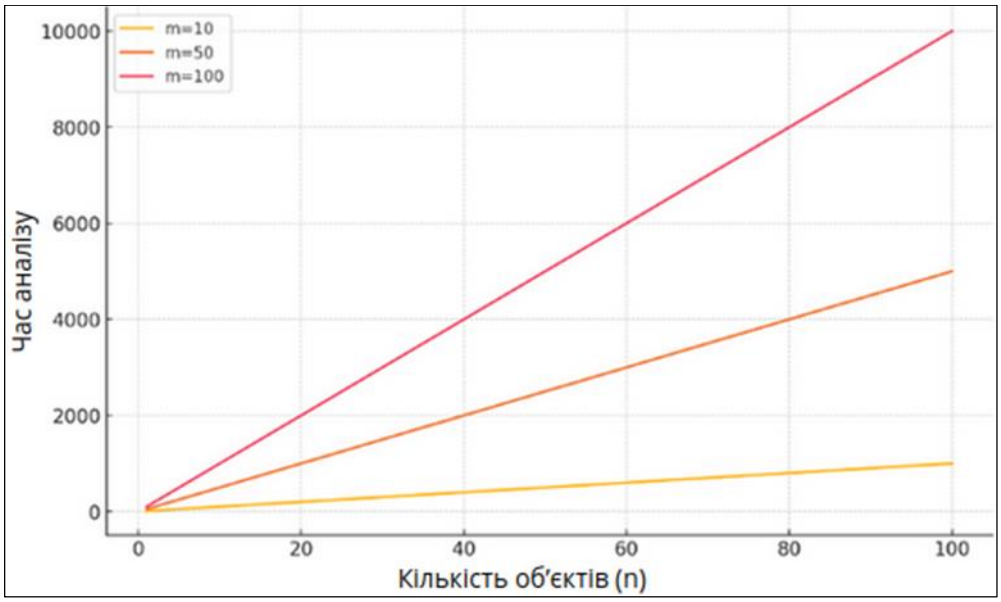

Рисунок 4.1 – Аналіз вихідних даних для структури колекцій

4.1.2 Методика визначення колекцій в базах даних типу «ключ-документ» в залежності від вихідних даних

Цей метод спрямований на класифікацію даних і визначення структури колекцій в базах даних типу "ключ-документ". Під час аналізу дані розбиваються на класи або категорії в залежності від їх характеристик [16].

Нехай n позначає кількість об'єктів, а k – коефіцієнт, що визначає складність класифікації (див. рис. 4.2). Тоді часова складність класифікації даних для цієї методики визначається як формула 4.2:

$$T_{classification} = O(k*n) \quad (4.2)$$

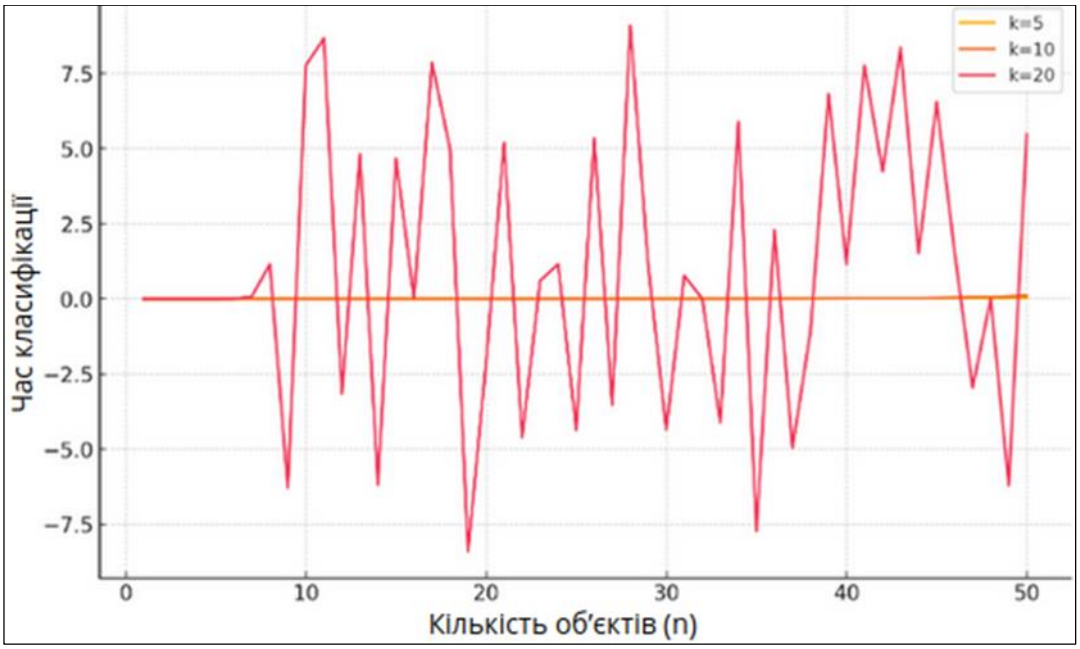

Рисунок 4.2 – Класифікація даних

4.1.3 Метод визначення структури вкладених документів в БД типу «ключ-документ»

Для баз даних типу "ключ-документ", що містять вкладені структури даних, необхідний метод визначення найефективнішої структури цих вкладених документів. При цьому проводиться аналіз глибини вкладеності даних і вибір оптимальної структури для їх зберігання [17].

Нехай d – позначає коефіцієнт глибини вкладеності даних, а n - кількість об'єктів (див. рис. 4.3). Тоді часова складність аналізу вкладеності даних визначається як формула 4.3:

$$T_{nesting_analysis} = O(d*n) \quad (4.3)$$

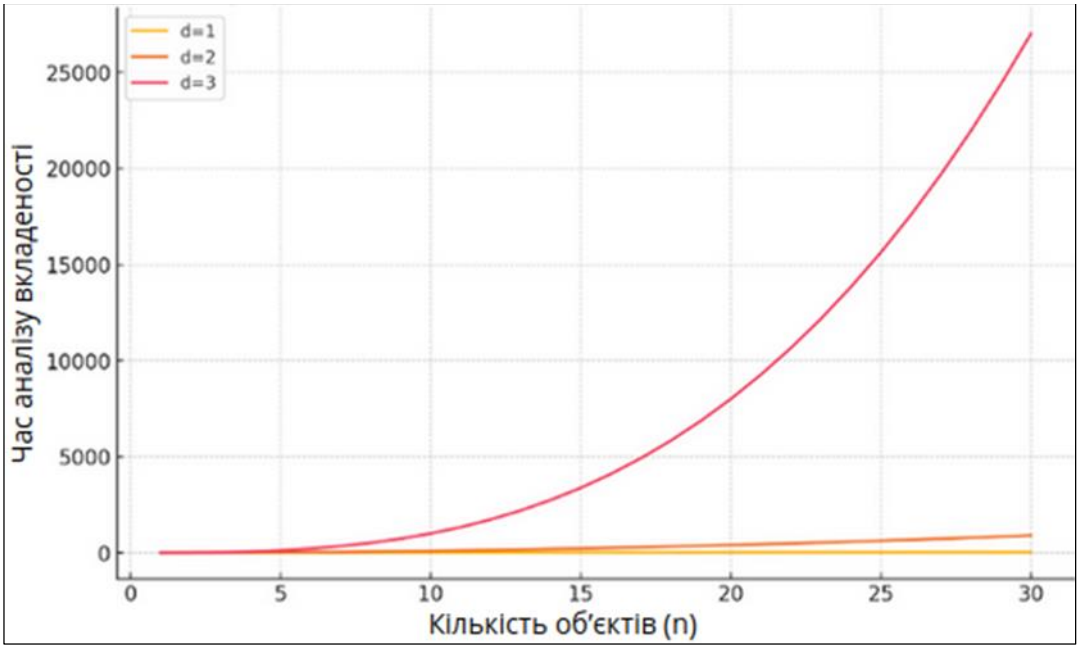

Рисунок 4.3 – Аналіз вкладених документів

4.2 Методи оптимізації структури розподілених баз даних NoSQL

Оптимізація структури розподілених баз даних NoSQL є ключовим етапом у процесі розробки та управління базами даних в сучасних інформаційних системах. Враховуючи великий обсяг даних та потребу в їх швидкому та ефективному доступі, правильно спроектована структура даних дозволяє підтримувати високу продуктивність та масштабованість системи. Методи оптимізації структури розподілених баз даних NoSQL включають в себе різноманітні стратегії та підходи до організації даних, забезпечення їх доступності та збереження консистентності.

Для розуміння і впровадження методів оптимізації структури розподілених баз даних NoSQL важливо враховувати особливості самої системи, потреби користувачів та характеристики даних. Оскільки розподілені бази даних оперують в розподіленому середовищі, де дані можуть розміщуватися на різних вузлах мережі, важливо враховувати такі аспекти, як мережева латентність, доступність та реплікація даних для забезпечення оптимальної швидкодії та надійності системи. Крім того, важливо враховувати масштабованість системи та здатність до росту обсягу даних та навантаження [18].

4.2.1 Метод побудови інформаційного графа запиту для розподіленої реляційної бази даних

Для розподілених реляційних баз даних потрібне побудова інформаційного графа запиту, який дозволяє оптимізувати виконання запитів і покращити продуктивність системи. Цей метод включає аналіз запитів до бази даних та побудову графа залежностей між сутностями та атрибутами даних [8].

Нехай q позначає кількість запитів, а t - коефіцієнт, що визначає складність аналізу запитів (див. рис. 4.4). Тоді часова складність побудови графа запиту визначається як формула 4.4:

$$T_{query_analysis} = O(t*q) \quad (4.4)$$

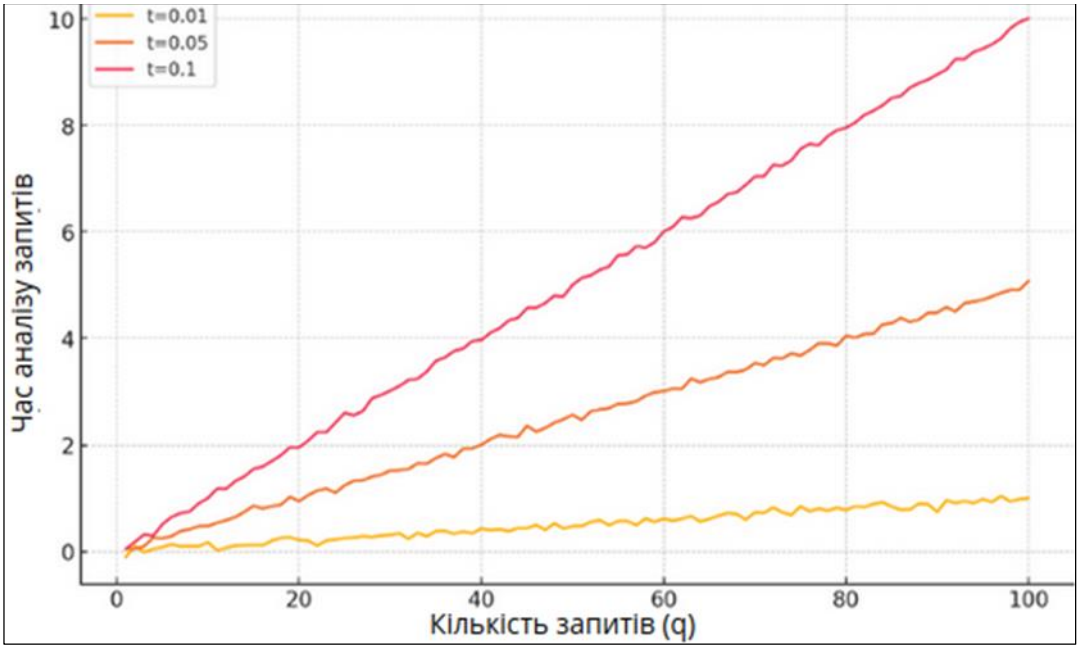

Рисунок 4.4 – Аналіз запитів

4.2.2 Метод побудови схеми розподіленої бази даних NoSQL типу «ключ-документ»

Метод побудови схеми розподіленої бази даних NoSQL типу «ключ-документ» включає аналіз інформації з декількох колекцій та розробку єдиної структури даних, яка дозволяє ефективно зберігати та оптимізувати доступ до

інформації. Цей процес вимагає обробки даних з різних джерел та врахування особливостей кожної колекції [8].

Нехай c позначає кількість колекцій для яких створюється схема (див. рис. 5). Тоді часова складність побудови схеми даних визначається як формула 4.5:

$$T_{schema_creation} = O(c) \tag{4.5}$$

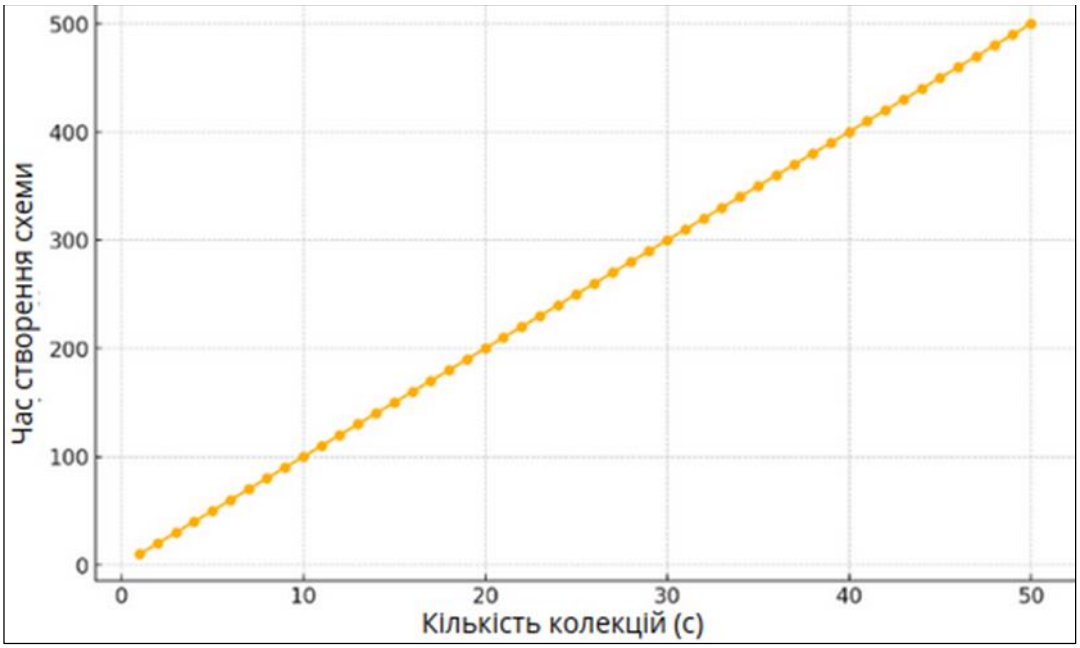

Рисунок 4.5 – Створення схеми даних

4.2.3 Метод оптимізації доступу до даних в розподіленій базі даних NoSQL типу «ключ-документ»

Оптимізація доступу до даних в розподіленій базі даних NoSQL включає в себе процес індексації даних та оптимізацію запитів для підвищення швидкодії та ефективності системи. Цей метод дозволяє швидше виконувати запити до бази даних та зменшити навантаження на систему [8].

Нехай q позначає кількість запитів, де log визначає логарифмічну функцію(див. рис. 6). Тоді часова складність оптимізації доступу до даних визначається як формула 4.6:

$$T_{acess_optimization} = O(\log(q)) \quad (4.6)$$

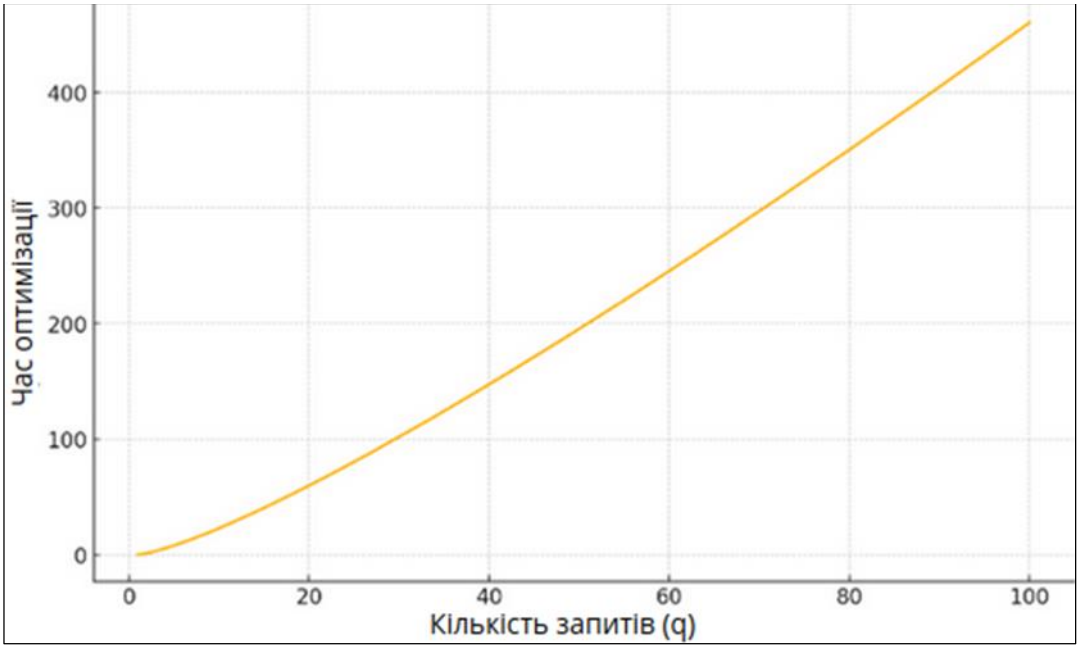

Рисунок 4.6 – Оптимізація доступу до даних

4.3 Висновок

Оптимізація структури баз даних NoSQL є важливим аспектом ефективного управління даними як у нерозподілених, так і в розподілених середовищах. Методи, розглянуті у даній роботі, демонструють різні підходи до оптимізації структури баз даних типу "ключ-документ".

Для нерозподілених баз даних NoSQL ключовими є методи визначення структури колекцій та методи класифікації даних, які дозволяють забезпечити ефективне зберігання та доступ до даних шляхом аналізу початкових даних і їх структурування. Визначення структури вкладених документів також є важливим, оскільки дозволяє більш раціонально організувати дані всередині документів, що, в свою чергу, покращує продуктивність бази даних.

Для розподілених баз даних NoSQL важливими є методи побудови інформаційного графа запитів та створення схеми даних. Ці методи забезпечують ефективний розподіл даних по різних вузлах мережі, що дозволяє оптимізувати час доступу до даних та підвищити надійність системи. Методи оптимізації доступу до даних шляхом індексації та оптимізації запитів також відіграють

важливу роль у забезпеченні високої продуктивності системи при великих обсягах даних.

Загалом, правильне застосування методів оптимізації структури баз даних NoSQL дозволяє забезпечити їхню продуктивність, масштабованість та надійність, що є критично важливим для сучасних інформаційних систем.

5 ПЕРЕТВОРЕННЯ ЗАПИТІВ ІЗ СТРУКТУРИ РЕЛЯЦІЙНОЇ БАЗИ ДАНИХ У ФОРМАТ БАЗИ ДАНИХ NOSQL ТИПУ "КЛЮЧ-ДОКУМЕНТ"

Запити до баз даних NoSQL вважаються більш складними порівняно з реляційними системами управління базами даних. Кожна база даних NoSQL (крім Cassandra) використовує власний формат мови запитів, такий як MongoQL для MongoDB або Cypher для Neo4j. Крім того, складність і різноманіття моделей даних у NoSQL роблять побудову складних запитів більш ускладненою.

Шляхом аналізу й огляду важливості збереження консистентності та точності даних у системах управління базами даних, незважаючи на їхню різнорідність, метод трансляції SQL-запитів у формат NoSQL баз даних, зокрема MongoDB, стає критичним для підтримки функціональності, яка відображається в існуючих реляційних базах даних. Цей підхід дозволяє підтримувати синтаксичну схожість з SQL, забезпечуючи при цьому ефективну обробку та збереження даних в NoSQL-системах.

Більшість баз даних NoSQL спочатку не підтримують можливостей використання мови запиту SQL для управління своїми даними. Тому трансляція запитів з формату SQL в формат NoSQL вважається кращим підходом до цієї проблеми.

У цьому розділі наведено опис методу трансляції запитів з формату SQL в формат MongoDB, який складається з чотирьох кроків: парсинг вихідного запиту SQL, формування словника пропозицій запиту MongoDB, переформування пропозицій словника з урахуванням структури бази даних MongoDB та синтез вихідного запиту. Представлено практичне втілення розробленого методу на мові програмування Java та опис кожного кроку методу. Розглянуто приклад застосування розробленого методу для трансляції різних запитів до бази даних MongoDB з різними варіантами структури бази даних: без вкладених документів та з вкладеними документами.

5.1 Мови запитів до баз даних

У сучасному світі бази даних відіграють ключову роль у зберіганні, управлінні та аналізі інформації. Існує два основних типи баз даних: SQL (реляційні) та NoSQL (нереляційні). Кожен з них має свої унікальні характеристики, переваги та недоліки, що робить їх придатними для різних випадків використання. Розуміння цих відмінностей є критично важливим для вибору правильної бази даних для конкретного проекту. У дипломній роботі розглянені основні особливості SQL і NoSQL баз даних, їх сильні та слабкі сторони, а також приклади використання, щоб допомогти вам зробити обґрунтований вибір, враховуючи потреби вашого проекту.

SQL і NoSQL бази даних мають різні характеристики та використовуються в залежності від вимог проекту. Основні відмінності між ними включають структуру даних, масштабованість, продуктивність, гнучкість і випадки використання.

SQL бази даних, також відомі як реляційні, зберігають дані в таблицях, що складаються з рядків і стовпців. Ці таблиці пов'язані між собою через зовнішні ключі, а для управління та запитів використовується мова SQL. Схема бази даних жорстко визначена, що забезпечує цілісність і структурованість даних, але потребує змін у структурі таблиць при необхідності зміни схеми даних. Сильними сторонами SQL баз даних є підтримка властивостей ACID (атомарність, узгодженість, ізоляція, довговічність), що гарантує надійність і передбачуваність транзакцій, а також потужні можливості для виконання складних запитів і аналітичних операцій. Крім того, SQL бази даних мають довгу історію і широку підтримку спільноти та документації.

Проте, SQL бази даних мають і слабкі сторони. Основною проблемою є обмежена горизонтальна масштабованість, що ускладнює їх використання в системах, що потребують розподілу даних на багато серверів. Також менша гнучкість схеми даних означає, що зміни в структурі даних можуть бути трудомісткими і витратними за часом. Крім того, продуктивність SQL баз даних

може знижуватися при роботі з дуже великими обсягами даних і високим навантаженням.

На відміну від SQL, NoSQL бази даних (нереляційні) зберігають дані в різних форматах, таких як документи, ключ-значення, графи і колоночні структури. Вони пропонують гнучку схему, де кожен документ або запис може мати різну структуру. Основними перевагами NoSQL баз даних є гнучкість схеми даних, що дозволяє зберігати різнорідні дані без необхідності попереднього визначення структури, і легкість горизонтальної масштабованості, що робить їх придатними для управління великими обсягами даних і високим навантаженням. NoSQL бази даних також оптимізовані для швидкого доступу і запису даних, особливо для простих операцій, і підтримують різні моделі даних.

Тим не менш, NoSQL бази даних мають і свої недоліки. Відсутність стандартів призводить до меншої узгодженості в синтаксисі і функціональності між різними NoSQL базами даних. Можливості для виконання складних запитів і аналізу даних у NoSQL баз даних обмежені в порівнянні з SQL. Крім того, не всі NoSQL бази даних підтримують транзакції ACID, що може бути критичним для деяких застосувань. Екосистема NoSQL баз даних менш зріла, що означає менший обсяг документації і підтримки спільноти в порівнянні з SQL базами даних.

SQL бази даних часто використовуються у фінансових і облікових системах, що потребують високої точності і транзакційної цілісності, а також у CRM і ERP системах і в застосуваннях, де необхідний складний аналіз даних і звітність. Прикладами SQL баз даних є MySQL, PostgreSQL, Oracle і Microsoft SQL Server. NoSQL бази даних, навпаки, підходять для великих даних і аналітики в реальному часі, веб-застосувань з високим навантаженням і великою кількістю користувачів (наприклад, соціальні мережі), а також для кешування даних і розподілених систем. Прикладами NoSQL баз даних є MongoDB (документ-орієнтовані), Redis (ключ-значення), Cassandra (колоночні) і Neo4j (графові).

Таким чином, вибір між SQL і NoSQL базами даних повинен базуватися на конкретних потребах проекту, враховуючи обсяг даних, тип операцій, вимоги до гнучкості, продуктивності і масштабованості [18].

5.2 Трансляція запитів з формату PostgreSQL у формат MongoDB

Ціль дослідницької роботи полягає у створенні формалізованих методів для трансляції даних з реляційної бази даних у формат NoSQL типу «ключ-документ», а також в оптимізації її структури та запитів для покращення швидкості обробки та зберігання даних.

У сучасному цифровому світі обробка великих обсягів даних стала не тільки нормою, але й ключовим елементом успіху для багатьох організацій та підприємств різних галузей. Підтримка високої продуктивності баз даних у розподілених обчислювальних мережах важлива для забезпечення швидкого доступу до інформації та ефективного виконання операцій.

Дослідницька робота спрямована на дослідження та оптимізацію продуктивності баз даних у розподілених обчислювальних мережах. Це дослідження стало актуальним через стрімкий розвиток технологій, який супроводжується збільшенням обсягів даних, а також через поширення використання розподілених обчислювальних середовищ у великих корпораціях та малих підприємствах.

Розподілені обчислювальні мережі використовуються для обробки великих обсягів даних, розподілу завдань обробки між різними вузлами та забезпечення доступності даних в різних місцях одночасно. Проте, ефективність роботи баз даних у таких середовищах залежить від різноманітних факторів, включаючи алгоритми реплікації даних, стратегії маршрутизації запитів та механізми синхронізації даних.

Метою даної дослідницької роботи є проведення комплексного дослідження продуктивності баз даних у розподілених обчислювальних мережах та розробка оптимальних методів їх оптимізації. Для досягнення цієї мети передбачається:

- аналіз сучасного стану технологій баз даних у розподілених середовищах шляхом огляду існуючих методів реплікації, розподілення та доступу до даних у розподілених системах, виявлення їх переваг та недоліків;
- експериментальне дослідження продуктивності шляхом виконання серії експериментів для оцінки продуктивності різних алгоритмів та стратегій роботи баз даних у розподілених обчислювальних мережах, а також для виявлення чинників, що впливають на їх ефективність;
- розробка оптимальних підходів до оптимізації на основі отриманих результатів розроблення нові стратегії та методи оптимізації роботи баз даних у розподілених середовищах з урахуванням специфіки завдань та вимог користувачів.

Ця робота має важливе теоретичне та практичне значення. Результати дослідження можуть бути використані для покращення продуктивності баз даних у розподілених обчислювальних мережах, що сприятиме підвищенню ефективності обробки даних та забезпеченню більш високої якості обслуговування користувачів. Враховуючи складність та масштабність проблеми, дослідження, яке виконано в рамках цієї роботи, є кроком до подальших досягнень в області оптимізації обробки даних у розподілених обчислювальних мережах.

Наукова частина дослідницької роботи зосереджена на створенні та описі методу трансформації запитів з SQL до NoSQL. Цей метод реалізований з використанням баз даних PostgreSQL та MongoDB та складається з чотирьох етапів:

- розбір(парсинг) початкового SQL-запиту;
- створення словника з елементів запиту для MongoDB;
- переробка представлень словника з урахуванням структури бази даних MongoDB;
- формування вихідного запиту з елементів словника.

Метод був реалізований на мові програмування Java, а кожен етап був детально описаний. Також розглянуто приклади використання розробленого методу для трансформації різних запитів до бази даних MongoDB з різними структурами даних.

Під час перетворення запитів з однієї мови на іншу можуть виникати різноманітні труднощі, наприклад, відмінності у синтаксисі мов запитів, що наведені в таблиці 5.1. Це зумовлено тим, що кожен тип баз даних має свою власну граматику для обробки даних. Ще однією складністю є різниця у моделях зберігання даних кожної бази даних. Наприклад, реляційні дані зберігаються у вигляді таблиць, які пов’язані між собою, тоді як дані в MongoDB організовані у колекціях, де кожен рядок вважається окремим документом.

Таблиця 5.1 – Порівняння основних об’єктів між PostgreSQL та MongoDB.

Основні об’єкти	PostgreSQL	MongoDB
База даних	База даних	База даних (Database)
Схема	Схема (Schema)	-
Таблиця	Таблиця (Table)	Колекція (Collection)
Рядок	Рядок (Row) або Запис (Record)	Документ (Document)
Стовпець	Стовпець (Column)	Поле (Field)
Індекс	Індекс (Index)	Індекс (Index)
Запит	SQL-запит (SQL Query)	Агрегаційний запит (Aggregation Pipeline)

Ця таблиця дає загальний огляд основних об'єктів в обох системах баз даних і показує їх відповідності. В PostgreSQL та MongoDB є свої власні терміни та підходи до моделювання даних, але багато з них виконують подібні функції.

Далі у процесі перетворення бази даних із формату SQL у формат NoSQL, такий як MongoDB, виконується створення методу автоматичного перекладу

запиту з SQL у формат NoSQL, який задовольняє такі умови: Запит до бази даних MongoDB автоматично буде створений з урахуванням структури використовуваної бази даних MongoDB. Результати виконання запиту в MongoDB після перекладу мають збігатися з результатами виконання вихідного SQL-запиту.

Метод автоматичної трансляції запитів з формату SQL в формат MongoDB, розроблений у рамках дослідницької роботи, враховує структуру баз даних PostgreSQL і MongoDB. Вхідні дані включають код запиту до бази даних PostgreSQL та метадані про їхню структуру. Вихідні дані - це код запиту до бази даних MongoDB.

5.2.1 Парсинг запиту в форматі PostgreSQL

Парсинг, як метод аналізу текстів і команд, відіграє важливу роль у багатьох сферах комп'ютерних наук та програмування. Це процес розбору і структурування вхідних даних, таких як текстові рядки або код, з метою отримання значущої інформації та побудови структурованих уявлень. Цей процес особливо важливий при роботі з мовами програмування та запитів, де сувора синтаксична та семантична перевірка критична для правильного виконання операцій.

Парсинг SQL-запитів є спеціалізованою формою парсингу, що спрямована на обробку та аналіз запитів, написаних мовою SQL (Structured Query Language). Цей процес необхідний для перетворення текстового подання запиту в структуровану форму, яка може бути інтерпретована та виконана системою управління базами даних (СУБД). Завдяки парсингу, можна розділити запит на ключові компоненти, такі як SELECT, FROM, WHERE, GROUP BY, HAVING та ORDER BY, і зрозуміти їх взаємозв'язки та ієрархію.

Парсинг SQL-запитів має безліч практичних застосувань, зокрема:

Оптимізація запитів: Аналіз структури запитів дозволяє оптимізувати їх виконання, знижуючи навантаження на сервер і прискорюючи отримання результатів. Це особливо важливо для великих баз даних з великими обсягами даних і складними запитами.

Аналіз безпеки: Розбір запитів допомагає виявляти потенційні вразливості, такі як SQL-ін'єкції, і запобігати їм. Це критично для захисту даних та забезпечення безпеки інформаційних систем.

Міграція даних: При перенесенні даних між різними системами управління базами даних (СУБД) необхідно розбирати і перетворювати SQL-запити для забезпечення їх сумісності. Парсинг дозволяє автоматизувати цей процес, зменшуючи ймовірність помилок.

Логування та моніторинг: Парсинг запитів дозволяє вести детальний облік і аналіз виконуваних запитів, що корисно для моніторингу продуктивності, аудиту та виявлення аномалій у поведінці системи.

Для ефективного парсингу SQL-запитів використовуються різні інструменти та бібліотеки, серед яких JSQLParser є одним з найпопулярніших серед Java-розробників. JSQLParser надає зручні засоби для розбору та аналізу SQL-запитів, полегшуючи розробку застосунків, що працюють з базами даних.

Основні етапи створення парсера SQL-запитів з використанням JSQLParser:

- створення граматики SQL: Визначення синтаксичних правил для різних конструкцій SQL-запитів, таких як SELECT, FROM, WHERE та інших, що дозволяє автоматизувати процес розбору та забезпечує точну відповідність запитів заданим правилам;
- використання JSQLParser для парсингу або розбору вхідних SQL-запитів і аналізу їх структури, ще JSQLParser надає API для зручної роботи з парсингом та побудовою синтаксичних дерев;
- реалізація парсера включає інтеграцію JSQLParser у проєкт на Java, налаштування обробки вхідних SQL-запитів та отримання необхідних даних, що включає обробку помилок та винятків, котрі виникають при парсингу;
- тестування та відлагодження включає перевірку правильності роботи парсера на різних прикладах SQL-запитів, відлагодження та оптимізація коду для підвищення продуктивності та надійності.

Граматика SQL-запитів є ключовим елементом у процесі парсингу вхідного запиту PostgreSQL. Для парсингу запиту його необхідно розділити на граматичні підзапити PostgreSQL, такі як: SELECT, FROM, WHERE, GROUP BY, HAVING та ORDER BY [21].

Нижче перераховані та схематично описані дії стосовно “розбору” основних запитів (див.рис.5.1-5.5).

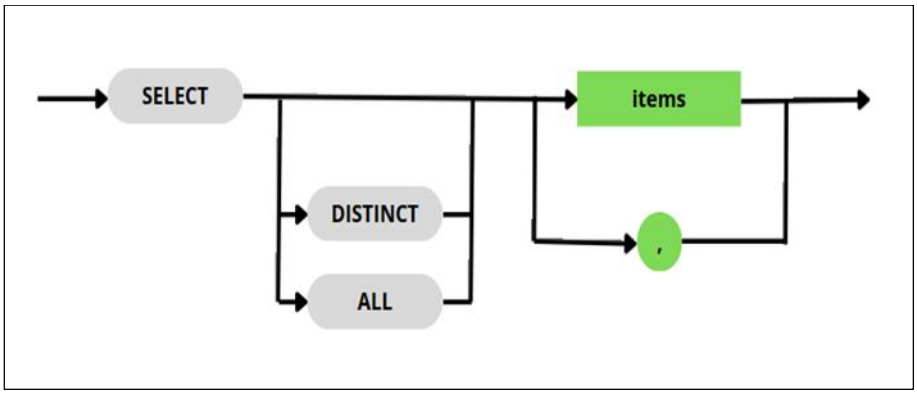

Рисунок 5.1 – Граматика “SELECT”.

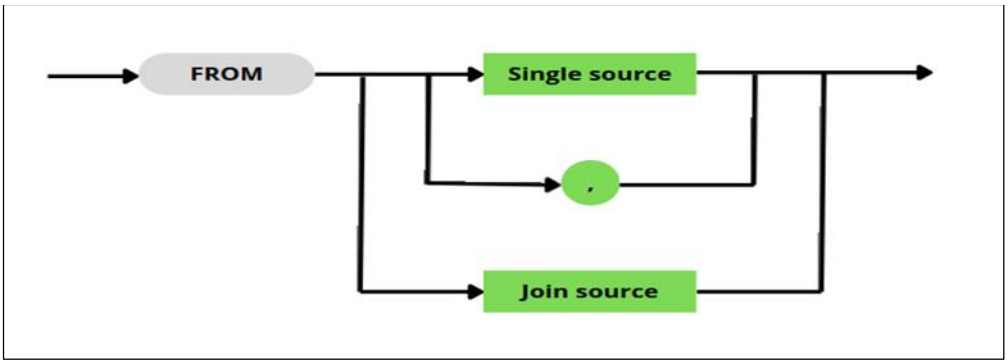

Рисунок 5.2 – Граматика “FROM”.

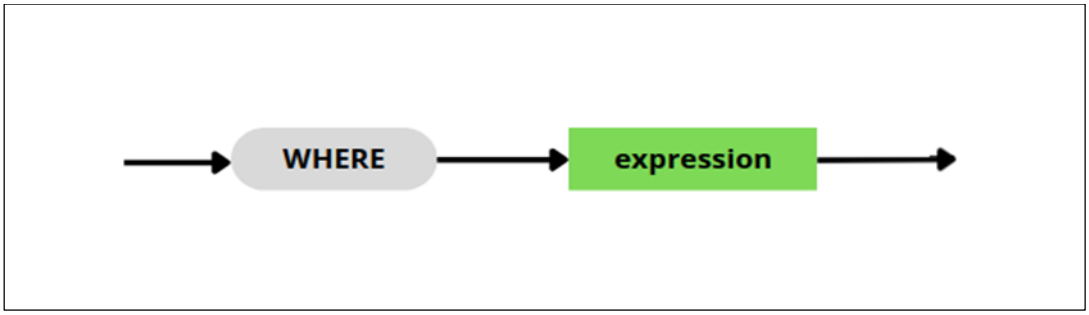

Рисунок 5.3 – Граматика “WHERE”.

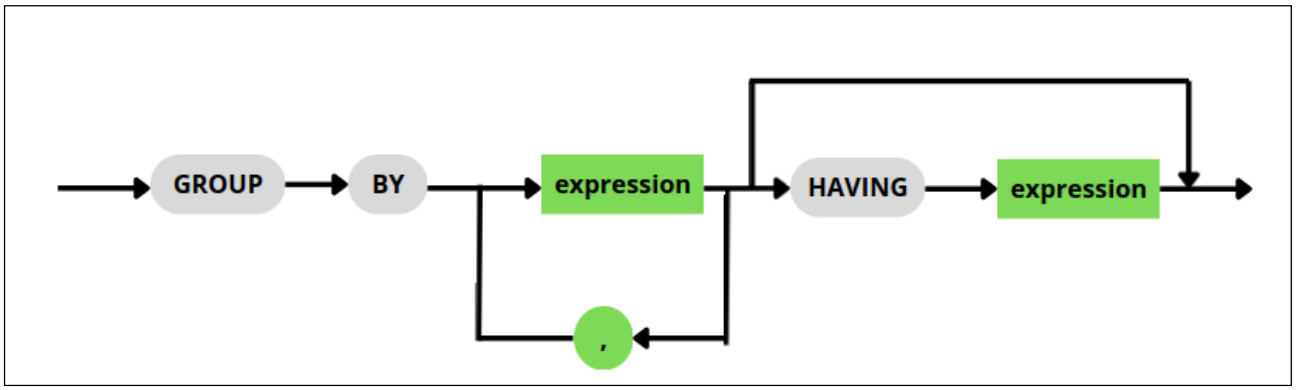

Рисунок 5.4 – Граматика “GROUP BY HAVING”.

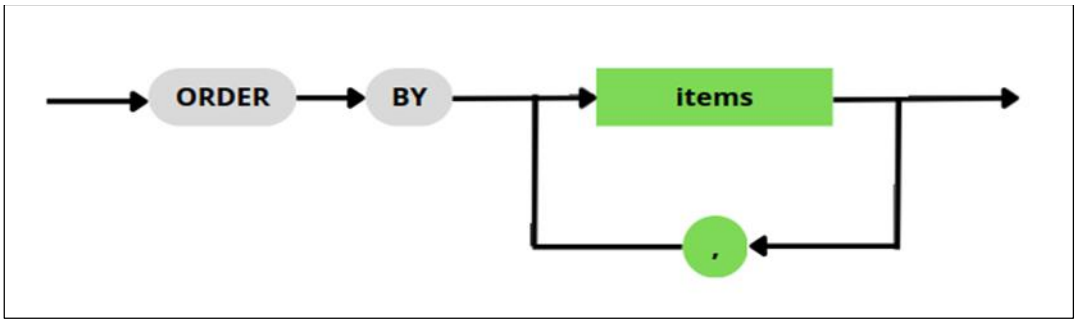

Рисунок 5.5 – Граматика “ORDER BY”.

Приклади парсингу SQL-запитів

Приклад 1:

Запит: "SELECT Customer, Product_name, SUM(Quantity) FROM Orders GROUP BY Customer, Product_name"

Результат парсингу: [SELECT Customer, Product_name, SUM(Quantity) FROM Orders GROUP BY Customer, Product_name]

Приклад 2:

Запит: "SELECT o.Customer, p.Product_name, p.Price FROM Orders o JOIN Products p ON o.Product_id = p.Product_id"

Результат парсингу: [SELECT Customer, Product_name, SUM(Quantity) FROM Orders WHERE Date_order BETWEEN '2023-01-01' AND '2024-12-31' GROUP BY Customer, Product_name]

Приклад 3:

Запит: "SELECT Customer, Product_name, SUM(Quantity) FROM Orders WHERE Date_order BETWEEN '2023-01-01' AND '2024-12-31' GROUP BY Customer, Product_name"

Результат парсингу: [Stage{name='$match', value=Filter{fieldName='$expr', value=Document{{}}}}, Stage{name='$project', value=And Filter{filters=[{"Customer": 1}, {"Product_name": 1}, {"SUM(Quantity)": 1}]}}, Stage{name='$group', id=And Filter{filters=[Document{{_id=$Customer}}, Document{{_id=$Product_name}}]}, fieldAccumulators=[]}]

Ці приклади показують, як можна використовувати функцію parseSQL для розбору різних типів SQL-запитів. Вони ілюструють, як легко можна отримати ключові частини запиту, такі як SELECT, FROM, WHERE, GROUP BY, HAVING та ORDER BY, використовуючи бібліотеку JSQLParser

Таким чином, розробка парсера SQL-запитів представляє важливу та актуальну задачу, вирішення якої сприяє покращенню продуктивності, безпеки та сумісності систем управління базами даних.

5.2.2 Формування словника запитів MongoDB.

У сучасному світі баз даних часто виникає необхідність інтеграції та взаємодії між різними системами управління базами даних (СУБД). Одним із завдань, що виникає при цьому, є конвертація SQL-запитів, які традиційно використовуються у реляційних базах даних, у відповідні запити для нереляційних баз даних, таких як MongoDB. Цей процес потребує детального аналізу та розуміння структури SQL-запитів, а також особливостей синтаксису та функціональності MongoDB.

Формування словника речень для запитів MongoDB передбачає перетворення SQL-запиту у відповідний запит MongoDB. У цій частині буде розглянуто процес трансформації SQL-запитів у MongoDB-запити, використовуючи відповідні методи та синтаксис MongoDB.

Одним із ключових етапів цього процесу є формування словника запиту MongoDB, який містить усю необхідну інформацію для побудови коректного

запиту до нереляційної бази даних. У цьому контексті буде розглянуто, як саме відбувається цей процес, використовуючи конкретні приклади SQL-запитів та їх парсинг.

Формування словника запиту MongoDB починається з аналізу SQL-запиту та виділення ключових компонентів, таких як вибірка полів, джерела даних, умови фільтрації, групування, умови для груп, сортування та з'єднання. Кожен з цих компонентів обробляється окремо та додається до словника з відповідними ключами [22].

Вибірка полів (SELECT).

На цьому етапі визначимо, які саме поля вибираємо з бази даних. Наприклад, у запиті "SELECT Customer, Product_name, SUM(Quantity) FROM Orders" обирано поля Customer, Product_name, а також агрегатну функцію SUM для поля Quantity. Ці поля додаються до словника під ключем "selectFields". Крім простих вибірок, можуть зустрічатися й агрегатні функції (SUM, COUNT, AVG), які також повинні бути коректно оброблені.

Джерело даних (FROM).

Джерело даних визначає, з якої таблиці або колекції було витягнуто дані. У простих випадках це лише одна таблиця, як у запиті "FROM Orders". Однак можуть бути й складніші випадки з підзапитами або об'єднаннями, які теж потрібно враховувати. Джерело даних додається до словника під ключем "fromTable".

Умови фільтрації (WHERE).

Умови фільтрації можуть бути досить складними, включаючи порівняння, логічні операції (AND, OR, NOT), діапазони тощо. Наприклад, у запиті "WHERE Date_order > '2024-01-01'" умова фільтрації вказує на вибірку даних, де дата замовлення пізніше певної дати. Ці умови додаються до словника під ключем "whereCondition".

Групування (GROUP BY).

Групування даних дозволяє об'єднати записи за певними критеріями. У запиті "GROUP BY Customer, Product_name" було згруповано записи за полями

Customer та Product_name. Ця інформація додається до словника під ключем "groupByFields". При цьому важливо враховувати також можливі агрегатні функції, які можуть використовуватися разом з групуванням.

Умови для груп (HAVING).

Умови для груп застосовуються після групування даних. Наприклад, у запиті "HAVING SUM(Quantity) > 10" фільтруються групи за умовою, що сума Quantity повинна бути більше 10. Ці умови додаються до словника під ключем "havingCondition".

Сортування (ORDER BY).

Сортування визначає порядок, у якому будуть повертатися результати запиту. Наприклад, у запиті "ORDER BY name" результати будуть відсортовані за полем name. Інформація про поля для сортування додається до словника під ключем "orderByFields".

З'єднання (JOIN).

З'єднання таблиць дозволяє комбінувати дані з різних джерел. У запиті "JOIN Products p ON o.Product_id = p.Product_id" з'єднуються таблиці Orders та Products за полем Product_id. При цьому можуть використовуватися різні типи з'єднань (INNER JOIN, LEFT JOIN тощо). Інформація про таблиці та умови з'єднання додається до словника під ключами "joinTables" та "joinConditions".

Таким чином, формування словника запиту MongoDB включає детальний аналіз кожного компонента SQL-запиту та коректне перетворення його у відповідну структуру для MongoDB. Це дозволяє забезпечити правильне виконання запиту в нереляційній базі даних, зберігаючи логіку та функціональність оригінального SQL-запиту.

Загально кажучи, при формуванні словника запиту MongoDB необхідно ретельно аналізувати кожну частину SQL-запиту та враховувати всі можливі варіанти його конструкцій для коректного перетворення в структуру даних MongoDB [14].

Приклад 1:

Запит SQL: [SELECT Customer, Product_name, SUM(Quantity) FROM Orders GROUP BY Customer, Product_name]

Результат в MongoDB: {selectFields=[Customer, Product_name, SUM(Quantity)], groupByFields=[Customer, Product_name], fromTable=Orders}

Приклад 2:

Запит SQL: [SELECT o.Customer, p.Product_name, p.Price FROM Orders o JOIN Products p ON o.Product_id = p.Product_id]

Результат в MongoDB: {selectFields=[o.Customer, p.Product_name, p.Price], joinTables=[Products p], fromTable=Orders o, joinConditions=[o.Product_id = p.Product_id]}

Приклад 3:

Запит SQL: [SELECT Customer, Product_name, SUM(Quantity) FROM Orders WHERE Date_order BETWEEN '2023-01-01' AND '2024-12-31' GROUP BY Customer, Product_name]

Результат в MongoDB: {selectFields=[Customer, Product_name, SUM(Quantity)], whereCondition=Date_order BETWEEN '2023-01-01' AND '2024-12-31', groupByFields=[Customer, Product_name], fromTable=Orders}

5.2.3 Переробка представлень словника з урахуванням структури бази даних MongoDB

Переформування пропозицій словника з урахуванням структури бази даних MongoDB є важливим етапом у процесі конвертації SQL-запитів до MongoDB-запитів. Цей крок включає підключення до бази даних, отримання списку колекцій, а також аналіз структури та зв'язків між документами. Це дозволяє правильно синтезувати запити MongoDB на основі словника, сформованого на попередніх етапах.

Перший крок у визначенні структури бази даних MongoDB – це підключення до бази даних. Це можна зробити за допомогою MongoClient, вказавши відповідний рядок підключення та ім'я бази даних. Підключення до бази даних є важливим етапом, оскільки воно дозволяє взаємодіяти з базою даних та

отримувати інформацію про її структуру. Це включає підключення до відповідного MongoDB-сервера, отримання доступу до необхідної бази даних та ініціалізацію об'єктів для взаємодії з колекціями та документами.

Після підключення до бази даних можна отримати список колекцій, щоб зрозуміти, які дані доступні в базі. Це можна зробити за допомогою методу listCollectionNames(), який повертає імена всіх колекцій у базі даних. Отримання списку колекцій дозволяє нам бачити, які саме колекції існують у базі даних, що є аналогом таблиць у реляційних базах даних. Ця інформація важлива для побудови коректних запитів та розуміння, де знаходяться необхідні дані.

Аналіз структури та зв'язків між документами є ключовим для коректної побудови запитів. У MongoDB дані можуть бути вкладеними, що значно відрізняється від реляційної моделі. Наприклад, документи в колекції Orders можуть містити вкладені документи з колекції Products, що впливає на спосіб формування запитів. Для аналізу структури можна використати методи MongoDB для отримання інформації про схему колекцій та вкладені документи. Це може включати перегляд прикладів документів, аналіз полів та типів даних, а також виявлення зв'язків між документами.

На основі аналізу структури та зв'язків можна формувати запити MongoDB, які відповідають словнику запиту. Наприклад, для запиту "SELECT o.Customer, p.Product_name, p.Price FROM Orders o JOIN Products p ON o.Product_id = p.Product_id" можна побудувати MongoDB-агрегацію, використовуючи lookup для з'єднання колекцій.

Переформування пропозицій словника з урахуванням структури бази даних MongoDB є необхідним кроком для правильної конвертації SQL-запитів. Цей процес включає підключення до бази даних, отримання списку колекцій та аналіз структури даних. Це дозволяє створити коректні та ефективні запити, зберігаючи логіку та функціональність оригінальних SQL-запитів.

5.2.4 Формування вихідного запиту з елементів словника до бази даних MongoDB.

На основі попередніх кроків, а саме аналізу та розбору SQL-запитів та створення словника запиту MongoDB, перейдемо до синтезу вихідного запиту MongoDB з використанням утвореного словника. У цьому кроці створимо послідовність операцій агрегації MongoDB, які відповідають структурі вихідного SQL-запиту:

- отримання інформації зі словника включає отримання всієї необхідної інформації зі словника запиту MongoDB, таку як поля для вибору (selectFields), таблицю (fromTable), умови WHERE (whereCondition), поля для сортування (orderByFields) та інші;
- побудова послідовності операцій агрегації будується на основі отриманої інформації було створено послідовність операцій агрегації MongoDB, які відповідають функціональності SQL-запиту, ці операції можуть включати match (для умов WHERE), project (для вибору полів), group (для групування), sort (для сортування) та інші;
- формування запиту MongoDB створюється після побудови послідовності операцій агрегації, які слід об'єднати у один запит MongoDB, цей запит може бути представлений у вигляді списку об'єктів Bson, кожен з яких представляє одну операцію агрегації;
- виконання запиту в MongoDB, що є остаточним можна виконати в базі даних MongoDB за допомогою відповідного драйвера.

Цей процес забезпечує перетворення SQL-запитів у відповідні запити MongoDB, зберігаючи при цьому логіку та функціональність оригінальних запитів.

Для синтезу вихідного запиту до бази даних MongoDB необхідно враховувати структуру та синтаксис мови запитів MongoDB, яка базується на документах JSON. В таблиці 2 наведені приклади запитів, котрі можуть бути використані для виконання будь-яких операцій.

Таблиця 5.2 – Результати трансляції запитів.

Приклад	SQL-запит	MongoDB-запит
1	"SELECT Customer, Product_name, SUM(Quantity) FROM Orders GROUP BY Customer, Product_name"	[Stage{name='$project', value=And Filter{filters=[{"Customer": 1}, {"Product_name": 1}, {"SUM(Quantity)": 1}]}}, Stage{name='$group', id=And Filter{filters=[Document{{_id=$Customer}}, Document{{_id=$Product_name}}]}, fieldAccumulators=[]}]
2	"SELECT o.Customer, p.Product_name, p.Price FROM Orders o JOIN Products p ON o.Product_id = p.Product_id";	[{"$unwind": "$Products p"}, Stage{name='$project', value=And Filter{filters=[{"o.Customer": 1}, {"p.Product_name": 1}, {"p.Price": 1}]}}]
3	"SELECT Customer, Product_name, SUM(Quantity) FROM Orders WHERE Date_order BETWEEN '2023-01-01' AND '2024-12-31' GROUP BY Customer, Product_name"	[Stage{name='$match', value=Filter{fieldName='$expr', value=Document{{}}}}, Stage{name='$project', value=And Filter{filters=[{"Customer": 1}, {"Product_name": 1}, {"SUM(Quantity)": 1}]}}, Stage{name='$group', id=And Filter{filters=[Document{{_id=$Customer}}, Document{{_id=$Product_name}}]}, fieldAccumulators=[]}]

З таблиці 5.2 видно, що в залежності від структури бази даних MongoDB (з вкладеними документами або без них) запити автоматично формувалися по-різному.

5.3 Висновки

У ході розробки конвертера SQL до MongoDB було здійснено кілька важливих кроків, починаючи з аналізу та розбору SQL-запитів, створення словника запиту MongoDB, визначення структури бази даних MongoDB та синтезу вихідного запиту MongoDB. Цей процес дозволяє автоматизувати конвертацію SQL-запитів у відповідні запити MongoDB, що є важливим завданням для розробників, які переходять від SQL до NoSQL баз даних, таких як MongoDB.

Результатом нашої роботи є конвертер, який здатний перетворювати різноманітні SQL-запити у відповідні запити MongoDB, зберігаючи логіку та функціональність оригінальних запитів у новому середовищі бази даних. Це важливий інструмент для розробників, які стикаються з необхідністю перетворення великої кількості SQL-запитів або переходять на платформу MongoDB з SQL-систем.

Завдяки цьому конвертеру розробники можуть зосередитися на розробці програмного забезпечення, не витрачаючи час на рутинні операції перетворення SQL-запитів у формат, придатний для використання з MongoDB. Такий підхід сприяє підвищенню продуктивності та зниженню ризику помилок при міграції бази даних або розробці нових додатків.

6 РОЗРОБКА ТА РЕАЛІЗАЦІЯ ПРОГРАМНИХ МОДУЛІВ

Розробка та реалізація програмних модулів є важливою складовою сучасної інформаційної технології. Модульне програмування дозволяє покращити структуру програмного коду, зробити його більш організованим, підтримуваним і розширюваним. В даному контексті модуль може розглядатися як окрема частина програми, яка виконує певну функцію або набір функцій.

Під час розробки програмних модулів важливо керуватися принципами модульності, які включають у себе розділення функцій на логічно відокремлені блоки, збереження їх незалежності та взаємодії через чітко визначені інтерфейси. Крім того, використання модульного підходу сприяє більшій перевикористовуваності коду, а також полегшує тестування і супровід програм.

У цьому контексті представлений код демонструє процес розробки та реалізації програмного модуля для перетворення PostgreSQL запитів на вирази MongoDB. Код складається з кількох функціональних частин, які послідовно виконують різні етапи перетворення.

Далі буде розглянуто основні кроки цього процесу: від парсингу SQL запитів до створення відповідного запиту MongoDB, використовуючи модульний підхід до розробки програмних модулів.

6.1 Опис програмних модулів мовою Java, що розділені на кроки

Розбір запиту в форматі PostgreSQL [19]:

```
    public static List<String> extractNonKeyTokens(String sqlQuery) {
            List<String> nonKeyTokens = new ArrayList<>();
            try {
                Statement statement = CCJSqlParserUtil.parse(sqlQuery);
                if (statement instanceof Select) {
                    Select selectStatement = (Select) statement;
                    SelectBody selectBody =
selectStatement.getSelectBody();
                    if (selectBody instanceof PlainSelect) {
                        PlainSelect plainSelect = (PlainSelect)
selectBody;
                        // Додавання не ключових токенів з запиту
                        nonKeyTokens.add(plainSelect.toString());
                    }
```

```
                }
            } catch (Exception e) {
                e.printStackTrace();
            }
            return nonKeyTokens;
        }
```

Формування словника пропозицій запиту MongoDB [20]:

```
    public static void populateMongoDBDictionary(String sqlQuery,
MongoDBQueryDictionary dictionary) {
            try {
                Statement statement = CCJSqlParserUtil.parse(sqlQuery);
                if (statement instanceof Select) {
                    Select selectStatement = (Select) statement;
                    SelectBody selectBody =
selectStatement.getSelectBody();
                    if (selectBody instanceof PlainSelect) {
                        PlainSelect plainSelect = (PlainSelect)
selectBody;
                        // SELECT items
                        List<SelectItem> selectItems =
plainSelect.getSelectItems();
                        List<String> selectFields = new ArrayList<>();
                        for (SelectItem item : selectItems) {
                            selectFields.add(item.toString());
                        }
                        dictionary.addSelectFields(selectFields);
                        // FROM table
                        Table fromTable = (Table)
plainSelect.getFromItem();
                        dictionary.addFromTable(fromTable.toString());
                        // WHERE condition
                        Expression where = plainSelect.getWhere();
                        if (where != null) {

dictionary.addWhereCondition(where.toString());
                        }
                        // JOIN conditions
                        List<Join> joins = plainSelect.getJoins();
                        if (joins != null) {
                            for (Join join : joins) {
                                Table joinTable = (Table)
join.getRightItem();

dictionary.addJoinTable(joinTable.toString());
                                Expression joinCondition =
join.getOnExpression();
                                if (joinCondition != null) {

dictionary.addJoinCondition(joinCondition.toString());
                                }
                            }
                        }
                        // GROUP BY items
```

```
                        GroupByElement groupBy =
plainSelect.getGroupBy();
                        if (groupBy != null) {
                            List<Expression> groupByExpressions =
groupBy.getGroupByExpressions();
                            List<String> groupByFields = new
ArrayList<>();
                            for (Expression expr : groupByExpressions) {
                                groupByFields.add(expr.toString());
                            }
                            dictionary.addGroupByFields(groupByFields);
                        }
                        // HAVING condition
                        Expression having = plainSelect.getHaving();
                        if (having != null) {

dictionary.addHavingCondition(having.toString());
                        }
                        // ORDER BY items
                        List<OrderByElement> orderByElements =
plainSelect.getOrderByElements();
                        if (orderByElements != null) {
                            List<String> orderByFields = new
ArrayList<>();
                            for (OrderByElement orderBy :
orderByElements) {
                                orderByFields.add(orderBy.toString());
                            }
                            dictionary.addOrderByFields(orderByFields);
                        }
                    }
                }
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
```

Визначення структури бази даних MongoDB:

```
    public static class MongoDBStructure {
            private final MongoDatabase database;

            public MongoDBStructure(String connectionString, String
dbName) {
                MongoClient mongoClient =
MongoClients.create(connectionString);
                database = mongoClient.getDatabase(dbName);
            }

            public void printCollectionNames() {
                for (String name : database.listCollectionNames()) {
                    System.out.println("Collection: " + name);
                }
            }
```

```
            public MongoDatabase getDatabase() {
                return database;
            }
    }
```

Синтез вихідного запиту [19]:

```
    public static List<Bson> buildMongoDBQuery(MongoDBQueryDictionary
dictionary) {
            // Отримання інформації зі словника
            List<String> selectFields = (List<String>)
dictionary.getQueryDictionary().get("selectFields");
            String fromTable = (String)
dictionary.getQueryDictionary().get("fromTable");
            String whereCondition = (String)
dictionary.getQueryDictionary().get("whereCondition");
            List<String> joinTables = (List<String>)
dictionary.getQueryDictionary().get("joinTables");
            List<String> joinConditions = (List<String>)
dictionary.getQueryDictionary().get("joinConditions");
            List<String> groupByFields = (List<String>)
dictionary.getQueryDictionary().get("groupByFields");
            String havingCondition = (String)
dictionary.getQueryDictionary().get("havingCondition");
            List<String> orderByFields = (List<String>)
dictionary.getQueryDictionary().get("orderByFields");

            // Побудова запиту MongoDB з використанням Aggregates
            List<Bson> pipeline = new ArrayList<>();

            // Можемо розпочати з Match Stage для WHERE умови, якщо вона
є
            if (whereCondition != null) {

pipeline.add(Aggregates.match(Filters.expr(parseCondition(whereCondition)))
);
            }

            // Далі можемо додати Unwind Stages для кожної таблиці в
JOIN, якщо вони є
            if (joinTables != null) {
                for (String joinTable : joinTables) {
                    pipeline.add(Aggregates.unwind("$" + joinTable));
                }
            }

            // Потім йде Projection Stage для вибірки полів
            List<Bson> projectFields = new ArrayList<>();
            for (String field : selectFields) {
                projectFields.add(Projections.include(field));
            }
            pipeline.add(Aggregates.project(Filters.and(projectFields)));

            // Далі Group Stage для GROUP BY
            if (groupByFields != null && !groupByFields.isEmpty()) {
```

```
                List<Bson> groupFields = new ArrayList<>();
                for (String field : groupByFields) {
                    groupFields.add(new Document("_id", "$" + field));
                }
                pipeline.add(Aggregates.group(Filters.and(groupFields)));
            }

            // Додаємо Match Stage для HAVING умови
            if (havingCondition != null) {

pipeline.add(Aggregates.match(Filters.expr(parseCondition(havingCondition))
));
            }

            // Завершуємо з Sort Stage для ORDER BY
            if (orderByFields != null && !orderByFields.isEmpty()) {
                List<Bson> sortFields = new ArrayList<>();
                for (String field : orderByFields) {
                    sortFields.add(Sorts.ascending(field));
                }
                pipeline.add(Aggregates.sort(Filters.and(sortFields)));
            }

            return pipeline;
        }
```

6.2 Тестування програмних частин

Тестування є важливою складовою розробки програмного забезпечення, оскільки воно дозволяє перевіряти правильність роботи коду і підтверджувати, що програма працює відповідно до очікувань. У цьому контексті використовуються юніт-тести, які перевіряють роботу окремих "одиниць" програми, таких як методи чи класи. Давайте розглянемо вступ до теми тестування, використовуючи наведений код тестів:

– тестування методу `testExtractNonKeyTokens()` цілком перевіряє правильність роботи методу `extractNonKeyTokens()`, який вилучає не ключові токени з SQL-запиту, цей тест перевіряє, чи повертається список, чи не є він порожнім та чи містить він правильний токен:

```
    @Test
        public void testExtractNonKeyTokens() {
            // Тестуємо метод, який вилучає не ключові токени
            String sqlQuery = "SELECT Customer, Product_name,
SUM(Quantity) FROM Orders GROUP BY Customer, Product_name";
            List<String> nonKeyTokens =
SQLToMongoDBConverter.extractNonKeyTokens(sqlQuery);
            assertNotNull(nonKeyTokens);
```

```
            assertFalse(nonKeyTokens.isEmpty());
            assertEquals(1, nonKeyTokens.size());
            assertEquals(sqlQuery, nonKeyTokens.get(0));
    }
```

– тестування методу `testPopulateMongoDBDictionary()` аналогічно перевіряє правильність роботи методу `populateMongoDBDictionary()`, який заповнює словник запиту MongoDB на основі SQL-запиту, він визначає, чи словник не є порожнім та чи містить він ключові елементи, такі як "selectFields", "fromTable", "groupByFields" і т.д.:

```
    @Test
        public void testPopulateMongoDBDictionary() {
            // Тестуємо метод, який заповнює словник запиту MongoDB
            String sqlQuery = "SELECT Customer, Product_name,
SUM(Quantity) FROM Orders GROUP BY Customer, Product_name";
            SQLToMongoDBConverter.MongoDBQueryDictionary dictionary = new
SQLToMongoDBConverter.MongoDBQueryDictionary();
            SQLToMongoDBConverter.populateMongoDBDictionary(sqlQuery,
dictionary);
            Map<String, Object> queryDictionary =
dictionary.getQueryDictionary();
            assertNotNull(queryDictionary);
            assertFalse(queryDictionary.isEmpty());
            assertTrue(queryDictionary.containsKey("selectFields"));
            assertTrue(queryDictionary.containsKey("fromTable"));
            assertTrue(queryDictionary.containsKey("groupByFields"));
        }
```

– тестування методу `testBuildMongoDBQuery()` перевіряє правильність роботи методу `buildMongoDBQuery()`, який будує запит MongoDB на основі словника запиту, метод перевіряє, чи будується коректний пайплайн запиту MongoDB, який включає необхідні стадії операцій:

```
    @Test
        public void testBuildMongoDBQuery() {
            // Тестуємо метод, який будує запит MongoDB
            SQLToMongoDBConverter.MongoDBQueryDictionary dictionary = new
SQLToMongoDBConverter.MongoDBQueryDictionary();
            dictionary.addSelectFields(List.of("Customer",
"Product_name", "SUM(Quantity)"));
            dictionary.addFromTable("Orders");
            dictionary.addGroupByFields(List.of("Customer",
"Product_name"));
```

```
            List<Bson>                          pipeline                          =
SQLToMongoDBConverter.buildMongoDBQuery(dictionary);
            assertNotNull(pipeline);
            assertFalse(pipeline.isEmpty());
            assertEquals(2,   pipeline.size());   //   Припускаємо,   що
whereCondition та groupByFields є
        }
```

– передтестові і післятестові налаштування, реалізовані методами `setUp()` і `tearDown()`, встановлюють з'єднання з базою даних перед тестуванням і закривають його після завершення тестування:

```
    @BeforeAll
        public static void setUp() {
            // Налаштовуємо з'єднання з базою даних перед тестуванням
            mongoClient = MongoClients.create(connectionString);
            database = mongoClient.getDatabase(dbName);
        }

        @AfterAll
        public static void tearDown() {
            // Закриваємо з'єднання з базою даних після тестування
            mongoClient.close();
        }
```

– тестування методу `testPrintCollectionNames()` перевіряє правильність роботи методу `printCollectionNames()`, який виводить імена колекцій у базі даних, тест включає перевірку наявності створеної тестової колекції в списку виведених імен колекцій:

```
    @Test
        public void testPrintCollectionNames() {
            // Створюємо тестову колекцію
            database.createCollection("testCollection");

            // Створюємо об'єкт для тестування
            SQLToMongoDBConverter.MongoDBStructure mongoDBStructure = new
SQLToMongoDBConverter.MongoDBStructure(connectionString, dbName);

            // Перевіряємо наявність створеної колекції в списку колекцій
            List<String>                  collectionNames                  =
database.listCollectionNames().into(new ArrayList<>());
            assertTrue(collectionNames.contains("testCollection"));

            // Перевіряємо метод printCollectionNames()
            ByteArrayOutputStream         outContent         =         new
ByteArrayOutputStream();
```

```
        System.setOut(new PrintStream(outContent));
        mongoDBStructure.printCollectionNames();
        assertTrue(outContent.toString().contains("testCollection"));
}
```

– тестування ініціалізації класу `MongoDBStructure` перевіряє правильність створення об'єкту цього класу, який відповідає структурі бази даних MongoDB, метод перевіряє, чи об'єкт створюється без помилок та чи він не є порожнім:

```
@Test
public void testMongoDBStructure() {
SQLToMongoDBConverter.MongoDBStructure mongoDBStructure = new
SQLToMongoDBConverter.MongoDBStructure("mongodb://localhost:27017",
"testDB");
assertNotNull(mongoDBStructure);
}
```

У результаті виконання тестів для класу `SQLToMongoDBConverter` було підтверджено його правильність та ефективність у конвертації SQL-запитів в запити MongoDB. Перший тест перевіряв коректність виділення не ключових токенів з SQL-запиту, що є важливим етапом перед конвертацією. Другий тест визначав, чи правильно заповнюється словник запиту MongoDB, що дозволяє правильно структурувати дані перед їх обробкою. Третій тест перевіряв правильність побудови запиту MongoDB з урахуванням усіх необхідних умов та групувань.

Також, тестування включало перевірку правильності налаштування з'єднання з базою даних перед тестуванням та його коректного закриття після завершення тестів. Це допомагає забезпечити чистоту середовища для кожного окремого тесту та виключити можливі проблеми з підключенням.

У цілому, результати тестів свідчать про те, що клас `SQLToMongoDBConverter` працює належним чином і може бути використаний для успішної конвертації SQL-запитів в формат, зрозумілий для MongoDB, що є важливим кроком у розробці додатків, що використовують різні системи управління базами даних.

ВИСНОВКИ

Дослідження представляє собою глибокий аналіз ключових аспектів оптимізації та вибору баз даних для різноманітних проектів. Шляхом дослідження різноманітних методів та стратегій оптимізації було ретельно дослыджено, як підвищити продуктивність та ефективність систем баз даних.

У дослідженні було звернено увагу на оптимізацію базових структур та алгоритмів, визначаючи їх ключову роль у покращенні швидкодії та ефективності систем. Оптимізація баз даних також включала розгляд різних методів, таких як мікросервіси, кешування та масштабування, кожен з яких має свої переваги та виклики. Дослідження також охопило оптимізацію запитів в розподілених базах даних, де було досліджено вплив різних стратегій на продуктивність системи.

Окрім того, було ретельно досліджено різні типи баз даних та їх застосування. Від реляційних до NoSQL баз даних, проаналізовано основні характеристики кожного типу, його переваги та недоліки. Цей аналіз надав нам можливість зрозуміти, який тип бази даних найбільш підходить для конкретного проекту в залежності від його вимог та обмежень.

В цілому, дослідження проливає світло на складний та важливий аспект розробки програмного забезпечення. Здобуті знання надають можливість зробити інформований вибір при розробці та підтримці баз даних, що відповідають найкращим практикам та вимогам ефективності.

Нарешті, надані в цьому дослідженні висновки та рекомендації стануть корисним ресурсом для інженерів програмного забезпечення, архітекторів систем та всіх зацікавлених сторін, які прагнуть підвищити ефективність та надійність своїх інформаційних систем.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. "Designing Data-Intensive Applications" by Martin Kleppmann (дата звернення: 01.03.2024).

2. "Introduction to Algorithms" by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein (дата звернення: 03.03.2024).

3. Науковий журнал "IEEE Transactions on Knowledge and Data Engineering" (дата звернення: 03.03.2024).

4. "Building Microservices" by Sam Newman (дата звернення: 04.03.2023).

5. Oracle Database Documentation. URL: https://docs.oracle.com/en /database/oracle/oracle-database/index.html (дата звернення: 10.03.2024).

6. Özsu, M. T., & Valduriez, P. (2011). Principles of Distributed Database Systems. Springer (дата звернення: 11.03.2024).

7. "MongoDB: The Definitive Guide" by Kristina Chodorow (дата звернення: 12.03.2024).

8. Наукова конференція VLDB (Very Large Data Bases) (дата звернення: 15.03.2024).

9. "High Performance MySQL" by Baron Schwartz, Peter Zaitsev, Vadim Tkachenko (дата звернення: 17.03.2024).

10. "Database System Concepts" by Abraham Silberschatz, Henry Korth, and S. Sudarshan (дата звернення: 23.03.2024).

11. PostgreSQL Documentation. (2024). SQL Standard Compliance. Retrieved from PostgreSQL Official Documentation (дата звернення: 24.03.2024).

12. Craig, T. (2022). The Art of PostgreSQL. PostgresConf (дата звернення: 28.03.2024).

13. Eric Vanier, Birju Shah, Tejaswi Malepati. Advanced MySQL 8: Discover the full potential of MySQL and ensure high performance of your database. – Packt Publishing, 2019 (дата звернення: 31.03.2024).

14. "MongoDB: The Definitive Guide" by Kristina Chodorow, Shannon Bradshaw (дата звернення: 02.04.2024).

15. "NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence" by Pramod J. Sadalage and Martin Fowler (дата звернення: 04.04.2024).

16. "Couchbase Essentials" by John Zablocki (дата звернення: 05.04.2024).

17. Науковий журнал ACM Digital Library (дата звернення: 05.04.2024).

18. "NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence" by Pramod J. Sadalage and Martin Fowler (дата звернення: 07.04.2024).

19. JSqlParser Documentation URL: https://github.com/JSQLParser/JSqlParser (дата звернення: 10.04.2024).

20. Richardson, C. (2018). Microservices Patterns: With examples in Java (дата звернення: 15.04.2024).

21. Константинов Олександр, Вечур Олександр. «Дослідження та оптимізація продуктивності баз даних у розподілених обчислювальних мережах». Збірник 25ої Міжнародної науково- технічної конференції «Current Trends in the Development of Scientific Research in Today's Conditions» (ICT-2024) Комп'ютерна інженерія секція (дата звернення: 15.05.2024).

22. Chetverikov G., Puzik O., Vechirska I. Multiple-valued structures of intellectual systems //Proceedings of the with Internations Computer Sciences and Information Technologies (CSIT). 2016, 7589907. -pp. 204-207 (дата звернення: 19.05.2024).