

Міністерство освіти і науки України
Державний заклад
«Луганський національний університет імені Тараса Шевченка»

Навчально-науковий інститут математики та інформаційних технологій

Кафедра інформаційних технологій та систем

Червинський Максим Владиславович

**ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ РІЗНИХ ТИПІВ БАЗ ДАНИХ У
РОЗРОБЦІ ДИНАМІЧНИХ REST API НА БАЗІ ФРЕЙМВОРКА
EXPRESS.JS**

кваліфікаційна робота

здобувача вищої освіти другого (магістерського) рівня

освітньої програми «Мультимедійні системи»

за спеціальністю 121 Інженерія програмного забезпечення

Особистий підпис _____ Максим ЧЕРВИНСЬКИЙ

Науковий керівник _____, Микола СЕМЕНОВ,
кандидат педагогічних наук, доцент
кафедри інформаційних технологій
та систем

Завідувач кафедри _____ Микола СЕМЕНОВ,
кандидат педагогічних наук, доцент
кафедри інформаційних технологій
та систем

АНОТАЦІЯ

Тема: Дослідження ефективності різних типів баз даних у розробці динамічних REST API на базі фреймворка Express.js.

Спеціальність: 121 «Інженерія програмного забезпечення».

Установа: ЛНУ імені Тараса Шевченка, 2025 р.

Магістерська робота містить: 91 с., 20 рис., 8 табл., 43 джерел.

Об'єкт дослідження – характеристики продуктивності динамічних REST API, розроблених на основі фреймворку Express.js при інтеграції з різними типами баз даних.

Предмет дослідження – вплив різних типів баз даних на ефективність розробки динамічних REST API на базі фреймворка Express.js.

Мета роботи – розробити методику оптимального вибору бази даних та оптимізації продуктивності для динамічних REST API на базі Express.js.

Результати роботи – розроблено динамічні REST API на основі Express.js, інтегровані з п'ятьма різними базами даних (MySQL, PostgreSQL, MongoDB, Cassandra та Redis); проведено комплексні тести продуктивності за різних сценаріїв навантаження; виміряні та проаналізовані ключові метрики продуктивності, такі як час відповіді, пропускну здатність, латентність та використання ресурсів; сформовано профілі продуктивності для кожного типу бази даних в контексті розробки API на основі Express.js.

Ключові слова: БАЗА ДАНИХ, ПРОДУКТИВНІСТЬ, ВЕБРОЗРОБКА, БЕКЕНД, REST API, EXPRESS.JS, MYSQL, POSTGRESQL, MONGODB, CASSANDRA, REDIS.

ANNOTATION

Topic: Performance Analysis of Various Databases in Express.js-based Dynamic REST API Development.

Specialty: 121 "Software engineering".

Institution: Luhansk Taras Shevchenko National University (LTSNU), 2025.

Master's thesis consists of: 91 p., 20 im., 8 tables, 43 sources.

Object of the study – performance characteristics of dynamic REST APIs developed using Express.js when integrated with various types of databases.

Subject of the study – impact of various types of databases on the performance and efficiency of Express.js-based dynamic REST API development.

Objective of the study – to evaluate and compare the performance of various databases within the context of Express.js-based dynamic REST APIs to provide insights and recommendations for optimal database selection and API performance optimization.

Results of the study – Express.js-based dynamic REST API integrated with five different databases (MySQL, PostgreSQL, MongoDB, Cassandra, and Redis) were developed. Comprehensive performance tests were conducted under various workload scenarios. Key performance metrics such as response time, throughput, latency, and resource utilization were measured and analyzed. The comparative analysis revealed distinct performance profiles for each database type, highlighting their strengths and limitations in the context of Express.js-based API development.

Keywords: REST APIs, EXPRESS.JS, DATABASE, MYSQL, POSTGRESQL, MONGODB, CASSANDRA, REDIS, PERFORMANCE, WEB DEVELOPMENT, BACKEND.

ЗМІСТ

ВСТУП	5
РОЗДІЛ 1. БАЗИ ДАНИХ У REST API	9
1.1. Огляд REST API та Express.js	9
1.2. Типи баз даних у веброзробці	16
1.2.1. Реляційні бази даних	16
1.2.2. Нереляційні бази даних	20
1.2.3. Бази даних у пам'яті (in-memory)	23
1.2.4. Бази даних NewSQL	28
1.3. Оптимізація продуктивності REST API	30
Висновки до розділу 1	35
РОЗДІЛ 2. ПРАКТИЧНА РЕАЛІЗАЦІЯ ДИНАМІЧНОГО REST API НА БАЗІ EXPRESS.JS	36
2.1. Визначення критеріїв та вибір баз даних	36
2.2. Проектування і розробка REST API	39
2.3. Методологія тестування	56
2.4. Підготовка середовища тестування	63
Висновки до розділу 2	68
РОЗДІЛ 3. ОЦІНКА ПРОДУКТИВНОСТІ ТА АНАЛІЗ	69
3.1. Проведення базового тестування	69
3.2. Тестування з інтеграцією Redis як рівня кешування	73
3.3. Тестування масштабованості й відмовостійкості RESTful-системи	76
3.4. Інтерпретація результатів	80
Висновки до розділу 3	83
ВИСНОВКИ	84
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ	86

ВСТУП

У сучасному середовищі веброзробки архітектурна парадигма REST (Representational State Transfer) стала основою для проєктування масштабованих вебсервісів. RESTful API забезпечує безперешкодну комунікацію між клієнтом і сервером, що лежить в основі функціонування багатьох сучасних вебдодатків [24]. Поширення RESTful API корелює зі зростаючим попитом на динамічні, інтерактивні та ефективні вебсервіси, здатні обробляти великі обсяги даних у реальному часі.

Express.js, мінімалістична й гнучка платформа для вебдодатків на Node.js, здобула значну популярність серед розробників завдяки здатності спрощувати створення надійних і високопродуктивних REST API [9]. Її архітектура, що базується на використанні проміжного програмного забезпечення (middleware), і велика екосистема плагінів забезпечують швидке створення прототипів та масштабовану розробку додатків, що робить її доцільним вибором як для стартапів, так і для інформаційних систем великих підприємств. Ефективність Express.js у створенні API тісно пов'язана з базами даних (БД), які забезпечують збереження, отримання та операції з даними.

Продуктивність RESTful API критично залежить від характеристик продуктивності бази даних, з якою вона взаємодіє. Продуктивність бази даних охоплює такі аспекти, як швидкість виконання запитів, обробка паралельності, масштабованість і надійність. Ці параметри безпосередньо впливають на відгук та пропускну здатність API, а отже, і на загальний користувацький досвід та операційну ефективність вебдодатка [16]. У ситуаціях, коли API на базі Express.js є основою для роботи з даними, вибір бази даних може стати визначальним фактором у досягненні оптимальних результатів продуктивності.

Попри визнане значення продуктивності баз даних для швидкодії API, існує брак комплексних досліджень, що систематично оцінюють вплив різних систем БД на продуктивність динамічних REST API на базі Express.js. Швидка еволюція технологій баз даних, які охоплюють як реляційні, так і нереляційні

парадигми, вимагає емпіричного дослідження для виявлення їхніх переваг і обмежень у контексті сучасних фреймворків для розробки API.

Це дослідження спрямоване на розв’язання нагальної потреби в аналізі впливу різних систем баз даних на продуктивність динамічних REST API, створених на основі Express.js. Шляхом проведення порівняльного аналізу продуктивності планується висвітлити диференційований вплив обраних баз даних на ключові показники продуктивності. Отримані результати допоможуть сформулювати найкращі практики вибору баз даних та стратегій їх оптимізації для розробників і організацій, що прагнуть підвищити ефективність і надійність своїх RESTful сервісів.

Об’єкт дослідження – характеристики продуктивності динамічних REST API, розроблених на основі фреймворку Express.js при інтеграції з різними типами баз даних.

Предмет дослідження – вплив різних типів баз даних на ефективність розробки динамічних REST API на базі фреймворка Express.js.

Мета роботи – розробити методику оптимального вибору бази даних та оптимізації продуктивності для динамічних REST API на базі Express.js.

Для досягнення поставленої мети необхідно вирішити такі завдання:

- 1) провести огляд поточного стану REST API, Express.js і різних технологій баз даних;
- 2) установити критерії відбору та обрати відповідні бази даних для аналізу продуктивності;
- 3) розробити та реалізувати узгоджені RESTful API, використовуючи Express.js для кожної вибраної бази даних;
- 4) виконати тести продуктивності за різних сценаріїв навантаження та рівнів паралельності;
- 5) зібрати та статистично проаналізувати дані продуктивності, отримані під час тестування;
- 6) провести порівняльний аналіз та інтерпретацію результатів продуктивності для різних баз даних;

- 7) розробити рекомендації щодо оптимального вибору бази даних і оптимізації роботи API на основі отриманих результатів.

Наукова новизна результатів дослідження полягає в цілеспрямованому аналізі впливу різних типів баз даних на продуктивність динамічних REST API, створених на базі Express.js. На відміну від попередніх досліджень, що розглядали продуктивність БД у загальному контексті або з використанням інших фреймворків, це дослідження пропонує емпіричний порівняльний аналіз баз даних різних типів (реляційних, нереляційних, in-memory, NewSQL) в ідентичних умовах.

Методи дослідження. Для досягнення поставлених завдань у дослідженні застосовані такі методи: експерименту - для проведення тестів продуктивності REST API, інтегрованих з різними базами даних за різних сценаріїв робочого навантаження; вимірювання – для збору кількісних даних про ключові показники продуктивності; порівняння – для аналізу відмінностей у продуктивності вибраних баз даних з точки зору ефективності та масштабованості; моделювання – для розробки узгоджених кінцевих точок API та моделей даних у БД для забезпечення однорідності тестування; аналізу – для інтерпретації зібраних даних про ефективність та визначення переваг і недоліків; синтезу – для інтеграції висновків у дійсну методику вибору бази даних та оптимізації API.

У першому розділі розглянуто теоретичне підґрунтя та наявні дослідження REST API, фреймворку Express.js і різних технологій баз даних. Розглянуті принципи архітектури REST, особливості Express.js і характеристики реляційних, нереляційних та in-memory баз даних. Проаналізовані фактори, що впливають на продуктивність API, і визначено прогалини в наявних знаннях.

У другому розділі викладено методологічний підхід, включаючи критерії відбору для баз даних і проєктування REST API з використанням Express.js. Описано створення узгоджених кінцевих точок API та моделей даних, конфігурацію контрольованого середовища тестування продуктивності

та визначення ключових показників, таких як час відгуку та пропускна здатність. Описано процеси впровадження та тестування, охоплюючи сценарії робочого навантаження та симуляції паралельного виконання.

У третьому розділі представлені та проаналізовані дані про продуктивність, зібрані під час тестування, включаючи візуалізацію результатів за допомогою графіків і таблиць та виконання порівняльного аналізу баз даних за різних навантажень. Проведено інтерпретацію результатів, визначені переваги й недоліки кожної бази даних та сформульована методика вибору бази даних і оптимізації продуктивності API на підставі виявлених технологічних особливостей.

РОЗДІЛ 1

БАЗИ ДАНИХ У REST API

1.1. Огляд REST API та Express.js

REST (Representational State Transfer) є архітектурним стилем, який визначає набір правил і властивостей, спрямованих на створення масштабованих, підтримуваних і ефективних вебсервісів. Запропонований Роєм Філдінгом у його знаковій дисертації [24], REST став фундаментальною парадигмою у розробці сучасних API.

Основоположний принцип REST – безстанність (statelessness), яка вимагає, щоб кожна взаємодія між клієнтом і сервером була незалежною та автономною. У безстанній архітектурі сервер не зберігає жодної інформації про сесію клієнта між запитами, тому кожен запит має містити всю необхідну інформацію для його обробки сервером. Це підвищує масштабованість, дозволяючи серверу обробляти численні одночасні запити без витрат на підтримку станів сесії. Безстанність також спрощує проєктування серверів і підвищує надійність, оскільки поведінка сервера залишається послідовною незалежно від попередніх взаємодій.

REST забезпечує чіткий розподіл між клієнтом і сервером, залишаючи питання інтерфейсу користувача на стороні клієнта, а збереження даних і бізнес-логіку - на стороні сервера. Такий поділ дозволяє обом компонентам незалежно розвиватися і масштабуватися, а також забезпечує можливість реалізації клієнтів на різних платформах, які можуть взаємодіяти з однією і тією ж серверною API без необхідності змін у архітектурі сервера.

Уніфікований інтерфейс - ключове правило REST, яке спрощує та роз'єднує архітектуру, дозволяючи кожній частині розвиватися незалежно. Філдінг визначає чотири стандарти, які формують уніфікований інтерфейс [24]:

1. **Ідентифікація ресурсів.** Ресурси однозначно ідентифікуються за допомогою URI (уніфікованих ідентифікаторів ресурсів). Їхні

представлення можна отримати або модифікувати за допомогою стандартних HTTP-методів, що забезпечує узгодженість API.

2. **Маніпуляція ресурсами через представлення.** Клієнти взаємодіють із ресурсами шляхом обміну представленнями (наприклад, JSON, XML). Це дозволяє змінювати стани ресурсів без розуміння внутрішньої реалізації сервера.
3. **Самоописові повідомлення.** Кожне повідомлення містить достатню інформацію для його обробки, включаючи метадані, наприклад, тип вмісту та кодування, що дозволяє клієнтам і серверам правильно інтерпретувати повідомлення.
4. **Гіпермедіа як рушій стану додатку (HATEOAS).** Клієнти динамічно навігують API за допомогою гіперпосилань, наданих у представленнях ресурсів, що сприяє інтуїтивній і адаптивній моделі взаємодії.

Уніфікований інтерфейс знижує зв'язність між клієнтом і сервером, полегшуючи інтеграцію різних систем.

REST підтримує багаторівневу архітектуру, де система складається з ієрархічних рівнів із чітко визначеними функціональними можливостями. Міжрівневі шари, такі як проксі, шлюзи та балансувальники навантаження, можуть бути додані між клієнтом і сервером без зміни загальної системи поведінки. Така багаторівнева архітектура покращує масштабованість, безпеку та керованість, дозволяючи кожному рівню обробляти окремі аспекти. Наприклад, механізми кешування можуть бути реалізовані на проміжних рівнях для покращення продуктивності, тоді як протоколи безпеки можуть бути впроваджені в певних точках архітектури.

Крім того, у REST допускається можливість передачі сервером коду (наприклад, JavaScript), який клієнт може виконати. Цей необов'язкове правило розширює функціональність клієнтських додатків, забезпечуючи динамічність, наприклад, через скриптування на льоту. Однак його

опціональний характер означає, що не всі RESTful системи використовують код на вимогу, часто через міркування, пов'язані з безпекою та складністю.

Ефективне кешування є важливим елементом архітектури REST, спрямованим на підвищення продуктивності та зниження навантаження на сервер. Відповіді сервера повинні явно вказувати їхню кешованість через відповідні заголовки, що дозволяє клієнтам і проміжним кешам зберігати та повторно використовувати відповіді на ідентичні запити. Завдяки кешуванню RESTful API можуть досягати суттєвого зменшення затримок і оптимізації пропускну здатності, покращуючи загальний користувацький досвід.

Дотримання принципів REST сприяє створенню роз'єднаної, масштабованої та підтримуваної архітектури систем, що є критично важливим для сучасних вебдодатків із динамічними та високими вимогами до продуктивності [7, 25]. Уніфікований інтерфейс і безстанність спрощують інтеграцію та забезпечують сумісність, дозволяючи різним клієнтам безперешкодно взаємодіяти з API.

Водночас суворе дотримання принципів REST може створювати перешкоди, наприклад, складнощі в управлінні переходами станів через суворий поділ клієнта й сервера. Однак основні принципи REST надають надійну основу, яка, за умови коректного застосування, значно покращує якість і продуктивність вебсервісів.

У сфері розробки на стороні сервера вибір відповідного фреймворку має ключове значення для створення ефективних вебдодатків. **Express.js** – фреймворк екосистеми Node.js, який набув поширення завдяки лаконічному дизайну і набору функцій, що сприяють швидкій розробці веб- і мобільних додатків. Він має відкритий вихідний код для Node.js, розроблений з метою надання специфічного рівню основних функцій вебдодатків, не приховуючи внутрішніх можливостей Node.js [23]. Мінімалістична філософія Express.js полягає в тому, що розробники зберігають детальний контроль над архітектурою програми, заохочуючи гнучкість і адаптивність у вирішенні

різноманітних вимог розробки. Express.js служить каналом зв'язку між сервером і клієнтом, керуючи запитами та відповідями HTTP.

Основою архітектури Express.js є простота та модульність. Використовуючи керовану подіями, неблокуючу модель вводу-виводу Node.js, він дозволяє створювати високопродуктивні масштабовані додатки, здатні обробляти велику кількість одночасних з'єднань [39, 40]. Основні функції Express.js включають:

1. Маршрутизація. Express.js пропонує механізм маршрутизації, який відображає команди HTTP та URL-шляхи до певних функцій обробки, полегшуючи легке створення кінцевої RESTful-точки.
2. Підтримка проміжного програмного забезпечення. В основі Express.js лежить його архітектура middleware, яка дозволяє інтегрувати багаторазові функції, що можуть обробляти запити та відповіді на різних етапах циклу «запит-відповідь» [9].
3. Інтеграція механізму шаблонів. Express.js легко інтегрується з різними механізмами шаблонів (наприклад, Pug, EJS), забезпечуючи динамічний рендеринг вмісту та розділення проблем між логікою програми та рівнями презентації.
4. Розширюваність. Гнучкість фреймворку дозволяє широке налаштування за допомогою низки плагінів і розширень, що задовольняє широкий спектр потреб конкретної програми.

Функції middleware в Express.js впроваджені пошарово, кожен з шарів виконує певні завдання, які сприяють загальній функціональності додатку. Ці функції мають доступ до об'єктів запиту та відповіді, а також до наступної функції middleware в циклі «запит-відповідь». Проміжне програмне забезпечення може виконувати безліч операцій, такі як:

- реєстрація запитів: збір і реєстрація деталей запиту для моніторингу та налагодження;

- автентифікація та авторизація: впровадження протоколів безпеки для перевірки аутентичності користувачів і прав доступу;
- синтаксичний аналіз даних: обробка та перетворення даних вхідного запиту (наприклад, JSON, закодовані дані в URL) у формат, придатний для читання.

Модульність middleware сприяє чіткому розподілу завдань, покращуючи обслуговування та масштабованість коду. Крім того, проміжне програмне забезпечення може слугувати для створення складних конвеєрів обробки, де кожен крок залежить від попереднього для досягнення поставлених цілей.

Окремо слід розглянути складну систему маршрутизації Express.js, яка дозволяє розробникам визначати кінцеві точки та пов'язувати їх із певними методами HTTP та функціями обробки. Ця система підтримує:

- параметри динамічного маршруту: створення гнучких і придатних для повторного використання маршрутів, які можуть обробляти змінні вхідні дані (наприклад, /users/:userId);
- вкладені маршрути: сприяння організації маршрутів в ієрархічній формі, що забезпечує модульність;
- ланцюги маршрутів: дозволяє об'єднувати декілька обробників маршрутів для одного маршруту, що забезпечує послідовну обробку запитів.

Механізм маршрутизації є невід'ємною частиною реалізації REST API, оскільки забезпечує структурований підхід до обробки ресурсно-орієнтованих операцій. Абстрагуючи складність обробки запитів, Express.js дозволяє розробникам зосередитися на основній функціональності додатку.

Як було зазначено, Express.js внутрішньо пов'язаний з Node.js, використовуючи його асинхронну архітектуру, керовану подіями, для забезпечення продуктивності без блокування. Цей зв'язок дозволяє Express.js ефективно керувати пов'язаними операціями вводу-виводу, такими як

взаємодія з базою даних і мережі запити, не перешкоджаючи швидкості реагування програми.

Щодо екосистеми Express.js, то вона пропонує широкий вибір модулів middleware, плагінів та сторонніх бібліотек для інтеграції різноманітних функціональних можливостей. Наприклад, за допомогою Passport.js розробники можуть реалізувати стратегії автентифікації для забезпечення безпечного доступу до додатків. Бібліотеки, такі як Joi та express-validator, сприяють перевірці введених даних, забезпечуючи їхню цілісність. Крім того, інструменти на кшталт Swagger та аріDoc спрощують створення інтерактивної документації API, покращуючи досвід розробників і полегшуючи колаборацію.

Доступність цих розширень прискорює терміни розробки, зменшує надмірність і сприяє дотриманню рекомендованих практик. Крім того, внески спільнотою готових рішень з відкритим кодом для Express.js, сприяє його безперервній еволюції і підтримці відповідності фреймворку сучасним парадигмам розвитку та технологічним інноваціям.

Продуктивність є ключовим фактором при виборі вебфреймворку, особливо для додатків, що вимагають високої пропускну здатності та мінімальної затримки. Express.js демонструє високу ефективність завдяки своїй легкій архітектурі та оптимізованим механізмам обробки запитів [16, 18], однак продуктивність цього фреймворку залежить від кількох критичних аспектів. По-перше, це оптимізація використання проміжного програмного забезпечення, яка може значно зменшити час відгуку та ресурсоемність, по-друге - застосування асинхронних шаблонів у функціях обробки дозволяє задля уникнення точок уповільнення і підтримки високого рівня паралелізму, по-третє - ефективне управління серверними ресурсами, такими як пам'ять і процесор, яке забезпечує стабільну роботу навіть під значним навантаженням.

Емпіричні дослідження показали, що Express.js досягає конкурентоспроможних показників продуктивності порівняно з іншими

усталеними фреймворками, особливо якщо він оптимізований для конкретних випадків використання [2, 9]. Тим не менш, необхідно проводити оцінювання продуктивності в залежності від контексту, щоб переконатися в придатності інфраструктури для конкретних вимог додатку.

Express.js часто співставляють з іншими вебфреймворками, такими як Koa.js, Hapi.js і Sails.js, кожен з яких пропонує різні функції та філософію проєктування. Порівняльний аналіз показує, що:

- Koa.js - розроблений тією ж командою, що стоїть за Express.js, але використовує сучасніший підхід, використовуючи нативний синтаксис `async/await`, що спрощує керування асинхронним кодом [33];
- Hapi.js - робить акцент на розробці, орієнтованій на конфігурацію, та пропонує багатий набір функцій за замовчуванням, що робить його придатним для великомасштабних додатків зі складними вимогами [19];
- Sails.js - призначений для створення керованих даними API і програм реального часу, він забезпечує архітектуру MVC і легко інтегрується з різними базами даних [32].

Хоча Express.js залишається найпоширенішим фреймворком через його простоту та розгалужену екосистему проміжного програмного забезпечення, вибір серед цих фреймворків має ґрунтуватися на конкретних потребах проєкту, міркуваннях продуктивності та кваліфікації розробника.

Express.js є особливо доцільним для розробки API-інтерфейсів RESTful завдяки невід'ємній підтримці маршрутизації, інтеграції проміжного програмного забезпечення та масштабованості [16]. Здатність фреймворку обробляти методи HTTP та коди стану повністю узгоджується з принципами архітектури REST, полегшуючи створення ресурсно-орієнтованих служб. Крім того, сумісність Express.js з різними системами баз даних, як реляційними, так і нереляційними, підкреслює його універсальність в

управлінні збереженням даних і операціями пошуку, критичними для функціональності API.

У контексті динамічної розробки REST API Express.js служить посередником, який організовує взаємодію між клієнтом, сервером і рівнями бази даних. Його архітектура проміжного ПЗ дозволяє реалізувати наскрізні завдання, такі як автентифікація, перевірка введених даних і обробка помилок. Крім того, підтримка фреймворком асинхронних операцій гарантує, що API можуть підтримувати високу швидкість реагування та обробляти значний трафік без зниження продуктивності.

1.2. Типи баз даних у веброзробці

В процесі веброзробки вибір відповідної системи БД має пріоритетне значення, оскільки від цього залежить зберігання, пошук та операції з даними, що безпосередньо впливає на продуктивність і надійність вебдодатків. Розмежовуючи відмінності між реляційними (SQL), нереляційними (NoSQL), базами даних у пам'яті та NewSQL, подальший теоретичний огляд з'ясовує, як кожен тип баз даних узгоджується з конкретними вимогами програми та показниками продуктивності.

1.2.1. Реляційні бази даних

Реляційні бази даних ґрунтуються на моделі, представленій Е. Ф. Коддом у 1970 році, яка організовує дані в таблиці (зв'язки), що складаються з рядків і стовпців. Кожна таблиця представляє певну сутність, а зв'язки між цими сутностями встановлюються через зовнішні ключі та об'єднання. Реляційні бази даних використовують мову структурованих запитів (SQL) як основний інтерфейс для обробки та визначення даних. SQL забезпечує декларативний синтаксис, який дозволяє виконувати складні запити, транзакції та адміністративні завдання. Стандартизований характер SQL забезпечує взаємодію та узгодженість між різними системами керування реляційними базами даних (RDBMS).

MySQL - одна з найбільш поширених реляційних баз даних, розроблена компанією MySQL AB, а наразі належить корпорації Oracle. MySQL користується особливим попитом в веброзробці завдяки бездоганній інтеграції з різними вебтехнологіями та підтримці великомасштабних програм [5, 10]. Ключові переваги MySQL полягаються в таких особливостях, як:

- використання ефективних механізмів зберігання, таких як InnoDB і MyISAM, які оптимізують операції зберігання та пошуку даних; InnoDB, зокрема, підтримує транзакції та обмеження зовнішнього ключа, підвищуючи цілісність даних;
- надійні механізми безпеки, включаючи автентифікацію користувачів, контроль доступу та шифрування даних;
- підтримує master-slave реплікацію, яка забезпечує резервування даних і балансування навантаження, що необхідно для високої доступності та відмовостійкості;
- наявність великої спільноти розробників, яка робить внесок у розширення екосистеми інструментів, бібліотек і плагінів, сприяючи оптимальній розробці та вирішенню проблем.

Хоча MySQL набула поширення завдяки простоті та продуктивності в багатьох вебдодатках, розширеною альтернативою їй є PostgreSQL, що є особливо доцільною для сценаріїв, які вимагають розширеної функціональності та відповідності стандартам.

PostgreSQL - вдосконалена реляційна СУБД з відкритим вихідним кодом, яку часто називають «найдосконалішою у світі базою даних з відкритим вихідним кодом» [35]. Їй віддають перевагу в сценаріях, що вимагають складних можливостей обробки даних і дотримання стандартів SQL. Основними особливостями PostgreSQL є [13]:

- можливість для розробників визначати власні типи даних, оператори та функції;

- підтримка широкого спектру методів індексування, включаючи індекси B-tree, hash і GIN, які оптимізують продуктивність запитів;
- повна підтримка віконних функцій, загальних табличних виразів (СТЕ) і повнотекстового пошуку;
- відповідність властивостям ACID, що забезпечує надійну обробку транзакцій і цілісність даних;
- механізм керування паралельним доступом за допомогою багатоверсійності (MVCC), що дозволяє одночасно зчитувати та оновлювати дані без взаємних блокувань і сприяє продуктивності системи.

Розуміння надійності реляційних баз даних вимагає ретельного вивчення властивостей ACID, які лежать в основі надійності транзакцій і цілісності даних. Властивості **ACID (Atomicity, Consistency, Isolation, Durability)** є фундаментальними для надійності та узгодженості транзакцій у реляційних базах даних. Ці властивості гарантують, що операції з базою даних обробляються надійно, навіть за наявності системних збоїв, одночасного доступу або інших аномалій.

1. Атомарність (Atomicity) - транзакція розглядається як єдиний нероздільний блок операцій. Вона або виконується повністю, або не виконується зовсім. Якщо під час виконання транзакції сталася помилка, відбувається відкат всіх змін, внесених до даних, і БД повертається до попереднього стану.
2. Узгодженість (Consistency) - транзакція переводить базу даних з одного узгодженого стану в інший, зберігаючи всі попередньо визначені правила, такі як обмеження цілісності та тригери. Ця властивість зберігає коректність даних під час обробки транзакцій.
3. Ізоляція (Isolation) - передбачає, що одночасні транзакції виконуються незалежно, не заважаючи одна одній. Система бази даних гарантує, що проміжні стани транзакції є невидимими для інших транзакцій, тим

самим запобігаючи таким аномаліям, як «брудне» читання або втрачені оновлення.

4. Довговічність (Durability) - гарантує, що після здійснення транзакції її наслідки є постійними та витримують наступні системні збої. Зазвичай це досягається за допомогою таких механізмів, як ведення журналу з попереднім записом і постійне зберігання.

Суворе дотримання властивостей ACID є основою придатності бази даних для додатків, де цілісність і надійність даних є найбільш критичним аспектом, наприклад фінансових систем, платформ електронної комерції та корпоративних застосунків.

Окрім транзакційної цілісності, ключову роль у ефективності бази даних відіграють організація та структура даних, оскільки реляційні БД покладаються на **структуровані схеми** для моделювання та підтримки взаємопов'язаних даних. Схема описує таблиці, стовпці, типи даних, обмеження та зв'язки, надаючи план того, як дані зберігаються та взаємопов'язані.

Одним із ключових аспектів структурованих схем є нормалізація даних. Вона полягає в розділенні таблиць для усунення надмірностей і аномалій залежності. Організовуючи дані в пов'язані таблиці, нормалізація підвищує цілісність даних і оптимізує ефективність їх збереження. Структуровані схеми також забезпечують суворе дотримання типів даних і обмежень, що гарантує введення лише валідних даних. Це запобігає створенню некоректних або пошкоджених записів у базі. Визначення зв'язків між таблицями за допомогою первинних і зовнішніх ключів дозволяє виконувати складні запити, які можуть отримувати та обробляти взаємозв'язані дані з кількох таблиць.

Попри статичність реляційних схем, вони також підтримують еволюцію схем через міграційні скрипти та контроль версій. Це дозволяє поступово вдосконалювати структуру бази даних і масштабувати її відповідно до нових вимог додатків.

Хоча MySQL і PostgreSQL є провідними реляційними базами даних, вони мають відмінності в характеристиках. Особливістю MySQL є швидкість у завданнях, орієнтованих на читання даних, що робить її використання доцільним у вебдодатках з великими обсягами запитів на отримання інформації. PostgreSQL, зі свого боку, ефективно функціонує в середовищах, орієнтованих на запис даних, і при обробці складних запитів завдяки вдосконаленим методам індексації та оптимізації [10].

Загалом PostgreSQL пропонує ширший набір функцій, включаючи підтримку розширених типів даних (наприклад, JSONB, XML), повнотекстовий пошук і процедурні мови. MySQL натомість пропонує більш спрощений набір функцій, орієнтований на простоту використання. Так само PostgreSQL перевершує MySQL в аспекті розширюваності, вона дозволяє ширше налаштування та інтеграцію функцій, визначених користувачем, і розширень. Це робить PostgreSQL кращим вибором для додатків, які потребують спеціалізованої обробки даних.

Розглянуті бази даних мають активну підтримку з боку спільноти, але PostgreSQL сприймається як більш орієнтована на розробників, з фокусом на інноваціях та відповідності стандартам. MySQL виграє від широкої комерційної підтримки компанією Oracle, що забезпечує потреби розгортання на рівні підприємств. Тож вибір між MySQL і PostgreSQL має базуватися на конкретних вимогах застосування, включаючи продуктивність, складність взаємодії з даними та необхідність використання розширених функцій.

1.2.2. Нереляційні бази даних

Оскільки вимоги до вебдодатків розвиваються, особливо щодо обробки великих обсягів неструктурованих та напівструктурованих даних, нереляційні бази даних (NoSQL) постали як альтернатива реляційним БД [30]. Бази даних NoSQL представляють собою відмінну категорію СУБД, призначених для роботи з широким спектром моделей даних, включаючи формати «ключ-значення», документи, стовпці та графіки.

На відміну від реляційних баз даних, системи NoSQL не покладаються на фіксовані схеми чи використання SQL для операції з даними і пропонують більшу гнучкість і масштабованість для адаптації до динамічної природи сучасних додатків [22]. Потреба в цій гнучкості зокрема виникає в сценаріях, що вимагають швидких циклів розробки, обробки даних у реальному часі та горизонтального масштабування між розподіленими системами [14]. Відсутність жорстких схем дозволяє нереляційним базам даних ефективно керувати неструктурованими даними та адаптуватися до нових вимог додатків без суттєвої міграції схем.

MongoDB - це документно-орієнтована база даних NoSQL, призначена для ефективного зберігання та отримання великих обсягів напівструктурованих і неструктурованих даних [18]. На відміну від реляційних баз даних, вона зберігає дані в колекціях як JSON-подібні документи. Ключовими особливостями MongoDB є:

- безсхемний дизайн: MongoDB не застосовує попередньо визначену схему, що дозволяє документам в одній колекції мати різні поля та типи даних;
- JSON-подібні документи: дані зберігаються у форматі BSON (двійковий JSON), який підтримує багатофункціональні типи даних, зокрема масиви, вкладені об'єкти та двійкові дані;
- індексування та запити: підтримуються різні методи індексування, включаючи складені індекси, текстові індекси та геопросторові індекси;
- горизонтальна масштабованість: підтримується горизонтальне масштабування за допомогою сегментування, розподіляючи дані між кількома серверами для керування великими наборами даних і високопродуктивними робочими навантаженнями;

- реплікація та висока доступність: забезпечується надлишковість даних і стійкість до відмов за допомогою наборів реплік, які зберігають копії даних на кількох вузлах.

Відсутність схем і масштабованість MongoDB роблять її доцільною для таких додатків, як системи управління контентом, аналітика в реальному часі та платформи IoT, які характеризуються різноманітністю й значними обсягами даних [39].

Cassandra - це розподілена NoSQL база даних, розроблена для створення високомасштабованих та високонадійних сховищ великих обсягів даних. Вона відноситься до категорії сховищ підвищено стійких до збоїв: поміщені в БД дані автоматично реплікуються на кілька вузлів розподіленої мережі або рівномірно розподіляються до декількох дата-центрів.

Основні характеристики Cassandra:

1. **Гнучкість схеми.** Хоча Cassandra використовує більш структурований підхід, ніж MongoDB, вона пропонує гнучкість у визначенні стовпців у таблицях. Нові стовпці можна додавати динамічно, не впливаючи на наявні дані, що робить їх придатними для моделей даних, що розвиваються.
2. **Децентралізована архітектура.** Cassandra використовує однорангову архітектуру, де всі вузли є рівноправними та взаємодіють без головного вузла. Така конструкція забезпечує високу доступність і відмовостійкість, оскільки немає ані єдиної точки відмови.
3. **Горизонтальна масштабованість.** Подібно до MongoDB, Cassandra ефективно справляється з горизонтальним масштабуванням, розподіляючи дані між кількома вузлами в кластері. Вона підтримує автоматичне розбиття та реплікацію, що забезпечує лінійну масштабованість у міру додавання нових вузлів
4. **Висока пропускна здатність.** Розроблена для навантажень з високою інтенсивністю запису, Cassandra може обробляти великі обсяги

одночасних записів з мінімальною затримкою, що актуально для таких застосувань, як аналітика в реальному часі, рекомендаційні системи та обробка часових рядів.

Архітектура Cassandra надає переваги в розробці додатків, які вимагають високої доступності, відмовостійкості та масштабованості в кількох геолокаціях [26].

Однією з визначальних характеристик баз даних NoSQL, таких як MongoDB і Cassandra, є їх дизайн без схем, який контрастує з жорсткими схемами реляційних баз даних. Розробники можуть зберігати дані з різними структурами в одній колекції або таблиці, що сприяє швидкій ітерації та врахуванню змін у вимогах програми. Базы даних без схем спрощують інтеграцію різнорідних джерел даних і зменшують накладні витрати на розробку й спрощують процеси розгортання, оскільки немає необхідності визначати та переносити схеми [3, 31]. Однак дизайн без схем також створює виклики, включаючи потенційну неузгодженість даних і необхідність перевірки на рівні програми та управління даними.

Критичним фактором при виборі баз даних NoSQL є здатність до масштабованості, особливо для додатків, які оброблятимуть великі обсяги даних і зазнаватимуть високих навантажень трафіку [22, 36, 43]. Як було зазначено, і MongoDB, і Cassandra підтримують горизонтальну масштабованість, хоча і за допомогою різних механізмів.

Так, MongoDB використовує шардинг, розподіляючи дані по шардах, кожен з яких розміщується на окремому сервері. Маршрутизатор (mongos) направляє запити до відповідного шарда, забезпечуючи безшовне розподілення та масштабування. Cassandra ж використовує механізм послідовного хешування для розподілу даних по вузлах. Дані розподіляються на основі первинного ключа, що забезпечує рівномірний розподіл і мінімізує перевантаження.

1.2.3. Базы даних у пам'яті (in-memory)

Враховуючи особливості баз даних NoSQL у роботі з великими обсягами даних і динамічними структурами, не менш важливо розглянути бази даних у пам'яті (in-memory). Базы даних у пам'яті зберігають дані переважно в системній основній пам'яті (RAM), а не на диску, що забезпечує швидкий доступ до даних і обробку. На відміну від традиційних баз даних, бази даних у пам'яті оптимізовані для швидких операцій читання та запису з механізмами збереження, які використовуються для захисту даних від потенційної втрати через системні збої.

Redis (Remote Dictionary Server) - одна з найпоширеніших баз даних у пам'яті. Вона працює як сховище «ключ-значення», де дані можуть зберігатися в різних форматах, включаючи рядки, списки, набори, хеші тощо. Основні характеристики Redis:

1. **Високошвидкісне отримання даних.** Redis зберігає дані в оперативній пам'яті, забезпечуючи час отримання в діапазоні мікросекунд. Ця продуктивність має вирішальне значення для таких випадків використання, як рейтингові таблиці в реальному часі, керування сесіями та черги повідомлень.
2. **Структури даних.** Redis підтримує широкий спектр структур даних, крім простих пар «ключ-значення», такі як списки, набори, відсортовані набори, хеші, растрові зображення та гіперлогіти.
3. **Параметри збереження.** Хоча Redis працює в пам'яті, він пропонує такі механізми збереження, як RDB (знімок) і AOF (файл лише для додавання), щоб забезпечити довговічність даних.
4. **Обмін повідомленнями публікації/підписки (Pub/Sub).** Дозволяє клієнтам підписуватися на канали та отримувати оновлення в реальному часі, що робить його придатним для створення систем сповіщень і служб зв'язку в реальному часі.
5. **Кластеризація та реплікація.** Redis підтримує реплікацію master-slave і Redis Cluster, що забезпечує горизонтальну масштабованість і

високу доступність. Дані можна розподіляти між кількома вузлами, щоб справлятися зі збільшеним навантаженням.

6. **Механізм кешування.** Redis зазвичай використовується як рівень кешування, щоб зменшити навантаження на основні бази даних і покращити час відповіді. Його політика вилучення LRU (Least Recently Used) і параметри TTL (Time-to-Live) оптимізують керування пам'яттю.

Використання Redis передбачає виконання таких завдань, як керування сесіями, кешування, аналітика в реальному часі, черги повідомлень, лічильники і рейтингові таблиці.

Redis використовує складний механізм кешування, який використовує його архітектуру зберігання даних у пам'яті для забезпечення швидкого доступу до даних і підвищення продуктивності додатку. Однією з основних функцій, які підтримують кешування в Redis, є можливість встановити час життя (TTL) для кешованих даних. Коли TTL присвоюється ключу, Redis автоматично видаляє ключ після закінчення зазначеного періоду. Це запобігає нескінченному зберіганню застарілих даних і дозволяє кешу ефективніше керувати пам'яттю. Наприклад, ця команда встановлює термін дії ключа «user:123» у 60 секунд.

SET user:123 "Grygorii Skovoroda" EX 60

Щоб керувати обмеженою пам'яттю, Redis реалізує кілька політик вилучення, які визначають, які ключі потрібно видалити, коли кеш досягає ліміту пам'яті. Загальні правила виселення включають:

- LRU (найменше використовуваний): спочатку видаляє ключі, до яких зверталися найменше за останній час;
- LFU (найрідше використовуваний): спочатку видаляє ключі з найменшою частотою звернень;
- випадкове вилучення: видаляє випадкові ключі, коли потрібно звільнити пам'ять;

- без вилучення: повертає помилку, коли досягнуто ліміту пам'яті, запобігаючи додаванню нових даних.

Політику вилучення можна налаштувати відповідно до конкретних потреб програми:

CONFIG SET maxmemory-policy allkeys-lru

Хоча Redis в основному функціонує як кеш, він також пропонує механізми збереження даних для захисту даних. Двома основними параметрами збереження є знімки RDB (файл бази даних Redis) і ведення журналу AOF (файл лише для додавання). Ці механізми гарантують, що кешовані дані можуть пережити перезавантаження або збої, якщо це необхідно, хоча постійність зазвичай вимикається, коли Redis використовується виключно як кеш.

Анулювання кешу - це процес видалення з кешу застарілих або нерелевантних даних. Redis підтримує явне видалення ключів за допомогою команди DEL. У вебдодатках для синхронізації кешу з основною базою даних використовуються такі стратегії недійсності кешу, як наскрізний запис (write-through), зворотний запис (write-back) і обхідний запис (write-around).

У контексті REST API на базі Express.js Redis використовується як рівень кешування для зменшення навантаження на базу даних і часу відповіді. Кешуючи дані, до яких часто звертаються, Redis зменшує потребу в повторюваних запитах до бази даних, тим самим підвищуючи масштабованість і швидкість реагування API. Крім того, підтримка Redis для обміну повідомленнями pub/sub може полегшити синхронізацію даних у реальному часі та керовану подіями архітектуру в програмі [1, 11].

Блок-схема на рис. 1.1 демонструє функціонування Redis як високопродуктивного рівня кешування.

Caching with Redis Cache

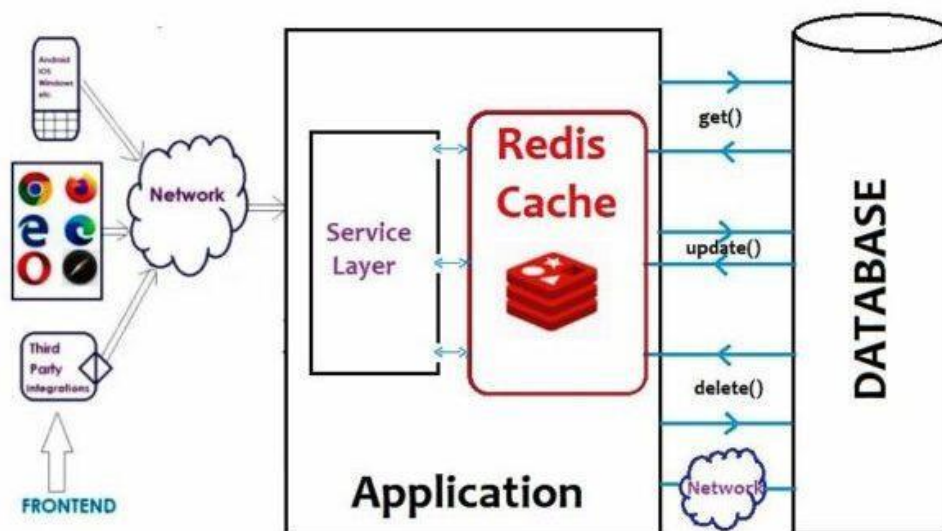


Рис. 1.1 – Механізм кешування в Redis [27]

Redis підтримує різні стратегії кешування, які можна адаптувати до конкретних потреб проєкту:

- **наскрізний запис (write-through):** дані записуються як у кеш, так і в основне сховище даних одночасно - це гарантує, що кеш завжди оновлюється з найновішими даними, але такий спосіб може ввести додаткову затримку під час операцій запису;
- **зворотній запис (write-back):** дані спочатку записуються в кеш, а потім асинхронно зберігаються в основному сховищі даних - це може покращити продуктивність запису, але потенційно збільшує ризик втрати даних у разі системних збоїв до того, як дані будуть повністю збережені;
- **окремо від кешу (cache-aside):** у цій стратегії програма спочатку перевіряє кеш на наявність необхідних даних, і, якщо дані відсутні (промах кешу), вони витягуються з основного сховища даних, повертаються до програми та згодом зберігаються в кеші для

майбутніх запитів – це мінімізує обсяг кеш-пам'яті, кешуючи лише ті дані, які явно запитуються;

- **обхідний запис (write-around):** коли потрібно записати нові дані, вони «обходять» кеш і записуються безпосередньо в базове сховище (наприклад, жорсткий диск або база даних) – такий підхід є доцільним, коли система зосереджена на записі даних або коли цільові дані записуються один раз і майже не використовуються.

Оскільки Redis працює в пам'яті, основним обмеженням цієї БД є розмір доступної оперативної пам'яті. Програми, що обробляють великі набори даних, можуть зіткнутися з проблемами масштабованості, якщо не використовувати ретельне керування пам'яттю та стратегії шардингу. Механізми стійкості Redis, незважаючи на надійність, не забезпечують такий самий рівень довговічності, як традиційні бази даних, що може призводити до ризиків для програм, які потребують гарантій цілісності даних. Крім того, є значними витрати, пов'язані з підтримкою великомасштабних баз даних у пам'яті, оскільки оперативна пам'ять дорожча за дискове сховище. У деяких випадках використання однопотокової архітектури Redis для обробки команд може стати вузьким місцем (bottleneck) у сценаріях із надзвичайно високим рівнем паралелізму, незважаючи на її здатність розподіляти навантаження через кластеризацію.

Для додатків, яким потрібен доступ до даних у реальному часі з керованими розмірами набору даних і мінімальним ризиком втрати даних, Redis пропонує значні переваги в продуктивності [1]. Однак ці обмеження вимагають всебічного розгляду вимог і обмежень програми при виборі Redis як основного сховища даних або рішення для кешування.

1.2.4. Бази даних NewSQL

Бази даних NewSQL є результатом еволюційного етапу в розвитку систем управління даними та поєднують у собі надійні транзакційні гарантії

традиційних SQL-баз даних із горизонтальною масштабованістю, властивою NoSQL-рішенням [21]. Виникнувши як відповідь на обмеження реляційних та нереляційних баз даних, NewSQL поєднує переваги обох підходів: зберігає властивості ACID і одночасно підтримує розподілену архітектуру, здатну ефективно обробляти навантаження даних високої швидкості та обсягів.

Традиційні системи керування реляційними базами даних (RDBMS), такі як MySQL і PostgreSQL, відзначаються надійними моделями узгодженості та можливістю структурованих запитів [37]. Однак вони важко масштабуються горизонтально через властиві архітектурні обмеження. Натомість бази даних NoSQL, такі як MongoDB і Cassandra, характеризуються горизонтальним масштабуванням і гнучкістю, але жертвують суворими гарантіями послідовності на користь можливих моделей узгодженості для досягнення продуктивності та масштабованості.

Бази даних NewSQL долають цю дихотомію, інтегруючи узгодженість транзакцій і можливості експресивних запитів SQL із розподіленою масштабованою архітектурою систем NoSQL. Алгоритми розподіленого консенсусу, такі як Paxos і Raft, використовуються для підтримки узгодженості між розподіленими вузлами, гарантуючи, що всі транзакції будуть виконуватися, незважаючи на збої вузлів або розділи мережі. Крім того, можливості обробки в пам'яті підвищують продуктивність за рахунок зменшення затримки, пов'язаної з операціями введення-виведення диска.

Поділ даних, або шардинг, є ще одним важливим компонентом, що дозволяє базам даних NewSQL ефективно розподіляти дані між кількома вузлами. NewSQL здебільшого використовують багатоверсійний контроль паралелізму (MVCC) для керування одночасними транзакціями, мінімізуючи конфлікт блокувань і підвищуючи пропускну здатність у багатокористувацьких середовищах.

У галузі набули поширення кілька баз даних NewSQL, кожна зі специфічними функціями, адаптованими до конкретних вимог додатків:

- Google Spanner - надає службу глобально розподіленої бази даних, яка підтримує надійну узгодженість і підтримує запити SQL;
- CockroachDB - за допомогою розподіленої архітектури автоматично обробляє збої вузлів і реплікацію даних;
- VoltDB - виконує обробку даних у пам'яті для забезпечення високої пропускної здатності та низької затримки;
- Nuodb - поєднує традиційні можливості надсилання запитів SQL із розподіленою архітектурою, забезпечуючи як цілісність транзакцій, так і масштабованість.

Бази даних NewSQL здебільшого пропонують повну інтеграцію з існуючими інструментами та екосистемами на основі SQL, що полегшує їх впровадження для організацій, які вже інвестували в технології реляційних баз даних. Підтримка розподілених архітектур підвищує відмовостійкість і доступність, завдяки чому додатки залишаються в робочому стані навіть у разі збоїв апаратного забезпечення або мережових проблем [42].

Незважаючи на зазначені переваги NewSQL, складність підтримки чіткої узгодженості між розподіленими вузлами може створити проблеми, особливо з точки зору конфігурації та керування. Переваги продуктивності систем NewSQL можуть відрізнятись залежно від конкретного випадку використання та характеристик робочого навантаження, що потребує ретельної оцінки перед розгортанням.

Як відносно нова категорія баз даних, рішенням NewSQL може бракувати зрілості та широкої підтримки спільноти, якими користуються усталені системи RDBMS і NoSQL. Це може вплинути на доступність ресурсів, документації та сторонніх інтеграцій, потенційно перешкоджаючи прийняттю в певних середовищах. Окрім того, вартість, пов'язана з розгортанням і масштабуванням розподілених баз даних NewSQL, є перешкодою для організацій з обмеженими бюджетами інфраструктури.

1.3. Оптимізація продуктивності REST API

Продуктивність REST API є критично важливим фактором, що визначає користувацький досвід, масштабованість системи та загальну ефективність додатку [4, 7, 29]. У контексті розробки динамічних REST API на основі Express.js виділяють кілька критичних аспектів, які впливають на показники продуктивності.

Час виконання запиту. Є ключовим фактором, що впливає на швидкість реагування REST API. Він охоплює тривалість від ініціювання запиту до бази даних до отримання потрібних даних. Декілька елементів впливають на час виконання запиту, включаючи складність запиту, стратегії індексування та ефективність основної бази даних.

Складні запити, які включають кілька об'єднань, підзапитів або агрегацій, вимагають більше часу обробки, тим самим збільшуючи затримку. Наявність і оптимізація індексів відіграють значну роль у скороченні часу виконання запиту. Добре структуровані індекси дозволяють системі баз даних ефективніше знаходити та отримувати дані, зменшуючи потребу у повному скануванні таблиці. І навпаки, невідповідне індексування може призвести до подовження часу запиту, особливо у великих потоках даних.

Вибір системи бази даних - реляційної, NoSQL або NewSQL - впливає на час виконання запиту через різну оптимізацію архітектури та можливості обробки запитів. Так, розширені механізми індексування та планувальник запитів PostgreSQL можуть підвищити продуктивність для складних запитів, тоді як документоорієнтована природа MongoDB може запропонувати швидший пошук для певних шаблонів неструктурованих даних.

Пул підключень (Connection pooling) - це стратегія, що використовується для ефективного управління та повторного використання підключень до бази даних, що дозволяє зменшити накладні витрати, пов'язані зі встановленням нових підключень для кожного запиту API. У застосунках на основі **Express.js** пул підключень зазвичай реалізується за допомогою

проміжного програмного забезпечення (middleware) або інструментів **ORM** (Object-Relational Mapping).

Ефективне використання пулу підключень мінімізує затримки, і, таким чином, запити API не потребуватимуть часу на встановлення нового підключення для кожної взаємодії. Ця стратегія оптимізує використання ресурсів, обмежуючи кількість одночасних підключень і запобігаючи перевантаженню сервера бази даних [34].

Неправильне налаштування пулу підключень може негативно вплинути на продуктивність. Якщо розмір пулу занадто великий, ресурси бази даних можуть бути вичерпані, і це призведе до конфліктів та збільшення затримок. У разі занадто малого розміру пулу можуть виникнути вузькі місця підключень, що затримуватиме обробку запитів.

Упровадження адаптивних стратегій управління пулом підключень, які змінюють розмір пулу залежно від реального навантаження та моделей використання, може покращити стабільність продуктивності. Додатково використання інструментів моніторингу дозволяє отримувати інформацію про метрики пулу підключень, що дає можливість для тонкого налаштування та проактивного управління підключеннями до бази даних.

Швидкість зчитування та запису даних є критично важливою для продуктивності REST API і залежить від різних чинників, включно з архітектурою бази даних, індексацією даних та характером виконуваних операцій.

Швидкість зчитування оптимізується за допомогою ефективної індексації, механізмів кешування та оптимізації запитів. Бази даних, що підтримують операції у пам'яті, такі як Redis, забезпечують значно швидший доступ до даних порівняно з дисковими системами. Крім того, впровадження кешуючих шарів у застосунках на Express.js дозволяє знизити навантаження на основні бази даних.

Швидкість запису, зі свого боку, визначається здатністю бази даних обробляти одночасні операції запису, управління транзакціями та моделями

узгодженості даних. Бази даних, що використовують вдосконалені механізми контролю конкурентності, такі як MVCC (Multi-Version Concurrency Control) у PostgreSQL, здатні підтримувати високу пропускну здатність запису без втрати цілісності даних. Натомість бази даних з однопотоковою архітектурою можуть стикатися з вузькими місцями запису при великих навантаженнях.

Підвищення продуктивності динамічних REST API на базі Express.js вимагає впровадження надійних методів оптимізації. Як критичні компоненти, серед них виділяються стратегії індексування, сегментування та кешування, які суттєво впливають на ефективність пошуку даних, масштабованість і загальну швидкість реакції системи.

Індексація - це фундаментальна техніка оптимізації для прискорення виконання запиту шляхом мінімізації обсягу даних, які система бази даних повинна сканувати. У реляційних базах даних індекс зазвичай реалізується як структура даних (наприклад, В-дерево або хеш-таблиця), яка підтримує відсортоване відображення значень ключів у відповідних розташуваннях рядків. Коли запит включає стовпець, який індексується, механізм бази даних перейти за індексом, щоб безпосередньо знайти відповідні рядки, обходячи необхідність повного сканування таблиці. Цей підхід значно скорочує час виконання запиту, особливо для великих наборів даних [38].

Для баз даних NoSQL, таких як MongoDB, індекси функціонують аналогічним чином, забезпечуючи ефективні шляхи доступу до документів на основі певних полів. Крім того, складені індекси, які охоплюють кілька полів, полегшують складніші операції запитів, дозволяючи базі даних задовольняти декілька умов запиту одночасно.

Сегментування (шардування, шардинг) - це розширена техніка оптимізації, яка передбачає поділ бази даних на менші, більш керовані частини, які називаються сегментами. Кожен сегмент, або шард, працює як незалежна база даних, що містить підмножину загальних даних. Сегментування полегшує горизонтальне масштабування шляхом розподілу

даних між декількома серверами, тим самим усуваючи вузькі місця та підвищуючи здатність системи обробляти великі обсяги одночасних запитів.

Поділ даних у процесі сегментування відбувається на основі попередньо визначеного ключа, який може бути діапазоном значень, хешем атрибута ключа або іншою схемою логічного розділення. У базах даних NewSQL сегментуванням здебільшого керують прозоро, при цьому механізм бази даних обробляє розподіл даних і реплікацію між сегментами. І навпаки, у базах даних NoSQL, таких як Cassandra, сегментування, іменоване розділенням (partitioning) є невід'ємною частиною архітектури бази даних, яка використовує узгоджене хешування для рівномірного розподілу даних між вузлами.

Кешування — це техніка оптимізації, яка передбачає зберігання даних, до яких часто звертаються, на рівні сховища швидкого доступу, щоб скоротити час пошуку та зменшити навантаження на основні бази даних [41]. Воно працює шляхом тимчасового зберігання копій даних, отримання або обчислення яких потребує значних ресурсних витрат – наприклад, результати запитів до бази даних, відповіді API або візуалізовані шаблони. Коли робиться запит, програма спочатку перевіряє кеш на наявність необхідних даних. Якщо дані присутні (звернення до кешу), вони подаються безпосередньо звідти, оминаючи запит до бази даних або обчислення. Якщо дані відсутні (промах кешу), вони витягуються з первинного джерела, зберігаються в кеші для майбутніх запитів, а потім повертаються клієнту.

Бази даних у пам'яті, такі як Redis, зазвичай використовуються як рівні кешування завдяки їхнім можливостям високошвидкісного доступу до даних. Використовуючи ефективні структури даних Redis і операції з низькою затримкою, API можуть досягати швидкого отримання даних, значно підвищуючи продуктивність [11].

Висновки до розділу 1

У розділі систематично окреслено основні елементи, необхідні для розуміння динаміки продуктивності динамічних REST API на основі Express.js. Викладено принципи архітектури REST та окреслено рамки, які лежать в основі масштабованих і координованих вебслужб. Висвітлено особливості фреймворку Express.js, зокрема його орієнтовану на проміжне програмне забезпечення архітектуру, механізми маршрутизації та повну інтеграцію з Node.js.

Досліджено різноманітність систем баз даних, які використовуються у веброзробці, зокрема реляційні бази даних (SQL), нереляційні бази даних (NoSQL), бази даних у пам'яті та бази даних NewSQL. Розглянуті такі типи баз даних, як MySQL, PostgreSQL, MongoDB, Cassandra, Redis та новітні рішення NewSQL. Порівняльна оцінка надала розуміння, як різні архітектури баз даних узгоджуються з вимогами до продуктивності динамічних REST API.

Розглянуто аспект продуктивності, критичний для розробки API, та визначені такі ключові фактори, як час виконання запиту, пул підключень, а також швидкість отримання та запису даних. Описано вплив цих факторів на швидкість реагування та масштабованість API. У цьому контексті розглянуто методи оптимізації, а саме стратегії індексування, сегментування та кешування, а також розкрито практичні підходи до підвищення продуктивності бази даних і зниження проблем із затримкою. Інтегруючи ці стратегії оптимізації, розробники можуть значно підвищити продуктивність API, гарантуючи, що проекти на базі Express.js працюватимуть стабільно й ефективно за різних умов навантаження.

Таким чином, у розділі поєднано теоретичні концепції з практичними міркуваннями, необхідними для подальшого аналізу продуктивності різних типів баз даних у динамічних REST API.

РОЗДІЛ 2

ПРАКТИЧНА РЕАЛІЗАЦІЯ ДИНАМІЧНОГО REST API НА БАЗІ EXPRESS.JS

2.1. Визначення критеріїв та вибір баз даних

Щоб забезпечити комплексний аналіз продуктивності різних типів баз даних у динамічних REST API на базі Express.js, важливо встановити об'єктивні критерії вибору. Ці критерії визначатимуть, які бази даних та їх комбінації використовуватимуться в різних моделях тестування. Необхідно переконатися, що вибрані бази даних представляють широкий спектр архітектур і характеристик продуктивності, зберігаючи відповідність реальним сценаріям веброзробки. На підставі цих аспектів та рекомендацій, сформульованих у наукових роботах [6, 7, 12, 20, 25] було визначено критерії, на які спиратиметься процес вибору баз даних.

1. Тип БД та архітектура. Оскільки дослідження спрямоване на порівняння різних типів баз даних, щоб отримати повне розуміння їхніх характеристик продуктивності, добірка включатиме:

1. Реляційні бази даних (SQL). Для представлення традиційного структурованого зберігання даних із сумісністю з ACID будуть включені MySQL і PostgreSQL.
2. Бази даних NoSQL. Для аналізу продуктивності в сценаріях, що включають проекти без схем і горизонтальну масштабованість, будуть використовуватися MongoDB і Cassandra.
3. Бази даних у пам'яті. Для оцінки механізмів високошвидкісного пошуку та кешування даних використовуватиметься Redis.
4. Бази даних NewSQL. Для перевірки сучасних баз даних, які поєднують узгодженість SQL із масштабованістю NoSQL, буде розглянуто CockroachDB.

Різноманітність типів баз даних дозволить охопити низку варіантів використання та архітектурних парадигм, що стосуються сучасної розробки API.

2. Складність запиту. Вибрані бази даних мають підтримувати різні рівні складності запитів, щоб імітувати реальні сценарії. Будуть розглянуті такі типи запитів:

- прості запити: пошук по одній таблиці та основні операції CRUD (створення, читання, оновлення, видалення);
- складні запити: об'єднання, агрегації, вкладені запити та операції фільтрації;
- запити з високим рівнем конкурентності: одночасні операції читання та запису для оцінки продуктивності бази даних під навантаженням.

Реляційні бази даних будуть оцінюватися на їх здатність обробляти складні об'єднання та агрегації, тоді для баз даних NoSQL буде проведена оцінка ефективності у виконанні пошуку на основі документів і ключового значення. Тестування баз даних у пам'яті будуть зосереджені на швидких операціях CRUD і кешуванні.

3. Обсяг даних і масштабованість. Щоб перевірити, як бази даних працюють під різними навантаженнями даних, у дослідженні розглядатимуться різні обсяги даних:

- малі набори даних: для оцінки продуктивності з мінімальними даними (в межах 1000 записів);
- середні набори даних: для типових робочих навантажень вебдодатків (в межах 100 тисяч записів);
- великі набори даних: для масштабних проєктів (мільйони записів).

Тести масштабованості вимірюють, як кожна база даних обробляє зростаючі обсяги даних і одночасні підключення. Можливості горизонтального масштабування баз даних NoSQL і NewSQL перевірятимуться за допомогою розподіленого розгортання, тоді як реляційні бази даних оцінюватимуться на межі вертикального масштабування.

4. Продуктивність читання й запису. Враховуючи важливість адаптивності API, дослідження оцінюватиме:

- швидкість отримання даних за різних умов запиту;
- ефективність операцій вставлення, оновлення та видалення даних.

Бази даних перевірятимуться на сценарії, пов'язані з інтенсивним навантаженням на читання (поширене в програмах, що обслуговують контент) і навантаження на запис (поширене в журналах та аналітиці в реальному часі). Також буде проаналізовано баланс ефективності читання та запису.

5. Затримка та пропускна здатність. Затримка – це час, витрачений на виконання окремих запитів або транзакцій. Пропускна здатність - кількість запитів або транзакцій, оброблених за секунду під навантаженням. Ці показники допоможуть оцінити швидкість реагування та продуктивність кожного типу бази даних для обробки запитів REST API. Буде проведено порівняння різних архітектур баз даних, щоб визначити сильні та слабкі сторони в різних умовах навантаження.

6. Можливості інтеграції з Express.js. Вибрані бази даних мають безперешкодно інтегруватися з Express.js, щоб забезпечити практичну доцільність їх використання. Критерії інтеграції включають:

- наявність офіційних драйверів і бібліотек: підтримка драйверів Node.js (наприклад, **mongoose** для MongoDB, **pg** для PostgreSQL);
- сумісність проміжного програмного забезпечення: можливість інтеграції з проміжним програмним забезпеченням Express.js для пулу з'єднань, логування запитів і обробки помилок;
- підтримка спільноти та документація: наявність ресурсів для полегшення розробки та усунення несправностей.

7. Кешування. Для баз даних, продуктивність яких можна підвищити за допомогою кешування, дослідження оцінюватиме їхню сумісність із рішеннями для кешування, такими як Redis. Буде розглянуто здатність

реалізувати різні стратегії кешування (окремо від кешу, наскрізний запис, зворотній запис, обхідний запис) для оцінки покращення продуктивності.

На основі викладених критеріїв для різних моделей тестування було обрано такі бази даних:

1. Реляційні бази даних: MySQL, PostgreSQL.
2. Бази даних NoSQL: MongoDB, Cassandra.
3. База даних у пам'яті: Redis.
4. База даних NewSQL: CockroachDB.

Зазначені бази даних буде перевірено окремо та в поєднанні з рішеннями для кешування.

2.2. Проєктування і розробка REST API

Розробка кінцевих точок (endpoints) у REST API для операцій CRUD є основоположним аспектом проєктів на основі Express.js. Ці операції відповідають таким методам HTTP, як POST, GET, PUT і DELETE, забезпечуючи стандартизований спосіб взаємодії з даними, що зберігаються в різних базах даних. Далі розглядається впровадження кінцевих точок CRUD у Express.js, демонструючи інтеграцію з реляційними базами даних, базами даних NoSQL, базами даних у пам'яті та NewSQL.

1. Перш ніж розробляти кінцеві точки API, ініціалізується базова структура проєкту **Express.js**:

```
mkdir express-api
```

```
cd express-api
```

```
npm init -y
```

Цей крок ініціалізує новий проєкт **Node.js** шляхом створення файлу **package.json** із стандартними конфігураціями. Команда **npm init -y** автоматизує процес, приймаючи параметри за замовчуванням.

2. Встановлення залежностей:

```
npm install express body-parser mongoose pg redis cassandra-driver
```

Команда `npm install` встановлює **Express.js** для створення сервера, **body-parser** - для аналізу вхідних корисних даних JSON і бібліотек для бази даних (mongoose для MongoDB, pg для PostgreSQL, redis для Redis і cassandra-driver для Cassandra).

3. Створення файлу головного сервера (див. рис. 2.1). Виконується за допомогою коду, який налаштовує базовий сервер Express.js із проміжним програмним забезпеченням аналізатора тіла для обробки корисних даних JSON. Сервер прослуховує порт 3000 і реєструє повідомлення під час запуску.

```
// index.js
const express = require('express');
const bodyParser = require('body-parser');

const app = express();
app.use(bodyParser.json());

const port = 3000;
app.listen(port, () => {
  console.log(`Server is running on port ${port}`);
});
```

Рис. 2.1 – Створення файлу головного сервера

4. Визначення кінцевих точок CRUD, які відповідають принципам RESTful. Для ресурсу з назвою **items** кінцеві точки структуровані таким чином, як показано в табл. 2.1.

Таблиця 2.1 – Структура кінцевих точок

HTTP-метод	Кінцева точка	Опис
POST	/items	Створити новий елемент
GET	/items	Отримати всі елементи
GET	/items/:id	Отримати конкретний елемент
PUT	/items/:id	Оновити існуючий елемент
DELETE	/items/:id	Видалити існуючий елемент

Розглянемо реалізацію кінцевих точок CRUD у Express.js за допомогою різних типів баз даних.

Для PostgreSQL спочатку ініціалізується пул підключень до бази даних за допомогою бібліотеки **pg** (див. рис. 2.2). Він визначає такі параметри підключення, як користувач, хост, база даних, пароль і порт, що дозволяє ефективно використовувати підключення. З міркувань безпеки справжнє ім'я користувача, адреса хосту, ім'я бази даних та пароль на рисунках в подальшому будуть замінені шаблоном.

```
// db/postgres.js
const { Pool } = require('pg');

const pool = new Pool({
  user: 'your_user',
  host: 'localhost',
  database: 'your_db',
  password: 'your_password',
  port: 5432,
});

module.exports = pool;
```

Рис. 2.2 – Конфігурація бази даних PostgreSQL

Наступним кроком є визначення точок RESTful API для операцій CRUD. Кінцеві точки керують створенням нового елемента, отриманням усіх елементів і отриманням елемента за його ідентифікатором. Функція **pool.query** виконує SQL-запити з параметризованими вхідними даними, щоб запобігти ін'єкції SQL. На рис. 2.3 показано фрагмент коду, який визначає точку для операції GET за id елемента.

```
// Get item by ID
router.get('/items/:id', async (req, res) => {
  const { id } = req.params;
  try {
    const result = await pool.query('SELECT * FROM items WHERE id = $1', [id]);
    if (result.rows.length === 0) {
      return res.status(404).send('Item not found');
    }
    res.json(result.rows[0]);
  } catch (err) {
    res.status(500).send(err.message);
  }
});

module.exports = router;
```

Рис. 2.3 – Визначення кінцевої точки GET (отримання певного елемента)

Далі необхідно імпортувати маршрути на основі PostgreSQL та інтегрувати їх у головний сервер Express.js. Це робить за допомогою коду, показаного на рис. 2.4. Усі маршрути мають префікс **/api**, що забезпечує чітку та послідовну структуру API.

```
// index.js
const express = require('express');
const bodyParser = require('body-parser');
const itemRoutes = require('./routes/itemsPostgres');

const app = express();
app.use(bodyParser.json());
app.use('/api', itemRoutes);

const port = 3000;
app.listen(port, () => {
  console.log(`Server is running on port ${port}`);
});
```

Рис. 2.4 - Інтеграція маршрутів у файл головного сервера (PostgreSQL)

Для бази даних MongoDB, яка є нереляційною, з'єднання з базою даних виконується у файлі конфігурації з використанням бібліотеки **mongoose** (див. рис. 2.5). Файл конфігурації визначає схему колекції елементів із зазначенням полів для імені та опису. Модель **Item** експортується для використання в операціях CRUD.

```
// db/mongodb.js
const mongoose = require('mongoose');

mongoose.connect('mongodb://localhost:27017/express_api', {
  useNewUrlParser: true,
  useUnifiedTopology: true,
});

const itemSchema = new mongoose.Schema({
  name: String,
  description: String,
});

const Item = mongoose.model('Item', itemSchema);

module.exports = Item;
```

Рис. 2.5 – Конфігурація бази даних MongoDB

У наступному файлі визначаються кінцеві точки RESTful API для операцій CRUD за допомогою MongoDB (див. рис. 2.6). Як і у випадку PostgreSQL, кінцеві точки обробляють створення нових елементів і отримання всіх елементів. Модель Item взаємодіє з MongoDB, використовуючи методи **mongoose**, такі як **save** і **find**, для обробки даних.

```

// Create a new item
router.post('/items', async (req, res) => {
  try {
    const item = new Item(req.body);
    await item.save();
    res.status(201).json(item);
  } catch (err) {
    res.status(500).send(err.message);
  }
});

// Get all items
router.get('/items', async (req, res) => {
  try {
    const items = await Item.find();
    res.json(items);
  } catch (err) {
    res.status(500).send(err.message);
  }
});

```

Рис. 2.6 – Визначення кінцевих точок POST і GET для MongoDB

Аналогічним чином, як було показано на рис. 2.4 для PostgreSQL, виконується код, що імпортує маршрути MongoDB та інтегрує їх у головний сервер Express.js. Єдиною відмінністю буде шлях до файлу `itemsMongo.js` (рядок 3), який буде виглядати так:

```
const itemRoutes = require('./routes/itemsMongo')
```

Під час розробки кінцевих точок CRUD критичним аспектом є вирішення типових проблем, такі як обробка помилок, валідація, використання проміжного програмного забезпечення та масштабованість.

Обробка помилок гарантує, що API зможе коректно впоратися з непередбачуваними ситуаціями, такими як помилки підключення до бази даних, недійсні введення та відсутність ресурса, до якого звертаються. Належні відповіді на помилки надають значущий фідбек і запобігають збою API.

Зокрема, код на рис. 2.7 встановлює пул з'єднань PostgreSQL і включає обробник подій помилок. Якщо під час простою клієнта виникає помилка підключення до бази даних, ця помилка реєструється, і процес завершується, щоб запобігти непередбачуваній поведінці.

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

const pool = new Pool({
  user: 'your_user',
  host: 'localhost',
  database: 'your_db',
  password: 'your_password',
  port: 5432,
});

// Handle connection errors
pool.on('error', (err) => {
  console.error('Unexpected error on idle PostgreSQL client', err);
  process.exit(-1);
});
```

Рис. 2.7 – Обробка помилки при втраті з'єднання

Не виключенням є ситуація, коли кінцева точка намагається отримати елемент за його ідентифікатором. Якщо елемент не знайдено, він повертає статус «404 Not Found» з відповідним повідомленням про помилку. Якщо під час запиту виникає помилка на стороні сервера, повертається статус внутрішньої помилки сервера 500. Код реалізації обробки цієї помилки представлений на рис. 2.8.

```
// routes/items.js
router.get('/:id', async (req, res) => {
  const { id } = req.params;
  try {
    const result = await pool.query('SELECT * FROM items WHERE id = $1', [id]);
    if (result.rows.length === 0) {
      return res.status(404).json({ error: 'Item not found' });
    }
    res.json(result.rows[0]);
  } catch (err) {
    res.status(500).json({ error: 'Internal server error' });
  }
});
```

Рис. 2.8 – Обробка помилки 404

Валідація введених даних гарантує, що дані, отримані API, є коректними та безпечними. Це запобігає обробці або зберіганню недійсних даних, що допомагає підтримувати їхню цілісність і безпеку.

На рис. 2.9 показано, як кінцева точка використовує експрес-валідатор для перевірки даних вхідного запиту. Проміжне програмне забезпечення body у даному випадку перевіряє, чи поле назви не є порожнім і чи містить поле опису принаймні 10 символів. Якщо перевірка не вдається, повертається відповідь 400 Bad Request із детальними повідомленнями про помилки.

```
const { body, validationResult } = require('express-validator');

router.post(
  '/items',
  [
    body('name').notEmpty().withMessage('Name is required'),
    body('description').isLength({ min: 10 }).withMessage('Description must be at least 10'),
  ],
  async (req, res) => {
    const errors = validationResult(req);
    if (!errors.isEmpty()) {
      return res.status(400).json({ errors: errors.array() });
    }
  }
);
```

Рис. 2.9. – Валідація вхідного запиту

Проміжне програмне забезпечення (middleware) в Express.js допомагає керувати такими завданнями, як автентифікація, логування, парсинг запитів і обробка помилок. Функції middleware виконуються послідовно протягом циклу запит-відповідь, що забезпечує модульність коду та можливість його повторного використання. Інтеграція, показана на рис. 2.10, застосовує проміжне програмне забезпечення для логування глобально до всіх маршрутів. Кожен запит до сервера буде зареєстровано перед обробкою відповідним обробником маршруту.

```
const express = require('express');
const bodyParser = require('body-parser');
const logger = require('./middleware/logger');
const itemRoutes = require('./routes/items');

const app = express();
app.use(bodyParser.json());
app.use(logger); // Apply logging middleware

app.use('/api', itemRoutes);

const port = 3000;
app.listen(port, () => {
  console.log(`Server is running on port ${port}`);
});
```

Рис. 2.10 – Інтеграція проміжного програмного забезпечення

Коли middleware інтегровано, за допомогою нього можна реєструвати метод HTTP, URL запиту та позначку часу для кожного вхідного запиту. Так, функція next() у коді на рис. 2.11 передає керування наступному проміжному програмному забезпеченню в ланцюжку або кінцевому обробнику маршруту.

```
// middleware/logger.js
const logger = (req, res, next) => {
  console.log(`${req.method} ${req.url} at ${new Date().toISOString()}`);
  next();
};

module.exports = logger;
```

Рис. 2.11 – Використання проміжного ПЗ для ведення логів запитів

Забезпечення масштабованості API може бути необхідним у реальних проєктах для того, щоб справлятися зі зростаючими навантаженнями в міру зростання кількості користувачів і обсягів даних. Такі методи, як об'єднання з'єднань для реляційних баз даних і реплікація для баз даних NoSQL, допомагають підтримувати продуктивність під високими навантаженнями. Наприклад, для MongoDB в межах цього проєкту реплікацію було налаштовано за допомогою наборів реплік:

```
mongoose.connect('mongodb://primary-node:27017,secondary-  
node1:27017, secondary-node2:27017/your_db?replicaSet=rs0', {  
  useUrlParser: true,  
  useUnifiedTopology: true, });
```

Набір реплік складається з основного вузла, який обробляє операції запису, і кількох вторинних вузлів, які реплікують дані. Це налаштування підвищує доступність даних і стійкість до відмов.

Таким чином, було показано, що розробка кінцевих точок CRUD у Express.js вимагає ретельного розгляду рекомендованих практик. Обробка помилок, валідація вхідних даних, ефективне проміжне ПЗ та методи масштабованості уможовлює створення стійкого та придатного до обслуговування API. Ці практики закладають основу для наступних кроків інтеграції бази даних і дослідження їхньої ефективності.

Наступним етапом розробки динамічних вебдодатків є інтеграція баз даних із REST API. Цей процес передбачає налаштування підключень до бази даних, обробку запитів, керування пулами підключень і забезпечення ефективного пошуку й зберігання даних. Кожен тип бази даних вимагає певних конфігурацій і механізмів обробки запитів.

З реляційними базами даних Express.js взаємодіє за допомогою клієнтських бібліотек, таких як pg для PostgreSQL і mysql2 для MySQL. Правильна конфігурація пулів з'єднань підвищує продуктивність за рахунок повторного використання з'єднань замість створення нових для кожного запиту [8, 15]. На рис. 2.12 показано приклад коду, який ініціалізує пул

з'єднань для PostgreSQL. Пул з'єднань допомагає керувати кількома підключеннями до бази даних, зменшуючи затримку та споживання ресурсів.

```
const express = require('express');
const pool = require('../db/postgres');
const router = express.Router();

router.get('/items', async (req, res) => {
  try {
    const result = await pool.query('SELECT * FROM items');
    res.json(result.rows);
  } catch (err) {
    res.status(500).send('Database error');
  }
});

module.exports = router;
```

Рис. 2.12 – Встановлення пулу з'єднань для PostgreSQL

Аналогічним чином встановлення пулу з'єднань відбувається для MySQL. Бібліотека **mysql2/promise** дозволяє виконувати асинхронні запити за допомогою **async/await**.

До нереляційних баз даних Express.js підключається за допомогою бібліотек **mongoose** для MongoDB і **cassandra-driver** для Cassandra. У коді, представленому на рис. 2.13, відбувається підключення до MongoDB за допомогою **mongoose**, визначається схема для колекції елементів і експортується модель для операцій з даними.

```
const mongoose = require('mongoose');

mongoose.connect('mongodb://localhost:27017/express_api', {
  useNewUrlParser: true,
  useUnifiedTopology: true,
});

const itemSchema = new mongoose.Schema({
  name: String,
  description: String,
});

const Item = mongoose.model('Item', itemSchema);

module.exports = Item;
```

Рис. 2.13 – Конфігурація з'єднання з MongoDB

Інтеграція Redis із Express.js передбачає налаштування клієнта Redis, налаштування стратегій кешування та включення операцій Redis у кінцеві точки API. Для цього встановлюється підключення до сервера Redis за допомогою бібліотеки redis, яка надає клієнт для взаємодії з Redis із Node.js (див. рис. 2.14).

```
const redis = require('redis');

// Create a Redis client
const client = redis.createClient({
  host: 'localhost', // Redis server host
  port: 6379,        // Redis server port
});

// Handle connection events
client.on('error', (err) => {
  console.error('Redis connection error:', err);
});
client.on('connect', () => {
  console.log('Connected to Redis');
});

module.exports = client;
```

Рис. 2.14 – Конфігурація клієнта Redis

Представлений код ініціалізує клієнт Redis, налаштований на підключення до сервера Redis, який працює локально на порту 6379. Слухачі подій налаштовані на реєстрацію повідомлень після успішного з'єднання (подія з'єднання) і для обробки будь-яких помилок з'єднання (подія про помилку). Експорт клієнта дозволяє іншим модулям використовувати операції Redis у додатку.

Для зберігання та отримання часто використовуваних даних впроваджується інтеграція кешування Redis у маршрути Express.js. Модуль маршруту представляє кешування для кінцевої точки GET /items/:id. Функція проміжного програмного забезпечення checkCache намагається отримати запитаний елемент із Redis, використовуючи ідентифікатор елемента як ключ. Якщо дані існують у кеші (дані правдиві), вони аналізуються з JSON і

повертаються негайно, минаючи запит до бази даних. Якщо дані не знайдено в Redis, запит переходить до обробника маршруту, який запитує PostgreSQL для елемента. Після успішного отримання елемент зберігається в Redis із TTL (Time-to-Live) впродовж 60 секунд за допомогою `setex` (див. рис. 2.15).

```
router.get('/items/:id', checkCache, async (req, res) => {
  const { id } = req.params;
  try {
    const result = await pool.query('SELECT * FROM items WHERE id = $1', [id]);
    if (result.rows.length === 0) {
      return res.status(404).json({ error: 'Item not found' });
    }
    const item = result.rows[0];
    // Store the retrieved item in Redis with a TTL of 60 seconds
    redisClient.setex(id, 60, JSON.stringify(item));
    res.json(item);
  } catch (err) {
    console.error('Error retrieving item:', err);
    res.status(500).json({ error: 'Internal server error' });
  }
});
```

Рис. 2.15 – Функція отримання конкретного об'єкта з кешу

Щоб функція кешування працювала, її необхідно включити в API. У випадку інтеграції PostgreSQL з Redis основний серверний файл об'єднуватиме як маршрути CRUD, наприклад для PostgreSQL, так і маршрути кешування на основі Redis, імпортуючи та монтуючи їх у шляху `/api`. На рис. 2.16 показана реалізація інтеграції, де, зокрема, змінна `ItemRoutes` відповідає за обробку стандартних операцій CRUD за допомогою PostgreSQL, тоді як `itemRedisRoutes` запроваджує кешування для конкретних кінцевих точок за допомогою Redis. Проміжне програмне забезпечення `bodyParser.json()` аналізує вхідні корисні дані JSON, а проміжне програмне забезпечення реєстратора запитує деталі з метою моніторингу. Сервер прослуховує порт 3000, підтверджуючи успішну ініціалізацію консольним повідомленням.

```

const bodyParser = require('body-parser');
const logger = require('./middleware/logger');
const itemRoutes = require('./routes/itemsPostgres'); // Primary CRUD routes
const itemRedisRoutes = require('./routes/itemsRedis'); // Redis caching routes

const app = express();

// Middleware to parse JSON bodies
app.use(bodyParser.json());

// Apply logging middleware globally
app.use(logger);

// Use the primary PostgreSQL-based item routes under the /api path
app.use('/api', itemRoutes);

// Use the Redis-based item routes
app.use('/api', itemRedisRoutes);

```

Рис. 2.16 – Інтеграція маршрутів Redis в головний файл серверу

На додаток до інтеграції Redis для кешування окремих кінцевих точок, він застосовується більш широко в Express.js для оптимізації продуктивності додатку. Створення глобального проміжного програмного забезпечення для кешування, показане на рис. 2.17, дає можливість систематично кешувати відповіді для декількох кінцевих точок.

```

// middleware/cache.js
const redisClient = require('../db/redis');

// Global cache middleware
const cacheMiddleware = (req, res, next) => {
  const key = req.originalUrl;
  redisClient.get(key, (err, data) => {
    if (err) {
      console.error('Redis GET error:', err);
      return res.status(500).json({ error: 'Internal server error' });
    }
    if (data) {
      return res.json(JSON.parse(data));
    }
    next();
  });
};

```

Рис. 2.17 – Імплементація проміжного ПЗ для глобального кешування

Проміжне програмне забезпечення перевіряє наявність кешованої відповіді на основі URL-адреси запиту. Якщо відповідь уже є в кеші, вона негайно повертається клієнту, міняючи обробник маршруту. Якщо відповіді в кеші немає, запит передається наступному проміжному програмному забезпеченню або обробнику маршруту, де результат обробки може бути збережений у кеші для майбутніх запитів. Цей підхід є доцільним для кінцевих точок, які повертають статичні або рідко змінювані дані, що дозволяє підвищити продуктивність API.

Для впровадження глобального кешування проміжне програмне забезпечення глобального кешування до відповідних маршрутів, щоб увімкнути функцію кешування на кількох кінцевих точках (див. рис. 2.18).

```
router.get('/items', cacheMiddleware, async (req, res) => {
  try {
    const result = await pool.query('SELECT * FROM items');
    const items = result.rows;
    // Store the retrieved items in Redis with a TTL of 120 seconds
    redisClient.setex(req.originalUrl, 120, JSON.stringify(items));
    res.json(items);
  } catch (err) {
    console.error('Error retrieving items:', err);
    res.status(500).json({ error: 'Internal server error' });
  }
});
```

Рис. 2.18 – Застосування проміжного ПЗ глобального кешування до маршрутів

Так, у модулі маршруту для кінцевої точки GET/items застосовується cacheMiddleware, що дозволяє кешувати відповідь для запитів на отримання всіх елементів. При зверненні до /items проміжне програмне забезпечення перевіряє, чи є відповідь у кеші Redis. Якщо дані знайдені, вони повертаються клієнту негайно. Якщо даних у кеші немає, виконується запит до бази даних PostgreSQL, після чого отримані результати зберігаються в Redis із часом життя 120 секунд за допомогою команди setex.

Для активування кешування на рівні кількох кінцевих точок включаються глобально кешовані маршрути до основного сервера Express.js. Основний серверний файл об'єднує різні модулі маршрутів, що охоплюють первинні операції CRUD, кешування, специфічні для Redis, і глобальне кешування. Монтуючи ці маршрути під шляхом /api, програма забезпечує узгоджене застосування стратегій кешування до відповідних кінцевих точок. Додаткове проміжне програмне забезпечення, як `bodyParser.json()`, обробляє вхідні дані у форматі JSON, а логер фіксує всі вхідні запити для спрощення моніторингу. Сервер слухає порт 3000 і підтверджує успішне розгортання відповідним повідомленням у консолі.

Бази даних NewSQL, такі як CockroachDB, поєднують узгодженість і реляційні функції традиційних баз даних SQL із масштабованістю та відмовостійкістю систем NoSQL. CockroachDB - це розподілена база даних, яка підтримує горизонтальне масштабування, автоматичну реплікацію та SQL-запити [20, 21]. Інтеграція CockroachDB з Express.js передбачає налаштування з'єднання, визначення схем і реалізацію операцій CRUD.

CockroachDB використовує протокол зв'язку PostgreSQL, що дозволяє працювати з бібліотеками, сумісними з PostgreSQL, такими як pg. Ця сумісність дозволяє використовувати аналогічні інструменти, використовуючи розподілену архітектуру CockroachDB. Для початку інтеграції останньої необхідно налаштувати пул підключень до CockroachDB за допомогою бібліотеки pg (див. рис. 2.19).

Оскільки CockroachDB використовує SQL для визначення схем, створюється таблиця для зберігання даних про елементи, щоб забезпечити сумісність із функціями реляційного SQL, одночасно використовуючи переваги розподіленої архітектури CockroachDB. У сценарії SQL визначається таблиця елементів із трьома полями: `id` (серіальний первинний ключ), `name` (обов'язковий рядок до 100 символів) і `description` (обов'язкове текстове поле).

```
// db/cockroach.js
const { Pool } = require('pg');

// Configure the CockroachDB connection pool
const pool = new Pool({
  user: 'your_user',      // Database username
  host: 'localhost',      // CockroachDB node host
  database: 'your_db',    // Database name
  password: 'your_password', // Database password
  port: 26257,            // Default CockroachDB port
  ssl: {
    rejectUnauthorized: false, // Use SSL for secure connections
  },
  max: 20,                // Maximum number of connections
  idleTimeoutMillis: 30000, // Close idle clients after 30 seconds
});
```

Рис. 2.19 – Конфігурація з'єднання з CockroachDB

Маршрути Express.js для виконання операцій CRUD над таблицею елементів у CockroachDB визначаються у спосіб аналогічний показаному на рис. 2.4. Після цього маршрути на основі CockroachDB включаються у проєкт Express.js, щоб активувати кінцеві точки CRUD.

Для спрощення взаємодії між додатком на Express.js і різними типами баз даних у цьому проєкті використовуються інструменти ORM (об'єктно-реляційне відображення) і ODM (об'єктно-документне відображення) шляхом абстрагування низькорівневих операцій бази даних у більш інтуїтивно зрозумілі функції вищого рівня. Кожен тип бази даних інтегрується з певним ORM або ODM, що відповідає його архітектурі та моделі даних.

Для PostgreSQL проєкт використовує бібліотеку **pg** у поєднанні з базовими запитам SQL для виконання операцій CRUD. Хоча **pg** не є повноцінною ORM, вона дозволяє виконувати параметризовані запити. Використання необробленого SQL забезпечує точний контроль над структурою та виконанням запитів. У більш складних сценаріях використовується Sequelize для забезпечення повних можливостей ORM, таких як визначення моделей, асоціації та автоматичні міграції.

У випадку MySQL використовується бібліотека **mysql2** для полегшення взаємодії між програмою Express.js і базою даних MySQL. Хоча ця бібліотека дозволяє виконувати необроблені SQL-запити, її можна розширити за допомогою ORM Sequelize, щоб абстрагувати ці запити в операції JavaScript вищого рівня. Завдяки Sequelize, моделі, що представляють таблиці бази даних, можна визначити за допомогою атрибутів, типів даних і обмежень. Ця абстракція спрощує операції CRUD, дозволяючи виконувати створення, читання, оновлення та видалення записів без написання великого коду SQL. Sequelize також підтримує асоціації, полегшуючи керування зв'язками між таблицями, і забезпечує автоматичні міграції для розвитку схем бази даних з часом.

Під час роботи з MongoDB проект використовує **mongoose** як інструмент ODM, яка полегшує визначення схем і моделей, представляючи колекції як об'єкти JavaScript із чіткою структурою та правилами перевірки. Схеми дозволяють визначати типи полів, обмеження та значення за замовчуванням. Операції CRUD спрощуються за допомогою таких методів, як **save()**, **find()** і **findByIdAndUpdate()**, які абстрагують складності написання власних запитів MongoDB. Крім того, mongoose забезпечує перехоплення проміжного програмного забезпечення для таких завдань, як перевірка введення та попередня обробка.

При роботі з Cassandra інтеграція спирається на бібліотеку драйверів **cassandra**, адаптовану до розподіленої природи цієї бази даних. Драйвери **cassandra** підтримують виконання запитів CQL (Cassandra Query Language) безпосередньо в програмі Express.js, забезпечуючи контроль над ключами розділів і стовпцями кластеризації. Хоча повноцінні інструменти ORM менш поширені для Cassandra, бібліотека абстракцій запитів **express-cassandra**, була використана для визначення моделей і схем у JavaScript.

Для Redis використовується спрощений підхід без ORM або ODM. Redis працює як сховище ключ-значення і взаємодії керуються безпосередньо через клієнтську бібліотеку **redis**. Операції включають налаштування та отримання

даних за допомогою команд, таких як **set** і **get**, з додатковою підтримкою розширених методів кешування. Оскільки Redis не зберігає дані в структурованому реляційному або документальному форматі, немає потреби в рівні абстракції, як ORM або ODM.

У випадку бази даних CockroachDB проєкт використовує бібліотеку **pg** через сумісність CockroachDB з протоколом PostgreSQL. Подібно до інтеграції PostgreSQL, для взаємодії з базою даних використовуються параметризовані запити SQL. Використання необробленого SQL дозволяє використовувати розподілені функції CockroachDB, такі як автоматична реплікація та горизонтальне масштабування.

Вибір інструментів ORM або ODM у цьому проєкті ґрунтується на характері кожної бази даних і вимогах програми. Реляційні бази даних виграють від орієнтованих на SQL бібліотек, які підтримують складні запити та транзакції. Бази даних, орієнтовані на документи, такі як MongoDB, продуктивно обслуговуються ODM, які пропонують визначення та перевірку схем. Redis ефективно працює із прямою взаємодією на основі команд, тоді як CockroachDB використовує бібліотеки на основі SQL для використання розподілених реляційних можливостей.

2.3. Методологія тестування

Тестування продуктивності динамічного REST API на основі Express.js з різними базами даних вимагає чітко визначеного середовища тестування, щоб забезпечити точні та відтворювані результати. Середовище тестування складається з конкретних конфігурацій обладнання, налаштувань програмного забезпечення та мережевих умов, які наближені до реальних сценаріїв.

Тести продуктивності проводяться на виділеній машині з сучасними апаратними компонентами, щоб мінімізувати вузькі місця, не пов'язані з базами даних. Технічні характеристики машини, на якій проводитиметься випробування, є такими:

- **ЦП:** Intel Core i7-12700K (12 ядер, 20 потоків, базова тактова частота 3,6 ГГц, розширена тактова частота 5,0 ГГц).
- **Оперативна пам'ять:** 32 ГБ DDR4-3200 МГц.
- **Сховище:** 1 ТБ NVMe SSD (PCIe 4.0) для основного зберігання баз даних і програмного коду.
- **Графічний процесор:** NVIDIA GeForce GTX 1660 (не використовується для тестування, оскільки тестування продуктивності зосереджено на ЦП та ввіді-виводі).
- **Операційна система:** Ubuntu 22.04 LTS (Linux Kernel 5.15).

Процесор Intel Core i7 забезпечує надійні багатопотокові можливості, необхідні для імітації одночасних запитів API та операцій з базою даних. 32 ГБ оперативної пам'яті гарантують наявність достатньої пам'яті для одночасного запуску кількох екземплярів бази даних, механізмів кешування та інструментів тестування продуктивності. Використання NVMe SSD зменшує затримку дискового вводу-виводу, щоб обмеження пам'яті не спотворили вимірювання продуктивності бази даних. Ubuntu 22.04 обрано через його стабільність, оптимізацію продуктивності та сумісність із програмами на стороні сервера.

Кожен тип бази даних - реляційна (MySQL, PostgreSQL), нереляційна (MongoDB, Cassandra), база даних у пам'яті (Redis) і NewSQL (CockroachDB) - розгортається на одній машині для підтримки узгодженості апаратних ресурсів. Бази даних налаштовано з параметрами за замовчуванням, оптимізованими для середовищ розробки, з незначними коригуваннями для пулу з'єднань, кешування та реплікації за потреби.

Для баз даних, які вимагають розподілених налаштувань, таких як Cassandra та CockroachDB, локальний багатовузловий кластер був змодельований за допомогою контейнерів Docker. Кожному вузлу були

виділені окремі порти та каталоги зберігання, щоб імітувати реалістичне розподілене середовище, зберігаючи відтворюваність тестів на одній машині.

Умови мережі контролюються за допомогою **tc** (Traffic Control) у Linux. Наведені нижче мережеві конфігурації застосовуються для введення мінливості та спостереження за впливом на продуктивність API:

- імітована затримка від 10 мс до 50 мс введена для імітації затримок, які зазвичай виникають у середовищах локальної мережі (LAN) і глобальної мережі (WAN);
- пропускна здатність мережі обмежена 100 Мбіт/с, щоб імітувати звичайну швидкість зв'язку між сервером і базою даних;
- введено номінальний рівень втрати пакетів 0,1%, щоб імітувати незначну нестабільність мережі.

Ці мережеві параметри дозволяють тестам продуктивності відображати умови, подібні до тих, які існують у виробничих розгортаннях, наприклад, у хмарних службах або територіально розподілених системах. Завдяки введенню обмежень контрольованої затримки та пропускної здатності тести можуть оцінити, як кожен тип бази даних справляється з реальними проблемами мережі, такими як затримка відповідей, обмеження передачі даних і тимчасові проблеми з підключенням.

Середовище тестування продуктивності додатково ізольоване за допомогою Docker, щоб переконатися, що кожен тестовий запуск не втручається фоновими процесами або змінами на рівні системи. Контейнери Docker використовуються як для програми Express.js, так і для різних баз даних. Контейнерам призначаються конкретні ядра ЦП і обмеження пам'яті, щоб запобігти конкуренції за ресурси.

Основні інструменти, які використовуються для тестування продуктивності, включають:

1. **Apache JMeter** - для імітації одночасних запитів API, вимірювання часу відповіді та оцінки пропускної здатності.

2. **Artillery** - для створення навантажувальних і стрес-тестів зі звітами в реальному часі.
3. **k6** - для скриптування складних сценаріїв продуктивності та генерації детальних показників продуктивності.

Зазначені інструменти налаштовані для надсилання HTTP-запитів до кінцевих точок Express.js API за різних умов навантаження, починаючи від низького (10 запитів на секунду) до високого (1000 запитів на секунду). Результати записуються та аналізуються для виявлення вузьких місць у продуктивності, таких як повільний час виконання запитів, неефективне об'єднання з'єднань і затримки, спричинені затримкою мережі. Завдяки визначенню узгоджених специфікацій апаратного забезпечення, контрольованим умовам мережі та надійним інструментам тестування, представлена конфігурація дозволить підвищити суттєве розуміння поведінки різних баз даних у динамічному REST API на базі Express.js за результатами тестів продуктивності.

Для всебічної оцінки продуктивності різних типів баз даних у динамічному API REST на основі Express.js, методологія тестування охоплює три основні типи робочих навантажень: інтенсивне читання, інтенсивне записування та збалансовані операції. Зазначені робочі навантаження відображають моделі використання в реальному світі та дозволяють ідентифікувати сильні та слабкі сторони кожної БД. З метою аналізу ефективності обробки конкурентних запитів та визначення меж масштабованості буде змодельовано роботу системи під навантаженням від множини одночасних користувачів.

Навантаження з інтенсивним читанням (read-heavy) передбачає великий обсяг операцій пошуку даних порівняно з модифікацією даних. Цей тип робочого навантаження типовий для додатків, що обслуговують вміст, наприклад вебсайтів новин, платформ електронної комерції та інформаційних панелей, де користувачі часто отримують доступ до даних, але рідко їх оновлюють. У контексті REST API на основі Express.js операції з інтенсивним

читанням реалізуються шляхом виконання повторних запитів GET до кінцевих точок, таких як **/items** (отримати всі елементи) та **/items/:id** (отримати певний елемент за ідентифікатором).

У процесі тестування Apache JMeter, Artillery та k6 налаштовані на імітацію кількох клієнтів, які надсилають одночасні запити на читання до цих кінцевих точок. Так, симуляція може включати 500 користувачів, які надсилають запити зі швидкістю 1000 запитів на секунду. Мета полягає в тому, щоб виміряти ключові показники ефективності, такі як:

- затримка: час, необхідний для повернення відповіді на запит на читання;
- пропускна здатність: кількість успішних запитів на читання, оброблених за секунду;
- частота помилок: відсоток невдалих запитів через тайм-аути або перевантаження бази даних.

Також буде представлено механізми кешування за допомогою Redis, щоби спостерігати, як кешування покращує продуктивність читання та зменшує навантаження на базу даних. Буде проаналізовано вплив індексів на ефективність читання в реляційних базах даних і базах даних NewSQL, тоді як бази даних NoSQL, такі як MongoDB і Cassandra, оцінюватимуться на їх здатність швидко отримувати дані на основі документів або широких стовпців.

Навантаження з інтенсивним записуванням (write-heavy) зосереджується на частому вставленні, оновленні та видаленні даних. Ці робочі навантаження поширені в таких проєктах, як системи логування, аналітика в реальному часі та соціальні мережі, де дані постійно генеруються та зберігаються. В API Express.js операції з інтенсивним записом перевіряються шляхом повторного виконання запитів POST для створення нових елементів, запитів PUT для оновлення існуючих елементів і запитів DELETE для видалення елементів.

Для цих тестів інструменти створення навантаження імітують сценарії з одночасними запитами на запис від 500 до 1000 користувачів зі швидкістю до 1500 запитів на секунду. Показники продуктивності, що підлягають моніторингу:

- затримка запису: час, необхідний для виконання операцій вставки, оновлення або видалення;
- пропускна здатність запису: кількість успішно виконаних операцій запису за секунду;
- узгодженість даних: показник того, що всі записи належним чином фіксуються та реплікуються, особливо в розподілених базах даних, таких як CockroachDB і Cassandra;
- стійкість до втрат даних: наскільки добре кожна база даних обробляє збереження даних під час високого навантаження та потенційних збоїв системи.

У реляційних базах даних, таких як MySQL і PostgreSQL, обробка транзакцій і пул з'єднань матимуть вирішальне значення для оптимізації продуктивності запису. У базах даних NoSQL, таких як Cassandra, ключовим фактором буде здатність розподіляти записи між кількома вузлами. Бази даних NewSQL, такі як CockroachDB, будуть оцінюватися на предмет узгодженості розподілених транзакцій під час великих навантажень запису.

Збалансовані операції (balanced operations) передбачають поєднання операцій читання та запису, що є найпоширенішим шаблоном використання для програм загального призначення. Типові приклади включають платформи для колаборації, програми електронної комерції та системи керування клієнтами, де користувачі одночасно читають і оновлюють дані. У цьому сценарії співвідношення читань і записів зазвичай становить близько 70% читань і 30% записів, але також перевірятимуться такі варіації, як 50-50 або 80-20.

Кінцеві точки API Express.js, які використовуються для збалансованих операцій, включають маршрути GET, POST, PUT і DELETE. Інструменти тестування імітують від 500 до 1000 одночасних користувачів, які виконують змішані запити зі швидкістю від 1000 до 2000 запитів на секунду. Показники ефективності включатимуть:

- затримка: середній час, необхідний для відповіді на поєднання запитів на читання та запис;
- пропускна здатність: кількість комбінованих операцій читання та запису, оброблених за секунду;
- паралельна обробка: здатність бази даних керувати одночасним читанням і записом без значного зниження продуктивності;
- використання ресурсів: використання ЦП, пам'яті та диска за збалансованих навантажень для виявлення потенційних вузьких місць.

Для реляційних баз даних ефективність обробки змішаних транзакцій і підтримки індексів буде критичною. Базы даних NoSQL буде оцінено на предмет їх узгодженості та того, наскільки добре вони керують одночасним читанням і записом. Базы даних NewSQL буде проаналізовано на предмет їх здатності підтримувати стійку узгодженість під час розподілу збалансованого навантаження між вузлами.

Важливою для стрес-тестування API та розуміння того, як кожна база даних працює під навантаженням, є симуляція множини одночасних користувачів, яка досягається за допомогою інструментів навантажувального тестування Apache JMeter, Artillery та k6.

Apache JMeter сконфігуровано як групи потоків. Кожна група потоків визначає кількість одночасних користувачів, період нарощування (швидкість додавання користувачів) і тривалість тесту. Так, тестовий сценарій може включати 1000 користувачів, які розгортаються протягом 30 секунд і

підтримують навантаження протягом 5 хвилин. JMeter фіксує детальні показники, включаючи час відповіді, пропускну здатність і рівень помилок.

Сценарії **Artillery** визначають фази з різними рівнями конкурентності та швидкості запитів. Так, тест може початися з 200 користувачів, які роблять 500 запитів на секунду, поступово збільшуючись до 1000 користувачів, які роблять 2000 запитів на секунду. Artillery забезпечує зворотний зв'язок у режимі реального часу щодо затримки, запитів за секунду та частоти помилок.

Скрипти **k6** дозволяють точно контролювати віртуальних користувачів і шаблони завантаження. Сценарії можуть включати постійне навантаження, змінне навантаження та випробування на стрибки. Так, сценарій k6 може імітувати 500 віртуальних користувачів, які зберігають постійне навантаження протягом 10 хвилин, а потім стрибок до 2000 користувачів протягом 1 хвилини. **k6** виводить докладні показники продуктивності, включаючи час відповіді, частоту запитів і використання ресурсів.

Зазначені інструменти забезпечують можливість комплексного тестування продуктивності в контрольованих умовах, шляхом імітації реалістичних сценаріїв користувацької поведінки та дослідження функціонування кожної бази даних при різних ступенях конкурентності. Проведені за цією методологією випробування дозволять отримати кількісні показники затримки, пропускну здатність, коефіцієнт помилок та споживання ресурсів, що сприятиме обґрунтованому вибору оптимальної бази даних для різних типів робочих навантажень у REST API, реалізованому на базі Express.js.

2.4. Підготовка середовища

Узгоджене та контрольоване середовище має критичне значення для точної оцінки ефективності різних типів баз даних у динамічному REST API на основі Express.js. Для забезпечення неупередженого тестування продуктивності, кодова база для кожної реалізації бази даних дотримується узгодженої структури та шаблону дизайну. Кінцеві точки, маршрути та логіка

API реалізуються однаково, незалежно від базового типу бази даних. Кожна реалізація операцій CRUD дотримується тих самих умов іменування, форматів вхідних/вихідних даних і механізмів обробки помилок.

Повна структура проекту представлена в Додатку А і має такий вигляд:

1. Кореневі файли (**/root**):

- `index.js`: головна точка входу програми, де налаштовано та запущено сервер `Express.js`.
- `package.json`: містить список залежностей проекту та сценаріїв для запуску програми.
- `package-lock.json`: файл блокування для забезпечення узгоджених версій залежностей.

2. Каталог **/db**: містить конфігураційні файли для підключення до кожного типу бази даних. Кожен файл експортує з'єднання або пул з'єднань, що використовуються обробниками маршрутів.

3. Каталог **/routes**: містить модулі маршрутизації для операцій CRUD, що відповідають кожному типу бази даних. Кожен файл обробляє кінцеві точки API для певної бази даних.

4. Каталог **/middleware**:

- `logger.js`: проміжне програмне забезпечення для реєстрації вхідних запитів.
- `cache.js`: проміжне програмне забезпечення для реалізації стратегій кешування за допомогою `Redis`.
- `validator.js`: проміжне програмне забезпечення для перевірки введення за допомогою бібліотек, таких як `express-validator`.

5. Каталог **/scripts**: містить сценарії заповнення бази даних для заповнення кожної бази даних ідентичними наборами даних перед тестуванням.

6. Каталог **/tests**: містить тестові сценарії для оцінки інтенсивного читання, запису та збалансованого робочого навантаження. Додається файл звіту для документування результатів ефективності.

7. Каталог **/config**: зберігає конфігураційні файли для керування такими параметрами, як деталі підключення до бази даних, порти сервера та тривалість тайм-ауту.

8. Каталог **/db-schemas**: містить визначення схем для даних PostgreSQL, MySQL, CockroachDB і Cassandra.

Маршрути Express.js для кожного типу бази даних (MySQL, PostgreSQL, MongoDB, Cassandra, Redis і CockroachDB) організовані в окремих модулях у каталозі маршрутів. Кожен модуль містить ідентичні кінцеві точки для операцій GET, POST, PUT і DELETE з узгодженою обробкою запитів, перевіркою параметрів і форматуванням відповіді. Проміжне програмне забезпечення для журналювання, перевірки й обробки помилок однаково застосовується в усіх реалізаціях.

Файли конфігурації бази даних зберігаються в каталозі **db**, і кожен файл експортує екземпляр підключення або пул. Правила іменування змінних підключення, функцій і обробників маршрутів залишаються незмінними. Така уніфікованість гарантує, що відмінності в продуктивності можна віднести до самих систем баз даних, а не до варіацій у структурі коду чи логіці.

Для забезпечення коректного та значущого порівняння продуктивності всі досліджувані бази даних попередньо заповнюються ідентичними наборами даних. Структура набору даних включає елементи з полями для ідентифікатора, імені та опису.

Генерація вихідних даних здійснюється програмним шляхом за допомогою спеціально розробленого скрипту. Зокрема, генерується набір даних обсягом 100 000 записів з рандомізованими, але детерміновано відтворюваними значеннями для полів імені та опису. Згенерований набір даних імпортується в кожну базу даних за допомогою скриптів, адаптованих до специфічних методів імпорту відповідної СКБД.

Для MySQL, PostgreSQL та CockroachDB використовуються пакетні оператори SQL INSERT для одночасного імпорту декількох записів. Застосування транзакцій гарантує атомарність та узгодженість процесу заповнення. У випадку MongoDB імпорт даних здійснюється в форматі JSON-документів з використанням методу **insertMany()**, наданого бібліотекою Mongoose. Для Cassandra імпорт даних реалізується за допомогою команд CQL INSERT, оптимізованих для ефективного розподілу даних між вузлами кластера шляхом врахування ключів розділів. У Redis імпорт даних відбувається шляхом додавання пар ключ-значення, що репрезентують кожен елемент, з використанням єдиного формату ключів (наприклад, **item:{id}**).

Сценарії заповнення виконуються перед кожним тестом продуктивності з метою відновлення початкового стану бази даних та забезпечення ідентичності вихідних даних для всіх тестів. Застосований підхід мінімізує вплив варіативності, зумовленої модифікацією даних під час попередніх тестів, та забезпечує відтворюваність результатів.

Пул підключень має вирішальне значення для оптимізації продуктивності бази даних і ефективного керування одночасними запитами. Щоб забезпечити узгодженість у різних реалізаціях баз даних, розміри пулів підключень і параметри часу очікування стандартизовані. Такі параметри застосовуються однаково:

1. Розмір пулу підключень. Розмір пулу з 20 підключень використовується для всіх баз даних, щоб збалансувати використання ресурсів і конкурентність. Цієї кількості достатньо для обробки великого рівня одночасних запитів, не перевантажуючи сервер бази даних.
2. Час простою. Неактивні з'єднання закриваються через 30 секунд, щоб звільнити ресурси, зберігаючи доступність для нових запитів. Цей параметр запобігає нескінченному утриманню невикористаних з'єднань у пулі.

3. Час очікування. Для отримання з'єднання з пулу встановлено час очікування 5 секунд. Якщо з'єднання не вдається встановити протягом цього періоду, запит не виконується, дозволяючи програмі обробити невдале з'єднання та надати відповідний фідбек.

Для MySQL, PostgreSQL і CockroachDB, пул підключень налаштовується за допомогою бібліотек **pg** або **mysql2**. Ці бібліотеки забезпечують вбудовану підтримку для керування пулами з'єднань. У MongoDB пулом підключень керує **mongoose**, в якому ця підтримка присутня за замовчуванням. Розмір пулу та параметри часу очікування вказуються в параметрах підключення. Для Cassandra драйвер **cassandra** підтримує пул підключень між кількома вузлами в кластері. Драйвер налаштовано з однаковим розміром пулу та параметрами часу очікування. У Redis пул підключень керується за допомогою клієнта **redis**, який підтримує налаштувану кількість одночасних з'єднань.

Уніфікованість коду в різних реалізаціях баз даних, заповнення баз даних ідентичними наборами даних та стандартизація розмірів пулів з'єднань і параметрів тайм-ауту створюють послідовну та контрольовану структуру для тестування продуктивності. Цей підхід мінімізує вплив зовнішніх змінних та гарантує, що відмінності в продуктивності зумовлені характеристиками самих баз даних, а не варіаціями коду чи конфігурації [17, 28].

Висновки до розділу 2

У цьому розділі було детально розглянуто практичну реалізацію динамічного REST API на базі фреймворка Express.js із використанням різних типів баз даних. Спершу були визначені критерії вибору баз даних, що дозволило включити до аналізу реляційні бази (MySQL, PostgreSQL), нереляційні (MongoDB, Cassandra), бази даних у пам'яті (Redis) та NewSQL рішення (CockroachDB). Цей вибір забезпечив умови для всебічного тестування продуктивності з урахуванням архітектури баз даних, складності запитів, масштабованості та можливостей інтеграції з Express.js.

Для кожного типу бази даних було описано інтеграцію з Express.js з дотриманням єдиної структури коду, узгоджених визначень кінцевих точок і стандартизованих конфігурацій. Реалізація операцій CRUD для всіх баз даних слідувала однаковим шаблонам і конвенціям, що забезпечило уніфікований підхід до створення, читання, оновлення та видалення даних. Проміжне програмне забезпечення для логування, валідації вхідних даних і кешування застосовувалося послідовно для підтримання якості й надійності роботи API. Бази даних були заповнені ідентичними наборами даних для мінімізації упереджень у тестуванні, а конфігурації пулів з'єднань було стандартизовано для оптимізації управління ресурсами та обробки паралельних запитів.

Описано специфікації обладнання, контрольовані умови мережі та стратегії навантажувального тестування для різних сценаріїв: операцій, орієнтованих на інтенсивне читання, інтенсивний запис і збалансовані навантаження. Для імітації великої кількості одночасних користувачів було використано інструменти, такі як Apache JMeter, Artillery та k6. Це дозволить отримати релевантні показники продуктивності за різних умов експлуатації. Комплексне налаштування середовища має на меті забезпечити зібрання значущих даних для оцінки переваг і недоліків кожного типу баз даних у динамічних REST API на базі Express.js.

РОЗДІЛ 3

ОЦІНКА ПРОДУКТИВНОСТІ ТА АНАЛІЗ

3.1. Проведення базового тестування

На першому етапі тестування продуктивності було проведено для кожної бази даних без включення Redis як рівня кешування. Такі показники, як час виконання запиту, пропускна здатність, затримка та використання ресурсів, були проаналізовані та візуалізовані за допомогою графіків і таблиць. Спочатку продуктивність кожної бази даних оцінювалася за сценаріями інтенсивного читання (read-heavy), інтенсивного запису (write-heavy) та збалансованої роботи без будь-якого механізму кешування. Метою цього є встановлення базової лінії для порівняння за трьох типів навантажень. Тести включали реалізації MySQL, PostgreSQL, MongoDB, Cassandra та CockroachDB з ідентичними наборами даних та єдиною логікою API.

Інтенсивне читання (80% операцій читання, 20% операцій запису). Для цього типу навантаження було змодельовано сценарії, де більшість операцій включає отримання даних. Це загальний шаблон для багатьох додатків, таких як контент-платформи, каталоги продуктів електронної комерції та канали соціальних мереж.

У табл. 3.1. наведено детальний огляд показників продуктивності, включаючи пропускну здатність, середню затримку і частоту помилок для кожної бази даних.

Таблиця 3.1 – Базове тестування БД при інтенсивному читанні

База даних	Пропускна здатність (запитів у секунду)	Середня затримка	Рівень помилки
PostgreSQL	830	12 мс	0,1%
MySQL	780	15 мс	0,2%
MongoDB	720	18 мс	0,3%
Cassandra	700	20 мс	0,4%
CockroachDB	680	25 мс	0,5%

Як показали результати, PostgreSQL досягає найвищої пропускну здатності (830 запитів у секунду) із найменшою затримкою (12 мс) і

мінімальним рівнем помилок (0,1%). Це свідчить про його ефективність у обробці великого обсягу запитів на читання з мінімальною затримкою. У той час як інші бази даних пропонують достатню пропускну здатність, їх вищі затримки та дещо підвищений рівень помилок вказують на потенційні вузькі місця або компроміси в їхніх архітектурах під час роботи з навантаженнями, пов'язаними виключно з інтенсивним читанням. MongoDB і Cassandra пропонують прийнятні показники, але слід уважно розглянути стратегії індексування та вимоги узгодженості для оптимізації продуктивності читання. CockroachDB поступається за показниками, оскільки більше підходить для сценаріїв, де розподілені транзакції та стійка узгодженість є абсолютною необхідністю, навіть за рахунок швидкості читання.

Інтенсивний запис (20% операцій читання, 80% операцій запису). Для цього типу навантаження проводилося моделювання сценаріїв, де основні операції передбачають вставлення, оновлення або видалення даних, що актуально для додатків, які генерують великі обсяги даних, такі як логування в реальному часі, прийом даних датчиків або високочастотні торгові платформи. Результати тестування для інтенсивного запису наведені в табл. 3.2.

Таблиця 3.2 - Базове тестування БД при інтенсивному записі

База даних	Пропускна здатність (запитів у секунду)	Середня затримка	Рівень помилок
Cassandra	950	10 мс	0,1%
MongoDB	800	12 мс	0,2%
PostgreSQL	650	15 мс	0,3%
MySQL	600	17 мс	0,3%
CockroachDB	550	20 мс	0,5%

У цьому випадку, згідно з результатами, Cassandra досягає найвищої пропускну здатності (950 запитів у секунду) з найнижчою затримкою (10 мс) і мінімальним рівнем помилок (0,1%), демонструючи свою придатність для додатків з інтенсивним записом. MongoDB демонструє наступні за

продуктивністю показники, які також свідчать про її придатність для систем з інтенсивним записом.

Реляційні бази даних (PostgreSQL і MySQL) пропонують достатню продуктивність, але з дещо вищими затримками, змушуючи прийняти компроміс між узгодженістю та продуктивністю необробленого запису. Це очікувано, оскільки вони надають пріоритет узгодженості даних (властивості ACID) і реляційній цілісності, що створює накладні витрати під час операцій запису. Управління транзакціями, механізми блокування та перевірка даних сприяють цій різниці в продуктивності.

CockroachDB має найнижчу пропускну здатність і найвищу затримку в цьому сценарії з інтенсивним записом: накладні витрати на керування розподіленими транзакціями та забезпечення узгодженості даних на кількох вузлах значно впливають на продуктивність запису в цьому сценарії. Рівень помилок у всіх базах даних залишається відносно низьким, що свідчить про стабільну роботу під час тестування.

Збалансоване навантаження (50% операцій читання, 50% операцій запису). Для цього типу навантаження проводилося моделювання сценаріїв, де операції читання та запису відбуваються приблизно в рівних пропорціях. Це змішане робоче навантаження забезпечує більш повну оцінку продуктивності бази даних порівняно зі сценаріями, пов'язаними виключно з інтенсивним читанням або записом. Результати тестування БД при здійсненні збалансованого навантаження наведені в табл. 3.3.

Таблиця 3.3 - Базове тестування БД при збалансованому навантаженні

База даних	Пропускна здатність (запитів у секунду)	Середня затримка	Рівень помилок
PostgreSQL	700	14 мс	0,2%
MySQL	680	16 мс	0,2%
MongoDB	640	18 мс	0,5%
Cassandra	620	18 мс	0,3%
CockroachDB	580	22 мс	0,8%

Згідно з результатами, PostgreSQL і MySQL зберігають достатній баланс між пропускнуою здатністю, затримкою та частотою помилок, що підкреслює їх придатність для програм із змішаними шаблонами читання й запису. MongoDB і Cassandra пропонують прийнятну пропускну здатність, але з дещо вищими затримками та/або частотою помилок, що вказує на компроміси в їхніх архітектурах під час обробки збалансованих робочих навантажень. CockroachDB демонструє найнижчу пропускну здатність, найвищу затримку та найвищий рівень помилок – отже і в цьому випадку її архітектура менш оптимізована для сценаріїв із великою кількістю одночасних операцій читання та запису.

Оцінка продуктивності чітко висвітлила переваги та недоліки розглянутих систем керування базами даних (СКБД). Реляційні СКБД, що спираються на десятиліття розвитку та суворе дотримання принципів ACID (атомність, узгодженість, ізоляція, довговічність), демонструють ефективність у робочих навантаженнях, орієнтованих переважно на читання, а також у випадках збалансованого поєднання операцій читання й запису.

У той же час бази даних NoSQL, зокрема MongoDB і Cassandra, підтверджують перевагу в умовах, де переважають операції запису. Їх висока продуктивність у середовищах із інтенсивним записом є результатом особливостей архітектури, включаючи схему без строгих структурних обмежень. Крім того, властива їм горизонтальна масштабованість дозволяє ефективно розподіляти навантаження запису між кількома вузлами, підвищуючи загальну продуктивність.

CockroachDB, натомість, пропонує цінні гарантії узгодженості в умовах розподіленої архітектури, однак стикається з труднощами у підтримці низької затримки під час високого навантаження. Це зумовлено складною координацією між вузлами, необхідною для забезпечення цілісності даних у розподіленій системі. Механізми, що забезпечують високу узгодженість і надійність даних у CockroachDB, створюють додаткові накладні витрати, які проявляються у вигляді підвищеної затримки за інтенсивного трафіку.

3.2. Тестування з інтеграцією Redis як рівня кешування

Для проведення наступної фази тестування в розроблений REST API на базі Express.js було інтегровано Redis, щоб імітувати його роль рівня кешування. Redis було налаштовано на зберігання даних, до яких часто звертаються, зменшуючи навантаження на основні бази даних. Випробування повторювалося в ідентичних умовах, щоб спостерігати зміни продуктивності.

Очікується, що інтеграція Redis як рівня кешування в RESTful API на основі Express.js значно вплине на продуктивність за рахунок зменшення навантаження на базу даних і покращення часу відповіді. Наступні тести були розроблені для оцінки покращень продуктивності, запроваджених Redis у різних конфігураціях баз даних (PostgreSQL, MySQL, MongoDB, Cassandra та CockroachDB). Redis було реалізовано як рівень кешування для оптимізації продуктивності важких навантажень читання з метою зменшення середньої затримки для даних, до яких часто звертаються.

Інтенсивне читання (80% операцій читання, 20% операцій запису). У цьому сценарії Redis використовувався для кешування результатів найчастіших запитів на читання, таких як інформація про продукт або профілі користувачів. Представлені у табл. 3.4 показники продуктивності порівнюють затримку, пропускну здатність і частоту помилок до та після інтеграції рівня кешування Redis.

Таблиця 3.4 – Тестування БД з інтеграцією Redis при інтенсивному читанні

База даних	Пропускна здатність (запитів у секунду)	Середня затримка	Рівень помилок
PostgreSQL	860	6 мс	0.1%
MySQL	800	8 мс	0.1%
MongoDB	740	9 мс	0.2%
Cassandra	720	12 мс	0.3%
CockroachDB	700	15 мс	0.4%

За результатами аналізу, PostgreSQL є лідером за всіма трьома показниками: найвища пропускна здатність, найнижча затримка та найнижчий

рівень помилок. MySQL посідає друге місце, демонструючи також дуже хороші результати за пропускну здатністю та затримкою, з таким же низьким рівнем помилок, як і PostgreSQL.

Інтенсивний запис (20% операцій читання, 80% операцій запису). У сценарії інтенсивного запису ефект Redis менш виражений з точки зору зменшення затримки, оскільки операції запису за своєю суттю обходять механізми кешування. Однак Redis все ще може бути корисним для оптимізації пропускну здатності запису, розвантажуючи важкі для читання запити та запобігаючи непотрібним читанням бази даних під час операцій запису.

У табл. 3.5 порівнюються показники продуктивності для інтенсивних операцій запису після інтеграції Redis.

Таблиця 3.5 – Тестування БД з інтеграцією Redis при інтенсивному записі

База даних	Пропускна здатність (запитів у секунду)	Середня затримка	Рівень помилок
Cassandra	950	10 мс	0.1%
MongoDB	830	12 мс	0.2%
PostgreSQL	700	16 мс	0.3%
MySQL	650	18 мс	0.3%
CockroachDB	600	22 мс	0.5%

На прикладі пропускну здатності дається взнаки ефекти інтеграції Redis. Так, Cassandra, яка зазвичай добре працює під час інтенсивного запису, зберігає високу пропускну здатність (950 запитів у секунду) із Redis. Це свідчить про те, що кеш Redis істотно не вплинув на продуктивність інтенсивних операцій запису. MongoDB також продемонструвала незначне покращення пропускну здатності, з 800 до 830 запитів у секунду. Реляційні бази даних (PostgreSQL і MySQL) продемонстрували незначне покращення пропускну здатності (від 600–650 до 650–700 запитів у секунду), оскільки ефект кешування Redis був більш очевидним у зменшенні навантаження на базу даних під час наступних операцій читання.

Збалансоване навантаження (50% операцій читання, 50% операцій запису) - передбачає комбінацію операцій читання та запису, які відображають реалістичні моделі використання в більшості RESTful API. Інтеграція кешування Redis у таких сценаріях дозволяє швидко обслуговувати дані, до яких часто звертаються, із кешу, одночасно зменшуючи навантаження на базову базу даних. Як показано в табл. 3.6, наскільки інтеграція Redis зменшила вплив частих читань бази даних, підвищивши загальну продуктивність кінцевих точок, що обробляють комбінацію запитів GET і POST.

Таблиця 3.6 - Тестування БД з інтеграцією Redis при збалансованому навантаженні

База даних	Пропускна здатність (запитів у секунду)	Середня затримка	Рівень помилок
PostgreSQL	740	10 мс	0.2%
MySQL	720	12 мс	0.2%
MongoDB	680	15 мс	0.3%
Cassandra	660	14 мс	0.3%
CockroachDB	640	16 мс	0.4%

Як видно з таблиці, PostgreSQL продемонструвала стабільну продуктивність, досягнувши пропускну здатності 740 запитів на секунду із середньою затримкою 10 мс. Низький рівень помилок у 0,2% вказує на те, що Redis ефективно обробляв часті операції читання, запобігаючи надмірним запитам до бази даних і забезпечуючи стабільну продуктивність. MySQL має наступні за результатами показники, з пропускну здатністю 720 запитів на секунду та середньою затримкою 12 мс. Рівень помилок був таким же низьким — 0,2%, що підкреслює, як Redis пом'якшував суперечності під час операцій читання, особливо для кінцевих точок, до яких часто надсилали запити.

Для MongoDB пропускна здатність була трохи нижчою — 680 запитів на секунду із середньою затримкою 15 мс. Рівень помилок у 0,3 % свідчить про те, що хоча інтеграція Redis зменшила навантаження на базу даних, власна структура MongoDB могла дещо обмежити повний ступінь підвищення

продуктивності порівняно з реляційними базами даних. Cassandra показала результати, порівняні з MongoDB із пропускнуою здатністю 660 запитів на секунду та середньою затримкою 14 мс. Коефіцієнт помилок у 0,3% свідчить про те, що Redis ефективно підтримував свою продуктивність у обробці змішаних робочих навантажень, зокрема шляхом кешування часто використовуваних пар ключ-значення.

CockroachDB, який спочатку мав високі затримки, продемонстрував найзначніше покращення, порівняно з тестуванням без інтеграції Redis. Його пропускна здатність досягла 640 запитів на секунду, а затримка скоротилася до 16 мс. Рівень помилок, який знизився до 0,4%, вказує на те, що інтеграція Redis усунула деякі вузькі місця в процесах виконання запитів, зокрема шляхом розвантаження важких операцій читання.

Загалом результати підтверджують, що інтеграція Redis є особливо ефективною для зменшення затримки та частоти помилок для збалансованих робочих навантажень у всіх типах баз даних. У той час як реляційні бази даних, такі як PostgreSQL і MySQL, отримали переваги від оптимізованої обробки запитів, бази даних NoSQL, такі як MongoDB і Cassandra, отримали суттєве підвищення продуктивності при обслуговуванні динамічних даних.

3.3. Тестування масштабованості й відмовостійкості RESTful-системи

Масштабованість і відмовостійкість є критично важливими параметрами для оцінки продуктивності REST API на основі Express.js за різних навантажень. Масштабованість вимірює, наскільки добре система адаптується до збільшення кількості одночасних користувачів і операцій, тоді як відмовостійкість зосереджується на здатності API підтримувати стабільну продуктивність і обробляти помилки під час стресу. У табл. 3.7 представлені результати оцінки розробленої системи за цими аспектами за допомогою контрольованих навантажувальних тестів для різних типів баз даних і конфігурацій, включаючи інтеграцію Redis.

Масштабованість системи оцінювалась шляхом поступового збільшення кількості одночасних користувачів і спостереження за пропускнуою здатністю та затримкою API. Випробування проводилися для кожного типу бази даних за трьома основними сценаріями: базова продуктивність бази даних, продуктивність, інтегрована з Redis, і змодельоване масштабування шляхом додавання реплік бази даних або коригування пулу з'єднань.

Тестування відмовостійкості оцінювало стійкість системи в умовах тривалого стресу, сценаріїв збоїв і несподіваних стрибків трафіку. Було змодельовано різні режими збоїв, у тому числі випадкові відключення бази даних, високі суперечки щодо запису та порушення обмеження швидкості API. У табл. 3.7 представлено узагальнені результати оцінки масштабованості та надійності для кожного типу бази даних, включаючи їхню поведінку в базових і інтегрованих конфігураціях Redis.

Таблиця 3.7 – Тестування масштабованості та відмовостійкості системи

Тип бази даних	Макс. кількість одночасних користувачів (Базовий рівень)	Макс. кількість одночасних користувачів (3 Redis)	Масштабованість	Надійність
PostgreSQL	3500	5000	Лінійна масштабованість до 5 000 користувачів з Redis. Мінімальне зростання затримки при великих навантаженнях на запис.	Висока надійність. Добре витримує тривалі навантаження, швидке відновлення після збоїв.
MySQL	3000	4500	Добре масштабується, але демонструє невелике зростання затримки при високій конкуренції за запис понад 4 500 користувачів.	Надійна, хоча відновлення після проблем з конкуренцією за запис повільніше, ніж у PostgreSQL.

Продовження таблиці 3.7

MongoDB	4500	6000	Сильна масштабованість при навантаженнях на читання. Обмежена конкуренцією за запис при високій кількості одночасних користувачів.	Надійна для навантажень на читання. Іноді трапляються збої операцій при навантаженнях на запис.
Cassandra	5000	7000	Майже лінійна масштабованість, особливо для навантажень на запис. Відмінна обробка розподілених операцій.	Висока надійність, навіть під час збоїв вузлів, завдяки механізмам реплікації.
CockroachDB	3000	4000	Ефективна масштабованість з Redis та горизонтальним масштабуванням, хоча завдання з великим навантаженням на запис створюють додаткові витрати.	Надійна під час збоїв, але демонструє збільшення затримки при понад 5 000 одночасних користувачів.

Результати показали відмінну поведінку масштабування для кожної бази даних. Реляційні бази даних (PostgreSQL і MySQL) показали лінійну масштабованість, коли розміри пулу з'єднань були оптимізовані. PostgreSQL обробляла до 5000 одночасних користувачів без помітного збільшення затримки, тоді як MySQL продемонструвала незначне зростання затримки понад 4500 користувачів завдяки своїм механізмам блокування під час інтенсивних операцій запису.

Бази даних NoSQL (MongoDB і Cassandra) продемонстрували високу масштабованість під час важких навантажень читання. MongoDB показала стабільну продуктивність до 6000 одночасних користувачів, вигравши від кешування Redis і безсхемного дизайну. Cassandra продемонструвала майже

лінійну масштабованість, особливо в сценаріях з інтенсивним записом, з максимальною продуктивністю в 7000 користувачів до початку деградації затримки.

У випадку бази даних NewSQL (CockroachDB) покращення масштабованості були помітні завдяки інтеграції Redis і горизонтальному масштабуванню. CockroachDB ефективно масштабувався до 4000 користувачів, але після цієї точки затримка зростає, що вказує на те, що обробка розподілених транзакцій створює накладні витрати при високому рівні паралелізму.

Щодо відмовостійкості, то PostgreSQL незмінно демонструвала високу надійність, зберігаючи стабільну продуктивність під час тривалого стресу та швидко відновлюючись після змодельованих збоїв завдяки надійному управлінню транзакціями. Відмовостійкість MySQL також була високою, але показала довший час відновлення під час сценаріїв із високим рівнем конкуренції на запис.

MongoDB продемонструвала стійкість під час великих навантажень читання, але надійність дещо знизилася під час інтенсивних стрес-тестів запису, де час від часу виникали збої в роботі. Cassandra, завдяки своїй розподіленій природі, виявилася високонадійною, продемонструвавши мінімальне зниження продуктивності та майже нульову втрату даних під час збоїв вузлів завдяки стратегіям реплікації.

База даних CockroachDB продемонструвала високу надійність під час моделювання несправностей, використовуючи свій розподілений дизайн для підтримки доступності. Однак її затримка значно зростає під час інтенсивних стрес-тестів із понад 5000 одночасними користувачами, що вказує на області для оптимізації координації транзакцій.

Таким чином, тестування показало, що масштабованість і надійність сильно залежать від архітектури та конфігурації бази даних. У той час як реляційні бази даних працювали краще при змішаних навантаженнях, бази даних NoSQL і NewSQL впорались краще з великомасштабними

розподіленими операціями. Кешування Redis постійно ставало критичним фактором у пом'якшенні вузьких місць у продуктивності та забезпеченні стабільності системи.

Отримані результати підкреслюють важливість вибору типу бази даних і конфігурації, яка відповідає очікуваним шаблонам робочого навантаження та вимогам надійності API. Коректно налаштований пул з'єднань, реплікація бази даних і стратегії кешування відіграють ключову роль у досягненні масштабованості й відмовостійкості в динамічних системах REST API.

3.4. Інтерпретація результатів

Розбіжності в продуктивності, які були зафіксовані під час дослідження, спричинені властивими архітектурними відмінностями між базами даних, що підлягали тестуванню. Реляційні бази даних, такі як PostgreSQL і MySQL, значною мірою покладаються на структуровані схеми та відповідність ACID, що забезпечує цілісність даних, але створює затримку під час великої конкуренції за запис. Це пояснює їхню порівняно повільну масштабованість під час інтенсивних операцій запису. Однак PostgreSQL продемонстрував високу продуктивність у цій категорії завдяки розширеним можливостям індексування та паралельної обробки.

У свою чергу, бази даних NoSQL, такі як MongoDB і Cassandra, відмінно справлялися з робочими навантаженнями з високим рівнем паралелізму, особливо в сценаріях, що фокусуються на операціях з інтенсивним читанням або записом. Структура MongoDB дозволяла гнучке отримання даних, хоча її продуктивність відставала в сценаріях із високим рівнем конкуренції щодо запису через відсутність внутрішньої підтримки транзакцій. Розподілений, оптимізований для запису дизайн Cassandra забезпечував майже лінійну масштабованість, що робить її доцільною для додатків, які потребують низької затримки запису та високої відмовостійкості.

База даних CockroachDB поєднує переваги реляційних баз даних із масштабованістю систем NoSQL, однак її додаткові накладні витрати від

забезпечення відповідності розподіленого ACID обмежили її продуктивність за високого паралелізму. Цей компроміс зробив CockroachDB придатним для збалансованих робочих навантажень, але менш конкурентоспроможним у середовищах зі значно нерівномірним розподілом операцій читання та запису.

Інтеграція Redis значно пом'якшила ці розбіжності, розвантаживши операції читання через механізми кешування. Зменшення навантаження на запити до бази даних, особливо для PostgreSQL і MySQL, продемонструвало, як кешування може подолати прогалини в продуктивності реляційних систем, роблячи їх життєздатними варіантами навіть у сценаріях високого паралелізму.

З точки зору розробки API, ці результати підкреслюють критичну роль архітектури бази даних у формуванні продуктивності програми. Реляційні бази даних залишаються надійним вибором для API, які суворо вимагають цілісності даних і складних запитів, таких як фінансові системи або корпоративні програми. Однак розробники повинні впроваджувати стратегії кешування, як продемонструвала інтеграція Redis, щоб покращити швидкість реагування та масштабованість.

Бази даних NoSQL добре узгоджуються з API, націленими на додатки реального часу, такі як платформи соціальних мереж або системи Інтернету речей, завдяки їхній здатності обробляти великі обсяги неструктурованих даних і підтримувати сценарії високого паралелізму. Тим не менш, відсутність надійної підтримки транзакцій вимагає ретельного розгляду вимог узгодженості даних під час розробки схеми. Відмовостійкість і розподілена архітектура Cassandra роблять її доречною для територіально розосереджених систем, де доступність даних і записування з низькою затримкою є пріоритетними.

Бази даних NewSQL є збалансованою альтернативою, особливо для сучасних програм, які вимагають глобальної узгодженості та масштабованості без шкоди для цілісності транзакцій. Однак їх накладні витрати на продуктивність під час стресу підкреслюють необхідність для розробників

оптимізувати шляхи виконання запитів і розглянути коригування пулу з'єднань, щоб мінімізувати затримку.

Зрештою, результати підкреслюють важливість узгодження вибору бази даних з характеристиками робочого навантаження програми. Інтерфейси API, які потребують читання, виграють від кешування та масштабованих рішень NoSQL, тоді як API зі складною транзакційною логікою потребують структурованих можливостей реляційних баз даних або баз даних NewSQL. Збалансування цих міркувань гарантує, що база даних служить фактором продуктивності, а не вузьким місцем, безпосередньо впливаючи на здатність API відповідати вимогам сучасних додатків.

Висновки до розділу 3

У розділі було надано комплексну оцінку продуктивності баз даних у контексті REST API на базі Express.js з зосередженням на впливі різних архітектур та конфігурацій баз даних. Базові тести продуктивності виявили критичні метрики для кожного типу бази даних за різних сценаріїв навантаження, підкреслюючи, як операції з великою кількістю читань, записів та збалансовані операції взаємодіють з системними обмеженнями, такими як час виконання запитів та обробка одночасних користувачів. Цей аналіз заклав основу для розуміння компромісів, властивих реляційним, нереляційним та NewSQL базам даних при їх застосуванні для розробки API.

Продемонстровано вплив кешування Redis на продуктивність та масштабованість бази даних. Завдяки розвантаженню операцій з великою кількістю читань, інтеграція Redis значно підвищила швидкість реагування та пропускну здатність для всіх типів баз даних. Покращення продуктивності були особливо помітними для реляційних баз даних PostgreSQL та MySQL. Нереляційні та NewSQL бази даних також показали покращення, хоча їхня властива масштабованість зробила вплив кешування менш трансформаційним. Ці результати підкреслюють важливість механізмів кешування в сучасних архітектурах API, особливо для додатків, що вимагають низької затримки при високій кількості одночасних користувачів.

Проаналізовано масштабованість та відмовостійкість кожного типу бази даних і сформовано уявлення про їхню поведінку в умовах стресових навантажень та збоїв. Нереляційні бази даних, особливо Cassandra, продемонстрували високу масштабованість та відмовостійкість, і, навпаки, реляційні та NewSQL бази даних потребували додаткової оптимізації для підтримки продуктивності в умовах високої кількості одночасних користувачів. Цей розділ підкреслив, що вибір та конфігурація бази даних є ключовими для забезпечення надійної, масштабованої та швидкої продуктивності API.

ВИСНОВКИ

Динамічні REST API стали основою сучасних вебдодатків, а їхня швидкість реагування та масштабованість безпосередньо пов'язані з архітектурою бази даних. Метою цього дослідження було оцінити продуктивність різних баз даних - реляційних, нереляційних і NewSQL - у контексті REST API на основі Express.js та зафіксувати їхні сильні сторони та обмеження за різноманітних навантажень.

В ході дослідження вдалося виявити значні варіації продуктивності різних типів баз даних, сформовані їхніми архітектурними відмінностями та характеристиками робочого навантаження, що відбивається в таких висновках:

1. Реляційні бази даних PostgreSQL і MySQL на високому рівні підтримують цілісність даних і обробляють складні запити, але потребують додаткової оптимізації, наприклад кешування Redis, для ефективного масштабування під час одночасних навантажень.
2. Нереляційні бази даних MongoDB і Cassandra продемонстрували високу масштабованість і керування паралелізмом, завдяки чому вони добре підходять для програм реального часу та розподілених додатків.
3. База даних NewSQL, а саме CockroachDB, запропонувала збалансований підхід шляхом поєднання узгодженості транзакцій із помірною масштабованістю, хоча вона демонструє накладні витрати на продуктивність у сценаріях високого стресу.
4. Кешування Redis покращувало інтенсивне читання та збалансувало робочі навантаження, зменшуючи затримку запитів і підвищуючи пропускну здатність усіх типів баз даних.

Результати підкреслили, що вибір бази даних повинен узгоджуватися з профілем робочого навантаження API. Реляційні бази даних були найбільш ефективними в сценаріях, які вимагали структурованих схем і цілісності транзакцій. Бази даних NoSQL виявилися більш доцільними для програм, які

вимагають високої доступності та масштабованості, тоді як система NewSQL запропонувала компроміс між цими парадигмами. Кешування стало важливою стратегією, особливо для важких робочих навантажень читання, подолання розриву продуктивності в реляційних системах і подальшої оптимізації реалізацій NoSQL і NewSQL. Ці висновки підкреслюють необхідність вибору бази даних залежно від робочого навантаження та оптимізації архітектури для досягнення ефективної продуктивності API.

Це дослідження зіткнулося з кількома обмеженнями, які вплинули на його обсяг і результати. Апаратні обмеження, включаючи використання одновузлової установки для баз даних, обмежували можливість повністю імітувати поведінку розподіленої бази даних за високого рівня паралелізму. Умови мережі були контрольованими та узгодженими, що не повністю відтворює різноманітність розгортань у реальних умовах. Крім того, використовувані набори даних, хоча і репрезентативні, можуть не охоплювати різноманітність і масштаб даних, які трапляються у виробничих середовищах.

Майбутні дослідження можуть розширити цю роботу шляхом включення додаткових БД, таких як графові бази даних або бази даних часових рядів. Тестування додатків у реальних умовах, особливо в середовищах із розподіленим розгортанням, може дати точнішу інформацію про масштабованість і відмовостійкість. Вивчення розширених стратегій кешування, таких як багаторівневе кешування або кешування меж, може ще більше підвищити продуктивність. Крім того, вивчення взаємодії між продуктивністю бази даних і шаблонами розробки API, такими як GraphQL або керованими подіями API, може запропонувати глибше розуміння оптимізації сучасних вебархітектур. Оцінка впливу нових технологій, таких як безсерверні бази даних і оптимізація запитів на основі штучного інтелекту, дасть змогу зрозуміти майбутнє розробки продуктивності API.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Використання Redis в якості засобу для зберігання сесій і кешу в Django. *DevZone*. URL: <https://devzone.org.ua/post/vykorystannia-redis-v-iakosti-zasobu-dlia-zberihannia-sesiy-i-keshu-v-django> (дата звернення: 24.05.2024).
2. Дмитрієнко О., Чміль Ю., Устименко Л. Оцінка інноваційних підходів до управління базами даних в сучасних інформаційних системах. *Наука і техніка сьогодні*. 2024. № 2(30). URL: [https://doi.org/10.52058/2786-6025-2024-2\(30\)-792-804](https://doi.org/10.52058/2786-6025-2024-2(30)-792-804) (дата звернення: 01.07.2024).
3. Козуб В. Технологія підвищення ефективності зберігання у No-SQL базах даних. *Інформаційні технології та суспільство*. 2024. № 3 (14). С. 14–22. URL: <https://doi.org/10.32689/maup.it.2024.3.2> (дата звернення: 18.06.2024).
4. Лукашевич Я., Поліхун А., Дмитрієнко О. Аналіз сучасних розробок ефективних систем управління базами даних. *Наука і техніка сьогодні*. 2024. № 14(28). URL: [https://doi.org/10.52058/2786-6025-2023-14\(28\)-562-577](https://doi.org/10.52058/2786-6025-2023-14(28)-562-577) (дата звернення: 17.06.2024).
5. Ляшенко О. А., Конашков О. О., Солодка Н. О. Порівняльний аналіз виконання запитів до серверів баз даних MySQL і MongoDB. *Вісник Херсонського національного технічного університету*. 2019. Т. 4. С. 114–124. URL: <https://doi.org/10.35546/kntu2078-4481.2019.4.13> (дата звернення: 28.05.2024).
6. Мельничук С. А. Аналіз характеристик продуктивності архітектур реляційних та нереляційних систем керування базами даних : кваліфікаційна робота на здобуття освітнього ступеня магістра : спец. 122 - комп'ютерні науки / наук. кер. В. В. Никитюк. Тернопіль : Тернопільський національний технічний університет імені Івана Пулюя, 2024. 66 с. URL: <https://elartu.tntu.edu.ua/handle/lib/47066> (дата звернення: 29.12.2024).

7. Найкращі методи розробки прагматичного RESTful API. *Data-hints - Дані, аналітика, стандарти та всяке різне*. URL: <https://data-hints.com/naykrashchi-metody-rozrobky-prahmatychnoho-restful-api/> (дата звернення: 01.07.2024).
8. Нікітіна Т. С., Морозова О. І. Порівняльний аналіз продуктивності баз даних SQL та NoSQL. *Системи управління, навігації та зв'язку. Збірник наукових праць*. 2019. Т. 1, № 53. С. 125–128. URL: <https://doi.org/10.26906/sunz.2019.1.125> (дата звернення: 02.06.2024).
9. Основи роботи з фреймворком Express.js. *Foxminded*. URL: <https://foxminded.ua/express-js/> (дата звернення: 01.06.2024).
10. Порівняння СУБД MySQL, PostgreSQL та MS SQL Server. *DataBI*. URL: <https://data-b-i.com/uk/article/porivnyannya-subd-mysql-postgresql-mssqlserver.html> (дата звернення: 05.05.2024).
11. Пранович К. Використання NoSQL бази даних Redis у сучасних додатках. Наукові досягнення – шлях до професії : Тези міжвуз. науково-практ. конф., м. Миколаїв, 22 груд. 2022 р. / ред. А. М. Старєва. Миколаїв, 2022. URL: https://uu.edu.ua/upload/Nauka/Electronni_naukovi_vidannya/2022/zbrnik_shlyah_do_profesii_2022.pdf#page=166 (дата звернення: 05.06.2024).
12. Стешко В. Ю. Дослідження архітектурних рішень та методів оптимізації для підвищення продуктивності додатків на Node.js і Vue.js : пояснювальна записка до кваліфікаційної роботи здобувача вищої освіти на другому (магістерському) рівні. Харків, 2023. 75 с.
13. Що таке PostgreSQL і для чого використовується? *FoxmindEd*. URL: <https://foxminded.ua/oplata-obucheniya-1/> (дата звернення: 05.05.2024).
14. Ярцев В. П. Розподілені бази даних. Київ : ДУТ, 2018. 97 с. URL: https://duikt.edu.ua/uploads/1_1754_30359302.pdf (дата звернення: 26.07.2024).1

15. Advanced query optimization in SQL databases for real-time big data analytics / M. M. Rahman et al. *Academic journal on business administration, innovation & sustainability*. 2024. Vol. 4, no. 3. P. 1–14.
16. Bachina B. Efficient RESTful API Development with NodeJS, Express, and PostgreSQL. *European Journal of Advances in Engineering and Technology*. 2023. Vol. 10, no. 6. P. 26–37. URL: <https://ejaet.com/PDF/10-6/EJAET-10-6-26-37.pdf> (date of access: 04.05.2024).
17. Behera T. K. Architecture Principles for Enterprise Software and Mobile Application Development. *Advances in Wireless Technologies and Telecommunication*. 2023. P. 1–20. URL: <https://doi.org/10.4018/978-1-6684-8582-8.ch001> (date of access: 17.06.2024).
18. Bojinov V. RESTful Web API Design with Node.js 10: Learn to create robust RESTful web services with Node.js, MongoDB, and Express.js, 3rd Edition. Packt Publishing, 2018. 178 p.
19. Brett J. Getting Started with hapi.js. Birmingham : Packt Publishing Ltd., 2016. 137 p.
20. CockroachDB – the cloud native, distributed SQL database designed for high availability, effortless scale, and control over data placement. *GitHub*. URL: <https://github.com/cockroachdb/cockroach> (date of access: 01.06.2024).
21. Database NewSQL Performance Evaluation for Big Data in the Public Cloud / M. Murazzo et al. *Communications in Computer and Information Science*. Cham, 2019. P. 110–121. URL: https://doi.org/10.1007/978-3-030-27713-0_10 (date of access: 21.06.2024).
22. Diogo M., Cabral B., Bernardino J. Consistency Models of NoSQL Databases. *Future Internet*. 2019. Vol. 11, no. 2. P. 43. URL: <https://doi.org/10.3390/fi11020043> (date of access: 16.06.2024).
23. Express - Node.js web application framework. URL: <https://expressjs.com/> (date of access: 01.05.2024).
24. Fielding R. T. Architectural Styles and the Design of Network-based Software Architectures : doctoral dissertation. Irvine, 2000.

25. Gowda P., Gowda AN. Best Practices in REST API Design for Enhanced Scalability and Security. *Journal of Artificial Intelligence, Machine Learning and Data Science*. 2024. Vol. 2, no. 1. P. 827–830. URL: <https://urfjournals.org/open-access/best-practices-in-rest-api-design-for-enhanced-scalability-and-security.pdf> (date of access: 01.08.2024).
26. Hewitt E. *Cassandra: The Definitive Guide*. Sebastopol, CA : O'Reilly Media, Incorporated, 2022. 92 p.
27. How to implement Redis Cache in Spring Boot Application?, 2024. Blogs - JavaTechOnline. URL: <https://javatechonline.com/wp-content/uploads/2021/08/Redis-Cache-1-600x463.jpg> (date of access: 28.05.2024).
28. Khasawneh T. N., AL-Sahlee M. H., Safia A. A. SQL, NewSQL, and NOSQL Databases: A Comparative Survey. *2020 11th International Conference on Information and Communication Systems (ICICS)*, Irbid, Jordan, 7–9 April 2020. 2020. URL: <https://doi.org/10.1109/icics49469.2020.239513> (date of access: 18.05.2024).
29. Kleppmann M. *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. O'Reilly Media, Incorporated, 2017. 616 p.
30. Meier A., Kaufmann M. *SQL & NoSQL databases*. Wiesbaden : Springer Fachmedien Wiesbaden, 2019. 229 p.
31. Nakhare D. A Comparative study of SQL Databases and NoSQL Databases for E-Commerce. *International Journal for Research in Applied Science and Engineering Technology*. 2021. Vol. 9, no. 12. P. 409–412. URL: <https://doi.org/10.22214/ijraset.2021.39263> (date of access: 15.06.2024).
32. Nathan I., McNeil M. *Sails.js in Action*. Manning Publications Co. LLC, 2017. 488 p.
33. Omole O. *Server Side Development with Node.js and Koa.js Quick Start Guide: Build Robust and Scalable Web Applications with Modern JavaScript Techniques*. Packt Publishing, Limited, 2018. 132 p.

34. Optimizing SQL databases for big data workloads: techniques and best practices / A. Uzzaman et al. *Academic journal on business administration, innovation & sustainability*. 2024. Vol. 4, no. 3. P. 15–29. URL: <https://doi.org/10.69593/ajbais.v4i3.78> (date of access: 01.08.2024).
35. Performance Analysis of Scaling NoSQL vs SQL: A Comparative Study of MongoDB, Cassandra, and PostgreSQL / D. Yedilkhan et al. *2023 IEEE International Conference on Smart Information Systems and Technologies (SIST)*, Astana, Kazakhstan, 4–6 May 2023. 2023. URL: <https://doi.org/10.1109/sist58284.2023.10223568> (date of access: 08.07.2024).
36. Preuveneers D., Joosen W. Automated Configuration of NoSQL Performance and Scalability Tactics for Data-Intensive Applications. *Informatics*. 2020. Vol. 7, no. 3. P. 29. URL: <https://doi.org/10.3390/informatics7030029> (date of access: 31.05.2024).
37. Raj P. A Detailed Analysis of NoSQL and NewSQL Databases for Bigdata Analytics and Distributed Computing. *Advances in Computers*. 2018. P. 1–48. URL: <https://doi.org/10.1016/bs.adcom.2018.01.002> (date of access: 18.06.2024).
38. Salamkar M. A. Scalable Data Architectures: Key Principles for Building Systems that Efficiently Manage Growing Data Volumes and Complexity. *International Journal of Science and Research (IJSR)*. 2021. Vol. 10, no. 1. P. 1737–1744. URL: <https://doi.org/10.21275/sr210115115205> (date of access: 08.05.2024).
39. Shkodra E., Jajaga E., Shala M. Development and Performance Analysis of RESTful APIs in Core and Node.js using MongoDB Database. 17th International Conference on Web Information Systems and Technologies, Online Streaming, 26–28 October 2021. 2021. URL: <https://doi.org/10.5220/0010621200003058> (date of access: 06.05.2024).

40. Subramanian H., Raj P. Hands-On RESTful API Design Patterns and Best Practices: Design, develop, and deploy highly adaptable, scalable, and secure RESTful web APIs. Packt Publishing, 2019. 378 p.
41. The Role of Caching in Future Communication Systems and Networks / G. S. Paschos et al. *IEEE Journal on Selected Areas in Communications*. 2018. Vol. 36, no. 6. P. 1111–1125. URL: <https://doi.org/10.1109/jsac.2018.2844939> (date of access: 16.08.2024).
42. Valduriez P., Jimenez-Peris R., Özsu M. T. Distributed Database Systems: The Case for NewSQL. *Transactions on Large-Scale Data- and Knowledge-Centered Systems XLVIII*. Berlin, Heidelberg, 2021. P. 1–15. URL: https://doi.org/10.1007/978-3-662-63519-3_1 (date of access: 08.01.2025).
43. Vemulapalli G. Optimizing NoSQL Database Performance: Elevating API Responsiveness in High-Throughput Environments. *International Machine Learning Journal and Computer Engineering*. 2023. Vol. 6, no. 6. P. 1–14. URL: <https://mljce.in/index.php/Imljce/article/view/31/14> (date of access: 01.06.2024).