МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

**ІНЖЕНЕРНИЙ НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІМ. Ю.М. ПОТЕБНІ
ЗАПОРІЗЬКОГО НАЦІОНАЛЬНОГО УНІВЕРСИТЕТУ**

КАФЕДРА ЕЛЕКТРОНІКИ, ІНФОРМАЦІЙНИХ СИСТЕМ ТА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Кваліфікаційна робота

другий (магістерський)
(рівень вищої освіти)

на тему **Особливості використання документо-орієнтованої та реляційної СКБД**

Виконав: студент 2 курсу, групи 8.1212-іпз-2
спеціальності 121 Інженерія програмного забезпечення
(код і назва спеціальності)
освітньої програми Інженерія програмного забезпечення
(код і назва освітньої програми)
С.А. Магометов
(ініціали та прізвище)
Керівник доцент, к.ф.-м.н. Г. П. Коломоєць
(посада, вчене звання, науковий ступінь, підпис, ініціали та прізвище)
Рецензент директор ТОВ «Алтер Віжн Груп»
.С. Тряпічко
(посада, вчене звання, науковий ступінь, підпис, ініціали та прізвище)

Запоріжжя
2023

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ІНЖЕНЕРНИЙ НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ім. Ю.М. Потебні
ЗАПОРІЗЬКОГО НАЦІОНАЛЬНОГО УНІВЕРСИТЕТУ

Кафедра ______електроніки, інформаційних систем та програмного забезпечення

Рівень вищої освіти другий (магістерський)_________

Спеціальність ___121_Інженерія програмного забезпечення______
(код та назва)

Освітня програма ___Інженерія програмного забезпечення
(код та назва)

ЗАТВЕРДЖУЮ

Завідувач кафедри Т.В. Критська__

“_01_” __вересня_____2023_року

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТОВІ

Магометову Сергію Алановичу
(прізвище, ім’я, по батькові)

1.Тема роботи _Особливості використання документо-орієнтованої та реляційної СКБД

керівник роботи к.ф. - м.н., доцент Г. П. Коломоєць.
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)

затверджені наказом ЗНУ від ____________________________

2. Строк подання студентом кваліфікаційної роботи_______30.11.2023_____

3. Вихідні дані магістерської роботи

- комплект нормативних документів;
- технічне завдання до роботи.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)

- огляд та збір літератури стосовно теми кваліфікаційної роботи;
- огляд та аналіз існуючих рішень та аналогів;
- дослідження особливостей використання документо-орієнтованої та реляційної СКБД;
- створення програмного продукту та його опис;
- перелік вимог для роботи програми;
- дослідження поставленої проблеми та розробка висновків та пропозицій.

5. Перелік графічного матеріалу: 13 слайдів презентації

6. Консультанти розділів магістерської роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата Завдання прийняв

7. Дата видачі завдання 01.09.2023

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів магістерської роботи	Строк виконання етапів магістерської роботи	Примітка
1	Аналіз предметної області	02.09-10.09.23	виконано
2	Формулювання основної задачі дипломної роботи та узгодження її з науковим керівником	11.09-12.09.23	виконано
3	Аналіз існуючих СКБД та мов програмування	13.09-14.09.23	виконано
4	Дослідження СКБД PostgreSQL	15.09-20.09.23	виконано
5	Дослідження СКБД MongoDb	21.09-26.09.23	виконано
6	Порівняння СКБД за характеристиками	27.09-28.09.23	виконано
7	Збір необхідної інформації для розробки застосунків	29.09-13.10.23	виконано
8	Аналіз модулів та вибір стеку для кожного застосунку	14.10-16.10.23	виконано
9	Розробка API та моделі БД	17.10-19.10.23	виконано
10	Розробка бекенд-застосунку для СКБД PostgreSQL	20.10-09.11.23	виконано
11	Розробка бекенд-застосунку для СКБД MongoDb	10.11-17.11.23	виконано
12	Порівняльний аналіз СКБД PostgreSql та MongoDb на базі практичного дослідження	15.11-19.11.23	виконано
13	Оформлення звіту	20.11-30.11.23	виконано

Студент _______________ Магометов С. А.
(підпис) (прізвище та ініціали)

Керівник роботи ________ Коломоєць Г. П.
(підпис) (прізвище та ініціали)

Нормоконтроль пройдено

Нормоконтролер _______________ Скрипник І. А.
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Сторінок: 126

Рисунків: 20

Таблиць: 2

Джерел: 39

Магометов С. А. Особливості використання реляційних та докуме-нто- орієнтованих СКБД : кваліфікаційна робота для здобуття ступеня вищої освіти магістра за спеціальністю 121-«Інженерія програмного забезпечення» / наук. керівник Г.П. Коломоєць. Запоріжжя : ЗНУ, 2023. 126 с.

Мета роботи полягає у порівняльному дослідженні реляційних та документо-орієнтованих систем керуваннями базами та напрацювання рекомендацій до їх вибору при розробці Веб-застосунків різного типу. Порівняльне дослідження виконане для створеної засобами платформи Nodejs серверної частини Веб-застосунку, що використовує розроблене API запитів до бази даних під управлінням PostgreSql або MongoDB системи керування базами даних.

Ключові слова: *API, РЕЛЯЦІЙНА БАЗА ДАНИХ, NOSQL БАЗА ДАНИХ, POSTGRESQL, MONGODB*

SUMMARY

Pages: 126

Figures: 20

Tables: 2

Sources: 39

Magometov S. A. Peculiarities of using document-oriented and relational DBMS : Qualification work for obtaining a master's degree of higher education in specialty 121 “Software engineering”, scientific supervisor, H.P. Kolomoets. Zaporizhzhia : ZNU, 2023. 126 p.

The purpose of the work is a comparative study of relational and document -oriented database management systems and development of recommendations for their selection in the development of various types of Web applications. A comparative study was carried out for the server part of the Web application created by means of the Nodejs platform, which uses the developed API of queries to the database under the control of the PostgreSql or MongoDB database management system.

Keywords: API, RELATIONAL DATABASE, NOSQL DATABASE, POSTGRESQL, MONGODB

ЗМІСТ

ВСТУП 9
1. Актуальність теми 9
2. Мета і завдання дослідження 10
Об’єкт дослідження 11
Предмет дослідження 11
Методи дослідження 11
Наукова новизна одержаних результатів 12
Практичне значення одержаних результатів 12
Апробація результатів кваліфікаційної роботи магістра 12
Глосарій 12
РОЗДІЛ 1 ДОСЛІДЖЕННЯ особливостей різних типів БД та їх використання в контексті веб-застосунків 15
1.1 Обґрунтування теми 15
1.2 Типи NoSQL СКБД та їх характеристика 16
1.3 Приклади задач для документно-орієнтованих баз даних 17
1.4 Приклади задач для реляційних баз даних 18
1.5 Обґрунтування обраних БД 19
1.6 Огляд обраних баз даних 23
1.6.1 Огляд PostgreSQL 23
1.6.2 Огляд MongoDB 25
РОЗДІЛ 2 ОСНОВИ СКБД ТА ВИЗНАЧЕННЯ критеріїв ЇХ порівняння 28
2.1 Формування Баз Даних 28
2.2 Реляційні бази даних та їх роль у веб розробці 30
2.3 NoSql бази даних та їх роль у веб розробці 32
2.4 Масштабування баз даних 33
2.5 Контейнерізація БД 35
2.6 Критерії порівняння 35
2.4.1 Модель даних 36
2.4.2 Гнучкість схеми даних 37
2.4.3 Транзакційна підтримка 39

2.4.4 Продуктивність та масштабованість 40
2.4.5 Аналіз мов запитів 42
2.4.6 Спільнота та підтримка 42
2.7 Теоретичне порівняння PostgreSQL та MongoDb 43
РОЗДІЛ 3 реалізація моделі БД конфігурація бекенду та створення API .. 45
3.1 Моделювання бази даних 45
3.2 Засоби реалізації 47
3.2.1 Мова програмування JavaScript 47
3.2.2 Платформа Nodejs 48
3.2.3 Фреймворк Express 49
3.2.4 CORS 50
3.2.5 Dotenv 51
3.2.6 Morgan 52
3.2.7 Mongoose 53
3.2.8 PG 54
3.2.9 Середовище розробки Visual Studio Code 55
3.2.10 Середовище розробки DataGrip 57
3.2.11 MongoDb Compass 58
3.2.12 Docker 59
3.2.13 Postman 61
3.2.15 Система контролю версій Git 62
3.2.16 MongoDb Atlas 63
3.3 Початкова конфігурація бекенд-застосунку 64
3.3.1 Головний файл app.js 65
3.3.2 Маршрутизація 69
3.4 Компоненти бекенду для MongoDb 71
3.5 Реалізация моделі бази данних в MongoDb 74
3.6 Опис CRUD операцій до MongoDb бази даних 78
3.7 Створення реляційної бази даних PostgreSql 84
3.8 PostgreSQL у Docker: Аналіз docker-compose.yml 87
3.9 Підключення серверної програми до PostgreSQL 88

3.10 Опис CRUD операцій у реляційній базі даних 89
3.11 Транзакції в реляційних БД та NoSql 94
РОЗДІЛ 4 ЕКСПЕРИМЕНТАЛЬНІ ДОСЛІДЖЕННЯ 98
4.1 Дослідження методів реалізації моделі даних 98
4.2 Дослідження гнучкості схеми даних.. 99
4.2.1 Дослідження тразакційної підтримки 101
4.3 Дослідження масштабованості 103
4.4 Аналіз мов запитів.. 105
4.5 Аналіз спільноти, підтримки та документації.................................. 106
4.7 Дослідження продуктивності.. 107
4.7.1 Підготовка до дослідження 107
4.7.2 Дослідження продуктивності MongoDb 110
4.7.3 Дослідження продуктивності PostgreSql 114
ВИСНОВКИ 120
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ.. 120

ВСТУП

Актуальність теми

У світі розробки програмного забезпечення виникає дедалі більше питань вибору типу системи керування базами даних (СКБД) для забезпечення ефективного і надійного зберігання даних. Особливо гостро це питання стоїть у контексті порівняння систем керування реляційними базами даних (RDBMS) та систем керування нереляційними базами (NoSQL DBMS). Кожна з таких систем вирішує завдання забезпечення персистентності даних, але має свої особливості, переваги та недоліки, що завдає складнощів у виборі оптимального варіанту. Питання "що краще використовувати?" набуває особливої актуальності у зв'язку зі зростанням обсягів даних, переходом до розміщення інформації у розподілених системах та розширенням можливостей технологій роботи з базами даних.

Вибір між системами керування реляційними та NoSQL базами даних визначається вимогами проекту. Реляційні СКБД, такі як MySQL [9] або PostgreSQL [2], широко застосовуються у класичних веб-застосунках з монолітною архітектурою та корпоративних системах, де структуровані дані та зв'язки між ними відіграють важливу роль. Однак з розвитком технологій розробки Веб та мобільних застосунків, збільшенням обсягів даних, які обробляються у реальному часі NoSQL СКБД, такі як MongoDB [19] або Cassandra [17], набули популярності. Такі системи дозволяють ефективно зберігати та обробляти неструктуровані дані, а також забезпечити відносно легке їх горизонтальне масштабування [5] при організації роботи з великими обсягами інформації.

Проблематика вибору між згаданими вище підходами полягає у необхідності збалансувати вимоги до консистентності, доступності та поділу (CAP-теорема [39]). Реляційні бази даних забезпечують сильні гарантії консистентності, але з цим пов'язана висока міра складності та обмеження в масштабованості. З іншого боку, NoSQL рішення надають більшу

доступність і гнучкість у роботі з неструктурованими даними, але можуть вести до меншої консистентності.

Додатково, проблема вибору між реляційними та NoSQL-базами даних актуалізується через стрімке зростання обсягів даних та необхідність забезпечення швидкого та ефективного доступу до них. Реляційні бази даних, з одного боку, пропонують сильну структуру та можливість використовувати складні SQL-запити для отримання даних з різних таблиць. З іншого боку, NoSQL-системи надають можливість працювати з великими обсягами неструктурованих даних, що часто виникає в сучасних проектах.

Однією з ключових аспектів є також гнучкість схеми даних. У реляційних базах даних схема фіксована і має бути строго дотримана для кожної таблиці. За необхідності змін, може виникнути необхідність виконання складних операцій міграції схем, що може бути затратним. У NoSQL базах даних, таких як MongoDB, схема є гнучкою і може змінюватися в процесі розвитку проекту без значних обмежень.

Ще однією важливою проблемою в контексті обрання між цими двома типами баз даних є вартість та складність обслуговування. Реляційні системи часто вимагають більше ресурсів та трудовитрат для налагодження та підтримки. У той час як NoSQL бази даних можуть бути менш витратними у відношенні до обладнання та можуть легше масштабуватися горизонтально за допомогою розподіленої архітектури.

Усе це створює ситуацію, коли не існує універсальної відповіді на питання "що краще використовувати?". Обираючи між реляційними та NoSQL-базами даних, розробники повинні враховувати усі вищезазначені фактори, враховуючи специфіку свого проекту, його масштаб, потреби щодо швидкодії та гнучкості. Все це робить проблематику вибору типу бази даних однією з найбільш важливих та актуальних в галузі розробки програмного забезпечення сьогодні.

Мета і завдання дослідження

Метою роботи є порівняльне дослідження Особливості використання реляційних та документо-орієнтованих СКБД.

Відповідно до мети були поставлені такі завдання:

1. Розробка серверної частини веб-застосунку, що працює з даними СКБД PostgreSql або MongoDB.
2. Дослідження моделі даних: порівняння структур даних та підходів до моделювання даних реляційних і документо-орієнтованих СКБД.
3. Гнучкість схеми: вивчення можливостей змін структур даних, що зберігають СКБД.
4. Підтримка транзакцій: дослідження способів організації транзакції та ефективності їх виконання.
5. Масштабованість: вивчення горизонтального масштабування СКБД кожного типу.
6. Аналіз мов запитів: порівняння синтаксису, можливостей та швидкодії запитів різного типу.
7. Спільнота і підтримка: порівняння підтримки обох СКБД та їх документації.

Об'єкт дослідження

Серверна частина веб-застосунку пошуку авіаквитків, що працює з даними СКБД PostgreSql або MongoDB.

Предмет дослідження

Кількісні та якісні характеристики, а також зручність та оптимізованість засобів, що надають PostgreSql та MongoDB СКБД.

Методи дослідження

Для розв'язання представлених завдань використовуються такі методи дослідження:

1. Аналіз інформаційних джерел про різні СКБД
2. Проведення аналогії з-поміж існуючих СКБД.
3. Порівняльний аналіз особливостей СКБД.
4. Синтез отриманих результатів досліджень.

Наукова новизна одержаних результатів

Наукова новизна даного дослідження полягає в аналізі ефективності та проблем, пов'язаних із використанням реляційних та нереляційних документо-орієнтованих СКБД. Виявлені підходи до оптимізації запитів і роботи з великим обсягом даних у реляційних базах даних, а також підвищення гнучкості схем у документо-орієнтованих базах.

Практичне значення одержаних результатів

Розглянуті особливості використання СКБД обох типів в реальних проектах і бізнес-середовищах розкривають практичний аспект наукового дослідження. Визначення найкращих випадків використання для кожного типу СКБД допомагає підвищити ефективність розробки та продуктивність, а також зменшити витрати.

Апробація результатів кваліфікаційної роботи магістра

Результати роботи було представлено на науково-технічної конференції студентів, аспірантів і молодих вчених «Молода наука -2023».

Глосарій

1. База даних (Database): Систематично організована колекція даних, яку можна легко зберігати, оновлювати та витягувати за допомогою комп'ютера.
2. Масштабування (Scalability): Здатність системи збільшувати свою продуктивність або обсяг обробки даних для відповіді на зростаюче навантаження чи обсяг користувацьких запитань.

3. Реляційні бази даних (Relational Databases): Тип баз даних, де дані зберігаються у вигляді табличних структур, а взаємозв'язки між ними визначаються за допомогою ключів.

4. Документо-орієнтовані бази даних (Document-Oriented Databases): Системи управління базами даних, де дані представлені у вигляді документів (наприклад, JSON або BSON), а взаємозв'язки можуть бути представлені вбудованими структурами.
5. MongoDB: Популярна документо-орієнтована база даних, яка використовує BSON для представлення даних та дозволяє гнучко керувати схемою.
6. SQL: Мова структурованих запитів, яка використовується для взаємодії з реляційними базами даних.
7. NoSQL: Група баз даних, які не використовують стандартні SQL-запити та можуть включати документо-орієнтовані бази даних.
8. CAP-теорема: Теорема, яка стверджує, що в розподіленій системі неможливо одночасно забезпечити консистентність (Consistency), доступність (Availability) і стійкість до розбиття (Partition Tolerance).
9. Індекс (Index): Структура даних, яка прискорює швидкість пошуку записів в базі даних, роблячи їх доступними за значенням конкретного поля.
10. ACID: Акронім для властивостей транзакцій у реляційних базах даних - Атомарність (Atomicity), Спростовність (Consistency), Ізольованість (Isolation), Довершеність (Durability).
11. Шардування (Sharding): Розподіл даних на різні сервери чи вузли для оптимізації масштабованості та покращення продуктивності.
12. MapReduce: Модель програмування та обчислення для обробки та генерації великих обсягів даних паралельно на розподіленому середовищі.
13. DynamoDB: Спрощена та повністю керована Amazon NoSQL база даних, яка надає високу доступність та масштабованість.

14. JSON (JavaScript Object Notation): Легкий формат обміну даними, який часто використовується в документо-орієнтованих базах даних.

15. Горизонтальне шардування (Horizontal Sharding): Техніка розподілу даних, при якій великий обсяг інформації розділяється на групи (шарди) на різних серверах чи вузлах. Кожен шард функціонує незалежно, обробляючи лише частину даних. Горизонтальне шардування спрямоване на поліпшення масштабованості та розділення навантаження між різними частинами системи, що дозволяє ефективно працювати з великими обсягами даних.

РОЗДІЛ 1 ДОСЛІДЖЕННЯ ОСОБЛИВОСТЕЙ РІЗНИХ ТИПІВ БД ТА ЇХ ВИКОРИСТАННЯ В КОНТЕКСТІ ВЕБ-ЗАСТОСУНКІВ

1.1 Обґрунтування теми

Бази даних є важливим елементом сучасної розробки програмного забезпечення, і вибір відповідного інструменту для роботи з ними є вирішальним завданням. У світі існує розмаїття систем керування базами даних (СКБД), і визначення оптимального вибору може викликати багато питань. Розглядаючи питання використання реляційних та документо-орієнтованих баз даних, можна виявити ключові особливості обох підходів.

Реляційні бази даних, такі як PostgreSQL, ґрунтуються на принципі табличної організації даних і використовують мову SQL для операцій з ними. Вони відомі своєю здатністю до структурованого зберігання та обробки даних, що дозволяє ефективно працювати зі зв'язками між різними частинами інформації. Однак, в роботі з неструктурованими даними або в умовах великого обсягу інформації, реляційні бази даних можуть виявитися менш гнучкими та менш продуктивними.

Документо-орієнтовані бази даних, такі як MongoDB, працюють на основі схеми документів у форматі JSON або подібних. Цей підхід надає велику гнучкість, особливо в роботі з неструктурованими даними, де кожен об'єкт може мати власну структуру. Вони ідеально підходять для сучасних завдань, пов'язаних з аналітикою даних, обробкою великих обсягів інформації та розробкою динамічних систем.

Обговорення особливостей використання реляційних та документо-орієнтованих баз даних актуальне не лише з точки зору веб програмування, але й у загальному контексті розробки програмного забезпечення. Визначення, коли і як використовувати кожен з цих типів баз даних, є ключовим для досягнення оптимальної продуктивності, ефективності та легкості супроводження програмних проектів.

Зараз існує два типи баз даних, а саме Sql і NoSql бази даних. Сьогодні NoSql бази даних є популярним напрямом у середовищі веб розробки з багатьох причин, хоч цей напрямок виник відносно давно, все ще NoSql бази даних продовжують розвиватися і ця тема залишається актуальною.

1.2 Типи NoSQL СКБД та їх характеристика

Нереляційні бази даних (NoSQL) є різновидом систем керування базами даних, які відрізняються від традиційних реляційних баз даних. Ці системи зазвичай призначені для роботи з великими обсягами неструктурованих або напівструктурованих даних. Основні типи NoSQL баз даних включають:

- **Ключ-значення (Key-Value) бази даних**:
  Характеристика: Кожен запис у базі даних має унікальний ключ і пов'язане з ним значення. Це найпростіший тип NoSQL баз даних.
  Приклади: Redis, DynamoDB.
- **Документо-орієнтовані бази даних**:
  Характеристика: Дані зберігаються у вигляді документів, зазвичай у форматі JSON або BSON (бінарний JSON). Кожен документ може містити в собі вкладені об'єкти або масиви.
  Приклади: MongoDB, CouchDB.
- **Стовпчато-орієнтовані бази даних:**
  Характеристика: Дані організовані у вигляді стовпців, а не рядків, що робить їх ефективними для операцій, де важливо вибіркове вилучення даних з кількох стовпців.
  Приклади: Apache Cassandra, HBase.
- **Графові бази даних:**
  Характеристика: Дані представлені у вигляді графів, де вузли представляють об'єкти, а ребра визначають взаємозв'язки між ними.
  Приклади: Neo4j, OrientDB.
- **Масиво-орієнтовані бази даних:**

Характеристика: Спеціалізовані для роботи з масивами даних. Зазвичай використовуються для аналізу великих обсягів даних.

Приклади: Apache Accumulo, SciDB.

Оскільки сьогодні робота з Json файлами є основною в розробці будь-якого додатка, а також особливо розробка веб-застосунків, то документо-орієнтовані бази даних є наймасовішими у використанні в порівнянні з іншими типами БД. Тому для порівняння з реляційними БД я взяв саме документо-орієнтований тип нереляційної бази даних.

1.3 Приклади задач для документо-орієнтованих баз даних

Документо-орієнтовані бази даних (DODB) відрізняються унікальною структурою та підходом до зберігання та обробки даних. Однією з ключових особливостей є їхня гнучкість схеми, яка дозволяє зберігати дані без жорстко визначеної структури [11]. Ця функція зробила СКБД ефективною для вирішення специфічних завдань, таких як керування змінними та неструктурованими даними.

Прикладом завдання для документо-орієнтованих баз даних є робота з даними, які мають змінну структуру [13]. Наприклад, у медичних системах, де можливі різні формати записів для різних типів медичних даних, СКБД дозволяють легко додавати нові поля або типи даних без необхідності змінювати загальну структуру бази даних.

Іншим важливим завданням є обробка напівструктурованих даних, таких як дані у форматі JSON або XML. СКБД дають можливість ефективно зберігати та обробляти ці дані, забезпечуючи швидкий доступ і зручний механізм для пошуку та фільтрації великої кількості інформації.

Іншим типовим завданням для баз даних, орієнтованих на документи, є керування даними для великих обсягів вмісту з різною структурою. Наприклад, у системах керування великими обсягами мультимедійних файлів або структурованих документів, де кожен файл може мати власні унікальні

атрибути, СКБД дозволяють ефективно зберігати та обробляти ці дані без необхідності заздалегідь визначати суворі схеми.

Ще одна задача — робота з даними в реальному часі та великою кількістю операцій читання та запису [1]. СКБД часто використовуються в системах миттєвого сповіщення, соціальних мережах або системах онлайн-комерції, де необхідно швидко зберігати та отримувати дані в реальному часі, а також забезпечувати гнучкість схеми для змін даних під час розробки проекту.

Крім того, СКБД ефективно використовуються для аналізу та обробки великих обсягів даних, де звичайні реляційні моделі можуть бути нестабільними або непрактичними через необхідність частих змін структури даних.

1.4 Приклади задач для реляційних баз даних

Одним із типових завдань для реляційних баз даних є керування даними з чіткою та чіткою структурою. У випадках, коли важливо забезпечити узгодженість і точність даних, особливо в бізнес-додатках або фінансових системах, РСКБД відображають їх ефективність, дозволяючи визначати таблиці, поля та зв’язки між ними заздалегідь.

Типовим завданням є використання реляційних баз даних для складних аналітичних операцій. Завдяки мові SQL і можливості використання операцій JOIN RDB дозволяють виконувати складні запити та отримувати консолідовані дані, що важливо для бізнес-аналітики та прийняття стратегічних рішень.

Реляційні бази даних також чудово справляються із завданням керування транзакціями, забезпечуючи атомарність, послідовність, ізоляцію та надійність (властивості ACID) [22]. Це особливо важливо у сферах, де важливо усунути помилки та забезпечити надійність даних, наприклад, медицина чи фінанси.

Використання реляційних баз даних є ефективним для роботи з великим обсягом структурованих даних, де здатність забезпечувати зв'язки між різними таблицями та виконувати пошук за допомогою запитів має вирішальне значення для ефективного управління цими даними.

Важливим завданням для реляційних баз даних є підтримка інтеграції та обробки даних з різних джерел. РБД дозволяють встановлювати зв'язки між таблицями для об'єднання інформації з різних джерел, що робить їх ефективними в ситуаціях [14], коли необхідно узгодити та агрегувати розрізнені дані.

Також важливим завданням є використання реляційних баз даних для забезпечення безпеки даних. Системи управління реляційними базами даних забезпечують механізми контролю доступу, шифрування даних і резервного копіювання, забезпечуючи високий рівень захисту конфіденційної інформації.

Ключовим завданням є оптимізація продуктивності для великої кількості транзакцій. Реляційні бази даних зазвичай використовуються в тих сферах, де важливі швидкість і низький відсоток збоїв, наприклад банківські системи або інтернет-магазини.

1.5 Обґрунтування обраних БД

Для даного дослідження я звернувся до рейтингів Stack Overflow Developer Survey 2023 (Рис. 5) [31]. Даний рейтинг визначає популярність тієї чи іншої СКБД, підраховуючи кількість запитів до кожної з них і проводячи опитування професіоналів і новачків по всьому світу. З діаграми можна побачити що найпопулярнішою NoSql базою даних є MongoDb в той же час PostgreSql являєтся однією з самих популярних БД взагалі та займає перше місце у різних рейтингах багато років. На рисунку 1 зображений рейтинг найпопулярніших БД за версією Stack Overflow.

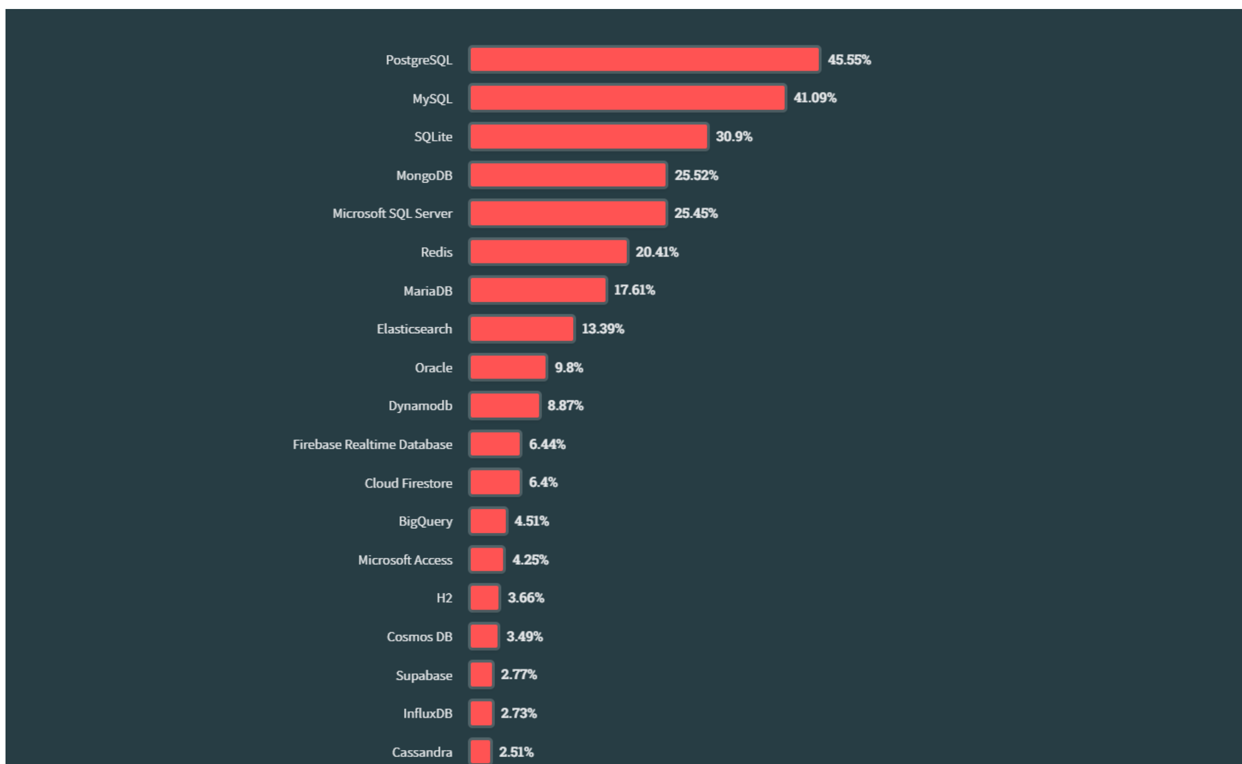

Рисунок 1 — Рейтинг БД 2023

Безумовними лідерами у рейтингу БД, які займають перші три місця це реляційні БД, в той час, як MongoDB на 4 місці. Це говорить про те, що в більшості для різних проектів використовують реляційні БД, але кожен рік рейтинги оновлюються, тому якщо у 2022 році MongoDb був на 5 місці, то сьогодні MongoDb використовується частіше ніж Microsoft Sql, і це вже може говорити нам про те, що популярність документо-орієнтованих БД, зокрема MongoDb, зростає з кожним роком. Також хочеться відзначити, що існує безліч інших баз даних, які були побудовані за принципом нереляційних БД, наприклад, таких як Redis, CouchDB і Dynamodb. Всі вони працюють за одним принципом, але завдання можуть виконувати різні та по-різному.

Якщо говорити про ці нереляційні БД, то неможливо і згадати ті екосистеми та інфраструктуру разом з якою вони використовуються. Можна навести досить простий приклад, як Redis. Практично Redis є швидким сховищем ключ значень і найчастіше використовується для зберігання деяких даних користувача після реєстрації. Ця технологія завжди застосовується у парі з такою технологією як RabbitMq. За допомогою RabbitMq

реалізуються черги запитів між мікросервісами, тоді як сама технологія мікросервісів є найпопулярнішою у веб-розробці великих та високонавантажених додатків. У такому випадку можна сміливо говорити про те, що фактично Redis зобов'язаний своєю популярністю мікросервісної архітектури і зокрема RabbitMq [32], оскільки він найчастіше використовується в парі з Redis.

MongoDB не має такого сильного "партнера" як RabbitMq. Фактично MongoDb є класичним прикладом нереляційних БД, розробка якої відбувається без прив'язки будь-яким іншим технологіям, але при цьому відмінно працює разом з усіма популярними бібліотеками і фреймворками для різних мов програмування.

MongoDb можливо не лідер серед усіх БД але безумовний лідер серед усіх нереляційних БД на сьогоднішній день, тому серед нереляційних БД я вибрав для порівняння саме MongoDB.

Серед реляційних баз даних я вирішив вибрати найпопулярнішу реляційну БД – PostgreSql. Окрім лідируючого місця в рейтингу на моє рішення вплинули інші важливі фактори та переваги PostgreSql, такі як:

1. Відкритий вихідний код: PostgreSQL є open-source, що означає, що він безкоштовно поширюється та має активну спільноту розробників. Це дозволяє користувачам отримати доступ до вихідного коду, вносити зміни та створювати доповнення, що сприяє появі широкого спектру розширень та інструментів.
2. Повна підтримка SQL: PostgreSQL повністю відповідає стандартам ANSI SQL, що полегшує переносимість даних та запитів між різними базами даних.
3. Масштабованість та продуктивність: Він пропонує можливості масштабування як вертикально, так і горизонтально. Завдяки можливості роботи на кластерах та використанню різних методів реплікації, PostgreSQL забезпечує високу продуктивність та масштабованість для великих обсягів даних.

4. Розширюваність: PostgreSQL підтримує безліч доповнень (extensions), що дозволяє розширювати його функціональність шляхом додавання нових можливостей та типів даних.
5. Надійність та стійкість: Завдяки механізмам транзакцій, контролю цілісності даних та механізмам відновлення, PostgreSQL забезпечує високу надійність та стійкість даних.
6. Підтримка різних типів даних: PostgreSQL надає широкий спектр вбудованих та розширюваних типів даних, включаючи географічні дані, JSON, XML та інші.
7. Підтримка збережених процедур і тригерів: Це забезпечує можливість створення функцій користувача, тригерів, а також роботи з процедурами, що спрощує обробку даних на рівні бази даних.
8. Спільнота та підтримка: Існує активна спільнота розробників та користувачів PostgreSQL, яка надає велику документацію, форуми підтримки та регулярні оновлення.

Ці переваги роблять PostgreSQL привабливим вибором для широкого кола додатків, від невеликих стартапів до великих корпоративних систем, де потрібна надійність, продуктивність та гнучкість в управлінні даними.

Для того, щоб остаточно переконатися в правильності вибору, я вирішив подивитися в інший рейтинг Google Trends, який показує кількість запитів за кожен рік по обох базах даних, вибудовуючи графік починаючи з 2004 року, закінчуючи сьогоднішнім днем [33]. З нього ми можемо помітити, що як у випадку з рейтингом Stack Overflow різниця залишається, але в період з початку до кінця 2021 року MongoDB був популярніше ніж PostgreSql. На рисунку 2 зображений графік рейтингу популярності обох БД.

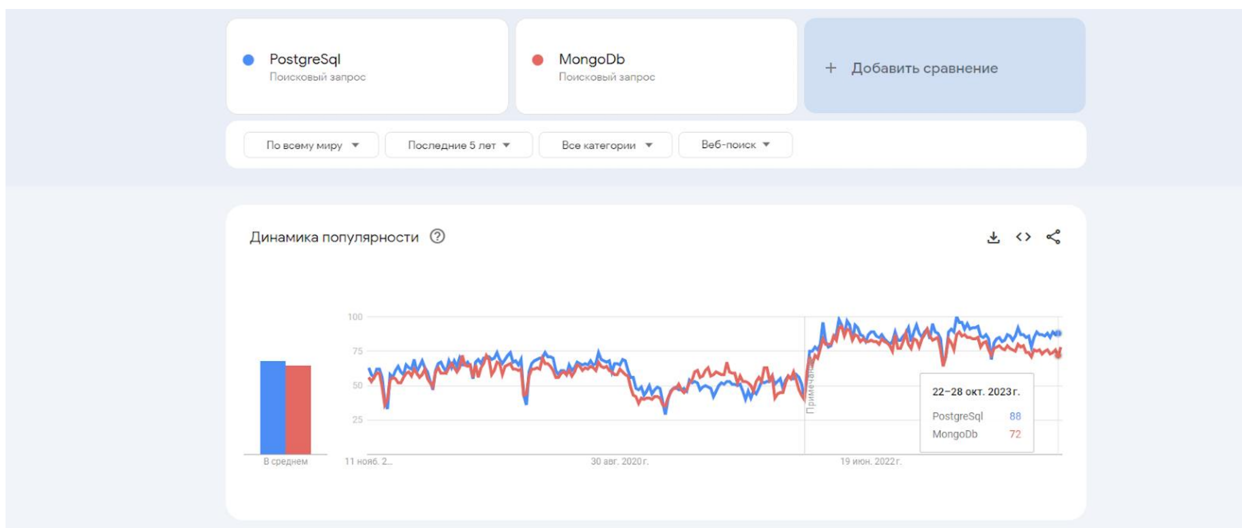

Рисунок 2 — Рейтинг MongoDB vs PostgreSql 2023

Отже виходячи з даних попереднього дослідження, а також рейтингів Google і StackOverflow, я вважаю, що я взяв дві найпопулярніші бази даних серед документо-орієнтованих БД і реляційних баз даних. З їхньою допомогою ми можемо вивчити докладніше світ документо-орієнтованих БД і світ реляційних баз даних, також дізнатися які особливості у тому чи іншому підході, у чому є кардинальні відмінності, а в чому подібність.

1.6 Огляд обраних баз даних

У сучасному світі бази даних для використання на різних платформах вимагає важливого вибору оптимальних технологій. Нижче наведено декілька ключових БД, які здатні ефективно працювати в умовах розробки веб-застосунків: PostgreSQL, MongoDB, SQLite, Firebase, Realm, MySQL, CouchDB, Microsoft SQL Server, DynamoDB, RethinkDB, Cassandra.

1.6.1 Огляд PostgreSQL

PostgreSQL – це потужна та високоефективна об'єктно-реляційна система управління базами даних (СКБД), яка надає надзвичайну гнучкість та функціональність для ефективного управління даними. Розроблена спільнотою відкритих розробників, PostgreSQL є вільною та відкритою

платформою, що забезпечує необмежені можливості для розробників та адміністраторів баз даних [12]. На рисунку 3 зображені основні переваги PostgreSQL.

Рисунок 3 — Основні переваги PostgreSQL

Початково створена університетом Каліфорнії в Берклі, PostgreSQL відрізняється своєю активною спільнотою та постійними оновленнями, що робить її з найсучасніших СКБД на ринку. Вона підтримує стандарт SQL та має багато додаткових функцій, які виходять за рамки стандарту, такі як розширення для геоданих, текстовий пошук та обробка JSON-даних.
Однією з ключових переваг PostgreSQL є його велика гнучкість у роботі з ефективними типами даних. Система підтримує реляційну модель, а також дозволяє зберігати та опрацьовувати неструктуровані дані. Це робить ідеальним вибором для проектів з різними вимогами до обробки даних.

PostgreSQL славиться надійністю та масштабованістю. Вона може легко обробляти великі обсяги даних і запитів, забезпечуючи стабільну роботу під навантаженням. Крім того, система має високий рівень безпеки, що робить її популярним вибором для проектів, де захист даних є високо пріоритетним завданням.

Повна підтримка транзакцій та механізми контролю версій виробляють PostgreSQL відмінним вибором для бізнес-застосунків, де важлива консистентність та надійність даних. Вона також підтримує асинхронний реплікаційний механізм, що забезпечує можливість створення резервних копій та підвищення доступності системи.

PostgreSQL є відкритою системою, що означає обмеження ліцензійних обмежень та використання вартості. Це робить її доступною для широкого кола користувачів і дозволяє використовувати її в проектах будь-якого розміру та напрямку [15].

1.6.2 Огляд MongoDB

MongoDB – це документо-орієнтована NoSQL база даних, яка користується широкою популярністю у сфері веб-розробки. Її історія розпочалася у 2007 році, коли компанія 10gen (нині MongoDB Inc.) розробила систему, яка поєднувала гнучкість неструктурованого зберігання даних та масштабованість. Перша версія MongoDB була випущена в 2009 році, і з того часу вона стала однією з найпоширеніших баз даних NoSQL.

Особливістю MongoDB є його документо-орієнтована модель даних. Замість традиційної таблиці та рядків РБД MongoDB зберігає дані у вигляді гнучких документів у форматі BSON (бінарно кодований JSON). Це дозволяє розробникам зберігати та отримувати дані у природному форматі, що полегшує розробку та інтеграцію з додатками [20].

Однією з переваг MongoDB є його гнучкість. Схема бази даних може бути змінена динамічно, що дозволяє додавати та видаляти поля в документах без необхідності міграції даних. Це особливо корисно в проектах, що швидко змінюються, де потрібна гнучкість при роботі з даними.

MongoDB також забезпечує високу продуктивність та масштабованість. Завдяки горизонтальному масштабуванню та розподіленій архітектурі, MongoDB здатна обробляти великі обсяги даних та витримувати високе навантаження. Вона також підтримує шардування, що дозволяє розподілити дані на кілька серверів та забезпечити балансування навантаження.

Однак, MongoDB має деякі недоліки. Через свою гнучку схему даних, MongoDB не забезпечує жорстку узгодженість даних і транзакційність, як традиційні реляційні бази даних. Це може бути обмеженням для певних типів додатків, які потребують точності та цілісності даних.

В цілому MongoDB є потужним інструментом для веб-розробки, який забезпечує гнучкість і масштабованість при роботі з неструктурованими даними. Його документо-орієнтована модель, висока продуктивність та гнучкість зробили його популярним вибором для багатьох додатків та проектів веб-розробки.

Іншою перевагою MongoDB є багатий набір функціональних можливостей. Ця СКБД підтримує потужну мову запитів, індекси для покращення продуктивності запитів, агрегаційні пайплайни для обробки та аналізу даних, а також реплікацію та шардування для забезпечення відмовостійкості та масштабованості [34].

Незважаючи на безліч плюсів, MongoDB має і деякі обмеження та недоліки. Зокрема, під час роботи з великими обсягами даних MongoDB може вимагати більше пам'яті та ресурсів, ніж реляційні бази даних. Також важливо враховувати, що MongoDB не є універсальним рішенням для всіх типів програм та сценаріїв використання. У деяких випадках, особливо коли потрібна сувора узгодженість даних або складні транзакції, традиційні реляційні бази даних можуть бути більш підходящим вибором. На рисунку 4 зображені основні переваги MongoDb.

Рисунок 4 — Основні переваги MongoDB

РОЗДІЛ 2 ОСНОВИ СКБД ТА ВИЗНАЧЕННЯ КРИТЕРІЇВ ЇХ ПОРІВНЯННЯ

2.1 Формування Баз Даних

Історія баз даних сягає своїм корінням в середину XX століття. Тоді комп'ютери тільки починали розвиватися, і з появою електронних машин виникла потреба в ефективному зберіганні та керуванні великими обсягами даних. Перші бази даних були засновані на ідеях та концепціях, закладених до теорії реляційних баз даних, запропонованої Едгаром Коддом у 1970 році [8].

Реляційні бази даних (РБД) стали домінуючим стандартом для зберігання та управління даними. Їхні основні принципи роботи включають структурування даних у таблиці з використанням ключів, зв'язування таблиць між собою за певними правилами та використання мови SQL для виконання запитів та маніпуляції даними. РБД стали широко поширені та використовувалися в різних галузях, включаючи банківську справу, бізнес та наукові дослідження.

Однак з розвитком Інтернету та веб-застосунків наприкінці XX століття виникла потреба у зберіганні та обробці неструктурованих даних, таких як текст, зображення, аудіо та відео. У відповідь на це з'явилися NoSQL бази даних, які відрізнялися від реляційних баз гнучкістю зберігання та обробки різнорідних даних. NoSQL бази даних використовували різні моделі даних, такі як ключ-значення, документо-орієнтована, стовпцева та графова моделі [18].

Сьогодні бази даних стали невід'ємною частиною багатьох сфер діяльності, включаючи соціальні мережі, електронну комерцію, медицину та багато іншого. Вони продовжують розвиватися та вдосконалюватися, пропонуючи нові функціональності, покращену продуктивність та можливості масштабування. Технології, такі як розподілені бази даних, бази даних у

пам'яті та хмарні бази даних, стають все більш популярними та затребуваними в сучасному світі.

Бази даних стали невід'ємною складовою інформаційного суспільства, і їхня еволюція продовжується. З розвитком нових технологій, таких як штучний інтелект, великі дані (Big Data) та Інтернету речей (IoT), бази даних стикаються з новими викликами та вимогами. Наприклад, великі обсяги даних, що генеруються сенсорами IoT, вимагають спеціальних рішень для їх зберігання та аналізу. Також, у контексті розвитку машинного навчання та аналітики даних, бази даних стають основою для обробки та вилучення цінної інформації з великих наборів даних [29].

Сучасні бази даних прагнуть забезпечення високої продуктивності, масштабованості і відмовостійкості. Розподілені бази даних, такі як Apache Cassandra та Apache Hadoop, дозволяють обробляти та зберігати дані на кластері серверів, забезпечуючи високу доступність та стійкість до відмов [17]. Бази даних у пам'яті, такі як Redis та Memcached, дозволяють швидко обробляти дані, прискорюючи процеси читання та запису.

Важливим трендом у світі баз даних є хмарні бази даних. Хмарні сервіси, такі як Amazon Web Services (AWS) та Microsoft Azure, надають готові інфраструктурні рішення для зберігання та обробки даних, з можливістю автомасштабування та гнучкої оплати в міру використання. Це спрощує розгортання та управління базами даних, а також знижує витрати на інфраструктуру [35].

В цілому, бази даних продовжують еволюціонувати, щоб відповідати зростаючим вимогам та викликам сучасного світу. Вони відіграють важливу роль у обробці та зберіганні даних, забезпечуючи надійність, ефективність та гнучкість для широкого спектру додатків та галузей. І з розвитком технологій та появою нових потреб, бази даних продовжуватимуть розвиватися пропонуючи ще більш потужні та гнучкі рішення для управління даними.

2.2 Реляційні бази даних та їх роль у веб розробці

На початку 1990-х років РБД почали застосовувати у веб-розробці. На той час веб-сайти були переважно статичними, і інформація зберігалася у файлах. Однак з появою веб-браузерів, які дозволили користувачам взаємодіяти з веб-застосунками в режимі реального часу виникла потреба у більш ефективному способі зберігання даних. РБД задовольнили цю потребу, оскільки вони дозволяють легко зберігати та отримувати доступ до даних в режимі реальної години. Внаслідок цього РБД стали стандартом для веб-розробки. РБД використовуються у веб-розробці для зберігання різноманітних типів даних, таких як:

1. BIGINT: Це цілі числа більшого розміру, ніж INTEGER, зазвичай використовується для дуже великих чисел.
2. INTEGER (INT): Цілі числа без десяткових знаків. Наприклад, 1, 10, -5.
3. FLOAT та DOUBLE: Числа з плаваючою точкою, де FLOAT є меншою точністю, ніж DOUBLE. Використовуються для зберігання чисел із десятковою частиною. Наприклад, 3.14.
4. CHAR(N) та VARCHAR(N): CHAR використовується для зберігання рядків фіксованої довжини, а VARCHAR — для рядків змінної довжини. N — максимальна кількість символів. Наприклад, CHAR(10) та VARCHAR(255).
5. DATE та TIME: DATE використовується для зберігання дати, а TIME – для зберігання часу.
6. DATETIME та TIMESTAMP: DATETIME поєднує інформацію про дату та час, в той час, як TIMESTAMP представляє момент у часі, зазвичай у вигляді кількості секунд від початку епохи.
7. BOOLEAN: Використовується для зберігання логічних значень, таких, як TRUE або FALSE.
8. BLOB (Binary Large Object): Використовується для зберігання великих двійкових об'єктів, таких як зображення або відео.

9. ENUM: Перерахований тип, який дозволяє встановити набір можливих значень для конкретного поля.
10. DECIMAL (або NUMERIC): Використовується для зберігання чисел з фіксованою точністю та масштабом, зазвичай застосовується у фінансових обчисленнях.
11. JSON: У деяких сучасних реляційних базах даних, таких як PostgreSQL, додано тип даних для зберігання даних JSON.

Реляційні бази даних засновані на концепції відносин. Відносини — це зв'язок між двома чи більше таблицями. Відношення визначається за допомогою ключів. Ключ – це стовпець чи кілька стовпців, які однозначно ідентифікують запис у таблиці. Реляційні бази даних забезпечують низку переваг, які роблять їх ідеальними для веб-розробки. Ось деякі з цих переваг:

1. Структурованість: РБД зберігають дані у структурованому вигляді, що робить їх легкими для розуміння та використання.
2. Ефективність зберігання: РБД використовують ефективні алгоритми зберігання, що дозволяє зберігати великі обсяги даних.
3. Забезпечення відсутності дублювання даних: Реляційні бази даних спираються на принцип нормалізації, який дозволяє ефективно управляти даними та забезпечувати їх консистентність. Нормалізація дозволяє розбити таблиці на менші та пов'язані елементи, уникнути повторюваності інформації та зменшити ризик виникнення аномалій даних.
4. Консистентність та цілісність даних: Завдяки нормалізації, реляційні бази даних підтримують високий рівень консистентності та цілісності даних. Це означає, що дані завжди знаходяться в стані, який відповідає визначеним правилам та обмеженням, що є важливим для довіри до інформації.

2.3 NoSql бази даних та їх роль у веб-розробці

NoSQL бази даних мають цікаву історію, пов'язану з виникненням нових вимог у галузі зберігання та обробки даних. Наприкінці 2000-х років з появою масштабованих веб-застосунків та соціальних мереж стало очевидно, що традиційні реляційні бази даних (РБД) не завжди можуть ефективно керувати величезними обсягами даних та забезпечувати високу продуктивність за великої кількості одночасних запитів [24].

NoSQL бази даних були розроблені у відповідь на ці виклики. Вони пропонували альтернативні моделі даних, відмінні від таблиць і реляцій РБД. Ключовими принципами NoSQL є гнучкість структури даних, горизонтальне масштабування, простота розробки та висока продуктивність.

Одним із плюсів NoSQL баз даних є гнучкість структури даних. Вони дозволяють зберігати неструктуровані дані, такі як JSON-документи або ключ-значення без необхідності визначення фіксованої схеми заздалегідь. Це особливо корисно в контексті розробки веб-застосунків, де дані можуть змінюватися або мати різну структуру.

Ще однією перевагою NoSQL баз даних є їхня здатність горизонтального масштабування. Вони дозволяють розподілити дані на кілька серверів та обробляти запити паралельно, що забезпечує високу доступність та продуктивність навіть за великої кількості даних та користувачів. Це важливо для сучасних веб-застосунків, які повинні обробляти величезні обсяги інформації та масштабуватись у міру зростання користувальницької бази [28].

Однак, у NoSQL баз даних є деякі недоліки. В силу гнучкості структури даних, відсутності жорсткої схеми та можливості зберігати неструктуровані дані, складність побудови складних запитів може зрости. Крім того, не всі NoSQL бази даних забезпечують транзакційність та цілісність даних, що може бути проблемою у певних сценаріях використання.

У веб-розробці сьогодні бази даних NoSQL відіграють важливу роль. Вони використовуються для зберігання та обробки різних типів даних, таких як профілі користувача, логи подій, графи зв'язків та інші. NoSQL бази

даних дозволяють розробникам створювати масштабовані та відмовостійкі додатки, здатні обробляти великі обсяги даних та витримувати високі навантаження.

У веб-розробці NoSQL бази даних забезпечують гнучкість під час роботи з різними типами даних. Наприклад, документо-орієнтовані бази даних, такі як MongoDB, дозволяють зберігати та обробляти документи у форматі JSON, що зручно для роботи з веб-сторінками або API. Ключ-значення бази даних, наприклад Redis, надають простий та ефективний спосіб зберігання та доступу до даних по ключу, що корисно для кешування або зберігання тимчасових даних.

NoSQL бази даних також відіграють важливу роль у галузі аналітики даних та машинного навчання. Завдяки можливості зберігання великих обсягів неструктурованих даних та горизонтальному масштабуванню, вони надають інструменти для отримання цінної інформації та виконання складних аналітичних запитів. Це важливо для прийняття бізнес-рішень та оптимізації процесів на основі даних.

В цілому, NoSQL бази даних мають значне значення у веб-розробці сьогодні. Вони пропонують гнучкість, масштабованість та продуктивність, необхідні для сучасних додатків, особливо тих, які працюють з великими обсягами даних або потребують гнучкості структури даних. З їх допомогою розробники можуть створювати інноваційні та ефективні веб-програми, здатні обробляти вимоги сучасних користувачів та бізнес-завдань [8].

2.4 Масштабування баз даних

Масштабування баз даних є фундаментальним аспектом проектування та обслуговування систем зберігання даних. Цей процес спрямований на забезпечення ефективності та стійкості роботи зі збільшенням обсягів даних, інтенсивності запитів та зростаючих потреб бізнесу. Масштабування баз даних означає способи розширення можливостей зберігання та обробки даних з метою забезпечення високої продуктивності та відмовостійкості.

Одним з основних підходів до масштабування є горизонтальне масштабування (шардинг). Цей метод передбачає розподіл даних по кількох вузлах (шардах), де кожен вузол відповідальний за зберігання та обробку своєї частини даних. Горизонтальне масштабування забезпечує лінійне зростання можливостей системи зі збільшенням числа вузлів, що робить його особливо придатним для обробки великих обсягів даних та високонавантажених додатків.

У процесі горизонтального масштабування виявляються дві основні стратегії:

1. Реплікація: Цей метод включає створення точних копій даних різних вузлів. Реплікація забезпечує підвищену доступність і стійкість до відмови, оскільки кожен вузол може служити як резервний у разі збою іншого. Однак він також стикається з проблемою обробки записів, що надходять на різні вузли.
2. Шардинг: Цей метод передбачає поділ даних на частини, кожна з яких зберігається на своєму вузлі. Шардинг може бути реалізований горизонтальною (по рядках) або вертикальною (по стовпцях) осі даних. У разі горизонтального шардингу різні вузли містять частини даних, що сприяє рівномірному розподілу навантаження і забезпечує можливість лінійного масштабування.

Масштабування баз даних знаходить застосування у різних галузях, включаючи електронну комерцію, фінансовий сектор, обробку транзакцій та аналітику даних. Цей підхід виявляється особливо корисним при обробці великих обсягів транзакційних даних або за необхідності забезпечення швидкого та надійного доступу до даних для багатьох користувачів. Коли зростає кількість користувачів, кількість даних або вимоги до відгуку програми, масштабування баз даних стає стратегічним рішенням для підтримки зростаючих потреб та забезпечення ефективності роботи інформаційних систем.

2.5 Контейнеризація БД

Використання PostgreSQL у середовищі Docker — це ефективний спосіб встановлення та управління базою даних, використовуючи контейнеризацію. Docker дозволяє ізолювати PostgreSQL та всі його залежності в контейнері, що спрощує процес розгортання та забезпечує консистентність середовища між різними системами [26].

Створення PostgreSQL у контейнері Docker дозволяє швидко створити та запустити ізольовану інсталяцію бази даних. Це дозволяє легко розгортати PostgreSQL на будь-якому пристрої або в хмарному середовищі, не потребуючи встановлення та налаштування його з початку.

Docker дозволяє масштабувати PostgreSQL шляхом розгортання декількох контейнерів з базами даних та їх координації, що дозволяє забезпечити високу доступність та надійність. Контейнери легко розширюються та зменшуються в залежності від потреб, забезпечуючи ефективну роботу з великими обсягами даних.

Крім того, використання PostgreSQL у Docker спрощує резервне копіювання та відновлення даних. Docker дозволяє зберігати стан контейнера, що робить процес резервного копіювання та відновлення більш простим та швидким, що є важливим для забезпечення надійності та безпеки даних.

Використання PostgreSQL у Docker надає зручний спосіб роботи з базою даних, спрощує розгортання та управління, дозволяє ефективно використовувати ресурси та покращує надійність системи.

2.6 Критерії порівняння

Порівнюючи MongoDB та PostgreSQL важливо враховувати кілька ключових критеріїв, щоб обрати найбільш відповідну систему управління базами даних для конкретного проекту:

- **Модель даних:**

MongoDB базується на документ-орієнтованій моделі, у той час, як PostgreSQL використовує реляційну модель даних. Потрібно оцінити, яка модель краще відповідає потребам у зберіганні та операціях з даними.

- **Гнучкість схеми даних:**

MongoDB дозволяє гнучко працювати зі схемами даних, у той час, як PostgreSQL вимагає строго визначеної схеми.

- **Транзакційна підтримка:**

PostgreSQL має сильну транзакційну підтримку, тоді як MongoDB має обмежену підтримку транзакцій.

- **Продуктивність та масштабованість:**

Обидві системи мають свої особливості у продуктивності та масштабованості. MongoDB часто використовується для великих обсягів неструктурованих даних, тоді як PostgreSQL може бути ефективним для операцій з великою кількістю даних, включаючи складні операції.

- **Аналіз мов запитів:**

Порівняння синтаксису, можливостей та швидкодії запитів різного типу

- **Спільнота та підтримка:**

Обидві бази даних повинні мати активні спільноти користувачів та розробників. Важливо враховувати наявність документації, оновлення та підтримки спільноти для вашого вибору.

2.4.1 Модель даних

Модель даних у контексті баз даних є способом організації та структурування інформації у межах системи управління базами даних (СКБД). Вона визначає, як дані будуть зберігатися, представлятися та взаємодіяти між собою.

Модель даних визначає структуру даних, їх типи, зв'язки та правила доступу до цих даних. Основна мета моделі даних полягає у наданні стандартизованого способу організації даних для забезпечення консистентності, ефективності та легкості управління. Вона визначає типи даних, які можуть бути збережені у базі даних та їхню структуру, забезпечуючи коректність

та цілісність даних у СКБД. Модель даних визначає, які типи даних можуть бути збережені та як вони взаємодіють між собою.

Реляційна модель даних використовує таблиці для зберігання даних та їх зв'язків за допомогою ключів. З іншого боку, документо-орієнтована модель даних структурує дані у вигляді документів, що спрощує їхню обробку. Модель даних важлива для побудови бази даних та розробки програм, оскільки надає рамки для створення оптимальної структури даних, що відповідає потребам конкретного проекту. Це допомагає забезпечити правильність та ефективність зберігання та обробки даних в системі.

Модель даних у базі даних - це не лише спосіб організації даних, але й концептуальна схема, яка визначає способи створення, зберігання, оновлення та взаємодії з даними в системі управління базами даних. Вона дозволяє визначити структуру даних та їх відношення, встановлюючи правила, які об'єднують окремі елементи інформації.

Різні моделі даних мають свої особливості та переваги. Реляційна модель використовує таблиці для зберігання даних та зв'язків, що спрощує операції з даними. Документо-орієнтована модель даних дозволяє структурувати дані у вигляді документів, що полегшує їх обробку. Модель даних також впливає на швидкість доступу до даних та продуктивність системи, дозволяючи розв'язувати питання щодо ефективності та оптимальності зберігання та маніпулювання даними.

Загалом, модель даних є ключовим елементом будь-якої бази даних, визначаючи структуру та організацію даних для ефективного управління інформацією в системі.

2.4.2 Гнучкість схеми даних

Гнучкість схеми даних у контексті бази даних відноситься до можливості змінювати чи розширювати структуру даних без значних обмежень чи змін у вже існуючій інформації. Це означає, що база даних може легко адаптуватися до нових вимог, без необхідності великого перепроектування чи переписування існуючих даних.

Гнучкість схеми даних особливо актуальна в системах, де дані можуть змінюватися в ході розвитку проекту або коли потрібно швидко реагувати на нові вимоги [16]. Вона дозволяє додавати нові поля, змінювати вже існуючі атрибути чи структури даних без значних перешкод для системи в цілому.

Системи з гнучкою схемою даних, такі як NoSQL бази даних чи деякі реляційні бази даних, можуть легко пристосовуватися до різноманітних форматів та типів даних, включаючи текстові файли, графічні дані, аудіо, відео та інші форми неструктурованої інформації.

За допомогою гнучкої схеми даних, розробники можуть здійснювати швидкі зміни у структурі даних, що дозволяє реагувати на потреби бізнесу чи користувачів без великих зусиль на перепроектування чи переналаштування бази даних.

Однак гнучкість схеми даних може мати свої недоліки. Недостатня структура чи контроль може призвести до неоднорідності даних, труднощів у розвитку чи ускладненого управління у майбутньому. Тому необхідно докладно розглянути вигоди та ризики при виборі гнучкої схеми даних для конкретного проекту.

Гнучкість схеми даних є ключовою у деяких типах баз даних, таких як NoSQL (наприклад, MongoDB) або певні реляційні бази даних із можливістю динамічного розширення схеми. Вона дає можливість додавати, видаляти чи змінювати типи даних та структури вже існуючих записів без значного впливу на існуючу інформацію чи без необхідності переконструювання всієї бази даних.

Така гнучкість корисна у випадках, коли вимоги до цього можуть змінюватися швидко, або коли система розвивається, і важко передбачити всі можливі варіанти даних наперед. Вона дозволяє швидше реагувати на зміни, не блокуючи розвитку через строгі обмеження схеми.

Це особливо важливо в умовах розвитку сучасних додатків, де швидкість впровадження нового функціоналу чи зміни вимог може визначити

конкурентні переваги. Здатність швидко адаптуватися до нових потреб користувачів чи бізнесу є важливою для успішного функціонування продукту.

Однак, варто пам'ятати, що занадто велика гнучкість схеми даних може призвести до складнощів у підтримці чи ускладненого керування базою даних у майбутньому. Недостатня структура даних може ускладнити їх аналіз та обробку. Тому вибір між гнучкою та суворою схемою даних повинен базуватися на конкретних потребах проекту та його майбутніх перспективах [16].

2.4.3 Транзакційна підтримка

Транзакції у контексті масштабованості баз даних є важливою частиною, оскільки вони визначають рівень конкурентоспроможності, швидкодії та розподілу навантаження при великому обсязі транзакцій.

Здатність до розподіленої транзакції (distributed transactions) є ключовою для масштабованості. Це означає, що транзакції можуть виконуватися на різних серверах або частинках бази даних одночасно без втрати ACID-властивостей. Виконання транзакцій у розподіленій середі дозволяє оптимізувати роботу та сприяти збалансованості навантаження на сервери.

Проте масштабованість транзакцій може бути складним завданням через необхідність забезпечення ізольованості та консистентності між різними частинами бази даних. Системи, що дозволяють робити це ефективно, використовують різні методи та підходи, такі як керування транзакціями через протоколи розподіленого блокування, асинхронне підтвердження чи використання реплікації даних для забезпечення надійності транзакцій у розподіленій середі.

Технології, що забезпечують гнучкість у роботі з транзакціями, дозволяють виконувати транзакції на різних серверах чи розподілених областях бази даних шляхом оптимізації та контролю. Це дозволяє базам даних працювати ефективно та надійно навіть у великих та розподілених середах.

Розподілені транзакції можуть включати в себе взаємодію з кількома різними ресурсами чи базами даних. Це може бути складним завданням,

оскільки необхідно забезпечити консистентність та ізоляцію у межах усіх взаємодіючих систем. Механізми та протоколи розподілених транзакцій мають забезпечувати цілісність даних та виконання вимог ACID, навіть при паралельному виконанні транзакцій на різних серверах чи різних частинах системи.

Технології, що дозволяють розподіляти транзакції, повинні бути гнучкими, адаптивними та забезпечувати контроль за всіма етапами транзакційного процесу. Це включає управління блокуванням, реплікацію даних та механізми узгодження між частинами бази даних для забезпечення ізольованості та консистентності даних [22].

Оптимізація процесу розподілених транзакцій може бути складним завданням через необхідність збалансування швидкодії та надійності. З одного боку, система має бути швидкою та здатною обробляти велику кількість транзакцій, а з іншого — забезпечувати цілісність даних та надійність у всіх умовах роботи.

Отже, ефективні розподілені транзакції вимагають комплексного підходу та використання передових технологій для забезпечення ефективності та надійності в роботі з великим обсягом даних та розподіленими системами.

2.4.4 Продуктивність та масштабованість

Продуктивність та масштабованість бази даних — це ключові аспекти, які визначають її ефективність у використанні та здатність працювати з великим обсягом даних або зростати з розвитком проекту. Продуктивність оцінює швидкодію та реакцію системи на запити, включаючи годину відповіді на запити та загальну продуктивність операцій.

Здатність масштабованості бази даних означає, наскільки легко чи ефективно система може збільшувати свій обсяг роботи чи даних без втрати продуктивності. Це важливо для зростання бізнесу, оскільки збільшення обсягу даних чи кількості користувачів може призвести до розширення бази даних.

Для досягнення високої продуктивності та масштабованості база даних повинна бути ефективно оптимізована для обробки запитів та швидкого доступу до даних. Це включає використання індексів, кешування даних, оптимізацію запитів та розподіл даних для балансування навантаження.

Деякі бази даних мають вбудовані механізми для масштабованості, такі як горизонтальне чи вертикальне розширення, що дозволяє розподіляти дані на кілька серверів чи збільшувати потужність системи без великих зусиль.

Здатність масштабованості та продуктивності є важливими для проектів, які потребують ефективної обробки великих обсягів даних або очікують на зростання користувачів чи обсягу інформації. Оптимізація та гнучкість системи бази даних відіграють критичну роль у забезпеченні її успішної роботи в умовах зростаючих вимог та обсягу інформації.

Продуктивність бази даних визначається швидкістю операцій зберігання, доступу та опрацювання великого обсягу даних. Швидкодія є ключовим фактором для задоволення потреб користувачів відповідно до їх очікувань. Вона оцінюється як час відповіді на запити, операції вставки, оновлення та видалення даних, а також завантаження бази даних у цілому.

Ефективність бази даних може бути вплинути такими факторами, як оптимізація запитів, налаштування конфігурацій, використання індексів та кешування даних. Використання цих методів може покращити швидкодію та загальну продуктивність бази даних.

Надійність та доступність бази даних також є важливими аспектами продуктивності. Це означає, що система повинна бути стійкою до відмов та забезпечувати постійний доступ до даних. Резервне копіювання, реплікація та відновлення даних — це лише кілька засобів, які забезпечують надійність та доступність бази даних.

Зрозуміння та належне впровадження стратегій для підвищення продуктивності та масштабованості бази даних дозволяє забезпечити стабільну та ефективну роботу системи, а також готовність до майбутніх викликів та зростання обсягу даних.

2.4.5 Аналіз мов запитів

Аналіз мов запитів у системах керування базами даних (СКБД) визначається великою мірою специфікою кожної конкретної системи. У контексті порівняння PostgreSQL та MongoDB, виявляються різні підходи до мов запитів та роботи з даними.

PostgreSQL, як реляційна база даних, використовує мову запитів SQL. SQL є стандартом в області реляційних баз даних і визначає стандартні способи роботи з даними, такі як SELECT, INSERT, UPDATE та DELETE. Використання SQL дозволяє виконувати складні запити, об'єднувати дані з різних таблиць, використовувати підзапити та застосовувати тригери.

Навпаки, MongoDB, як система управління нереляційною базою даних, використовує власну мову запитів, відому як MongoDB Query Language (MQL). MQL є мовою, схожою на JavaScript, і використовується для взаємодії з нереляційними документами, які зберігаються у колекціях. Запити в MongoDB можуть бути менш інтуїтивними для тих, хто звик до SQL, але вони дають гнучкість у роботі з невстановленою структурою даних.

Спрощена структура MongoDB дозволяє зберігати дані у вигляді JSON-подібних документів, що визначає нереляційний підхід. У випадку PostgreSQL, таблиці, зовнішні ключі та інші елементи реляційної бази даних додатково структурують дані.

2.4.6 Спільнота та підтримка

Спільнота та підтримка грають ключову роль у виборі технологій для управління базами даних. У світі PostgreSQL та MongoDB ці аспекти виявляються в різних вимірах. PostgreSQL, як реляційна база даних, має велику та активну спільноту, взяту від відкритого програмного забезпечення, що забезпечує неперевершену кількість експертів і розробників. Документація, форуми та регулярні оновлення свідчать про велику активність спільноти PostgreSQL.

У випадку MongoDB, система управління нереляційною базою даних також користується сильною спільнотою. MongoDB надає засоби співпраці,

такі як форуми та групи користувачів, а також отримує підтримку від комерційної компанії MongoDB, Inc. Обидві системи забезпечують користувачів різноманітними ресурсами для обміну досвідом та вирішення проблем.

Спільнота та рівень підтримки стають важливими факторами для розробників, які цінують активність, обмін досвідом та широкий спектр ресурсів для вирішення завдань.

2.7 Теоретичне порівняння PostgreSQL та MongoDb

PostgreSQL та MongoDB — дві різні системи управління базами даних, кожна з яких має свої особливості та переваги. MongoDB є документ-орієнтованою NoSQL базою даних, яка дозволяє зберігати дані у вигляді документів, які використовують JSON-подібні структури. З іншого боку, PostgreSQL — реляційна база даних, що використовує таблиці та SQL для зберігання та операцій з даними.

MongoDB відома своєю гнучкістю та легкістю розширення схеми даних. Вона дозволяє зберігати набагато більше різноманітних типів даних, що забезпечує більшу вільність у моделюванні даних порівняно з традиційними реляційними базами даних.

Однак PostgreSQL відзначається найсильнішою структурою даних через використання таблиць та суворих схем даних. Це робить його більш відповідним для застосування, де потрібна точність та стабільність структури, таких як фінансові системи або деякі типи додатків з великим обсягом даних.

MongoDB часто використовують у сучасних додатках, де швидкість розробки та потреба в гнучкості моделі даних є пріоритетними. У той час як PostgreSQL відомий своєю надійністю, ACID-властивістю (атомарність, консистентність, ізоляція, довіреність) та високою продуктивністю у складних операціях з даними.

MongoDB та PostgreSQL представляють різні підходи до зберігання та управління даними. MongoDB базується на моделі документів, де кожен

документ може містити різні поля та структури, що надає більшої гнучкості у роботі з неструктурованими даними. У той же час, PostgreSQL використовує традиційні таблиці та схеми, що забезпечує сувору структурованість даних.

MongoDB зазвичай використовується у випадках, де потрібна швидка реакція на зміни в структурі даних та колосальна гнучкість у зміні схеми даних без великої переробки. PostgreSQL, з іншого боку, частіше вибирають для проектів, де потрібна сувора консистентність даних та структурованість.

У MongoDB немає обмежень щодо схеми даних, що може бути як перевагою, так і недоліком. Це дозволяє швидше розгортати та змінювати структуру даних, але одночасно вимагає більшої уваги до дизайну та контролю за даними.

РОЗДІЛ 3 РЕАЛІЗАЦІЯ МОДЕЛІ БД КОНФІГУРАЦІЯ БЕКЕНДУ ТА СТВОРЕННЯ API

3.1 Моделювання бази даних

Головною метою створення веб-застосунків є порівняння систем NoSql баз даних та реляційних баз даних. Для цього було розроблено два веб-застосунка, кожний з яких складався з бекенд частини, бази даних та клієнтської частини для MongoDB та PostgreSql відповідно.

Для реалізації було обрано веб-застосунок для пошуку та вибору авіа-квитків. Також було визначено модель бази даних, за допомогою якої ми зможемо спроектувати однаково структуровані бази даних з точки зору організації інформації.

Це дуже важливо при порівнянні запитів, зручності написання та використання. Використовуючи одну модель бази даних, ми зможемо наочно побачити ключову різницю між двома СКБД. Ця модель була спроектована таким чином, що ми зможемо реалізувати різні типи запитів для кожної бази даних. На рисунку 5 зображена модель бази даних.

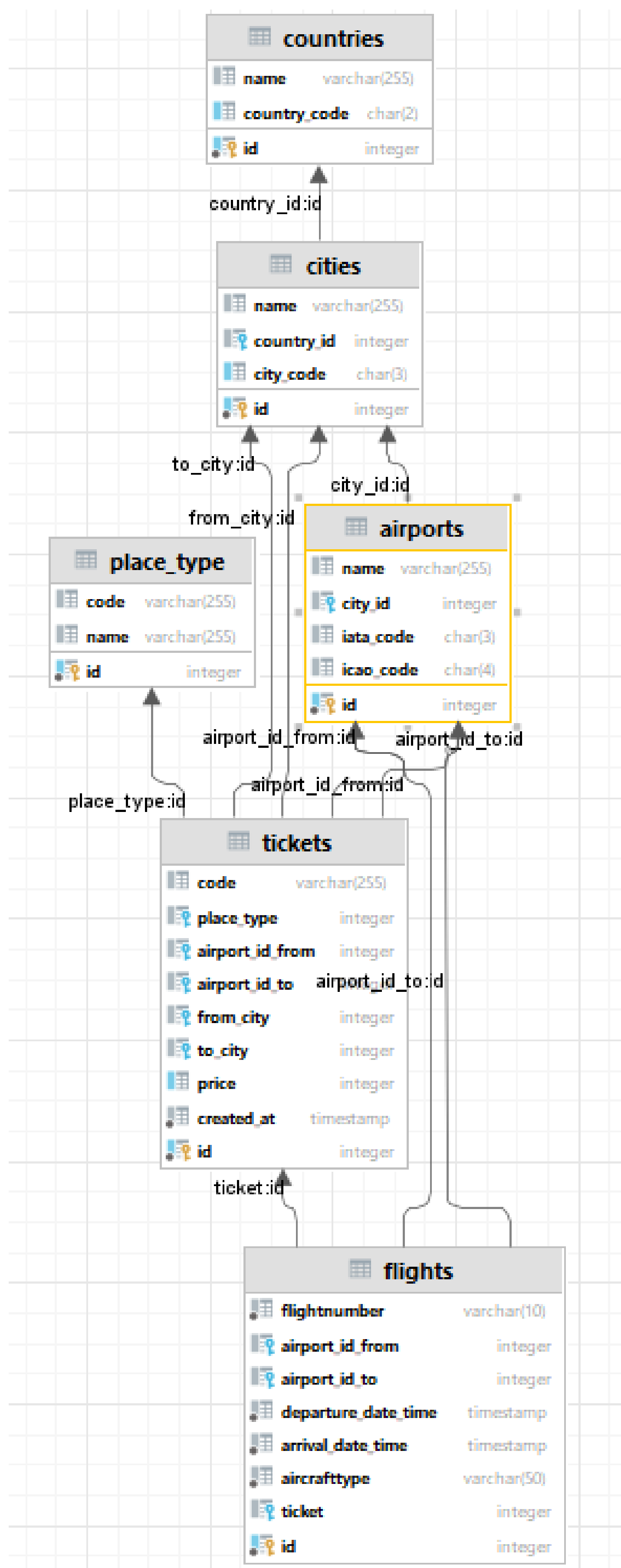

Рисунок 5 — Модель БД

3.2 Засоби реалізації

Для реалізації бекенд додатків для кожної з баз даних, був обраний наступний стек технологій:

- Мова програмування JavaScript.
- Платформа NodeJs.
- Фреймворк Express.
- Пакет Cors.
- Пакет Dotenv.
- Пакет Morgan.
- Пакет Mongoose.
- Пакет Pg.
- Середовище розробки Visual Studio Code.
- Середовище розробки DataGrip.
- MongoDb Compass.
- Docker.
- Postman.
- Менеджер пакетів Node Package Manager.
- Система контролю версій Git.
- MongoDb Atlas.

3.2.1 Мова програмування JavaScript

JavaScript — це високорівнева, інтерпретована мова програмування, яка використовується для розробки динамічних веб-сайтів та взаємодії з користувачем на стороні клієнта. Вона була створена компанією Netscape під назвою Mocha, а пізніше перейменована в LiveScript та, нарешті, в JavaScript. Мова стала стандартом для веб-розробки та інтегрується з HTML та CSS, щоб надавати динамічність та інтерактивність веб-сторінкам.

Історія JavaScript почалася в 1995 році, коли Брендан Айк створив мову для взаємодії з браузером Netscape Navigator. Початково вона

задумувалася як мова для скриптів, але завдяки своїм можливостям швидко стала важливою складовою для веб-розробки. У 1997 році мова була стандартизована Ecma International як ECMAScript.

JavaScript має численні переваги, такі як висока розповсюдженість, зручна інтеграція з HTML та CSS, асинхронний код для ефективної взаємодії з сервером та можливість використання на різних платформах. Вона також підтримує функціональне та об'єктно-орієнтоване програмування, що дозволяє розробникам вибирати підхід відповідно до вимог проекту.

Незважаючи на свої переваги, JavaScript має й недоліки, такі як відсутність строгого контролю типів, що може вести до помилок в роботі програми. Також, у великих проектах може виникати проблема зі складністю коду через відсутність простору імен та інші обмеження [36].

3.2.2 Платформа Nodejs

Node.js — це безкоштовна та високопродуктивна платформа, заснована на двигуні JavaScript V8 від Google Chrome, яка дозволяє виконувати JavaScript-код на серверному боці. Ця технологія стала популярною завдяки своїй здатності ефективно обробляти багатооперативні вхідно-вихідні операції та робити акцент на асинхронності, що робить її ідеальною для створення високоефективних серверних додатків [37].

Історія Node.js розпочалася у 2009 році, коли розробник Райан Дал, працюючи в компанії Joyent, створив цей проект як відповідь на проблеми сучасних серверних застосунків, які вимагали швидкої та ефективної обробки багатооперативних запитів. Node.js швидко здобув популярність завдяки своїй простоті та продуктивності, стаючи основою для численних веб-застосунків та сервісів.

Однією з ключових переваг Node.js є його можливість працювати асинхронно, що полегшує обробку багатьох запитів одночасно, і реалізація механізму подій сприяє створенню ефективних та швидких додатків. Node.js

також відкриває доступ до широкого вибору модулів та бібліотек, що сприяє розширенню можливостей розробників.

Платформа Node.js стала основою для створення великої кількості інструментів, таких як фреймворк Express.js для швидкої розробки серверних додатків та пакетний менеджер npm для управління залежностями. Node.js здатний взаємодіяти з базами даних, обробляти HTTP-запити та створювати масштабовані мережеві додатки.

Ще однією перевагою Node.js є можливість використання JavaScript для розробки і на клієнтському, і на серверному боці, що спрощує розробку та уніфікує мовний стек. Завдяки активній спільноті та постійному оновленню, Node.js залишається важливою технологією у світі веб-розробки.

Недоліками Node.js можуть бути відсутність вбудованої підтримки деяких стандартів та обмеження в роботі з великими високовитратними обчисленнями через однопоточну модель. Однак розробники знаходять рішення цих проблем за допомогою асинхронного програмування та використання засобів масштабування [37].

Node.js може бути порівняно з іншими серверними технологіями, такими як Apache чи Nginx, та відзначається високою швидкістю та продуктивністю. Ця технологія дозволяє створювати легкі та ефективні серверні додатки, що зробило її популярною серед розробників у сфері веб-розробки.

3.2.3 Фреймворк Express

Express.js — це мінімалістичний та гнучкий веб-фреймворк для Node.js, призначений для розробки серверних додатків та API. Його простота та лаконічність дозволяють розробникам швидко створювати високопродуктивні веб-застосунки, а активна спільнота та велика кількість розширень додають додатковий функціонал і зручність в роботі. Express.js з'явився в 2010 році і став одним із перших фреймворків для розробки на платформі Node.js. Виникнення Express сприяло швидшому та зручнішому створенню серверних додатків, оскільки воно визначило кращі практики та стандарти для організації коду. В основі філософії Express лежить принцип

"робити максимум, надаючи мінімум". Фреймворк не нав'язує конкретний шаблон організації коду, що дає розробникам велику свободу вибору архітектури свого додатку.

Express забезпечує базовий набір функцій, які включають маршрутизацію, обробку запитів та відповідей, middleware для обробки проміжних етапів обробки запиту, та інші. Однією з переваг Express є його легкість вивчення та швидкість розробки. Розробники можуть швидко створювати прості серверні додатки, але в той же час використовувати його для створення складних API або веб-застосунків. Express також підтримує велику кількість додаткових модулів (middleware), які дозволяють додавати нову функціональність та оптимізувати роботу додатку. Такі middleware можуть використовуватися для автентифікації, обробки cookie-файлів, логування та багатьох інших задач.

Важливим аспектом Express є його велика спільнота та активна підтримка. Це означає, що розробники можуть швидко знаходити відповіді на свої питання, а також використовувати готові рішення та код від інших у своїх проектах. Недоліками можуть бути відсутність вбудованих засобів для оркестрації та керування структурою проекту при роботі з великими та складними додатками. Однак ці недоліки можна компенсувати за допомогою сторонніх бібліотек та модулів, що підтримують структуру проекту та організацію коду.

Загалом, Express.js залишається одним із найпопулярніших веб-фреймворків для Node.js завдяки своїй простоті, гнучкості та широким можливостям розширення для розробників у створенні серверних додатків та API [37].

3.2.4 CORS

CORS (Cross-Origin Resource Sharing) — це стандарт технічної безпеки, який дозволяє веб-сайтам чи веб-застосункам запитувати ресурси з інших доменів, ніж той, з якого був завантажений вихідний веб-ресурс. Це механізм вбудований у браузери і забезпечує безпечний обмін ресурсами

між різними доменами, контролюючи доступ до ресурсів через HTTP-запити.

Історія CORS виникла як реакція на проблеми безпеки, пов'язані з тим, що веб-сайти намагалися взаємодіяти з ресурсами на інших доменах без належного контролю. Це стало актуальним через зростання складності веб-застосунків та використання різних доменів для розміщення ресурсів. Однією з ключових переваг CORS є те, що він дозволяє веб-застосункам безпечно отримувати доступ до ресурсів на інших доменах, зберігаючи при цьому важливі принципи безпеки. Він дозволяє серверу вказувати, які домени мають право отримати доступ до його ресурсів, а браузери, в свою чергу, автоматично дотримуються цих правил.

CORS реалізується через HTTP-заголовки, такі як Access-Control-Allow-Origin, які сервер повертає відповіддю на запит. Ці заголовки дозволяють чітко визначити, з яких доменів дозволено виконувати запити. Крім того, CORS підтримує передачу креденцій (cookies або HTTP-авторизацію) за допомогою відповідного налаштування заголовків.

Недоліками CORS може бути складність конфігурації на сервері та встановлення правильних заголовків. Деякі старі браузери можуть не підтримувати CORS повністю, але це стосується переважно застарілих версій.

У підсумку, CORS визначає стандарт, який дозволяє веб-застосункам взаємодіяти з ресурсами на інших доменах без порушення принципів безпеки та конфіденційності. Він виявляється ключовим елементом для сучасних веб-розробників, які стикаються з потребою обміну даними між клієнтом та сервером у розподіленому веб-середовищі [37].

3.2.5 Dotenv

Dotenv — це модуль для розробки на різних мовах програмування (зазвичай використовується в контексті JavaScript чи Node.js), який дозволяє завантажувати конфігураційні дані з файлу оточення (.env) у змінні середовища. Це допомагає розробникам зберігати конфіденційні дані, такі як

ключі API чи параметри підключення до баз даних, поза вихідним кодом, тим самим роблячи код більш безпечним та переносним.

Історія Dotenv пов'язана з необхідністю забезпечити безпеку конфіденційної інформації в забезпеченні доступу до зовнішніх служб чи баз даних. Використання .env файлів стало стандартом у веб-розробці, а Dotenv з'явився для спрощення процесу завантаження цих конфігураційних даних у змінні середовища.

Однією з ключових переваг Dotenv є його простота використання. Розробники можуть просто створити файл .env, вказати необхідні змінні та їх значення, і потім використовувати Dotenv, щоб автоматично завантажити ці значення у змінні середовища під час запуску додатку.

Додатково, Dotenv підтримує різні мови програмування та фреймворки, що робить його універсальним інструментом для розробників на різних технологічних стеках. Використання Dotenv стає особливо важливим в середовищі розробки та в умовах, коли додаток розгортається на різних серверах чи платформах.

Недоліками Dotenv може бути потенційна вразливість безпеки, якщо файли .env не належним чином зберігаються чи управляються, оскільки ці файли можуть містити конфіденційну інформацію. Також, при великому обсязі змінних середовища, управління та підтримка файлів .env може стати складнішою задачею.

У підсумку, Dotenv став необхідним інструментом у веб-розробці, допомагаючи розробникам ефективно та безпечно управляти конфігураційними даними у своїх додатках, забезпечуючи при цьому зручність і переносність коду.

3.2.6 Morgan

Morgan — це middleware для обробки логів HTTP-запитів в середовищі Node.js. Використовуючи Morgan, розробники можуть легко створювати докладні логи про запити, які надходять до їх серверів, що допомагає відстежувати та аналізувати діяльність сервера та додати зручності у

відладку. Історія Morgan пов'язана з необхідністю розробників отримувати інформацію про HTTP-запити, яка надходить до їх серверів. Зазвичай, вбудовані засоби логування Node.js не надають докладної інформації, тому Morgan створено для доповнення цього функціоналу.

Однією з ключових переваг Morgan є його простота використання. Для того, щоб почати логування HTTP-запитів, розробникам просто потрібно встановити його як middleware у своєму додатку. Morgan підтримує різні формати логів, і розробники можуть легко налаштовувати його для відображення необхідної інформації, такої як URL, метод запиту, статус відповіді тощо.

Ще однією важливою перевагою Morgan є його гнучкість. Він може використовуватися з будь-яким фреймворком для серверної частини Node.js, таким як Express чи Koa, або навіть у власних серверних додатках, що робить його універсальним інструментом для розробників.

Morgan також надає можливість логувати дані у реальному часі, що є корисним для відслідковування динаміки роботи сервера та виявлення можливих проблем.

Недоліків у Morgan майже немає, оскільки його основна мета - надати простий та ефективний спосіб логування HTTP-запитів. Можливо, для деяких розробників може здатися, що інтерфейс конфігурації міг би бути трошки більш розширеним.

У підсумку, Morgan є невід'ємною частиною веб-розробки на Node.js, допомагаючи розробникам легко відстежувати та аналізувати HTTP-запити, що робить його важливим інструментом для відладки та моніторингу серверних додатків.

3.2.7 Mongoose

Mongoose — це бібліотека для Node.js та фреймворк MongoDB, яка надає розробникам можливість моделювати та взаємодіяти з базою даних MongoDB, використовуючи зручний об'єктно-орієнтований спосіб програмування. Mongoose допомагає спростити взаємодію з MongoDB, надаючи

високорівневий інтерфейс для створення, читання, оновлення та видалення документів у базі даних.

Історія Mongoose пов'язана з необхідністю надати розробникам Node.js зручний інтерфейс для роботи з MongoDB, нереляційною базою даних, яка використовує документ-орієнтований підхід. Mongoose став відповіддю на це завдання, надаючи можливість використовувати MongoDB з більш об'єктно-орієнтованим стилем коду.

Однією з ключових переваг Mongoose є його зручний спосіб визначення та використання схем даних. Розробники можуть визначати моделі, які відповідають структурі їхніх даних, і використовувати ці моделі для взаємодії з базою даних. Mongoose також надає ряд зручних можливостей, таких як валідація даних та обробка хуків (hooks), які дозволяють реагувати на події в базі даних.

Ще однією перевагою Mongoose є його можливість використання middleware, яке дає розробникам можливість визначити функції, які виконуються перед або після певних операцій з базою даних. Це робить Mongoose потужним інструментом для реалізації складних логічних операцій та додаткової обробки даних.

Недоліками Mongoose може бути певний навчальний поріг, оскільки він вводить деякий рівень абстракції, і розробникам потрібно зрозуміти концепції схем та моделей. Також, в залежності від конкретного випадку використання, Mongoose може викликати деякий наклад на продуктивність [2].

3.2.8 PG

Пакет pg для Node.js є потужним драйвером, що забезпечує взаємодію з базою даних PostgreSQL з додатків, написаних на JavaScript або TypeScript. Цей пакет забезпечує простий та гнучкий спосіб підключення до PostgreSQL, виконання SQL-запитів, керування транзакціями та обробки помилок.

Він відрізняється високою продуктивністю та підтримує різні функції, такі як виконання запитів, керування з'єднаннями, обробка транзакцій, а також можливість роботи з подіями та обробки помилок. pg надає зручний API для виконання різних операцій з базою даних, включаючи вибірку, вставку, оновлення та видалення даних.

Ключові особливості pg включають підтримку підготовлених запитів, пул з'єднань для ефективного управління ресурсами і можливість реагувати на зміни даних через подію модель. Він також забезпечує інтеграцію з різними мовами програмування та широкий набір офіційних драйверів, що робить його гнучким у використанні.

Серед аналогів pg можна виділити node-postgres і pg-promise, але pg часто вибирається завдяки активній спільноті, великій документації та регулярним оновленням. Цей пакет надає стійке та надійне рішення для роботи з PostgreSQL в екосистемі Node.js, із зручністю використання та ефективністю роботи [15].

3.2.9 Середовище розробки Visual Studio Code

Visual Studio Code (VS Code) — це безкоштовний, легкий та потужний редактор коду, розроблений компанією Microsoft. Його призначенням є надання зручного середовища для написання, редагування та відлагодження коду на різних мовах програмування. Завдяки своїй гнучкості та розширюваності, Visual Studio Code отримав широкий розповсюдження серед розробників.

Історія Visual Studio Code розпочалася у 2015 році, коли Microsoft представила його як новий, легкий редактор коду для всіх платформ. Заснований на Electron, цей редактор отримав широкий захід завдяки своїй швидкості, зручності та великому спектру функцій, що були доступні розробникам у безкоштовному пакеті.

Однією з ключових переваг Visual Studio Code є його відкритість та підтримка різних мов програмування, включаючи JavaScript, Python, Java, C#, PHP і багато інших. Завдяки розширенням, розробник може налаштовувати середовище під свої потреби, додавати плагіни для підтримки конкретних технологій або розширювати можливості редактора.

Інтерфейс Visual Studio Code простий та інтуїтивно зрозумілий, але в той же час має велику кількість функцій для продуктивної роботи. Редактор підтримує такі різноманітні можливості, як автодоповнення коду, відлагодження, систему контролю версій, інтеграцію з системами збірки та багато іншого.

Visual Studio Code активно оновлюється та розвивається, регулярно випускаючи нові версії з покращеннями та новими функціями. Редактор підтримує велику спільноту розробників, яка активно співпрацює над покращенням та доповненням його функціональності.

Недоліків у Visual Studio Code досить мало, але деякі розробники можуть відмітити обмежену функціональність для великих проектів порівняно з іншими інтегрованими середовищами розробки. Проте, це питання в основному стосується особливостей великих та складних проектів, оскільки для більшості задач Visual Studio Code надто відмінно справляється. На рисунку 6 зображене середовище розробки Visual Studio Code.

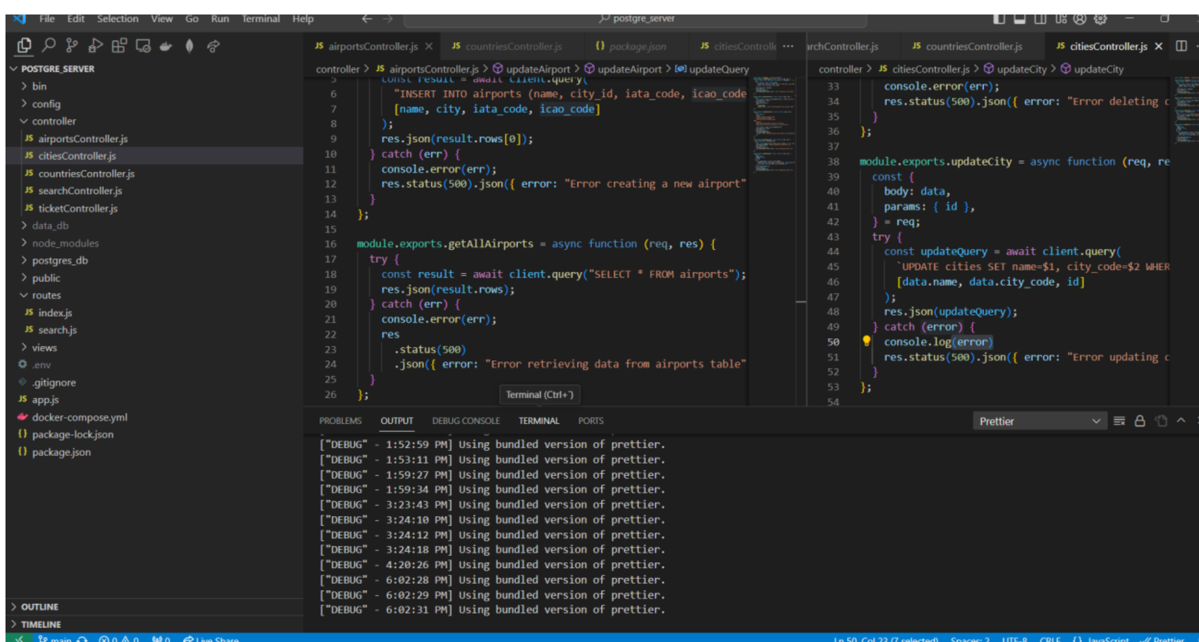

Рисунок 6 — Середовище розробки Visual Studio Code.

3.2.10 Середовище розробки DataGrip

DataGrip — це інтегроване середовище розробки (IDE) для роботи з базами даних, розроблене компанією JetBrains. Запущений в 2016 році, він став необхідним інструментом для розробників, які працюють з різними типами баз даних, включаючи MySQL, PostgreSQL, Oracle, SQL Server та інші. DataGrip пропонує багатий функціонал для зручного редагування та відлагодження SQL-коду, а також для візуальної навігації та аналізу даних в базі.

Однією з ключових переваг DataGrip є його мультиплатформеність, що дає можливість розробникам працювати на різних операційних системах, включаючи Windows, macOS та Linux. Він інтегрується з іншими продуктами JetBrains, такими як IntelliJ IDEA та PyCharm, що створює зручне середовище для розробки в одній екосистемі.

Один з визначних аспектів DataGrip - це велика кількість підтримуваних баз даних. Вибір інструменту не обмежується однією платформою, що дозволяє розробникам легко перемикатися між різними системами управління базами даних з однаковим зручним інтерфейсом.

Інтерфейс DataGrip пропонує велику кількість інструментів для зручної роботи. Зокрема, він включає можливості автоматичного доповнення, візуального плану виконання запитів, а також інструменти для аналізу структури бази даних та її даних.

DataGrip надає потужні інструменти для відлагодження SQL-коду, включаючи можливість поетапного виконання запитів та аналізу їх ефективності. Інтеграція з системами контролю версій дозволяє розробникам ефективно взаємодіяти з кодом та базою даних у командному середовищі.

Хоча DataGrip надає велику кількість функцій у безкоштовній пробній версії, повноцінна ліцензія може визначатися як вартістьна. Однак для професіоналів, які регулярно працюють з базами даних, це інвестиція у продуктивність та зручність роботи.

В цілому, DataGrip виступає як потужний та зручний інструмент для розробників, що працюють з різними базами даних, надаючи їм інструменти для продуктивної роботи, ефективного аналізу та візуалізації даних у їх проектах. На рисунку 7 зображене середовище розробки DataGrip.

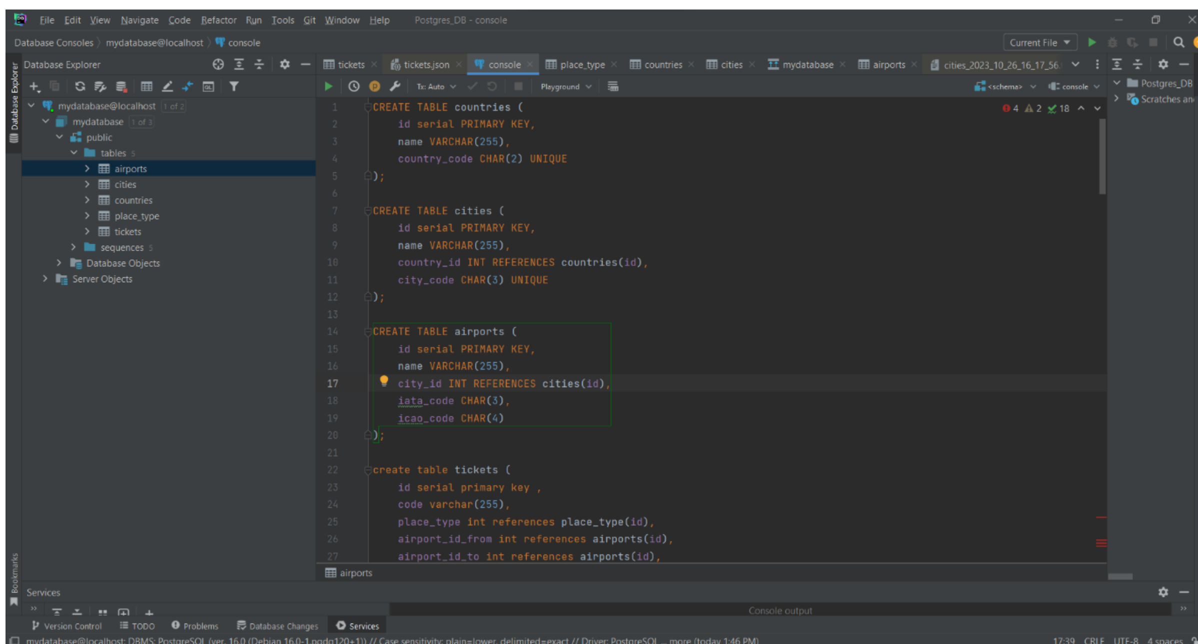

Рисунок 7 — Середовище розробки DataGrip.

3.2.11 MongoDb Compass

MongoDB Compass є офіційним візуальним інтерфейсом для роботи з базою даних MongoDB. Цей інструмент розроблений для полегшення процесу адміністрування, моніторингу та взаємодії з MongoDB через графічний інтерфейс. MongoDB Compass надає розробникам та адміністраторам зручний спосіб дослідження даних, створення запитів та аналізу структури колекцій.

Однією з ключових особливостей MongoDB Compass є візуальний конструктор запитів, який дозволяє користувачам будувати складні запити до бази даних без необхідності вводити JSON-структури вручну. Це робить роботу з даними більш інтуїтивною та зручною, особливо для тих, хто віддає перевагу візуальним коштам.

Крім того, MongoDB Compass надає функціонал для моніторингу та профілювання запитів, що дозволяє відстежувати продуктивність бази даних. Користувачі можуть переглядати виконані запити, аналізувати структуру колекцій та стежити за станом сервера MongoDB.

Інтерфейс MongoDB Compass інтуїтивно зрозумілий та підтримує різні операційні системи, що робить його зручним інструментом для розробників на всіх етапах роботи з MongoDB. Він також забезпечує безпечне з'єднання з сервером бази даних та підтримує роботу з кластерами MongoDB.

Серед аналогів MongoDB Compass можна згадати Robo 3T та Studio 3T, але MongoDB Compass відрізняється своєю офіційною підтримкою, активним оновленням та сумістістю з іншими інструментами екосистеми MongoDB. На рисунку 8 зображено Візуальний інтерфейс MongoDb Compass.

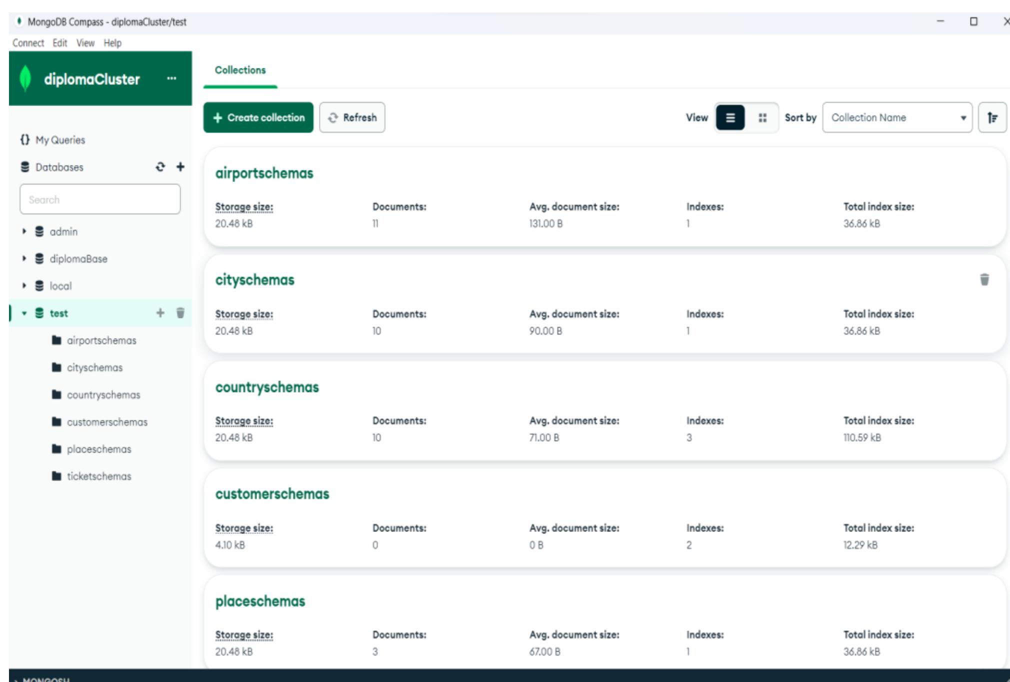

Рисунок 8 — Візуальний інтерфейс MongoDb Compass.

3.2.12 Docker

Docker — це відкрита платформа, що надає засоби для автоматизації розгортання та керування програмами в контейнерах. Контейнери в Docker включають додаток і всі його залежності, такі як бібліотеки та інші

необхідні компоненти, що робить їх переносними і легко масштабуються між різними оточеннями.

Однією з ключових переваг Docker є ізоляція контейнерів від навколишньої системи, що забезпечує консистентність роботи програми у різних умовах. Docker використовує контейнери для пакування додатків та їх залежностей, що робить їх переносимими між різними середовищами виконання.

Docker дозволяє упаковувати програми у контейнери з використанням Dockerfile, де визначаються кроки для створення образу. Ці образи можуть бути легко поширені та розгорнуті на будь-якій системі, що підтримує Docker, що усуває проблеми сумісності та забезпечує єдине оточення виконання.

Однією з ключових концепцій Docker є використання Docker Hub, публічного репозиторію образів, який дозволяє розробникам обмінюватись та використовувати готові образи. Це спрощує процес розгортання додатків, оскільки розробники можуть використовувати стандартні образи, і навіть створювати свої власні задоволення конкретних потреб.

Docker також забезпечує інструменти для оркестрації контейнерів, такі як Docker Compose та Kubernetes, які дозволяють керувати та масштабувати контейнери у розподілених середовищах.

Основні переваги Docker включають простоту у використанні, швидке розгортання, ізоляцію ресурсів, стандартизацію оточення та підтримку мікросервісної архітектури. Docker став широко використовуваним інструментом у розробці та розгортанні додатків, спрощуючи процеси розробки, тестування та обслуговування [38]. На рисунку 9 зображено візуальний інтерфейс контейнера з PostgreSql у Docker.

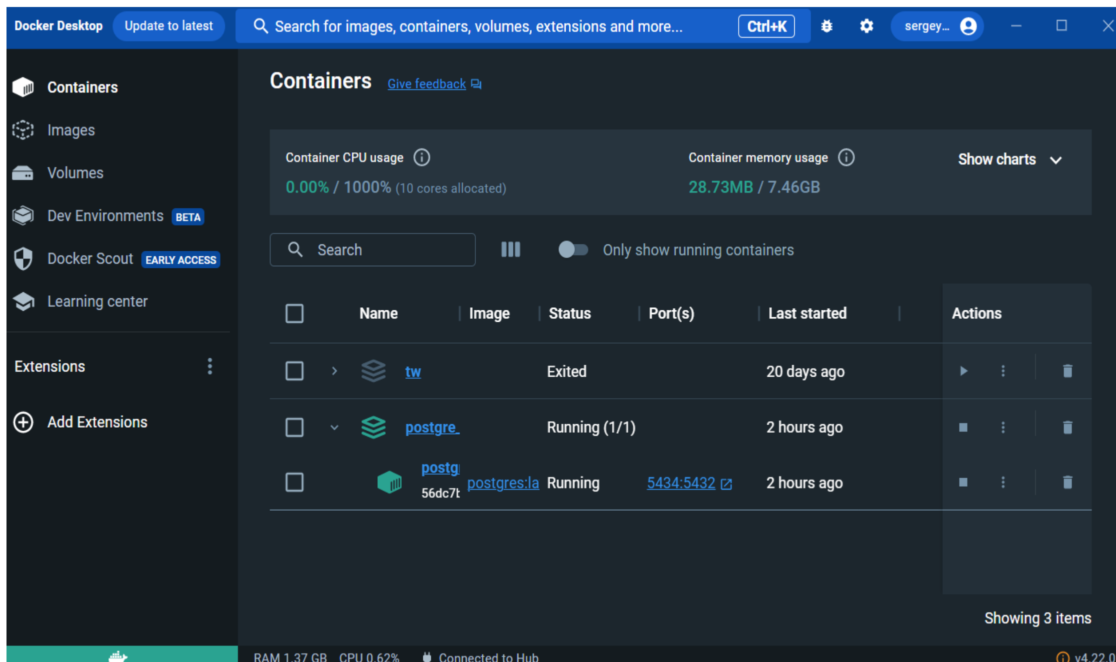

Рисунок 9 — Візуальний інтерфейс контейнера з PostgreSql у Docker.

3.2.13 Postman

Postman — це невід'ємний інструмент в арсеналі розробників, що надає великий функціонал для тестування та розробки веб-сервісів. Цей багатофункціональний клієнт забезпечує комфортну взаємодію з API та спрощує процеси створення, тестування та налагодження HTTP-запитів. Надаючи інтуїтивно зрозумілий інтерфейс користувача, Postman дозволяє розробникам легко виконувати різні завдання.

Важливою особливістю Postman є можливість надсилання HTTP-запитів будь-якого типу: GET, POST, PUT, DELETE та інших. Розробники можуть легко настроювати параметри запитів, включаючи заголовки, параметри запиту, тіло запиту та багато іншого. Це особливо корисно при тестуванні та налагодженні API, де необхідно перевірити різні сценарії взаємодії.

Інструмент також має функціональність управління оточеннями, що дозволяє зберігати та перемикатися між різними наборами змінних оточення залежно від контексту роботи. Такий підхід спрощує тестування на різних етапах розробки та інтеграції.

Postman надає можливості для створення та організації запитів у колекції. Це зручно при роботі з кількома пов'язаними запитами або у разі потреби угруповання запитів для конкретного сценарію. Колекції також можна експортувати та імпортувати, забезпечуючи переносимість тестових сценаріїв між проектами та командами.

Вбудовані засоби тестування API у Postman дозволяють створювати та автоматизувати тести для перевірки коректності відповідей від веб-сервісів. Це суттєво прискорює процес розробки, оскільки дозволяє автоматизувати перевірку функціональності API.

Генерація коду для різних мов програмування – ще одна корисна можливість Postman. Розробники можуть отримати необхідний код для виконання запитів на різних платформах, що полегшує інтеграцію з клієнтськими програмами.

Postman забезпечує можливість спільної роботи, дозволяючи командам обмінюватися колекціями запитів та тестами. Це важливо для забезпечення узгодженості в роботі над API та забезпечення єдиних стандартів.

В цілому, Postman – це потужний інструмент, який полегшує та прискорює процес розробки веб-сервісів, надаючи розробникам зручність, гнучкість та ефективність при роботі з API.

3.2.15 Система контролю версій Git

Git – це розподілена система контролю версій, призначена для відстеження змін у будь-якому наборі файлів. Вона була спочатку розроблена для узгодження роботи програмістів, які співпрацюють над вихідним програмним кодом. Основні цілі Git включають забезпечення швидкості, цілісності даних і підтримки розподілених нелінійних робочих процесів.

Git дозволяє ефективно організовувати командну роботу та зручно працювати над проектами як індивідуально, так і в колективі. Система розгалужень надає інтерфейс для розподілу роботи та управління часом програміста, сприяючи ефективності та організації робочих процесів.

Однією з ключових переваг Git є можливість уникнути зберігання великої кількості архівів на комп'ютері, пропонуючи зручний механізм для перемикання між версіями проекту. За допомогою коммітів файлів система зберігає пронумеровані зміни, забезпечуючи бінарну структуру та підпис версії для кожного комміту.

Стандартний процес роботи з Git включає створення репозиторію, створення гілки і завантаження комміту для ініціалізації контролю версій. Файл .gitignore дозволяє визначити, які файли слід ігнорувати під час завантаження, такі як файли збирання, кеша та модулі проекту.

Система надає зручний процес отримання будь-якої збереженої версії протягом короткого часу. Однак є файли, які не потребують завантаження, і для них використовується .gitignore файл.

Git надає низку команд для роботи з версіями кожного проекту окремо, забезпечуючи гнучкість та ефективність в керуванні версіями проектів. На рисунку 10 зображена система контролю версій Git.

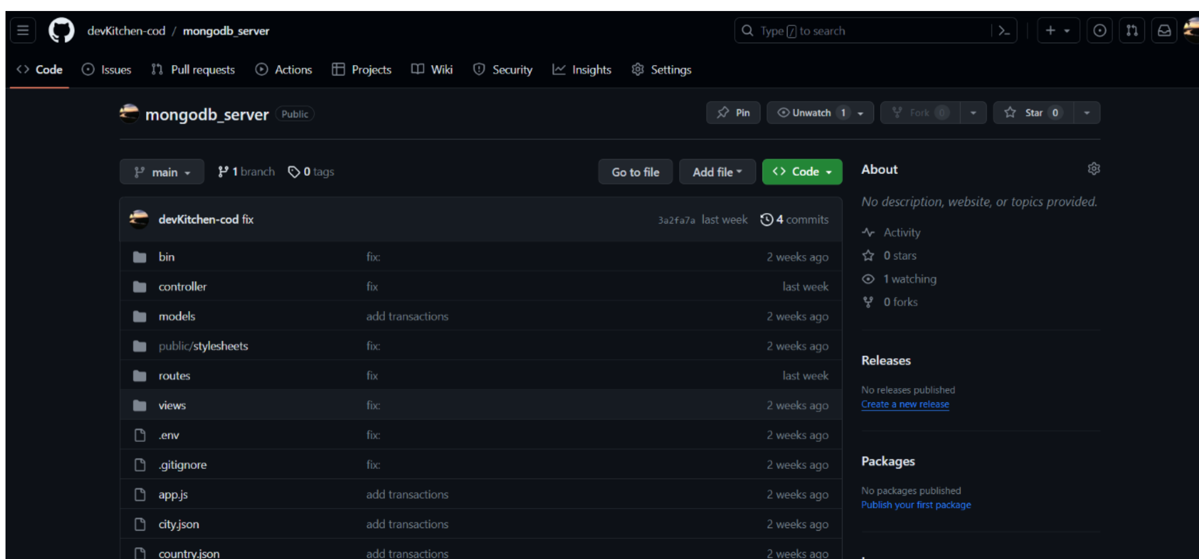

Рисунок 10 — Система контролю версій Git.

3.2.16 MongoDb Atlas

MongoDB Atlas — це керований сервіс, що надає розробникам та компаніям можливість розгортання та масштабування баз даних MongoDB у хмарі. Він відрізняється простотою розгортання через веб-інтерфейс,

дозволяючи створювати кластери MongoDB лише за кілька кліків. Це робить платформу доступною для широкого кола розробників.

Сервіс забезпечує гнучкість масштабування, дозволяючи легко керувати ресурсами бази даних залежно від потреб програми. Висока доступність даних забезпечується автоматичним керуванням реплікацією даних, що гарантує безперебійну роботу програм.

Безпека даних у MongoDB Atlas забезпечується засобами шифрування у спокої та у русі, а також ефективним керуванням доступом, що відповідає високим стандартам безпеки та конфіденційності.

Платформа надає інструменти для резервного копіювання та відновлення даних, що важливо для запобігання втраті інформації у разі збоїв або помилок.

Інтеграція з популярними хмарними платформами, такими як AWS, Azure та Google Cloud, забезпечує гнучкість вибору провайдера хмари та спрощує керування ресурсами.

MongoDB Atlas надає панель управління та інструменти моніторингу, що допомагає розробникам відстежувати продуктивність бази даних та отримувати інформацію про стан кластера [19].

3.3 Початкова конфігурація бекенд-застосунку

Бекенд — застосунок складаэтся з наступних компонентів:

- Головний файл app.js.
- Файл маршрутизації routes.js.
- Папка файли контроллерів.
- Docker — compose файл.
- Папка з данними та кешем бази даних.
- Конфігураційний файл серверу.

3.3.1 Головний файл app.js

Файл app.js є найголовнішим файлом у серверній програмі. Таким чином, під час спуску серверної програми цей файл виступає "точкою старту" для сервера. Так як я використовував фреймворк Express, то насамперед потрібно імпортувати його до файлу app.js а також імпортувати інші важливі бібліотеки і пакети, так як вони будуть використовуватися при запуску сервера.

В лістингу 1 наведені імпорти фреймворку express та інші бібліотеки та пакети.

Лістинг 1 — Імпорт пакетів та бібліотек

```
var express = require("express");
var path = require("path");
var cookieParser = require("cookie — parser");
var logger = require("morgan");
var createError = require("http-errors");
```

Наступною представленою ділянкою коду (див. Лістинг 2) є частина конфігурації серверної програми, що починається з встановлення шляхів представлень та вибору шаблонизатора представлень. Визначено шлях до директорії 'views', де зберігаються файли представлень, і обраний шаблонизатор 'jade' (або Pug) їхньої обробки.

Далі у конфігурації впроваджено middleware. Перший з них — логер у режимі розробки, який фіксує події та стан програми для полегшення налагодження.

Наступні middleware відповідають за обробку даних. Middleware express.json() обробляє JSON-запити, надаючи доступ до даних через об'єкт req.body. express.urlencoded({ extended: false }) аналізує дані, надіслані у формі URL-кодування.

Додатковий елемент конфігурації — middleware для обробки кукіс (cookieParser()), який витягує інформацію із заголовків запитів.

Особлива увага приділяється статичним файлам. Middleware express.static() обслуговує статичні файли з директорії 'public', що включає файли стилів, зображення і скрипти.

Маршрутизація налаштована за допомогою двох контролерів: indexRouter для кореневого маршруту '/' та usersRouter для маршруту '/users'.

Забезпечується підтримка CORS за допомогою middleware cors(), що дозволяє взаємодію з ресурсами з різних джерел.

Нарешті, middleware для обробки помилок 404 передбачає перенаправлення запитів до неіснуючих ресурсів на обробник помилки 404.

Лістинг 2 — Конфігурація пакетів серверу

```
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'jade');

app.use(logger('dev'));
app.use(express.json());
app.use(express.urlencoded({ extended: false }));
app.use(cookieParser());
app.use(express.static(path.join(__dirname, 'public')));

app.use('/', indexRouter);
app.use('/users', usersRouter);
app.use(cors())
app.use(function(req, res, next) {
  next(createError(404));
});
```

У наступному коді, зображеному на лістингу 3, реалізовано механізм перевірки статусу підключення до бази даних перед запуском серверної

програми. Функція checkDatabaseConnection забезпечує взаємодію з базою даних, а функція start координує цей процес.

Функція checkDatabaseConnection є реалізацією механізму перевірки підключення до бази даних. З використанням методу connect об'єкта db (див. Лістинг 3) вона ініціює запит на підключення. У разі виникнення помилки при підключенні відбувається коректне завершення з'єднання, і через колбек передається повідомлення про помилку. У разі успішного підключення, також з наступним коректним завершенням з'єднання, надсилається повідомлення про успішний статус підключення.

Асинхронна функція start є координатором процесу ініціалізації сервера після перевірки стану підключення до бази даних. Шляхом виклику функції checkDatabaseConnection здійснюється запит на статус підключення. Відповідно до результатів перевірки, а саме наявності помилки або її відсутності, в консоль виводяться відповідні повідомлення. У разі успішного статусу підключення, намагається запустити сервер на вказаному порту. Важливо відзначити, що ця функція охоплює можливі помилки та вживає відповідних заходів щодо їх обробки, що сприяє стабільності та надійності процесу ініціалізації.

Узагальнений сценарій функціонування полягає в тому, що спочатку здійснюється перевірка стану підключення до бази даних, після чого залежно від результату ініціюється або виведення повідомлення про успішний статус, або повідомлення про помилку. Далі, у разі успішного статусу, робиться спроба запуску сервера, що є ключовим етапом ініціалізації серверної програми.

Лістинг 3 — Перевірка підключення та підключення до БД

```
const { Pool } = require("pg");

const pool = new Pool({
  user: "user",
  host: "localhost",
```

```
  database: "mydatabase",
  password: "user",
  port: 5434,
});

module.exports = pool;
```

Лістинг 4 — Перевірка підключення та підключення до БД

```
const db = require("./postgres_db/db");

function checkDatabaseConnection(callback) {
  db.connect((error, client, done) => {
    if (error) {
      done();
      callback(error, null);
    } else {
      done();
      callback(null, "Підключення до бази даних успішно
встановлено.");
    }
  });
}

const start = async () => {
  checkDatabaseConnection((error, result) => {
    if (error) {
      console.error("Помилка підключення до бази даних:",
error);
    } else {
      console.log(result);
      try {
        app.listen(PORT, () => console.log(`Server
strated on ${PORT}`));
```

```
      } catch (e) {
        console.log("[error]", e);
      }
    }
  });
};
start();
```

3.3.2 Маршрутизація

Наступний фрагмент коду (див. Лістинг 4) конфігурує маршрути Express.js, призначені для обробки різноманітних запитів HTTP, спрямованих на взаємодію з базою даних. Кожен маршрут співвідноситься з конкретним ресурсом, таким як країни, міста, аеропорти та квитки. Для кожного ресурсу визначено маршрути, що забезпечують створення, читання, оновлення та видалення даних.

Кожен маршрут надсилає запити до відповідного контролера (наприклад, countriesController, citiesController, airportsController), який обробляє запити, взаємодіє з базою даних та формує HTTP-відповіді відповідно до вимог.

Також є маршрут організації транзакції, який призначений для виконання операцій з використанням транзакцій у контексті ресурсу "Country".

Для ресурсу "Place" визначено маршрут getAllPlaces, а також реалізовані маршрути пошуку, такі як searchAirport, searchCountry та searchCity, які ймовірно реалізують пошук відповідних даних (Лістинг 5).

Таким чином я зробив набір маршрутів, що забезпечує структурований і гнучкий інтерфейс для взаємодії з базою даних, надаючи можливість здійснювати різні операції над різними сутностями.

Лістинг 5 — Маршрути ендпоінтів

```
var express = require('express');
var router = express.Router();
```

```
const country_controller = require('../controller/coun-
triesController')
// http://localhost:8080/
router.post('/createCountry', country_controller.create-
Country);
router.get('/getAllCountries', country_controller.get-
AllCountries);
router.post('/getCountryById/:id', country_control-
ler.getCountryById);
router.delete('/deleteCountry/:id', country_control-
ler.deleteCountry);
router.put('/updateCountry/:id', country_controller.up-
dateCountry);
router.get('/useTransaction', country_control-
ler.useTransaction)
const city_controller = require('../controller/citiesCon-
troller')
router.post('/createCity/:id', city_controller.create-
City)
router.get('/getAllCities', city_controller.getAllCities)
router.get('/getCityById/:id', city_control-
ler.getCityById)
router.delete('/deleteCity/:id', city_controller.delete-
City)
router.put('/updateCity/:id', city_controller.updateCity
)
const airport_controller = require("../controller/air-
portsController")
router.post('/createAirport', airport_controller.cre-
ateAirport)
router.get('/getAllAirports', airport_controller.getAl-
lAirports)
router.get('/getAirportById/:id', airport_control-
ler.getAirportById)
```

```
router.delete('/deleteCity/:id', airport_controller.dele-
teAirport)
router.put('/updateCity/:id', airport_controller.up-
dateAirport)
const place = require('../controller/placeController')
router.get('/getAllPlaces', place.getAllPlaces)
const ticket_controller = require('../control-
ler/ticketController')
router.post('/createTicket', ticket_controller.cre-
ateTicket)
router.get('/getAllTickets', ticket_controller.get-
AllTickets)
router.get('/getTicketById/:id', ticket_controller.get-
TicketById)
router.delete('/deleteTicket', ticket_controller.dele-
teTicket)
const search = require('../controller/searchController')
router.post('/searchAirport', search.searchAirport)
router.post('/searchCountry', search.searchCountry)
router.post('/searchCity', search.searchCity)
// router.post
module.exports = router;
```

3.4 Компоненти бекенду для MongoDb

Оскільки MongoDb є документо-орієнтованою базою даних, створення, підключення і використання цієї БД буде відрізнятися від традиційних реляційних баз даних.

Для того, щоб створити базу даних я перейшов на офіційну сторінку MongoDB і вибравши найближчий за моїм географічним розташуванням сервер, створив кластер всередині якого вже в автоматичному режимі MongoDb розгортає базу даних.

На рисунку 11 зображена конфігурація кластеру MongoDb

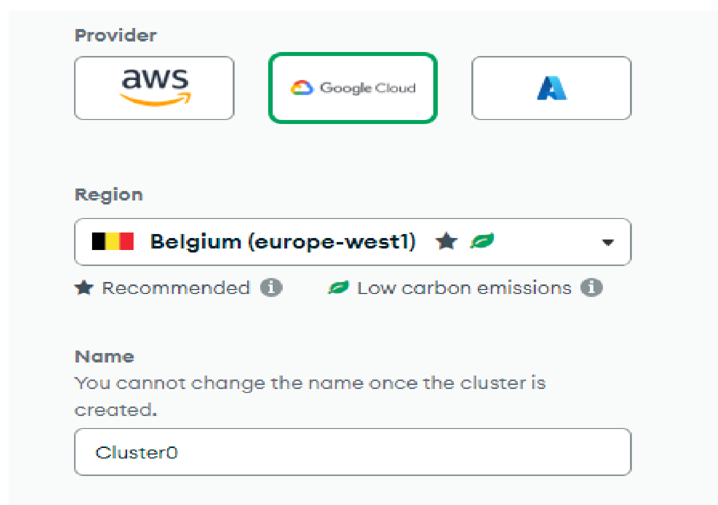

Рисунок 11 — Конфігурація кластеру MongoDb

Далі слідує етап підключення. Для підключення ми повинні використовувати строку підключення, який нам дається при створенні бази даних. На рисунку 12 зображена конфігурація строки підключення до кластеру та БД

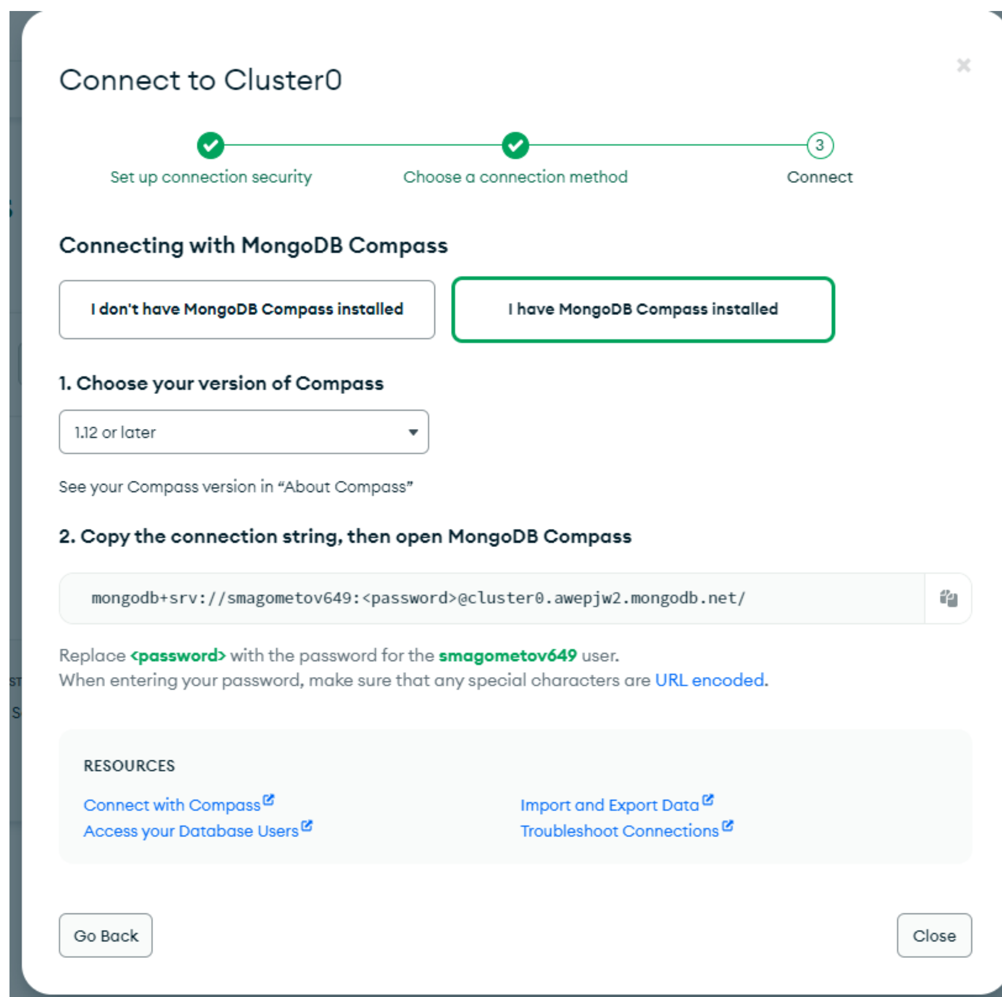

Рисунок 12 — Конфігурація строки підключення до кластеру та БД

Для зберігання рядка підключення і іншої важливої інформації, наприклад, як зберігання порту на якому ми запускаємо свій сервер, я використовував файл .env (див. Лістинг 6). Цей файл призначений для зберігання глобальних змінних, які я зможу використовувати будь-якої миті.

Лістинг 6 — Глобальні змінні

```
PORT=8080
URI=mongodb+srv://smagometov:omega94cskill@diplomaclus-
ter.orj9htk.mongodb.net/
```

Наступний важливий етап це саме підключення до БД. Так, як серверний додаток у рамках дослідження розрахований тільки на використання баз даних, я можу безперешкодно виконати підключення свого сервера до сервера бази даних в рамках однієї функції start() (див. Лістинг 7)

Лістинг 7 — Функція підключення до MongoDb кластеру.

```
const start = async () => {
  try {
    await mongoose.connect(URI)
    console.log("Server is connected to MongoDb")
    app.listen(PORT, () => console.log(`Server started on
${PORT}`))
  } catch (error) {
    console.log(error)
  }
}
```

3.5 Реалізация моделі бази данних в MongoDb

Для початку роботи з базою даних треба визначити схеми колекцій документів відповідно до моделі бази даних. Схеми в базі даних MongoDB є структурами, що визначають формат і організацію даних в колекціях. На відміну від традиційних реляційних баз даних, MongoDB, як NoSQL база даних, не вимагає строгої схеми даних, що дозволяє гнучкіше моделювати інформацію. Схеми MongoDB можуть бути легко змінені та розширені у процесі розвитку програми.

Кожен документ у колекції MongoDB є JSON-подібним об'єктом, який може містити різні поля та значення. Замість використання заздалегідь визначених таблиць з фіксованим набором стовпців, як у реляційних базах даних, MongoDB дозволяє створювати документи з різними полями.

Схеми MongoDB можуть бути простими або вкладеними. Прості схеми являють собою одиничні поля, такі як рядки, числа або булеві

значення. Вкладені схеми дозволяють створювати структуровані документи з вкладеними об'єктами чи масивами.

Додатково MongoDB підтримує індекси для покращення продуктивності запитів. Індекси можуть бути створені для одного або декількох полів у колекції, забезпечуючи швидкий доступ до даних та оптимізацію пошукових запитів.

У MongoDB схеми даних представлені як колекцій, де кожна колекція містить документи з різними полями. Це відбиває динамічну природу NoSQL баз даних, де структура даних може еволюціонувати у часі. Важливо відзначити, що, хоча MongoDB надає свободу в організації даних, хороша практика включає обдумане проектування схеми з урахуванням потреб конкретної програми.

Схеми MongoDB можуть включати різні типи даних, такі як рядки (String), числа (Number), булеви значення (Boolean), масиви (Array), об'єкти (Object), дати (Date), і навіть спеціальні типи даних, такі як ObjectId, що є унікальним ідентифікатором документа.

У контексті MongoDB вкладені схеми часто використовуються для створення складних структур даних. Наприклад, документ, який надає інформацію про людину, може містити вкладені документи для адреси, контактних даних та інших деталей.

Важливою характеристикою схем MongoDB є динамічне додавання полів у документи. Це означає, що програма може динамічно адаптуватися до нових вимог без зміни структури бази даних.

У лістингу 8 я створив схеми даних відповідно до моделі бази даних.

Лістинг 8 — Схеми MongoDb.

```
const mongoose = require("mongoose");
const Schema = mongoose.Schema;
const AirportSchema = new Schema({
  name: { type: String, required: true },
```

```
  city: { type: Schema.Types.ObjectId, ref: "CitySchema"
},
  country: { type: Schema.Types.ObjectId, ref: "Coun-
trySchema" },
  iata_code: { type: String, required: true},
  icao_code: {type: String, required: true},
});
module.exports = mongoose.model("AirportSchema", Air-
portSchema);
const mongoose = require("mongoose");
const Schema = mongoose.Schema;
const CitySchema = new Schema({
  name: { type: String, required: true },
  country: {
    type: Schema.Types.ObjectId,
    ref: "CountrySchema",
    required: true,
  },
  city_code: { type: String, required: true },
});
module.exports = mongoose.model("CitySchema", Cit-
ySchema);
const mongoose = require("mongoose");

const CountrySchema = new mongoose.Schema({
  name: { type: String, required: true, unique: true },
  country_code: { type: String, required: true, unique:
true },
});
module.exports = mongoose.model("CountrySchema", Coun-
trySchema);
const mongoose = require("mongoose");

const CustomerSchema = new mongoose.Schema({
```

```
  name: { type: String, required: true, unique: true },
});
module.exports = mongoose.model("CustomerSchema", Custom-
erSchema);
const mongoose = require("mongoose");

const PlaceSchema = new mongoose.Schema({
    name: {type: String},
    code:{type: String}
})

module.exports = mongoose.model("PlaceSchema",
PlaceSchema);
const mongoose = require("mongoose");

const TicketSchema = new mongoose.Schema({
  code: { type: String },
  place_type: {
    type: mongoose.Schema.Types.ObjectId,
    ref: "PlaceSchema",
    required: true,
  },
  airport_id_from: {
    type: mongoose.Schema.Types.ObjectId,
    ref: "AirportSchema",
    required: true,
  },
  airport_id_to: {
    type: mongoose.Schema.Types.ObjectId,
    ref: "AirportSchema",
    required: true,
  },
  from_city: {
    type: mongoose.Schema.Types.ObjectId,
```

```
    ref: "CitySchema",
    required: true,
  },
  to_city: {
    type: mongoose.Schema.Types.ObjectId,
    ref: "CitySchema",
    required: true,
  },
  price: { type: Number },
});

module.exports = mongoose.model("TicketSchema", Tick-
etSchema);
```

Таким чином при першому запуску методу **.connect(URI)** будуть створені колекції відповідно до схеми БД.

3.6 Опис CRUD операцій до MongoDb бази даних

Оскільки основною метою моєї кваліфікаційної роботи є дослідження особливостей використання документо-орієнтованої бази даних та реляційної бази даних, то одним із важливих аспектів є побудова запитів різних типів. Так як я не ставив за мету створення клієнтської частини, я реалізував для кожної схеми всі необхідні ендпоінти для роботи з ними. Всі ендпоінти включають один з чотирьох функцій роботи з базою даних (CRUD операції) і прийом даних з запиту і відправка результату на клієнт.

Для реалізації були використані стандартні методи NodeJS, наприклад .json() який дозволяє передавати результат запиту в json форматі, а також були використані методи mongoose бібліотеки для запитів в базу даних.

Лістинг 9 — Реалізація ендпоінту для створення нового запису в документі Airport

```
module.exports.createAirport = async function (req, res) {
  const { name, cityId, countryId, iata_code, icao_code }
= req.body;
    console.log(name, cityId, countryId, iata_code,
icao_code);
  try {
    const createdCity = await AirportSchema.create({
      name: name,
      city: cityId,
      country: countryId,
      iata_code: iata_code,
      icao_code: icao_code,
    });
    res.json(createdCity);
  } catch (err) {
    console.error(err);
     res.status(500).json({ error: "Error creating a new
airport" });
```

Цей код реалізує ендпоінт для створення нового аеропорту в базі даних MongoDB. На початку з тіла HTTP-запиту витягуються параметри, такі як назва аеропорту (name), ідентифікатор міста (cityId), ідентифікатор країни (countryId), код IATA (iata_code) та код ICAO (icao_code). Ці параметри виводяться в консоль для налагодження.

Далі ініціюється процес створення нового документа в колекції MongoDB, що представляє аеропорти. Використовується модель AirportSchema, що відповідає схемі даних для аеропортів у базі даних. Створений документ містить такі параметри, як назва, ідентифікатор міста, ідентифікатор країни, коди IATA та ICAO.

У разі успішного створення нового аеропорту сервер повертає JSON-відповідь з даними про створений об'єкт. В іншому випадку, якщо виникає помилка при створенні, сервер логує помилку в консоль і повертає клієнту

HTTP-відповідь з кодом стану 500 та об'єктом JSON, що містить повідомлення про помилку.

Цей ендпоінт надає інтерфейс для взаємодії з базою даних MongoDB, безпосередньо для операції створення нового аеропорту, та забезпечує обробку помилок для забезпечення стабільності роботи сервера за можливих проблем з базою даних.

Лістинг 10 — Реалізація зчите ння даних з колекції Country

```
module.exports.getAllCountries = async function (req, res)
{
  try {
    const country = await CountrySchema.find({});
    res.json(country);
  } catch (err) {
    console.error(err);
    res
      .status(500)
       .json({ error: "Error reading data from countries
table" });
  }
};
```

Операція читання даних із колекції країн (див. Лістинг 9) реалізована за допомогою методу find({}) моделі CountrySchema. Даний метод надає можливість отримати всі документи з колекції, що відповідає запиту на отримання всіх країн.

У блоці try-catch виконується спроба виконання запиту до бази даних. У разі успішного виконання запиту, отримані дані про країни у вигляді масиву документів передаються клієнту у форматі JSON з використанням методу res.json(country) і навпаки — у разі виникнення помилки у процесі виконання запиту, сервер також логує відповідну помилку в консоль.

Метод find({}) у цьому контексті виконує функцію вилучення всіх документів із колекції без використання фільтрів. Такий підхід застосовується

у сценаріях, де потрібно отримати всі доступні дані без будь-яких умов фільтрації.

Лістинг 11 — Реалізація зчитання даних за фільтром

```
module.exports.getCountryById = async function (req, res)
{
  const { params: id } = req;
  if (!id) return res.status(404).json({ message: "ID is
required" });
  try {
    const country = await CountrySchema.findById(id.id);
    console.log("[country]", country);
    res.json(country);
  } catch (error) {
    console.log(error);
    return res.status(500).json(JSON.stringify(error));
  }
};
```

Ендпоінт getCountryById є обробником запиту для отримання даних про країну за її унікальним ідентифікатором. Ідентифікатор витягується з параметрів запиту (req.params) та перевіряється на наявність. У разі відсутності ідентифікатора сервер повертає HTTP-відповідь з кодом 404 і повідомленням про помилку.

Основний функціонал реалізується з використанням методу findById моделі CountrySchema. Цей метод надається бібліотекою Mongoose для роботи з MongoDB у контексті схеми даних. findById приймає ідентифікатор як аргумент та виконує запит до бази даних для пошуку документа з відповідним ідентифікатором.

Метод findById забезпечує зручний спосіб пошуку документа за його унікальним ідентифікатором MongoDB. Він автоматично перетворює

переданий ідентифікатор на формат, зрозумілий базі даних, і виконує ефективний запит для отримання відповідних даних.

Лістинг 12 — Реалізація методу видалення документу.

```
module.exports.deleteTicket = async function (req, res) {
  const { params: cityId } = req;
  console.log(cityId.id);
  try {
    const deletedCity = await TicketSchema.deleteOne({
_id: cityId.id }); //findOneAndDelete
    res.json(deletedCity);
  } catch (err) {
    console.error(err);
    return res.status(500).json({ error: "Error deleting
city" });
  }
};
```

Ендпоінт deleteTicket є обробником запиту для видалення квитка за його унікальним ідентифікатором з бази даних MongoDB. Ідентифікатор витягується із параметрів запиту (req.params). У разі успішного виконання операції видалення сервер повертає JSON-відповідь з інформацією про видалення. У разі помилки сервер відправляє HTTP-відповідь з кодом 500 і повідомленням про помилку.

Лістинг 13 — Реалізація методу оновлення документу.

```
module.exports.updateTicket = async function (req, res) {
  const {
    body: data,
    params: { id },
  } = req;
  console.log(data, id);
```

```
  try {
      const  updatedTicket  =  await  TicketSchema.findOne-
AndUpdate(
      { _id: id },
      { $set: { code: data.code } }
    );
    res.json(updatedTicket);
  } catch (error) {
    console.log(error);
    res.status(500).json({ error: "Error updating city" });
  }
};
```

updateTicket є обробником запиту для оновлення інформації про квиток. Дані для оновлення та унікальний ідентифікатор квитка витягуються із параметрів запиту (req.body та req.params). При успішному виконанні операції оновлення сервер повертає JSON-відповідь з оновленими даними про квиток. У разі виникнення помилки сервер відправляє HTTP-відповідь з кодом 500 і повідомленням про помилку.

Лістинг 14 — Реалізація пошуку за допомогую where() та equals()

```
module.exports.searchCountry = async function (req, res) {
  const { body: data } = req;
  try {
    const country = await CountrySchema.find()
      .where("name")
      .equals(data.name)
      .where("country_code")
      .equals(data.country_code);
    if (country.length === 0) {
      res.status(404).json({ message: "Country not found"
});
    } else {
```

```
      res.status(200).json(country);
    }
  } catch (error) {
    res.status(500).json(JSON.stringify({ error }));
  }
};
```

Методи where, equals використовуються для додавання критеріїв пошуку. У цьому коді вони використовуються для визначення умов пошуку. Наприклад, .where("name").equals(data.name) вказує, що необхідно знайти документи, у яких поле name збігається з переданим значенням data.name, і аналогічно поля country_code.

3.7 Створення реляційної бази даних PostgreSql

У ході розробки бекенд-додатка, орієнтованого на порівняльне дослідження між реляційними базами даних та документо-орієнтованими, постало питання про ефективне розгортання PostgreSQL. Як стратегічне рішення було зроблено вибір на користь контейнеризації за допомогою Docker замість локальної установки. Цей вибір обумовлений кількома факторами, включаючи ізоляцію середовища, управління залежностями та легкість масштабування. Далі будуть розглянуті аргументи, які підтримують переваги розгортання PostgreSQL у контейнерах Docker у порівнянні з локальною установкою, забезпечуючи більш гнучке та кероване оточення для розробки та тестування.

Ізольованість та портативність: Docker контейнери забезпечують ізольоване середовище, в якому працює PostgreSQL. Це дозволяє уникнути конфліктів з іншими додатками та залежностями на хост-системі. Контейнери легко переносяться між різними середовищами (розробка, тестування, продуктивна) і навіть між різними хост-системами.

Управління залежностями: Docker забезпечує керування залежностями та версіями, дозволяючи вам явно вказувати необхідні версії PostgreSQL, бібліотек та інших компонентів. Це позбавляє проблем сумісності та полегшує перенесення додатків між оточеннями.

Легкість встановлення та видалення: Встановлення та видалення програм у Docker здійснюється з використанням контейнерів, що спрощує процес. Ви можете легко створювати, запускати, зупиняти та видаляти контейнери з необхідними додатками, включаючи бази даних, без впливу на хост-систему.

Масштабованість та управління ресурсами: Docker забезпечує можливість ефективного управління ресурсами контейнерів, включаючи виділення обмежених ресурсів (пам'ять, процесорний час) та горизонтальне масштабування.

Розробка та тестування: Використання Docker спрощує процеси розробки та тестування, оскільки контейнери забезпечують однорідні оточення для розробників та тестувальників.

Більш проста установка та оновлення: Docker-образи, що включають PostgreSQL, часто оновлюються та підтримуються спільнотою. Оновлення PostgreSQL у контейнері може бути більш простим процесом порівняно з оновленням, яке проводиться на хост-системі.

Зручність резервного копіювання та відновлення: Docker забезпечує можливість створення знімків (snapshots) контейнерів, що спрощує резервне копіювання та відновлення даних [38].

Використання Docker

Перед початком розгортання бази даних необхідно встановити систему Docker локально. Дотримуючись інструкцій з офіційного сайту Docker, мені вдалося це зробити досить просто.

Для створення контейнера та розгортання у ньому СКБД необхідний образ контейнера. Образ контейнера Docker є легковажним, автономним і виконуваним пакетом, який включає всі необхідні для запуску програми компоненти: код, середовище виконання, системні інструменти, бібліотеки

та налаштування. Ці образи використовуються для створення контейнерів, які є екземплярами виконання цих образів у ізольованому середовищі.

Образ можна створити самостійно так і використовувати вже доступні. Всі доступні образи зберігаються на Docker Hub, у тому числі в ньому доступний і офіційний образ для СКБД PostgreSql від розробників Postgre, тому я скористався саме ним.

Після встановлення образу слід перейти до створення самого контейнера на основі образу, який ми встановили раніше. Для цього створив новий файл docker-compose.yml. **docker-compose.yml** — це конфігураційний файл для Docker Compose, інструмента, який дозволяє визначити і запускати багатоконтейнерні програми в Docker-середовищі. Файл docker-compose.yml містить опис сервісів, їх налаштувань та інші параметри, необхідні для розгортання та конфігурації програм у кількох контейнерах. На рисунку 13 зображена конфігурація docker.

```yaml
docker-compose.yml
version: "3"
services:
  postgres_master:
    image: postgres:latest
    restart: always
    environment:
      - POSTGRES_PASSWORD=user
      - POSTGRES_USER=user
      - POSTGRES_DB=mydatabase
    volumes:
      - ./postgres_db/data_db:/var/lib/postgresql/data
    ports:
      - "5434:5432"
```

Рисунок 13 — Конфігурація docker-compose

3.8 PostgreSQL у Docker: Аналіз docker-compose.yml

У сучасній розробці програмного забезпечення, особливо в контексті багатоконтейнерних програм, ключову роль відіграє ефективне управління даними. У цьому контексті Docker та інструмент Docker Compose надають потужні засоби для розгортання та конфігурування безлічі сервісів в ізольованих контейнерах. У цьому дослідженні представлений конфігураційний файл Docker Compose (docker-compose.yml), призначений для розгортання та налаштування PostgreSQL, однієї з провідних систем управління реляційними базами даних.

Файл конфігурації є описом сервісу postgres_master, який є основним компонентом бази даних PostgreSQL, що розгортається. Кожна частина файлу має власну роль і спрямована на забезпечення надійності, керованості та ефективності в роботі бази даних у контейнеризованому середовищі.

Опис конфігурації:

Сервіс postgres_master:

Задається ім'я сервісу, postgres_master, що визначає контейнер з PostgreSQL як головний майстр бази даних.

Використовується офіційний образ PostgreSQL, версія latest, для забезпечення актуальності та безпеки.

Включено механізм автоматичного перезапуску сервісу з параметром restart: always, що забезпечує безперервну доступність бази даних навіть після збоїв.

Змінні оточення:

Встановлюються змінні оточення для налаштування PostgreSQL:

POSTGRES_PASSWORD, POSTGRES_USER, POSTGRES_DB визначають відповідно пароль, ім'я користувача та ім'я бази даних.

Том даних:

Створюється том для зберігання даних PostgreSQL, визначений як ./postgres_db/data_db:/var/lib/postgresql/data. Це забезпечує збереження даних між перезапуском контейнера та керування станом бази даних.

Прокидання портів:

Визначено прокидання портів з хост-системи на контейнер: "5434:5432". Це забезпечує доступність PostgreSQL на порту 5434 хост-системи, а контейнер слухає на порту 5432.

Даний файл конфігурації є гнучким і легко настроюваним середовищем для розгортання PostgreSQL, дозволяючи взаємодіяти з базою даних з мінімальними зусиллями з налаштування та забезпечуючи високий ступінь ізольованості та портативності в багатоконтейнерній архітектурі.

3.9 Підключення серверної програми до PostgreSQL

Даний фрагмент (див. Ліст.15) коду є конфігурацією та експортом об'єкта Pool з бібліотеки pg, яка використовується для взаємодії з базою даних PostgreSQL у середовищі Node.js. У коді створюється новий об'єкт Pool, який служить керувати пулом з'єднань з базою даних. Налаштування конфігурації передають облікові дані для підключення до бази даних, такі як ім'я користувача (user), хост (host), назва бази даних (database), пароль (password) і порт (port). Після налаштування об'єкта Pool він експортується для використання в інших модулях програми, що дозволяє ефективно керувати з'єднаннями з базою даних у різних частинах програми.

Лістинг 15 — Підклчення до БД

```
const { Pool } = require("pg");

const pool = new Pool({
  user: "user",
  host: "localhost",
  database: "mydatabase",
  password: "user",
  port: 5434,
});

module.exports = pool;
```

3.10 Опис CRUD операцій у реляційній базі даних

Лістинг 16 — Ендпоінт Create операції

```
const client = require("../postgres_db/db");

module.exports.createAirport = async function (req, res)
{
```

```
  const { name, city, iata_code, icao_code } = req.body;
  try {
    const result = await client.query(
      "INSERT INTO airports (name, city_id, iata_code, icao_code) VALUES ($1, $2, $3, $4) RETURNING *",
      [name, city, iata_code, icao_code]
    );
    res.json(result.rows[0]);
  } catch (err) {
    console.error(err);
    res.status(500).json({ error: "Error creating a new airport" });
  }
};
```

Цей код є серверним обробником запиту для створення нового аеропорту в базі даних.

Код починається з підключення модуля client, який є клієнтом для взаємодії з базою даних PostgreSQL.

Далі визначено функцію createAirport, яка є асинхронною функцією. Вона приймає два аргументи – req (об'єкт запиту) та res (об'єкт відповіді). У тілі функції відбувається вилучення необхідних параметрів створення нового аеропорту з тіла запиту (req.body). Ці параметри включають name (назва аеропорту), city (ідентифікатор міста), iata_code та icao_code (коди аеропорту).

Далі виконується спроба виконати запит до бази даних із використанням client.query. Запит використовує SQL-запит для вставлення даних у таблицю аеропортів. Значення вставляються з параметрів функції. У разі успішного виконання запиту результат відправляється у відповідь клієнту у форматі JSON через res.json(result.rows[0]).

У разі виникнення помилки (помилки при виконанні запиту до бази даних) код спіймає виняток, виведе повідомлення про помилку в консоль

(console.error(err)) і відправить клієнту відповідь з кодом стану 500 (внутрішня помилка сервера) і повідомленням про помилку в формат JSON (res.status(500).json({ error: "Error creating a new airport" })).

Таким чином, цей код відповідає за обробку запиту на створення нового аеропорту, вставку відповідних даних у базу даних та повернення результату або обробку помилки у разі невдачі.

Лістинг 17 — Реалізація API-ендпоінту для отримання зведеної інформації

```
module.exports.summaryAirports = async function (req, res) {
  try {
    const query = `
    SELECT
    airports.name AS airport_name,
    cities.name AS city_name,
    countries.name AS country_name
FROM
    airports
    JOIN cities ON airports.city_id = cities.id
    JOIN countries ON cities.country_id = countries.id;
        `;
    const result = await client.query(query);
    res.json(result.rows);
  } catch (err) {
    console.error(err);
    res.status(500).json({
      error: "Error retrieving data from airports and related tables",
    });
  }
};
```

Цей код є серверним обробником запиту для отримання зведеної інформації про аеропорти, міста та країни з відповідних таблиць бази даних. Функція summaryAirports використовує асинхронний підхід та приймає параметри req (об'єкт запиту) та res (об'єкт відповіді). Всередині функції формується SQL-запит для вибору даних із бази даних, що включають назви аеропортів, міст та країн. У разі успішного виконання запиту, результат відправляється у відповідь клієнту у форматі JSON, в іншому випадку обробляється помилка, виводиться повідомлення про помилку в консоль, і клієнту повертається відповідна відповідь з кодом стану 500 (внутрішня помилка сервера) та повідомленням про помилку.

Лістинг 18 — Реалізація API-ендпоінту для видалення аеропорту

```
module.exports.deleteAirport = async function (req, res)
{
  const airportId = req.params.id;
  try {
    await client.query("DELETE FROM airports WHERE id =
$1", [airportId]);
    res.json({ message: "Airport deleted successfully"
});
  } catch (err) {
    console.error(err);
    res.status(500).json({ error: "Error deleting air-
port" });
  }
};
```

Функція deleteAirport є асинхронною та приймає параметри req (об'єкт запиту) та res (об'єкт відповіді). Всередині функції витягується ідентифікатор аеропорту з налаштувань запиту (req.params.id), після чого виконується запит до бази даних для видалення запису з відповідним ідентифікатором. У разі успішного виконання операції, у відповідь клієнту

надсилається повідомлення про успішне видалення. У разі помилки виводиться повідомлення про помилку в консоль і клієнту повертається відповідна відповідь з кодом стану 500 (внутрішня помилка сервера) та повідомленням про помилку.

Лістинг 19 — Реалізація API-ендпоінту для оновлення інформації про країну

```
module.exports.updateCountry = async function (req, res)
{
  const {
    body: data,
    params: { id },
  } = req;
  try {
    const updateQuery = await client.query(
      `UPDATE countries SET name=$1, country_code=$2
WHERE id = $3`,
      [data.name, data.country_code, id]
    );
    res.json(updateQuery);
  } catch (error) {
    console.log(error);
    res.status(500).json({ error: "Error updating coun-
try" });
  }
};
```

Функція updateCountry є асинхронною та приймає параметри req (об'єкт запиту) та res (об'єкт відповіді).

У тілі функції використовується деструктуризація об'єкта req для отримання даних з тіла запиту (body: data) і параметрів запиту (params: { id

}). Отримані дані включають нове ім'я країни (data.name), новий код країни (data.country_code) та ідентифікатор країни (id).

Далі виконується спроба оновлення даних у базі даних із використанням запиту UPDATE. У разі успішного виконання запиту результат оновлення (updateQuery) відправляється у відповідь клієнту у форматі JSON через res.json(updateQuery). У разі виникнення помилки виводиться повідомлення про помилку в консоль (console.log(error)) і клієнту повертається відповідь з кодом стану 500 (внутрішня помилка сервера) та повідомленням про помилку у форматі JSON (res.status(500).json({ error: "Error updating country" })).

3.11 Транзакції в реляційних БД та NoSql

Транзакції є важливою частиною баз даних, забезпечуючи надійність та цілісність даних під час виконання операцій. У даному контексті проаналізуємо та порівняємо реалізації транзакцій у двох різних типах баз даних – реляційних та NoSQL. Для ілюстрації використовуються приклади коду Node.js з використанням PostgreSQL і MongoDB.

Транзакція є послідовністю однієї або декількох операцій бази даних, які повинні бути виконані атомарно, тобто або всі успішно виконуються, або жодна. Вона також забезпечує ізоляцію операцій, стійкість та можливість відкату до попереднього стану у разі помилки [22].

Лістинг 20 — Реалізація транзакції для PostgreSql

```
module.exports.getAllCountriesTransaction = async func-
tion (req, res) {
  try {
    await client.query("BEGIN");
    const result = await client.query("SELECT * FROM
countries");
    await client.query("COMMIT");
```

```
    res.json(result.rows);
  } catch (err) {
    console.error(err);
    res
      .status(500)
      .json({ error: "Error retrieving data from coun-
tries table" });
  }
};
```

Наведений код (див. Ліст. 20) демонструє використання транзакцій у PostgreSQL. Спочатку виконується команда BEGIN для початку транзакції, потім виконується запити на вибір усіх країн та міст з таблиці countries та cities відповідно, і нарешті команда COMMIT завершує транзакцію. У разі помилки використовується ROLLBACK для відкату змін.

Лістинг 21 — Реалізація транзакції у MongoDb

```
module.exports.useTransaction = async function (req, res)
{
  const session = await CountrySchema.startSession();
  let temp;
  const resOfTransaction = await session.withTransac-
tion(async () => {
    try {
      const allCountries = await Coun-
trySchema.find().session(session);
      const allCities = await CitySchema.find().ses-
sion(session);
      console.log("Result of transaction:", allCountries,
allCities);
      temp = { allCountries: allCountries, allCities:
allCities };
    } catch (error) {
```

```
      console.log(error);
      res.status(500).json({ message: "error" });
    } finally {
      session.endSession;
    }
  });
  if (!Object.values(resOfTransaction).length) {
    console.log("null"), res.status(500);
  }
  console.log("resOfTransaction", resOfTransaction);
  res.status(200).json({ resOfTransaction: resOfTransac-
tion, result: temp });
};
```

У наведеному коді (див. Лiст. 21) для MongoDB використовується сесія для керування транзакцією. Сесія створюється за допомогою startSession(), і весь блок операцій всередині з Transaction вважається атомарним. При успішному виконанні зміни фіксуються за допомогою session.commitTransaction, інакше виконується відкат з session.abortTransaction.

Обидва підходи забезпечують транзакційність, але в PostgreSQL використання SQL-запитів і явної вказівки початку і завершення транзакції, тоді як у MongoDB - використання сесій та блоку операцій з транзакцією.

Сесії в MongoDB є механізмом управління транзакціями, призначений для забезпечення консистентності даних у базі даних. Ці сесії були введені з метою забезпечити атомарність операцій, підтримуючи цим транзакційну модель взаємодії з базою даних.У контексті MongoDB сесія асоціюється з клієнтом (MongoClient) та забезпечує виконання групи операцій як єдиний, атомарний блок.

Реляційні бази даних надають жорсткіший контроль над транзакціями та атомарністю, тоді як NoSQL бази даних, зберігаючи певний ступінь гнучкості, вимагають явного використання сесій для транзакцій.

В результаті проведення дослідження реляційних та нереляційних баз даних було розроблено два серверні застосунки: один для MongoDB, інший для PostgreSQL. Кожен з цих застосунків надає API для взаємодії з відповідною базою даних. Основною метою цього дослідження є аналіз особливостей використання реляційних та нереляційних баз даних у контексті розробки веб-застосунків.

Спроектовано та реалізовано API для обох застосунків, і їх функціональність спрямована на легкість у проведенні подальших досліджень замість використання як готового програмного продукту для роботи з користувацькими колекціями.

Результати цього дослідження будуть використані для ретельного порівняння ефективності та особливостей використання реляційних та нереляційних баз даних у веб-розробці.

РОЗДІЛ 4 ЕКСПЕРИМЕНТАЛЬНІ ДОСЛІДЖЕННЯ

4.1 Дослідження методів реалізації моделі даних

У PostgreSQL модель даних описується за допомогою мови SQL, яка включає в себе створення таблиць та визначення їхньої структури. Кожна таблиця (див. Лістинг 22) має стовпці з визначеними типами даних (VARCHAR, serial, INT), а взаємозв'язки між таблицями встановлюються за допомогою зовнішніх ключів (REFERENCES).

Лістинг 22 — Створення таблиці Cities

```
CREATE TABLE cities (
    id serial PRIMARY KEY,
    name VARCHAR(255),
    country_id INT REFERENCES countries(id),
    city_code CHAR(3) UNIQUE
);
```

В той же час у MongoDB модель даних будується на основі вбудованих та посилальних документів за допомогую JavaScript та бібліотеки Mongoose (див. Лістинг 23). Вбудовані документи можуть включати однорівневу структуру внутрішніх документів, тоді як посилальні документи використовують ObjectId для встановлення зв'язків між документами. Також визначаеться структура документа, а саме типи полів (String, Int, та ін.) тоді як посилання реалізується шляхом визначення поля ref назвою схеми посилання.

Лістинг 23 — Створення коллекції City в MongoDb

```
const CitySchema = new Schema({
  name: { type: String, required: true },
  country: {
    type: Schema.Types.ObjectId,
```

```
    ref: "CountrySchema",
    required: true,
  },
  city_code: { type: String, required: true },
});
```

У PostgreSQL структура даних визначається явно з використанням SQL, що робить модель досить простою та прозорою. У MongoDB вбудовані та посилальні документи можуть зробити модель більш гнучкою, але це може призвести до складнішої структури та меншої чіткості.

4.2 Дослідження гнучкості схеми даних

В PostgreSQL гнучкість схеми даних досягається визначенням опціональних та обов'язкових полів при створенні таблиць. Використання різних типів даних і обмежень для полів дозволяє точно налаштовувати структуру даних. Наприклад, ключові слова NULL та NOT NULL вказують на можливість або обов'язок вводу значень у відповідному полі.

У MongoDB, гнучкість схеми даних досягається завдяки динамічній схемі, яка дозволяє документам в одній колекції мати різні поля. Нові поля можуть додаватися без необхідності визначення їх структури наперед. Такий підхід забезпечує велику гнучкість при змінах у вимогах до даних.

У ситуації, коли потрібні термінові зміни у структурі документа (потрібно додати нове поле), то в MongoDB є можливість зберегти документ, структура якого відрізняєтся від тієї структури, що вказана в схемі даних, але тільки тоді, коли схема є nonStrict (див. Ліст 24).

Лістинг 24 — Реалізация nonStrict схеми для коллекції країн

```
const CountrySchema = new mongoose.Schema({
  name: { type: String, required: true, unique: true },
```

```
  country_code: { type: String, required: true, unique: true },
}, {strict: false});
```

Таким чином, при запиті на ендпоінт createCountry ми можемо додати нові поля напряму до самого запиту (див. Лістинг 25) і данні будуть зберігатися.

Лістинг 25 — Реалізація запиту с новим полем

```
const newCountry = await CountrySchema.create({
      name: name,
      country_code: country_code,
      created_at: new Date().getTime(),
    });
```

Важливо відмітити що хоч це і є основною відмінністю від реляційних БД, в реальних проектах це погана практика на мою думку. Якщо використовувати несуворий режим схем даних, тоді в такому випадку схема даних не виконуватиме одну зі своїх основних задач, поставлених розробниками MongoDb, а саме перевірки даних. Допустимими в схемі є механізм, який забезпечує відповідність даним визначеним правилам, встановленим у схемі. Перевірка виконується при спробі зберегти новий документ у колекції бази даних або оновити існуючий документ. В кінцевому підсумку, валідація це важливий механізм для роботи з даними, при використанні якого зберігається повноцінність даних.

Оскільки PostgreSql це реляційна база даних, в ній немає динамічної схеми, тому при виконанні запиту на створення нового запису, де вказана колонка, якої немає в початковій схемі даних, запит видасть помилку (Рис. 32)

```
server strated on 8081
error: column "created_at" of relation "countries" does not exist
    at C:\Users\Сергей\Documents\GitHub\postgre_server\node_modules\pg-pool\index.js:45:11
    at process.processTicksAndRejections (node:internal/process/task_queues:95:5)
    at async module.exports.createCountry (C:\Users\Сергей\Documents\GitHub\postgre_server\controller\countriesController.js:7:20) {
  length: 132,
  severity: 'ERROR',
  code: '42703',
  detail: undefined,
  hint: undefined,
  position: '44',
  internalPosition: undefined,
  internalQuery: undefined,
  where: undefined,
  schema: undefined,
  table: undefined,
  column: undefined,
  dataType: undefined,
  constraint: undefined,
  file: 'parse_target.c',
  line: '1052',
  routine: 'checkInsertTargets'
}
POST /createCountry 500 191.085 ms - 40
```

Рисунок 14 — Помилка виконання запиту createCountry в PostgreSql

4.2.1 Дослідження тразакційної підтримки

Правила для РСКБД були запропоновані Коддом і опубліковані в 1985 році, але не всі існуючі РСКБД повністю відповідають цим правилам через їх високу строгість.

РСКБД, як транзакційні системи, дотримуються принципів ACID :

1. **Атомарність**: гарантує, що транзакція або виконується повністю, або не виконується зовсім.
2. **Узгодженість** (consistency): забезпечує, щоб успішно завершені транзакції не порушували узгодженості бази даних.
3. **Ізольованість** (isolation): обмежує виконання транзакцій, щоб вони не впливали на результати інших транзакцій.
4. **Надійність**: результати успішно завершеної транзакції зберігаються навіть у разі аварійного завершення роботи.

Дотримання цих принципів дозволяє передбачити поведінку системи та забезпечує цілісність бази даних.

У результаті основними перевагами реляційних баз даних є цілісність і структурованість даних, а також зниження ймовірності несподіваної поведінки системи.

Транзакції в PostgreSQL використовуються для забезпечення атомарності, консистентності, ізоляції та довіреності (ACID) операцій з базою

даних. Транзакції дозволяють групі операцій виконуватися як єдиний незалежний блок, який забезпечує ці ACID-властивості.

Основні команди, пов'язані з транзакціями в PostgreSQL:

1. Команда BEGIN починає нову транзакцію. Усі операції після цієї команди вважаються частинами транзакції
2. Команда COMMIT фіксує зміни, внесені в базу даних, завершуючи транзакцію. Це підтвердження, що всі зміни були успішно внесені.
3. Команда ROLLBACK скасовує всі зміни, внесені в базу даних під час поточної транзакції. Вона викликає відкат транзакції.
4. SAVEPOINT дозволяє створювати точки збереження в межах транзакції, а ROLLBACK TO SAVEPOINT дозволяє відкочуватися до конкретної точки збереження.
5. Використовуючи блок EXCEPTION, можна реалізувати обробку винятків в транзакціях (див. Лістинг 26).

Лістинг 26 — Приклад обробки вийнятків

```
BEGIN
  -- операції
EXCEPTION
  WHEN others THEN
    -- обробка винятків
    ROLLBACK;
END;
COMMIT;
```

6. PostgreSQL має режим AUTOCOMMIT, в якому кожна команда SQL автоматично виконується в окремій транзакції. Цей режим ввімкнено за замовчуванням.

Транзакції в MongoDB є механізмом для групування декількох операцій в єдиний атомарний блок. Вони дозволяють вам виконувати набір

операцій як один незалежний блок, який гарантує ACID-властивість (атомарність, консистентність, ізоляція та довіреність) для цих операцій.

1. Сесія в MongoDB дозволяє утримувати стан транзакції та реалізовує механізм блокування для управління конфліктами у доступі до ресурсів. Сесії створюються за допомогою startSession().
2. Транзакцію розпочинаємо за допомогою withTransaction(). При цьому вказується функція, яка містить набір операцій для виконання в межах транзакції.
3. Транзакція фіксується автоматично, якщо блок операцій в withTransaction() виконується без помилок. Фіксація також може бути викликана явно за допомогою commitTransaction().
4. За допомогою параметрів readConcern та writeConcern можна налаштовувати рівень ізоляції та контролювати поведінку запису.

Транзакції в MongoDB використовують оптимістичний механізм блокування, що дозволяє багатьом транзакціям одночасно працювати з даними, але при цьому вони здатні виявляти та вирішувати конфлікти при фіксації транзакції.

Цей механізм дозволяє MongoDB забезпечити високу масштабованість та швидкодію в розподілених середах. Однак важливо враховувати, що транзакції в MongoDB підтримуються лише в режимі replica set або в кластерах MongoDB, які використовують шардування.

4.3 Дослідження масштабованості

Масштабування баз даних Postgres і MongoDB є важливим аспектом для забезпечення високої продуктивності та доступності в розподілених системах.

PostgreSQL:

PostgreSQL підтримує як вертикальне, так і горизонтальне масштабування. Для вертикального масштабування можна збільшити ресурси

окремого сервера, додаючи процесори, пам'ять чи інші ресурси. Це дозволяє оптимізувати продуктивність на рівні окремого вузла.

Щодо горизонтального масштабування PostgreSQL підтримує розподілення навантаження за допомогою реплікації та шардування. Реплікація в PostgreSQL може бути реалізована за допомогою Master-Slave або Master-Master архітектур. Важливо відзначити, що шардування в PostgreSQL потребує більше ручного керування та конфігурації.

MongoDB:

MongoDB також підтримує масштабування, що реалізується таким же чином як і в PostgreSQL, але має певні відмінності. MongoDB покладається на горизонтальне масштабування, зокрема шардування. Дані можуть бути розділені між кількома серверами (шардами), що дозволяє ефективно обробляти великі обсяги даних. Також MongoDB має свій хмарний сервіс для керування БД - MongoDB Atlas.

MongoDB Atlas спрощує горизонтальне масштабування за допомогою автоматизованого шардування. Користувачам не потрібно проводити складні кроки для розподілу даних, оскільки це автоматизовано під капотом.

MongoDB Atlas також надає можливість вибору різних рівнів продуктивності та резервне копіювання даних у хмарних сервісах. Це спрощує процес адміністрування та надає можливість ефективно використовувати хмарні ресурси.

Одним з ключових відмінностей є те, що MongoDB Atlas призначений для легкого впровадження та управління в хмарних сервісах, тоді як PostgreSQL може вимагати більшої конфігурації та управління, особливо при розгортанні на своїх серверах.

Способи реплікації:

MongoDB використовує реплікацію за принципом Master-Slave або Replica Sets. Replica Sets дозволяють створити кілька копій основних (primary) та вторинних (secondary) вузлів для забезпечення доступності та надійності.

PostgreSQL також підтримує реплікацію, але може використовувати різні підходи, такі як асинхронна реплікація, синхронізація та каскадна реплікація.

4.4 Аналіз мов запитів

MongoDB використовує мову запитів MongoDB (MQL), що надає можливість взаємодії з документо-орієнтованою структурою MongoDB. MQL володіє розширеним набором функцій і підтримує проекцію, агрегаційні операції, запити до документів, конвеєри агрегації, геопросторові запити та текстовий пошук. Також слід відзначити, що при використанні різних драйверів для MongoDB, можна відступити від використання MQL і перейти до мови програмування, яка використовується у бекенд-застосунку.

У випадку PostgreSQL використовується варіант SQL під назвою Postgres SQL як мова запитів. Хоча вона схожа на SQL, вона має додаткові можливості, такі як розширювана система типів, функції та наслідування. Однак PostgreSQL залишається сумісним із стандартним SQL, тому ви також можете використовувати SQL-запити. Також слід відзначити, що при використанні різних ORM для PostgreSQL, можна також використовувати мову програмування, яка використовується у бекенд-застосунку.

Таблиця 1 — Підтримка різних мов програмування

База Даних	**Мови програмуваня**
MongoDb	JavaScript (Node.js) Python Java Ruby C# PHP Go Swift Scala
PostgreSql	C C++ Java Python Ruby PHP Go JavaScript (Node.js)

Таблиця 2 — Підримка драйверів баз даних наступних фреймворків

База Даних	**Фреймворки**
MongoDb	Express.js (Node.js) Flask (Python) Spring (Java) Ruby on Rails (Ruby) Meteor (JavaScript) Django (Python)
PostgreSql	Django (Python) Ruby on Rails (Ruby) Spring (Java) Laravel (PHP) Express.js (Node.js) Hibernate (Java)

4.5 Аналіз спільноти, підтримки та документації

Аналіз спільноти, підтримки та документації є важливим аспектом при виборі між MongoDB та PostgreSQL як системами управління базами даних.

Спільнота MongoDB велика та активна. MongoDB має широкий розподіл допоміжних екосистем, включаючи офіційний форум, блоги, а також активні спільноти на платформах соціальних мереж. Регулярно проводяться конференції та події, спрямовані на обмін досвідом та знаннями.

Офіційна документація MongoDB є детальною та зрозумілою. MongoDB також активно підтримується командою розробників, і нові версії та виправлення помилок регулярно випускаються.

PostgreSQL також має сильну та здорову спільноту користувачів. Форуми, блоги та соціальні мережі активно використовують для обговорення

проблем, обміну порадами та підтримки один одного. Крім того, існує велика кількість сторонніх ресурсів, присвячених PostgreSQL.

Офіційна документація PostgreSQL є детальною та широко покриває всі аспекти системи. Команда розробників PostgreSQL також активно підтримує проект і регулярно випускає нові версії з новими можливостями та виправленнями.

В обох випадках, велика та добре підтримувана спільнота, а також докладна та актуальна документація, роблять MongoDB та PostgreSQL популярними виборами серед розробників та архітекторів баз даних.

4.7 Дослідження продуктивності

4.7.1 Підготовка до дослідження

При створенні індексу я дотримувався основних правил і рекомендацій, які відносяться як до реляційних баз даних, так і для NoSql БД, а саме:

Унікальність значень: Виберіть стовпець, у якому значення є унікальними для кожного рядка. Унікальні значення забезпечують ефективніший доступ до даних.

Частий доступ до даних: Якщо у вас є часті запити до певних значень, виберіть стовпець, який часто використовується в умовах пошуку, фільтрації або сортування.

Сортування та діапазони: Якщо виконується сортування даних або проводите запити у певному діапазоні значень, виберіть стовпець, який добре сортується та має безперервні значення.

Тип даних: Перевірте, чи вибраний стовпець має відповідний тип даних для індексу. Наприклад, цілі стовпці зазвичай ефективніше рядкових.

Я припустив, що у таблиці квитків у стовпці price лежатимуть саме ті дані, які підходять за цими правилами. У такому випадку мені знадобилися великі обсяги даних, для створення яких була створена кінцева точка для обох баз даних (див. Ліст. 27 та 28), яка в якості параметра приймає кількість

записів/документів і в результаті генерації даних видає файл csv/json відповідно, після чого я зміг використовувати їх для імпорту даних.

Лістинг 27 — Кінцева точка для генерації даних csv для PostgreSql

```
module.exports.generateData = async function (req, res) {
  const { body: count } = req;
  let arr = [];
  for (let i = 0; i < count.count; i++) {
    const temp = {
      code: "T" + parseInt((Math.random() * 999999 +
1).toFixed(0)),
      place_type: parseInt((Math.random() * 2 +
1).toFixed(0)),
      airport_id_from: 1,
      airport_id_to: 2,
      from_city: 1,
      to_city: 2,
      price: i+1,
      created_at: generateRandomDate(new Date(2010, 5,
25), new Date()),
    };
    arr.push(temp);
  }
  function generateRandomDate(from, to) {
    return new Date(
      from.getTime() + Math.random() * (to.getTime() -
from.getTime())
    );
  }
  try {
    const parser = new Parser();
    let csv = parser.parse(arr);
    fs.writeFileSync("csv", csv);
```

```
    console.log(`Created ${count.count} tickets`)
  } catch (error) {
    console.log(error);
    res.status(500).json({ error: "Error creating a new
country" });
  }
};
```

Лістинг 28 — *Кінцева точка для генерації даних MongoDb*

```
module.exports.generateTickets = async function (req,
res) {
  const { body: count } = req;
  console.log(count);
  let arr = [];
  for (let i = 0; i < count.count; i++) {
    let temp = {
      code: `${i}`,
      place_type: {
        $oid: "654b6a0b32d5bfc2ab7ff050",
      },
      airport_id_from: {
        $oid: "654b801df4b97145405e0d67",
      },
      airport_id_to: {
        $oid: "654b808ff4b97145405e0d69",
      },
      created_at: {
        $date: generateRandomDate(new Date(2010, 5, 25),
new Date()),
      },
      deleted_at: null,
      price: 100 + i,
    };
```

```
    arr.push(temp);
  }
  console.log("arr.length", arr.length);
  fs.writeFileSync("tickets", JSON.stringify(arr));

  function generateRandomDate(from, to) {
    return new Date(
      from.getTime() + Math.random() * (to.getTime()  — from.getTime())
    );
  }
};
```

4.7.2 Дослідження продуктивності MongoDb

Оскільки у MongoDb є хмарний сервіс MongoDb Atlas, а також MongoDb Compass який надає зручний інтерфейс для роботи з базою даних, я скористався цією можливістю. Для того, щоб визначити швидкість виконання запиту, слід увійти в режим Explain який в свою чергу покаже нам результати тестів на продуктивність. У цьому режимі є кілька видів сканування:

CALLSCAN – сканування зазначеної колекції шляхом виконання раніше вказаного запиту. Результатом сканування є час виконання запиту і логи журналу планувальника запитів всередині кластера, і логи статусу виконання запиту в якому знаходиться внутрішньосерверний час виконання запиту.

FETCH — отримання документів, заданих у параметрах запиту. Проводить повне сканування всієї колекції Результатом виконання запиту є виведення логів статусу виконання даного запиту.

IXSCAN — Схоже на CALLSCAN сканування колекції та її документів, але з використанням встановлених індексів для даної колекції. Результатом виконання запиту є час сканування індексів.

Також у правій частині модального вікна результатів оцінки продуктивності () можна побачити час виконання самого вказаного тестового запиту (execution time) а також кількість повернутих документів, перевірених документів, перевірених індексів і мітка яка визначає був використаний індекс чи ні.

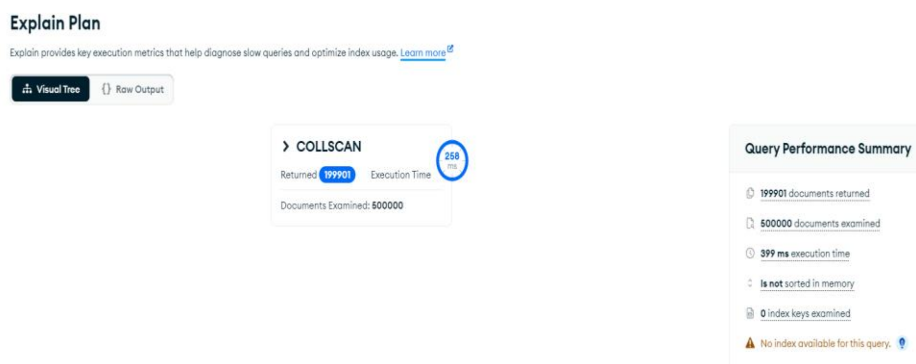

Рисунок 15 — Модальне вікно результатів тестів без індексу

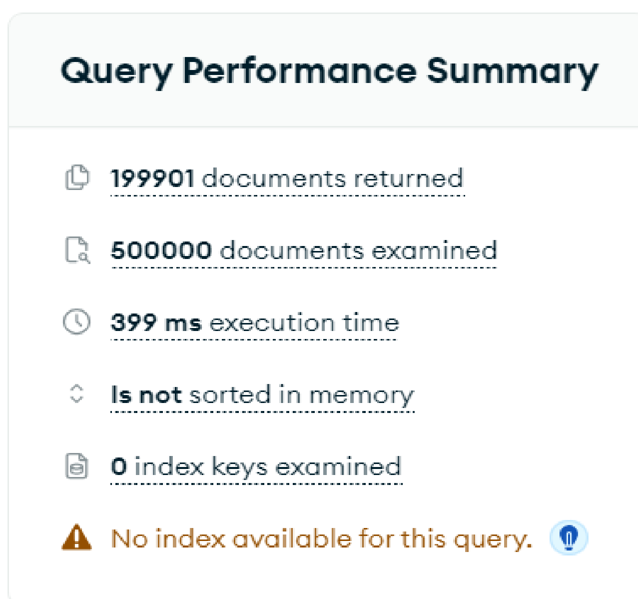

Рисунок 16 — Результати виконання запиту без індексу

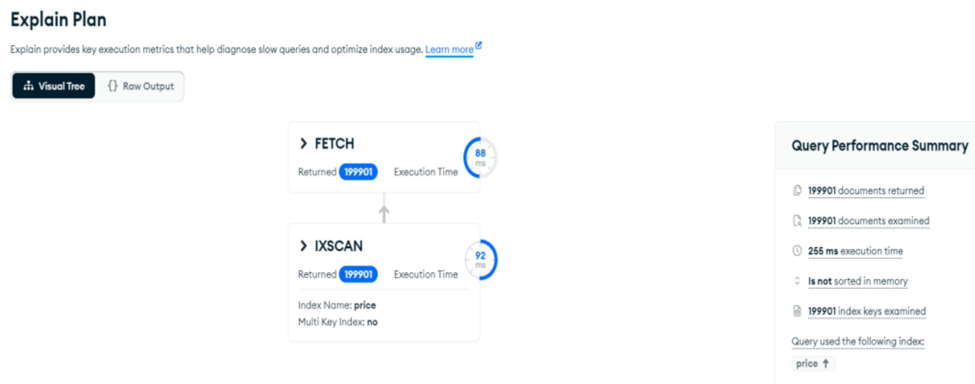

Рисунок 17 — Модальне вікно результатів тестів з індексом

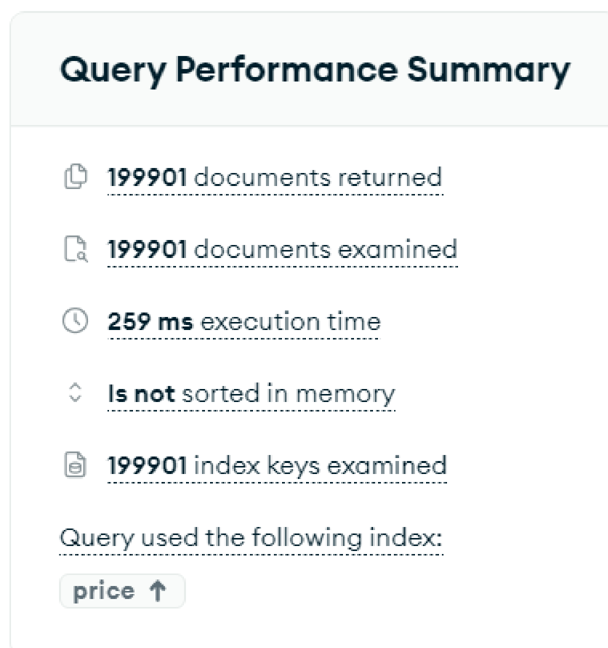

Рисунок 18 — Результати виконання запиту без з індексом

Процедура створення індексу

Для створення індексу, використовуючи інтерфейс MongoDb Compass треба перейти у вкладку Indexes. Далі після натискання на кнопку Create Index ми маємо можливість створити індекс з певними налаштуваннями. Серед усіх налаштувань при створенні індексу можна виділити кілька основних, а саме:

Create unique index — створення унікального індексу, що означає, що індексовані поля не будуть мати дублікатів даних.

Index name — ім'я індексу

Create TTL – спеціальні single — field індекси при використанні яких можна мати можливість автоматичного видалення документів з колекції у визначений вказаний час.

Після налаштування потрібно зазначати для якої колекції документів індекс буде використовуватися та тип індексу ()

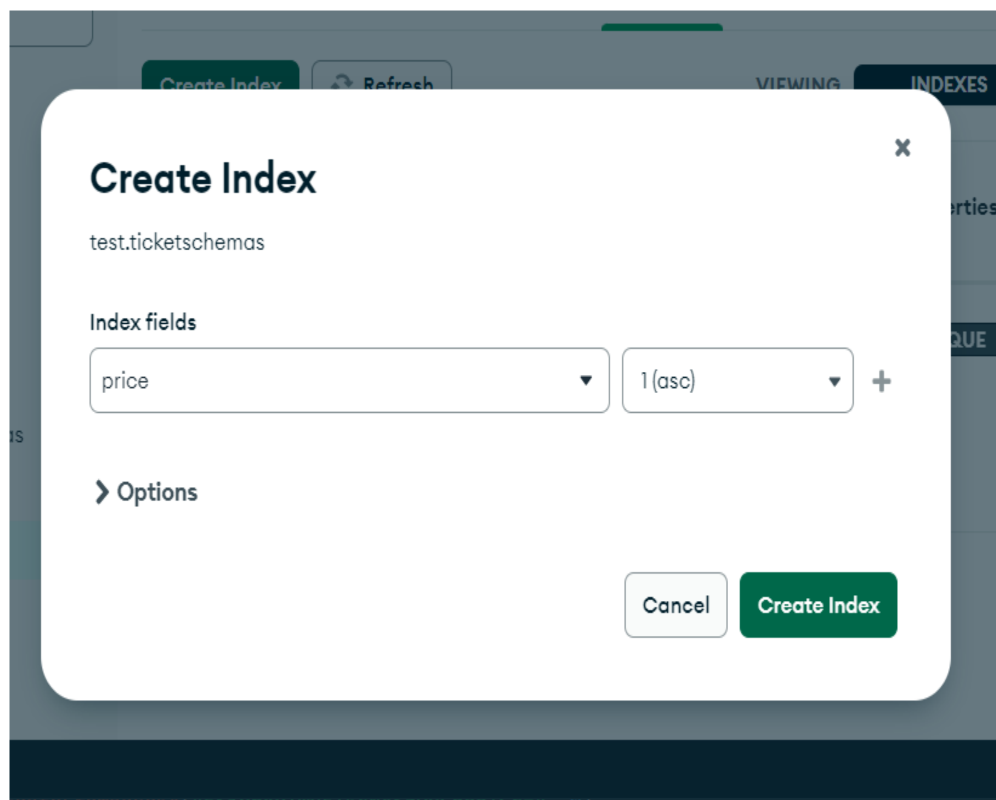

Рисунок 19 — Вікно створення індексу

Типів індуксу 3, а саме:

1. **1 (asc) та — 1 (desc):** Ці значення вказують на порядок сортування для полів індексу. Коли створюється складовий індекс, можна вказати порядок сортування кожного поля окремо.
    a. 1 означає сортування за зростанням (ascending).
    b. -1 позначає сортування за спаданням (descending).
2. **2dsphere:** Цей тип індексу використовується для індексації гео-графічних даних, представлених у вигляді географічних точок на Землі. Він підтримує різні геометричні операції, такі як пошук

об'єктів у межах певної області або визначення найближчих об'єктів до заданої точки.

3. **text:** Цей тип індексу використовується для індексації текстових даних у документах. Він дозволяє виконувати текстові пошукові запити, такі як пошук за словами чи фразами.

4.7.3 Дослідження продуктивності PostgreSql

Оскільки в PostgreSql немає подібного інтерфейсу як MongoDb Compass, то я використовував Postman для відправки тестового запиту. Як і в MongoDb у PostgreSql є команда EXPLAIN яка дозволяє дізнатися дані планувальника задач базі даних а також час витрачений на планування та час витрачений на виконання самого запиту. Як тестовий запит я використав таблицю квитків та запит на вибірку 199901 найдешевших квитків (див. Ліст. 29).

Лістинг 29 — Тестовий запит

```
module.exports.findChipest = async function(req, res) {
  try {
    const result = await client.query("EXPLAIN ANALYZE
SELECT * FROM tickets ORDER BY price LIMIT 199901");
    res.json(result.rows);
  } catch (err) {
    console.error(err);
    res.status(500).json({ error: "Error retrieving data
from tickets table" });
  }
```

Результатом виконання запиту є json в якому представлено покроково що саме було зроблено для виконання запиту, а також Planning Time і Execution Time (див. Лістинг 30).

Лістинг 30 — Результат тестування PostgreSql запиту без індексу

[{"QUERY PLAN":"Limit (cost=32573.05..55896.46 rows=199901 width=43) (actual time=340.561..496.632 rows=199901 loops=1)"},{"QUERY PLAN":" -> Gather Merge (cost=32573.05..81187.48 rows=416666 width=43) (actual time=340.560..486.565 rows=199901 loops=1)"},{"QUERY PLAN":"   Workers Planned: 2"},{"QUERY PLAN":" Workers Launched: 2"},{"QUERY PLAN":"   -> Sort (cost=31573.02..32093.86 rows=208333 width=43) (actual time=312.101..361.729 rows=67157 loops=3)"},{"QUERY PLAN":"   Sort Key: price"},{"QUERY PLAN":" Sort Method: external merge Disk: 9640kB"},{"QUERY PLAN":"   Worker 0: Sort Method: external merge Disk: 9640kB"},{"QUERY PLAN":"   Worker 1: Sort Method: external merge Disk: 9120kB"},{"QUERY PLAN":"   -> Parallel Seq Scan on tickets (cost=0.00..6756.33 rows=208333 width=43) (actual time=0.008..9.333 rows=166667 loops=3)"},{"QUERY PLAN":"Planning Time: 1.751 ms"},{"QUERY PLAN":"Execution Time: 510.143 ms"}]

Для дослідження продуктивності запитів з індексом був створений індекс для колонки price. Наступним кроком був повторний запуск тестового запиту (Ліст.30). Результати тестування наведені нижче (Ліст.31).

Лістинг 31 — Результати тестування з індексом

[{"QUERY PLAN":"Limit (cost=0.42..7065.72 rows=199901 width=43) (actual time=0.956..138.274 rows=199901 loops=1)"},{"QUERY PLAN":" -> Index Scan using tickets_price_index on tickets (cost=0.42..17672.42 rows=500000 width=43) (actual time=0.955..127.824 rows=199901 loops=1)"},{"QUERY PLAN":"Planning Time: 3.155 ms"},{"QUERY PLAN":"Execution Time: 143.574 ms"}]

Створення індексу

Оскільки для роботи з реляційною базою даних PostgreSql я використовую Data Grip, то маю можливість створення індексу за допомогою вбудованого інтерфейсу (Рис. 38). В цілому процес створення індексу з нічим не відрізняється від того ж процесу в MongoDb, але при цьому ми можемо також використовувати Sql для написання запиту на створення індексу. PostgreSQL підтримує кілька типів індексів: B-дерево, хеш, GiST, SP-GiST, GIN та BRIN. Для різних типів індексів застосовуються різні алгоритми, орієнтовані певні типи запитів. За замовчуванням команда CREATE INDEX створює індекси типу B-дерево, найефективніші у більшості випадків.

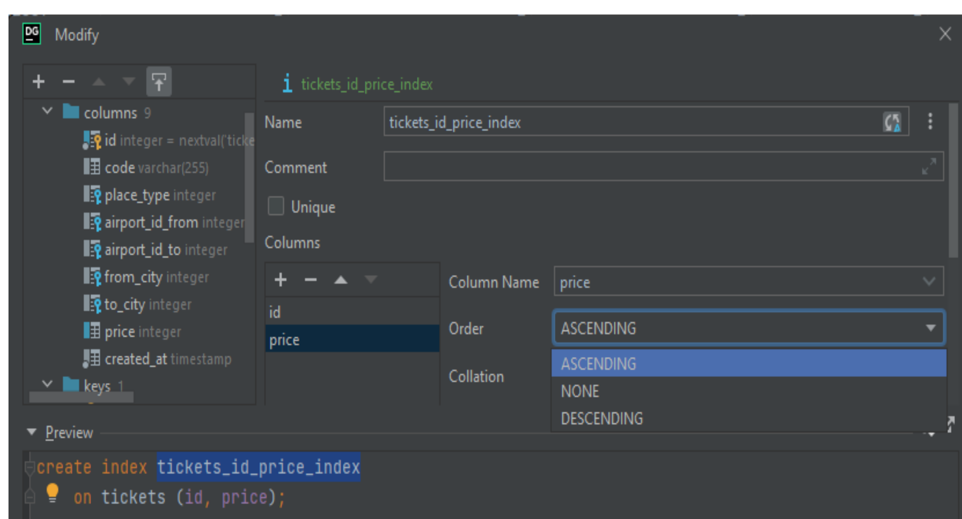

Рисунок 20 — Інтерфейс для створення індексу

Порівняльні результати

1. **Загальний час виконання запиту без індексу**:

- MongoDB: Час виконання запиту – 399 мілісекунд.
- PostgreSQL: Середній час виконання запиту – 510 мілісекунд.

2. **Загальний час виконання запиту з індексом:**

    - MongoDB: Час виконання запиту – 259 мілісекунд.
    - PostgreSQL: Середній час виконання запиту – 143 мілісекунд.

Проаналізувавши отримані вище результати, можна скласти характеристику кожної з розглянутих платформ і виробити рекомендації щодо їх застосування.

MongoDB:

Виходячи з глибокого аналізу, рекомендується використовувати MongoDB у сценаріях, де потрібна висока гнучкість та адаптивність до змінної структури даних. MongoDB виправдовує себе в проектах, де важливим є управління JSON-подібними документами, дозволяючи легко вносити зміни в схему даних без необхідності перебудовувати всю базу даних. Це особливо цінно в динамічних середовищах розробки, де потрібна швидка реакція зміни вимог.

При розробці веб — застосунків, особливо з використанням JavaScript (Node.js), MongoDB стає привабливим вибором, забезпечуючи нативну інтеграцію з цією мовою. Сумісність стеку технологій між серверною та клієнтською частинами програми створює більш узгоджену розробку та полегшує командну взаємодію.

У випадку, якщо проект передбачає високий трафік та необхідність горизонтального масштабування, MongoDB демонструє свою чудову продуктивність. Це особливо важливо для додатків, де потрібна обробка великих обсягів даних або де можливе динамічне масштабування ресурсів.

Одним із значних переваг MongoDB є MongoDB Atlas, хмарна платформа, що надає високий рівень управління та масштабованості. Інтеграція з хмарними платформами, такими як AWS, Azure та GCP, забезпечує зручність розгортання та управління інфраструктурою проекту.

Якщо проект включає геопросторові аспекти, MongoDB також виділяється завдяки PostGIS-подібним можливостям. Це дозволяє ефективно зберігати та аналізувати просторові дані, що особливо важливо для додатків, пов'язаних із картографією, геоаналітикою та локаційними послугами.

Таким чином, MongoDB є багатофункціональним рішенням, призначеним для сучасних проектів, де важливі гнучкість, продуктивність і зручність розробки.

PostgreSQL:

Рекомендується використовувати PostgreSQL у сценаріях, де потрібна строга схема даних та висока надійність. PostgreSQL виявляє себе у проектах, де цінується структурованість даних та можливість точного визначення відносин між ними. Це особливо цінно у додатках, де необхідно забезпечити цілісність даних та підтримувати складні взаємозв'язки.

При використанні різних мов програмування, PostgreSQL надає великий набір офіційних драйверів, що робить його гнучким у контексті багатомовних розробок. Це важливо для команд, які використовують різні технології на різних рівнях програми.

Якщо проект передбачає інтенсивні запити та агрегацію даних, PostgreSQL демонструє високу продуктивність. Він може бути кращим у сценаріях, де необхідні складні запити та операції агрегації даних, такі як аналітика та формування звітів.

PostgreSQL також привабливий для проектів, де потрібна підтримка геопросторових даних. З використанням розширення PostGIS, PostgreSQL надає потужні інструменти для зберігання та обробки просторової інформації, що може бути важливим фактором для проектів, пов'язаних із геолокацією чи картографією.

При виборі PostgreSQL також варто звернути увагу на можливості транзакцій та дотримання принципів ACID, що робить його правильним вибором для проектів, де важлива надійність та цілісність даних.

Таким чином, PostgreSQL є надійним і гнучким рішенням для проектів, де ключовими є структурованість даних, продуктивність і підтримка складних запитів.

Результати досліджень:

Виходячи з проведеного аналізу, обидві бази даних, MongoDB і PostgreSQL, мають унікальні характеристики, які роблять їх привабливими для різних сценаріїв використання. MongoDB демонструє себе як багатофункціональне рішення, що підтримує гнучкість та адаптивність до змін у структурі даних. Це особливо корисно в динамічних середовищах, де

потрібна швидка реакція на зміни вимог. З іншого боку, PostgreSQL рекомендується у сценаріях із суворішою схемою даних та високою надійністю, що робить його привабливим вибором для проектів, де цінується інтеграція даних та підтримка складних взаємозв'язків. Обидві бази даних виявляють видатну продуктивність у своїх контекстах використання та надають додаткові можливості, такі як геопросторові аспекти. Рішення про вибір між MongoDB і PostgreSQL значною мірою залежить від конкретних вимог проекту та переваг розробників.

ВИСНОВКИ

Реалізація серверних програм та API:

Обидві серверні програми були успішно реалізовані, включаючи API для виконання всіх операцій CRUD для реляційної бази даних PostgreSQL і документо-орієнтованої бази даних MongoDB.

Транзакції:

Реалізацію транзакцій для обох баз даних було успішно здійснено. У PostgreSQL використовувався механізм транзакцій із підтримкою принципів ACID, а MongoDB - транзакції з допомогою сеансів і обробки помилок.

Порівняльний аналіз продуктивності:

Проведено аналіз продуктивності обох баз даних у різних сценаріях, включаючи обсяг даних, швидкість виконання запитів та загальну продуктивність. Отримані результати підкреслюють особливості кожної бази даних у контексті швидкості та ефективності обробки даних.

Гнучкість схеми даних:

PostgreSQL як реляційна база даних забезпечує строгу структурованість даних, що вимагає певної схеми. MongoDB, у свою чергу, надає гнучку схему, дозволяючи зберігати JSON-подібні документи без жорстких обмежень.

Масштабованість та горизонтальне масштабування:

MongoDB виявляє чудову горизонтальну масштабованість, що робить його кращим для проектів з високим навантаженням та необхідністю масштабування. У той час як PostgreSQL має свої обмеження у горизонтальному масштабуванні.

Обробка помилок та керування транзакціями:

Досліджено механізми обробки помилок та управління транзакціями в обох базах даних, оцінено їх надійність та зручність впровадження.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Simon Riggs, Gianni Ciolli: PostgreSQL 14 Administration Cookbook. 2020, P. 50–68.
2. Shannon Bradshaw, Eoin Brazil, Kristina Chodorow: MongoDB: The Definitive Guide, 3rd Edition. 2021, P. 78-101.
3. Amit Phaltankar (Author), Juned Ahsan (Author), Michael Harrison (Author), Liviu Nedov (Author): MongoDB Fundamentals: A hands-on guide to using MongoDB and Atlas in the real world 2018 P. 316-352.
4. Date C.J. An Introduction to Database Systems. Eighth Edition. Addison-Wesley, 2003. P.85-96.
5. Silberschatz A., Korth H.F., Sudarshan S. Database System Concepts. Seventh Edition. McGraw-Hill Education, 2019 P. 350-380.
6. Atzeni P., Ceri S., Paraboschi S., Torlone R. Database Systems: Concepts, Languages and Architectures. McGraw-Hill Education, 1999.
7. Elmasri R., Navathe S.B. Fundamentals of Database Systems. Seventh Edition. Pearson, 2016.
8. Garcia-Molina H., Ullman J.D., Widom J. Database Systems: The Complete Book. Second Edition. Pearson, 2008.
9. Rob, P., Coronel C. Database Systems: Design, Implementation, and Management. Tenth Edition. Cengage Learning, 2016.
10. Pratt P., Adamski J. Concepts of Database Management. Ninth Edition. Cengage Learning, 2019.
11. Regina O. Obe, Leo S. Hsu: "PostgreSQL: Up and Running". 2017.
12. Hans-Jürgen Schönig: "Mastering PostgreSQL 12". 2019.
13. Chitij Chauhan: "PostgreSQL High Performance Cookbook". 2017.
14. Michael J. Hernandez: "Database Design for Mere Mortals". 2013.
15. Markus Winand: "SQL Performance Explained". 2018.
16. Pramod J. Sadalage, Martin Fowler: "NoSQL Distilled". 2012.
17. Eben Hewitt: "Cassandra: The Definitive Guide". 2010.
18. Kristina Chodorow, Michael Dirolf: "MongoDB: The Definitive Guide". 2010.

19.Kyle Banker: "MongoDB in Action". 2011.

20.Shannon Bradshaw, Eoin Brazil, Kristina Chodorow: "MongoDB: The Definitive Guide". 2010.

21.Joe Celko: "SQL Programming Style". 2005.

22.Jim Gray, Andreas Reuter: "Transaction Processing: Concepts and Techniques". 1992.

23.Christo Kutrovsky, Guy Harrison: "Oracle Wait Interface: A Practical Guide to Performance Diagnostics & Tuning". 2004.

24.Kimball Ralph, Ross Margy: "The Data Warehouse Toolkit: The Definitive Guide to Dimensional Modeling" (Third Edition). 2013.

25.John L. Hennessy, David A. Patterson: "Computer Architecture: A Quantitative Approach" (Fifth Edition). 2011.

26.Michael Stonebraker, Lawrence A. Rowe: "The Design of POSTGRES" . 1996.

27.James R. Groff, Paul N. Weinberg: "SQL: The Complete Reference" (Third Edition). 2009.

28.Chris Date: "An Introduction to Database Systems" (Ninth Edition) (2018).

29.Randal K. Michael: "Learning SQL" (Second Edition). 2009.

30.Richard T. Snodgrass: "Developing Object-Oriented Database Applications in SQL" , 1999.

31. Developer Survey 2023 URL :https://survey.stackoverflow.co/2023/

32.RabbitMQ URL: https://www.rabbitmq.com/ (дата звернення 20.11.2023).

33.Google trends. PostgreSql/MongoDb. URL: https://trends.google.com/trends/explore?date=today%205-y&q=%2Fm%2F05ynw,%2Fm%2F05z_r2n&hl=ru (дата звернення 20.11.2023).

34.MongoDB Scaling and Sharding URL: https://www.mongodb.com/docs/manual/sharding/(дата звернення 20.11.2023).

35.Cloud Computing: From Beginning to End Ray J. Rafaels та Richard Reese 2020

36. Marijn Haverbeke: "Eloquent JavaScript" 2018
37. Mario Casciaro "Node.js Design Patterns" 2014
38. Docker Docs URL: https://docs.docker.com/storage/storagedriver/device-mapper-driver/#snapshots (дата звернення 20.11.2023).
39. CAP-теорема URL: https://habr.com/ru/articles/328792/(дата звернення 20.11.2023).

Декларація

академічної доброчесності

здобувача вищої освіти ЗНУ

Я____Магометов Сергій Аланович______________________,
студент___2__курсу, форми здобуття освіти _____денна________,
Інженерного навчально-наукового інституту ім. Ю.М. Потебні ЗНУ
Спеціальності121 Інженерія програмного забезпечення,
адреса електронної пошти__ipz118bd-08@stu.zsea.edu.ua_____________,
підтверджую, що написана мною кваліфікаційна робота на тему «_Особливості використання документо-орієнтованої та реляційної СКБД» відповідає вимогам академічної доброчесності та не містить порушень, що визначені у ст. 42 Закону України «Про освіту», зі змістом яких ознайомлений/ознайомлена;

- заявляю, що надана мною для перевірки електронна версія роботи є ідентичною її друкованій версії;
- згоден/згодна на перевірку моєї роботи на відповідність критеріям академічної доброчесності у будь-який спосіб, у тому числі за допомогою Інтернет-системи, а також на архівування роботи в базі даних цієї системи.

Дата__30.11.2023__ Підпис___________ С.А. Магометов
(студент)

Дата___30.11.2023__ Підпис__________ Г. П. Коломоєць
(науковий керівник))