МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ВОЛИНСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ЛЕСІ УКРАЇНКИ
Кафедра комп'ютерних наук та кібербезпеки

На правах рукопису

НИКИТЮК АНГЕЛІНА ВІКТОРІВНА

ПОРІВНЯЛЬНИЙ АНАЛІЗ ЕФЕКТИВНОСТІ РЕЛЯЦІЙНИХ ТА НЕРЕЛЯЦІЙНИХ БАЗ ДАНИХ У ВЕБЗАСТОСУНКАХ

Спеціальність: 122 Комп'ютерні науки
Освітньо-професійна програма: Комп'ютерні науки та інформаційні технології
Кваліфікаційна робота на здобуття освітнього ступеня «бакалавр»

Науковий керівник:
БУЛАТЕЦЬКИЙ ВІТАЛІЙ ВІКТОРОВИЧ,
Доцент кафедри комп'ютерних наук та кібербезпеки

РЕКОМЕНДОВАНО ДО ЗАХИСТУ
Протокол №_____
засідання кафедри комп'ютерних наук
та кібербезпеки
від _____ 2025 р.
Завідувач кафедри
(_____) Гришанович Т. О.

ЛУЦЬК 2025

ЗМІСТ

ВСТУП

Актуальність теми. З розвитком інформаційних технологій значно зросли вимоги до обробки великих обсягів даних, забезпечення швидкого доступу до них та ефективного зберігання. Сучасні програмні продукти, особливо вебзастосунки та корпоративні інформаційні системи, оперують величезними обсягами різнорідної інформації, що потребує застосування потужних і водночас гнучких засобів управління даними.

Використання реляційних баз даних, забезпечує надійність та структурованість даних завдяки чіткій схемі даних і потужним засобам запитів на основі SQL. Але вони можуть бути менш ефективними при роботі з великими обсягами неструктурованих або напівструктурованих даних.

Нереляційні бази даних забезпечують гнучкість у збереженні неструктурованих даних та високу продуктивність у певних сценаріях використання. Вони все частіше використовуються для розробки сучасних вебдодатків, де важливі адаптивність до змін у структурі даних і швидкий доступ.

Сучасні інформаційні системи потребують обгрунтованого вибору бази даних відповідно до особливостей проєкту, що безпосередньо впливає на ефективність роботи системи. Це дослідження спрямоване на аналіз ключових характеристик та відмінностей між реляційними та нереляційними базами даних в контексті вебдодатку. Результати такого аналізу допоможуть визначити критерії вибору найбільш відповідного рішення для конкретних завдань, що є актуальним для розробників та архітекторів програмних систем у сучасних умовах розвитку IT-сфери.

Мета роботи – проведення порівняльного аналізу реляційних та нереляційних баз даних, визначення їхніх переваг, недоліків та оптимальних сфер застосування. Дослідження спрямоване на оцінку продуктивності баз даних

у реальних умовах на основі вебдодатку, що дозволить визначити, який підхід є найбільш ефективним для різних типів задач.

Для досягнення поставленої мети необхідно виконати такі завдання:

- провести аналіз концептуальних відмінностей між реляційними та нереляційними базами даних, зокрема їх моделей даних, способів зберігання та обробки інформації;
- розглянути архітектуру реляційних і нереляційних баз даних, принципи їх роботи, особливості реалізації та типові сценарії застосування;
- розробити вебдодаток, що ілюструватиме роботу обох типів баз даних, та забезпечить умови для проведення порівняльного тестування;
- виконати тестування, яке включатиме оцінку продуктивності баз даних за параметрами швидкості запису, читання, оновлення даних, а також здатності до масштабування та обробки великих обсягів інформації;
- проаналізувати отримані результати з метою визначення сильних і слабких сторін кожного типу баз даних у контексті конкретних задач вебдодатку;
- розробити рекомендації щодо вибору найбільш відповідної бази даних залежно від специфіки проєкту, що допоможе оптимізувати процеси зберігання, обробки та управління інформацією.

Об’єкт дослідження: бази даних як інструмент зберігання та обробки структурованої та неструктурованої інформації в інформаційних системах.

Предмет дослідження: ефективність використання реляційних та нереляційних баз даних під час взаємодії з вебдодатком.

Публікації:

Никитюк А. Порівняльний аналіз реляційних та нереляційних баз даних на основі вебдодатку, розробленого мовою Python. *Проблеми комп’ютерних наук, програмного моделювання та безпеки цифрових систем*.: матеріали ІІ міжнар. науково-практ. конф. Луцьк, 9–11 червн. 2025 р. Луцьк, 2025.

РОЗДІЛ 1.
ТЕОРЕТИЧНІ ОСНОВИ РЕЛЯЦІЙНИХ ТА НЕРЕЛЯЦІЙНИХ БАЗ ДАНИХ

1.1. Історія та розвиток баз даних в застосунках

Бази даних стали невід'ємною частиною інформаційних систем, забезпечуючи ефективне управління та збереження даних. Перші спроби організації збереження інформації в електронному вигляді відносяться до 1960-х років, коли з'явилися ієрархічні та мережеві моделі баз даних. Ці системи використовувалися переважно у великих підприємствах та державних установах, але мали складну структуру та вимагали значних ресурсів для адміністрування. З часом такі підходи поступилися місцем реляційним базам даних, які запропонували більш зручний спосіб організації інформації за допомогою таблиць та зв'язків між ними. Узагальнюючи, у контексті структур баз даних, їх подальший розвиток можна поділити на три періоди: навігаційний, реляційний та пост-реляційний [6].

У 1971 році група CODASYL, відповідальна за стандартизацію мови COBOL, представила свою модель баз даних, яка стала основою для багатьох комерційних продуктів [22]. Вона використовувала мережеву структуру для зберігання та навігації між даними, проте вимагала значної підготовки для ефективного використання.

Паралельно IBM у 1966 році розробила власну систему управління базами даних IMS, яка базувалася на ієрархічній моделі. Вона була простішою за CODASYL, проте також обмежувала гнучкість роботи з даними. Обидві концепції згодом отримали назву «навігаційні бази даних» через спосіб доступу до інформації.

У 1970-х роках, після публікації концепції реляційної моделі даних Едгаром Коддом, почався активний розвиток систем управління базами даних (СУБД), таких як IBM System R та Oracle [14]. Вони стали основою для сучасних реляційних баз даних, що широко використовуються і сьогодні.

У наступні десятиліття розвиток інформаційних технологій призвів до збільшення обсягу даних, що вимагало створення більш ефективних методів їх обробки та зберігання. Це стало поштовхом для появи NoSQL-рішень, які змогли задовольнити потреби масштабованості та гнучкості, особливо в умовах веб-додатків і великих розподілених систем.

Сучасний світ інформаційних технологій потребує комплексних рішень для управління даними, тому сьогодні активно використовуються як реляційні, так і нереляційні бази даних. Компанії впроваджують гібридні підходи, комбінуючи традиційні SQL-бази для транзакційної обробки даних із NoSQL-рішеннями для роботи з великими обсягами неструктурованої інформації [10]. Такі зміни у підходах до зберігання даних обумовлені зростаючими вимогами до швидкодії, масштабованості та обробки потокових даних у реальному часі.

1.2. Принципи роботи реляційних баз даних

Реляційні бази даних (RDBMS) забезпечують ефективне збереження структурованих даних у вигляді таблиць, де кожен рядок відповідає окремому запису, а стовпці репрезентують атрибути цього запису. Відмінною рисою реляційних баз є строге визначення схем даних, що гарантує узгодженість та цілісність інформації. Взаємозв’язки між таблицями будуються за допомогою зовнішніх ключів, що дозволяє організовувати дані у вигляді логічно зв’язаних структур. Це спрощує запити до бази даних та забезпечує цілісність інформації під час виконання операцій оновлення або видалення записів.

Мова структурованих запитів SQL (Structured Query Language) є стандартним інструментом для роботи з реляційними базами даних. Вона надає можливість виконувати складні операції фільтрації, сортування, групування та агрегації даних [28]. Використання SQL також дає змогу реалізовувати транзакції – послідовності операцій, які виконуються як єдине ціле. Це критично важливо для фінансових систем, де гарантується, що жодна операція не буде виконана частково або незавершеною.

Однією з ключових особливостей реляційних баз даних є підтримка ACID-властивостей (Atomicity, Consistency, Isolation, Durability). Ці властивості гарантують, що всі операції у базі даних виконуються надійно, забезпечують збереження цілісності інформації та запобігають втраті даних у разі системних збоїв. ACID-гарантії є критично важливими для фінансових систем, банківських транзакцій та інших застосунків, де необхідна висока точність і узгодженість даних [24].

Сучасні реляційні бази даних, такі як PostgreSQL, MySQL та Microsoft SQL Server, підтримують широкі можливості для складних запитів, транзакцій та розширеного аналізу даних. PostgreSQL, зокрема, відрізняється гнучкою підтримкою JSON, що дозволяє йому частково виконувати функції NoSQL-систем [3]. Завдяки своїй стабільності та можливості масштабування реляційні бази залишаються основним вибором для багатьох корпоративних застосунків.

1.3. Класифікація реляційних баз даних

Реляційні бази даних можна класифікувати за кількома критеріями, зокрема за моделлю розгортання, рівнем розподіленості, типом зберігання даних та призначенням.

За моделлю розгортання реляційні бази даних поділяються на локальні (on-premise) та хмарні (cloud-based) [13]. Локальні бази розгортаються на фізичних

серверах компанії або організації та забезпечують повний контроль над даними, що особливо важливо для корпоративних та фінансових установ. Хмарні бази, такі як Amazon RDS, Google Cloud SQL або Azure SQL Database, пропонують гнучкість, масштабованість та автоматичне управління ресурсами, що робить їх ідеальними для сучасних вебзастосунків та сервісів. Хмарні бази даних можна використовувати з будь-якого місця, де є доступ до інтернету, що особливо важливо для організацій із розподіленою структурою або великою кількістю даних. Оскільки хмарні рішення адмініструються досвідченими провайдерами, вони часто є більш надійними та стабільними, ніж локальні системи [26].

Однак у деяких сценаріях хмарні бази даних можуть мати недоліки. У таких базах дані передаються через інтернет і зберігаються на сторонніх серверах. Тому важливо ретельно перевіряти заходи безпеки, які впроваджує постачальник послуг. Іншим ризиком є залежність від стабільності інтернет-з'єднання, оскільки в разі перебоїв з мережею доступ до даних може бути ускладнений або повністю втрачений. Крім того, користувачі мають обмежений контроль над управлінням баз даних порівняно з локальними рішеннями, а перехід до іншої компанії-постачальника послуг може бути ускладнений через специфічні налаштування та технології.

За рівнем розподіленості виділяють централізовані та розподілені реляційні бази даних. Централізовані бази зберігають усі дані в одному серверному середовищі та використовуються в невеликих або середніх проєктах. Доступ до даних може бути реалізований через мережі LAN або WAN [25]. Таких тип баз даних забезпечує простоту контролю та роботи над даними. Однак централізовані бази можуть мати повільнішу швидкість обробки порівняно з іншими рішеннями, особливо коли одночасно кілька користувачів звертаються до даних. Також, у разі збою системи можуть виникнути проблеми з втратою даних, і їх відновлення може бути складним, якщо відсутні резервні копії.

Розподілені бази даних – це система, яка складається з кількох баз даних,

підключених до комп’ютерної мережі, що дозволяє зберігати та обробляти дані на різних комп’ютерах, які можуть бути в різних географічних локаціях. [7] Це забезпечує ефективність обробки та підтримки даних. Розподілені бази, такі як Google Spanner, дозволяють розподіляти дані між кількома серверами, що підвищує стійкість до відмов та збільшує продуктивність системи [1]. Перевагою розподілених баз даних є їхня надійність, оскільки збої в одній частині системи не впливають на всю систему. Однак, є й недоліки, такі як складність забезпечення цілісності даних, дублювання даних на різних базах даних і потенційні проблеми з розподілом даних, що може уповільнити обробку запитів.

За типом зберігання даних реляційні бази поділяються на традиційні дискові (disk-based) та бази даних на основі оперативної пам’яті (in-memory). Дискові бази, такі як PostgreSQL та MySQL, записують дані на фізичні накопичувачі та використовують складні механізми кешування для прискорення доступу. Ін-меморі бази, як Redis, працюють повністю в оперативній пам’яті, що дозволяє отримати надзвичайно високу швидкість обробки запитів, однак вимагає значних апаратних ресурсів.

Дискові бази даних вже давно є стандартом систем керування базами даних завдяки їхній здатності зберігати великі обсяги даних за низькою ціною. Однак доступ до даних на диску може бути повільніший приблизно в 10 разів за доступ до них з оперативної пам’яті [20]. Така різниця зумовлена через взаємодію дискової пам’яті з низькорівневими компонентами операційної системи. Наприклад, Neo4j, графова база даних, використовує диск як основне сховище, при цьому намагається кешувати якомога більше даних в оперативній пам’яті. Натомість, бази даних на основі оперативної пам’яті працюють по-іншому: вони зберігають усі свої дані безпосередньо в оперативній пам’яті, що спрощує архітектуру, усуваючи потребу в буферах. Однак їхнім головним недоліком є обмеження в розмірі доступної пам’яті. Якщо об’єм даних перевищує розмір RAM, доведеться перейти на потужнішу і, отже, дорожчу машину. Також,

враховуючи, що дані в оперативній пам’яті не зберігаються після вимкнення живлення, для забезпечення надійності таких систем використовують енергонезалежну пам’ять, таку як диск для резервного копіювання даних [16]. У разі збою система зчитує дані з журналу на диску та відновлює оперативну пам’ять. Таким чином досягається як висока швидкість роботи, так і збереження даних.

Альтернативою є так звана архітектура «більша, ніж пам’ять» (Larger-than-memory), яка поєднує продуктивність оперативної пам’яті з ємністю жорсткого диска. Ця модель дозволяє розрізняти «гарячі» (ті, що часто використовуються) дані та «холодні» (ті, що рідко використовуються) дані. Холодні дані ідентифікуються завдяки відслідковуванні активності транзації, або у вигляді фонового процесу аналізу даних. Після їх виявлення, такі дані переміщуються на диск для оптимізації продуктивності. Вилучення холодних даних може бути ініційовано або при наближенні верхньої межі пам’яті, або при виникненні потреби в просторі.

У цьому типі архітектури, коли транзакція потребує дані, що знаходяться на диску, є дві стратегії вирішення цієї задачі: її можна або перервати та перезапустити після завантаження даних, або призупинити під час завантаження даних з диску.

За призначенням реляційні бази можуть бути загального призначення або спеціалізовані. Загальні СУБД, такі як PostgreSQL, MySQL та Microsoft SQL Server, використовуються у широкому спектрі застосунків – від вебсайтів до корпоративних систем. Спеціалізовані бази, такі як Oracle Database, орієнтовані на великі підприємства та підтримують складні аналітичні функції, що необхідні для обробки фінансових даних або управління ресурсами.

1.4. Принципи роботи нереляційних баз даних

Нереляційні бази даних (NoSQL) розроблені для зберігання великих обсягів даних, що не є строго структурованими [15]. Вони пропонують альтернативні підходи до організації інформації, а також забезпечують ефективну масштабованість, високу швидкість обробки та гнучкість у роботі з різними типами даних. Однією з головних переваг NoSQL-баз є можливість горизонтального масштабування – вони легко розподіляються між кількома серверами, що дозволяє ефективно працювати з великими потоками даних у розподілених системах. [21]

MongoDB є одним із найпопулярніших представників документо-орієнтованих баз даних [8]. Вона використовує JSON-подібні документи для зберігання інформації, що дає змогу змінювати структуру записів без необхідності складних міграцій. Це особливо корисно для вебдодатків, де дані можуть змінюватися динамічно. MongoDB також підтримує вбудовані індекси, що значно підвищує швидкість запитів, а її модель роботи дозволяє розподіляти навантаження між кількома вузлами, зберігаючи при цьому швидкість обробки операцій.

Redis, на відміну від MongoDB, є базою типу «ключ-значення", яка працює переважно у пам’яті, що робить її надзвичайно швидкою [29]. Вона часто використовується для кешування даних, обробки черг повідомлень та збереження сесій користувачів. Завдяки підтримці структур даних, таких як списки, множини та хеші, Redis надає розширені можливості роботи з великими обсягами інформації у реальному часі. Проте обмежена стійкість до втрати даних і необхідність ручного управління масштабуванням можуть стати викликами під час використання у критично важливих системах [12].

NoSQL бази даних широко застосовуються у сфері великих даних, стрімінгових платформ, мобільних додатків та розподілених обчислень.

Наприклад, Kafka використовується для потокової передачі даних та аналітики у реальному часі [4]. Її архітектура дозволяє організовувати передачу великих потоків інформації між сервісами, що робить її ідеальним рішенням для великих розподілених систем, таких як фінансові платформи та аналітичні сервіси.

1.5. Класифікація нереляційних баз даних

Нереляційні бази даних (NoSQL) класифікуються за способом організації даних, що визначає їхню продуктивність, гнучкість та область застосування. Основними типами NoSQL баз є документо-орієнтовані, ключ-значення, колонкові та графові бази даних. Кожен із цих типів оптимізований для певних задач і використовується у різних сферах, від обробки великих масивів даних до моделювання складних взаємозв'язків між об'єктами.

Документо-орієнтовані бази даних (MongoDB, CouchDB) використовують JSON або BSON-документи для зберігання інформації. Вони забезпечують гнучку схему даних, що дозволяє зберігати записи без необхідності попереднього визначення структури таблиць. Також, їхня гнучкість проявляється у підтримці вкладених об'єктів. Наприклад, колекція «users» може містити документи про користувачів із полями імені, електронної пошти, адреси та налаштувань. У цьому документі адреса може бути представлена як вкладений об'єкт з відповідними даними (вулиця, місто, штат, індекс), а замовлення можуть зберігатися як масив об'єктів. У реляційній базі даних аналогічну інформацію довелося б розподілити між кількома таблицями. Наприклад, дані користувача могли б бути розділені на таблиці «Users», «Addresses», «Preferences» та «Orders». Для встановлення зв'язків між таблицями потрібно було б використати зовнішні ключі. Такий підхід дозволяє уникати дублювання даних і забезпечує їхню цілісність, проте ускладнює операції приєднання і обробки даних. Документо-орієнтовані бази даних, у порівнянні з реляційними, спрощують

зберігання складних структур за рахунок гнучкої схеми.

Документоорієнтовані бази підтримують CRUD-операції: створення, читання, оновлення та видалення документів. Більшість таких баз надають можливість індексації даних для прискорення пошуку. Також підтримується агрегація даних за певними критеріями, що нагадує GROUP BY у SQL. Для масштабування часто використовується шардинг, що дозволяє розподіляти дані між кількома серверами. Деякі документні бази підтримують валідацію даних, що дозволяє встановлювати правила їх збереження. Також доступна підтримка транзакцій для забезпечення атомарності змін кількох документів. [27]. Проте, такий тип баз даних програють реляційним у сценаріях, де потрібне виконання складних запитів, особливо тих, що передбачають багаторазові об’єднання даних з різних колекцій. Також деякі документо-орієнтовані бази даних мають обмежену підтримку ACID-властивостей, що може ускладнити забезпечення цілісності даних в атомарних операціях. Гнучкість та відсутність структурованості даних може супроводжуватися відсутністю вбудованої валідації даних на вході, що ускладнює підтримання якості та достовірності інформації.

Такий підхід ідеально підходить для вебдодатків, контент-менеджмент систем та мобільних застосунків, де дані можуть мати змінну структуру. Завдяки вбудованій підтримці індексів та можливості горизонтального масштабування документо-орієнтовані бази забезпечують високу швидкість доступу до інформації.

Бази даних типу «ключ-значення» (Redis, DynamoDB) є найпростішими серед NoSQL рішень. Вони зберігають дані у вигляді пар «ключ-значення», що забезпечує швидкий доступ до інформації. Такі бази ідеально підходять для кешування, управління сесіями користувачів та реального часу аналітики. Redis, наприклад, працює в оперативній пам’яті, що значно прискорює читання та запис, але потребує додаткових механізмів для збереження стійкості даних.

Завдяки своїй архітектурі, бази даних типу «ключ-значення» чудово справляються з обробкою великих обсягів даних у реальному часі, зокрема для аналітики IoT-пристроїв або моніторингу соціальних мереж. У сфері ігор такі бази використовуються для збереження рейтингів і таблиць лідерів через швидкий доступ до записів. Водночас, такі бази даних мають обмежені можливості запитів і не підходять для складної обробки даних або побудови зв'язків між об'єктами, як наприклад у реляційних базах даних. Їхня проста модель даних ефективно показує себе у системах, де потрібне горизонтальне масштабування і мінімальні вимоги до складності. Попри переваги у швидкості та гнучкості, такі бази іноді поступаються реляційним за підтримкою ACID-властивостей і цілісності даних. Ключові переваги цих баз включають високу продуктивність, простоту інтеграції, гнучкість та можливість розподіленого зберігання. Вибір на користь баз типу «ключ-значення» доцільний у випадках роботи з неструктурованими даними, побудови масштабованих розподілених систем або обробки даних з високою частотою змін. [11]

Колонкові бази даних (Apache Cassandra, HBase) використовуються для обробки великих розподілених обсягів даних. Вони зберігають інформацію у стовпцях, а не у рядках, що дозволяє ефективно працювати з аналітичними запитами та масовими операціями над даними. Так як дані в колонці однорідні та часто повторюються, це дає можливість стиснути їх пам'ять навіть до 10 разів за допомогою традиційних алгоритмів стиснення, як LZW [5]. Запити читання у таких базах даних можуть відбуватися швидше, ніж у рядкових, бо з диску зчитуються лише потрібні колонки. Також завдяки колонковій структурі, даними можна оперувати як векторами, замість ітерації кожного рядка, що дає значне прискорення запитів. Ще одна перевага – ефективні запити оновлення. У колонкових базах даних оновлення змінюють дані точково лише в тих колонках, які зазнали змін. У базах даних із рядковою орієнтацією будь-яке оновлення вимагає переписування всього рядка.

Проте, видалення у колонкових базах потребує використання спеціальних позначок (tombstones), щоб позначити видалені рядки, що може негативно впливати на рівень стиснення даних. Для вставки нових записів також необхідна певна реконструкція рядків. Тому операції оновлення та вставки залишаються ефективнішими в базах даних із рядковою орієнтацією.

Такі бази даних ідеально підходять для завдань зі зберігання даних, бізнес-аналітики та аналітики. Це обумовлено тим, що великі обсяги даних добре стискаються, запити часто виконують агрегування великої кількості записів, що ефективніше обробляється при колонковій структурі, а частота оновлень низька порівняно з кількістю операцій читання. Водночас це не найкращий варіант для сценаріїв з високим транзакційним навантаженням, частими вставками та оновленнями. З ними краще справляються рядкові бази даних.

Графові бази даних (Neo4j, ArangoDB) створені для роботи зі складними взаємозв’язками між об’єктами. У таких базах даних інформація зберігається у вигляді вершин (сутностей) та ребер (зв’язків між ними), що дозволяє виконувати швидкі запити для пошуку взаємозв’язаних даних, що неможливо реалізувати ефективно у традиційних реляційних або інших NoSQL базах. Вони особливо корисні у сферах із складною структурою даних, таких як соціальні мережі, системи рекомендацій, виявлення шахрайства, управління ланцюгами постачання та біоінформатика [9].

Графові бази даних краще підходять для даних з великою кількістю зв’язків. Вони орієнтовані на моделювання складних взаємозв’язків між об’єктами та працюють найкраще у тих сценаріях, де потрібно швидко та ефективно працювати з багатьма рівнями зв’язків — наприклад, у соціальних мережах, де можна легко визначати друзів друзів або поширення інформації через мережу контактів. Щодо запитів, реляційні бази даних використовують мову SQL, яка чудово справляється із запитами на основі табличних даних. У той час як графові бази даних застосовують спеціалізовані мови, як-от Cypher або

Gremlin, які оптимізовані для пошуку шляхів, виявлення взаємозв’язків та навігації по графу. У контексті масштабованості реляційні бази мають усталені методики горизонтального та вертикального масштабування для великих обсягів даних. Графові бази даних масштабуються складніше через специфіку збереження зв’язків, однак сучасні технології забезпечують розподілену обробку навіть для великих графових структур.

Кожен тип нереляційних баз даних має свої переваги та обмеження, тому вибір конкретного рішення залежить від вимог до продуктивності, масштабованості та типу даних, що використовуються в конкретному застосунку. Сучасні інформаційні системи часто комбінують кілька типів баз даних для досягнення оптимальної продуктивності та гнучкості роботи з інформацією.

1.6. Огляд використання баз даних в реальних застосунках

Реальні застосунки використовують бази даних відповідно до специфіки їх роботи та вимог до продуктивності.

Розглянемо соціальні мережі на прикладі Facebook. Це соціальна мережа, що дозволяє людям спілкуватися, обмінюватися контентом та підтримувати зв’язок із друзями та родиною онлайн. Станом на 2025, система налічує понад 3 мільярд користувачів, а отже, вона має справлятися з великим обсягом даних швидко та ефективно, щоб забезпечити приємний користувацький досвід.

Цей продукт використовує реляційну базу даних для зберігання своїх основних даних, включаючи соціальний граф та дані продукту Facebook Messenger. Для цього компанія використовує базу даних MySQL з двома різними типами сховища: InnoDB та RocksDB. InnoDB підтримує індекси B+ дерева, характеризується швидким читання, але повільними записами, тому компанія застосовує її для соціального графу. Натомість, завдяки можливостям швидкого

запису, RocksDB використовується для керування даними месенджеру. MySQL підтримує понад мільярд користувачів, причому пікові навантаження сягають кількох мільйонів запитів і модифікацій на секунду. Вона розгорнута на понад 1000 серверах та активно використовує технології сегментації (sharding) та реплікації даних. Це допомагає ефективно розподілити таке велике навантаження на систему.

З метою прискорення запитів та ефективного розподілу навантаження, використовується потужна система кешування: понад 99% запитів звертаються до кешу безпосередньо [19]. Окрім цього, Facebook застосовує систему кешування TAO, яка розуміє структуру графа Facebook.

Для аналізу даних та збереження метрик Facebook використовує Hadoop, Hive та Presto, а ведення журналу подій застосовується система зберігання LogDevice. Програмний продукт також застосовує Beringei – система зберігання часових рядів в оперативній пам’яті. У ньому зберігаються дані та статистика продуктивності програми, що дозволяє моніторити навантаження на сервіси в режимі реального часу.

Наступний приклад – Netflix. Цей програмний продукт світовим лідером у сфері розваг та стрімінгу медіа, який налічує понад 300 мільйонів користувачів станом на 2024 рік [18]. Щоб забезпечити зручне користування навіть у пікові навантаження, така система має забезпечувати швидкий доступ до даних.

Щоб забезпечити низьку затримку в усьому світі, дані реплікуються в різних регіонах. Окрім цього, реалізоване кешування за допомогою EVCache – сховища даних на основі оперативної пам’яті. Воно використовує SSD, що забезпечує швидкі операції читання та запису. Дана база даних зменшує навантаження на основні джерела даних, адже у ній зберігаються найбільш затребувана інформація [17].

Джерелом критично важливих для бізнесу даних являється реляційна база даних MySQL. Вона забезпечує надійність та послідовність, узгодженість даних,

тому це робить її хорошим рішенням для обробки транзакцій та бізнес-операцій. Окрім цього, дана база даних реплікуються на багатьох центрах обробки даних, щоб надати швидкі доступ та відновлення після аварій.

Для зберігання важливих даних також використовується нереляційна база даних Apache Cassandra. Вона потрібна у сценаріях, що вимагають великого обсягу читання та запису. Завдяки розподіленій архітектурі та здатності обробляти велику кількість запитів, Cassandra ідеально підходить для великих та динамічних даних, таких як історія активності користувачів.

На додачу, для ефективного моніторингу системи, реалізована система реєстрації подій. Щодня Netflix реєструє приблизно 500 мільярдів подій з різних джерел, таких як активність переглядів, системні помилки та показники продуктивності. Kafka виступає в ролі початкової системи обміну повідомленнями, здатної обробляти масивний потік даних у режимі реального часу. Вона забезпечує відмовостійкість та масштабованість. Для постійного збереження цих даних використовується база даних Apache Chukwa. Вона передає дані до централізованого сховища, такого як Amazon S3, для довгострокового аналізу. Разом ці інструменти забезпечують ефективне управління даними в режимі онлайн та офлайн.

Щоб реалізувати швидкий пошук та ефективну аналітику в режимі реального часу, використовується Elasticsearch – пошукова та аналітична система із сховищем даних. Завдяки постійному аналізу вподобань та взаємодії користувачів, вона дозволяє Netflix покращувати пошук контенту, персоналізувати рекомендації та виявляти технічні аномалії. Це забезпечує безперебійний, релевантний та якісний досвід.

Окрім цього, вищезгадане хмарне сховище Amazon S3 слугує для збереження великих масивів даних, включаючи медіадані, такі як відео, аудіо та зображення.

Як ми бачимо, реальні застосунки часто застосовують гібридну архітектуру баз даних, використовуючи як реляційні, так і нереляційні рішення. Аналіз таких систем дозволяє оцінити ефективність різних підходів до зберігання даних та обрати оптимальне рішення для конкретного проєкту.

РОЗДІЛ 2.
ПОРІВНЯЛЬНИЙ АНАЛІЗ РЕЛЯЦІЙНОЇ ТА НЕРЕЛЯЦІЙНОЇ БАЗИ ДАНИХ ПІД ЧАС ВЗАЄМОДІЇ З ВЕБДОДАТКОМ

2.1 Постановка задачі

Основною задачею кваліфікаційної роботи є проведення порівняльного аналізу реляційних та нереляційних баз даних і виявлення їх переваг та недоліків у межах одного програмного середовища на прикладі реального вебдодатку. Дослідження спрямоване на комплексну оцінку продуктивності, масштабованості та зручності використання реляційних і нереляційних баз даних у реальному прикладному середовищі.

Як приклад програмного середовища обрано вебдодаток для ведення персонального блогу. При його розробці необхідно врахувати функціональні вимоги до системи: підтримку авторизації, створення та редагування публікацій, коментування, а також забезпечення стабільної роботи при одночасному доступі кількох користувачів. Для досягнення мети кваліфікаційної роботи потрібно створити вебдодаток, який взаємодіє з двома різними СУБД – реляційною та нереляційною, кожна з яких має власну архітектуру даних, але наповнена однаковими даними. Це дозволить об’єктивно оцінити їх ефективність у типових сценаріях використання.

Крім того, з метою забезпечення порівняння різних типів баз даних потрібно розгорнути єдине середовище виконання із використанням Docker-контейнерів, розробити набір тестів, що охоплюють основні CRUD-операції (створення, читання, оновлення, видалення), а також врахувати такі аспекти, як:

- час виконання запитів;
- обсяг резервних копій;
- підтримка транзакцій і віконних функцій;

- реалізація складних запитів із фільтрацією, агрегацією та сортуванням.

На основі отриманих результатів аналізу потрібно сформулювати практичні рекомендації щодо доцільного використання реляційних та нереляційних баз даних у реальних програмних застосунках залежно від вимог до продуктивності, гнучкості структури даних та масштабованості.

2.2 Методологія дослідження та вибір моделі розробки вебдодатку

З метою забезпечення достовірності аналізу потрібно розробити архітектуру обох СУБД, що дозволить виконувати ідентичні операції над всіма необхідними сутностями. Для проведення дослідження також потрібно створити однакове заповнення двох баз даних.

Етапи дослідження включають написання скриптів для генерації тестових даних, синхронізації даних в обох базах, проведення вимірювань часу виконання CRUD-операцій (створення, читання, оновлення, видалення) та комплексних запитів з агрегаціями, віконними функціями та транзакціями.

Передбачається виконання всіх операцій багаторазово для мінімізації впливу випадкових відхилень. Результати запусків зберігатимуться у текстових файлах, на основі яких обчислюватимуться середні показники продуктивності. Вимірювання проводитимуться як для початкового стану баз даних, так і після додавання індексів. Також передбачається порівняння обсягу баз даних та їх резервних копій, з подальшим стисканням архіватором із однаковими параметрами.

Для реалізації вебдодатку обрано спіральну модель розробки програмного забезпечення, яка поєднує переваги ітеративного та каскадного підходів. Її основна ідея полягає у представленні процесу створення продукту у вигляді спіралі, що складається з послідовних витків, де кожен виток охоплює чотири

основні етапи: визначення цілей, оцінка ризиків, розробка і тестування, а також планування наступної ітерації.

Життєвий цикл розробки за цією моделлю нагадує спіраль, яка починається з планування і з кожною ітерацією рухається до створення повноцінного продукту. На кожному витку спіралі розробляється новий прототип, який доповнює існуючий функціонал. Цей підхід забезпечує створення проміжних результатів, які можна тестувати та вдосконалювати до отримання фінальної версії продукту (рис. 2.1).

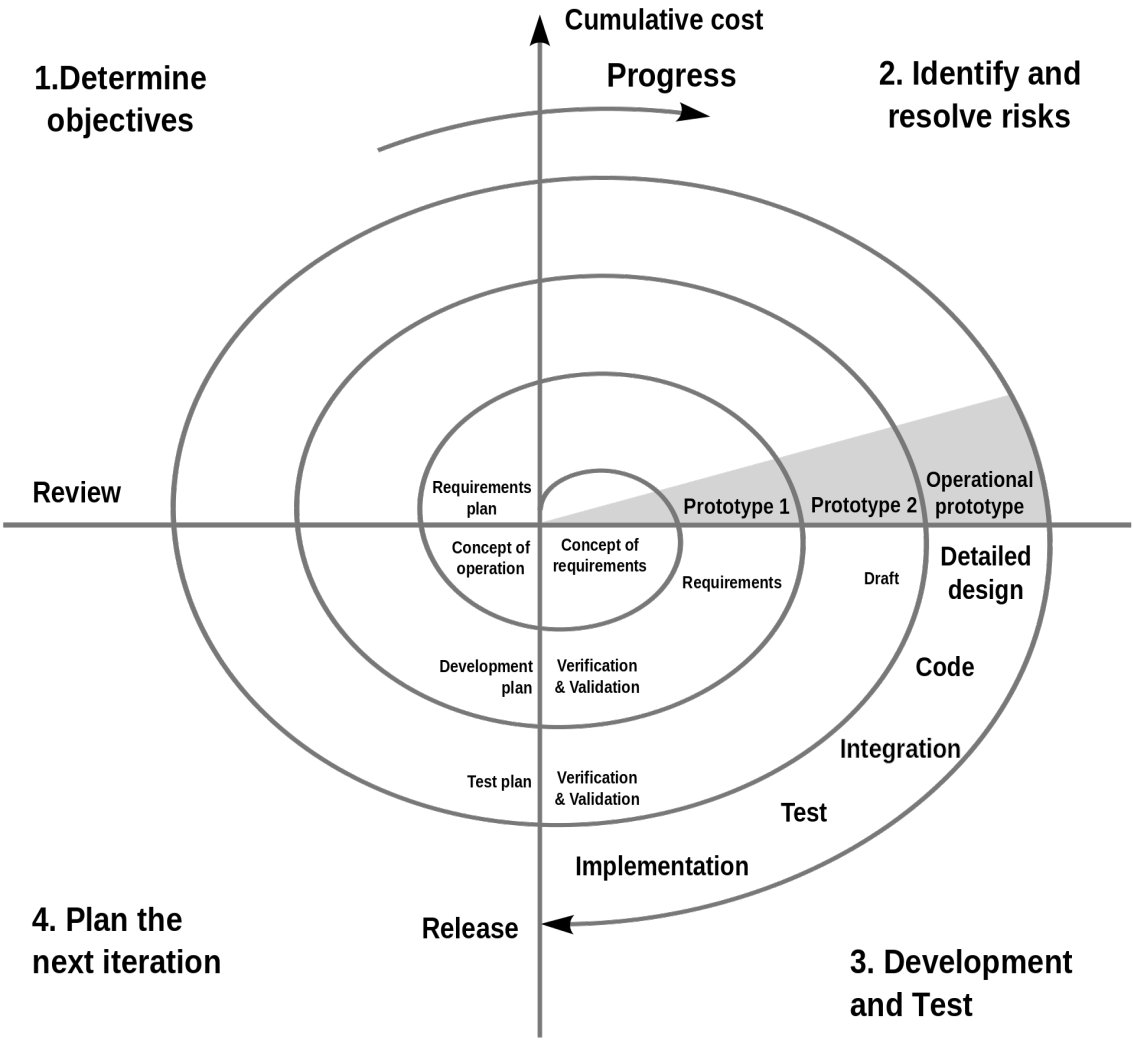

Рисунок 2.1 – Спіральна модель розробки.

Однією з ключових особливостей спіральної моделі є концентрація на оцінці можливих ризиків. Виділення окремої стадії для їх аналізу дозволяє розробникам своєчасно виявляти проблеми та знаходити ефективні способи їх вирішення. Це забезпечує високу гнучкість у процесі розробки і дає можливість адаптуватися до змінних вимог.

У процесі розробки персонального блогу спіральна модель дає можливість поступового впровадження функціоналу:

- на першій ітерації створюється базовий прототип із можливістю створення та відображення постів
- на наступних витках додається, система реєстрації користувачів і управління профілем
- завершальні етапи включають покращення дизайну, інтеграцію з додатковими базами даних і впровадження системи безпеки

2.3 Обґрунтування вибору інструментальних засобів

У якості реляційної бази даних було обрано PostgreSQL завдяки її високій продуктивності, стабільності та здатності працювати зі складними структурами даних, зокрема JSON. Вона забезпечує підтримку транзакцій і високий рівень захисту, що особливо важливо для додатків, які працюють із персональними даними користувачів. Окрім того, ця СУБД є однією з найпоширеніших серед реляційних баз даних, які активно застосовуються в сучасних вебдодатках.

MongoDB була обрана як нереляційна база даних, що теж відома як популярне рішення завдяки своїй гнучкості у зберіганні неструктурованих даних, високій масштабованості та швидкості обробки великих обсягів інформації. Завдяки можливості зберігання документів у форматі BSON, дана база даних дозволяє ефективно працювати з вкладеними структурами, такими як коментарі до публікацій.

Для реалізації вебдодатку був ретельно обраний набір інструментів, що гарантують гнучкість, продуктивність і безпеку. Основними критеріями при виборі були зручність використання, надійність і можливість легкої інтеграції з іншими системами та технологіями.

Flask є вебфреймворком для Python, що надає мінімалістичний підхід до розробки додатків. Він вирізняється своєю гнучкістю та модульною структурою, яка дозволяє розробникам самостійно обирати необхідні компоненти. Це робить Flask чудовим рішенням для проектів різного масштабу. Крім того, активна спільнота підтримує розвиток численних бібліотек, які значно спрощують процес розробки.

Для створення адаптивного інтерфейсу було використано CSS-фреймворк Bootstrap. Завдяки готовим компонентам, таким як кнопки, форми та меню навігації, цей інструмент дозволяє швидко створювати естетичний і зручний дизайн, що коректно відображається на різних пристроях – від мобільних телефонів до настільних комп’ютерів.

Такий вибір інструментів дозволяє здійснити аналіз із використанням актуальних і широко застосовуваних технологій, що підвищує релевантність та практичну цінність отриманих результатів.

Розгортання баз даних, додатку та запуск скриптів проводилось на машині, оснащеній процесором 11th Gen Intel® Core™ i5-11400H @ 2.70GHz × 12, відеокартою NVIDIA GeForce RTX 3050 Laptop GPU/PCIe/SSE2. Використовується операційна система Ubuntu 20.04.6 LTS 64-bit.

Для проведення тестування були запущені бази даних PostgreSQL версії 15.1 та MongoDB версії 7.0.20 в ізольованих Docker-контейнерах. Такий підхід дозволив забезпечити контрольоване середовище та гарантувати однакові умови виконання для обох СУБД.

2.4. Моделювання структури даних, наповнення та підготовка середовища для порівняльного дослідження PostgreSQL і MongoDB

Архітектура PostgreSQL включає в себе представлення сутностей користувачів, публікацій та дописів у трьох окремих таблицях (рис. 2.2). Завдяки підтримці неструктурованості даних та документо-орієнтованості бази даних MongoDB, сутності представлені у двох колекціях, замість трьох, як у реляційній базі (рис. 2.3).

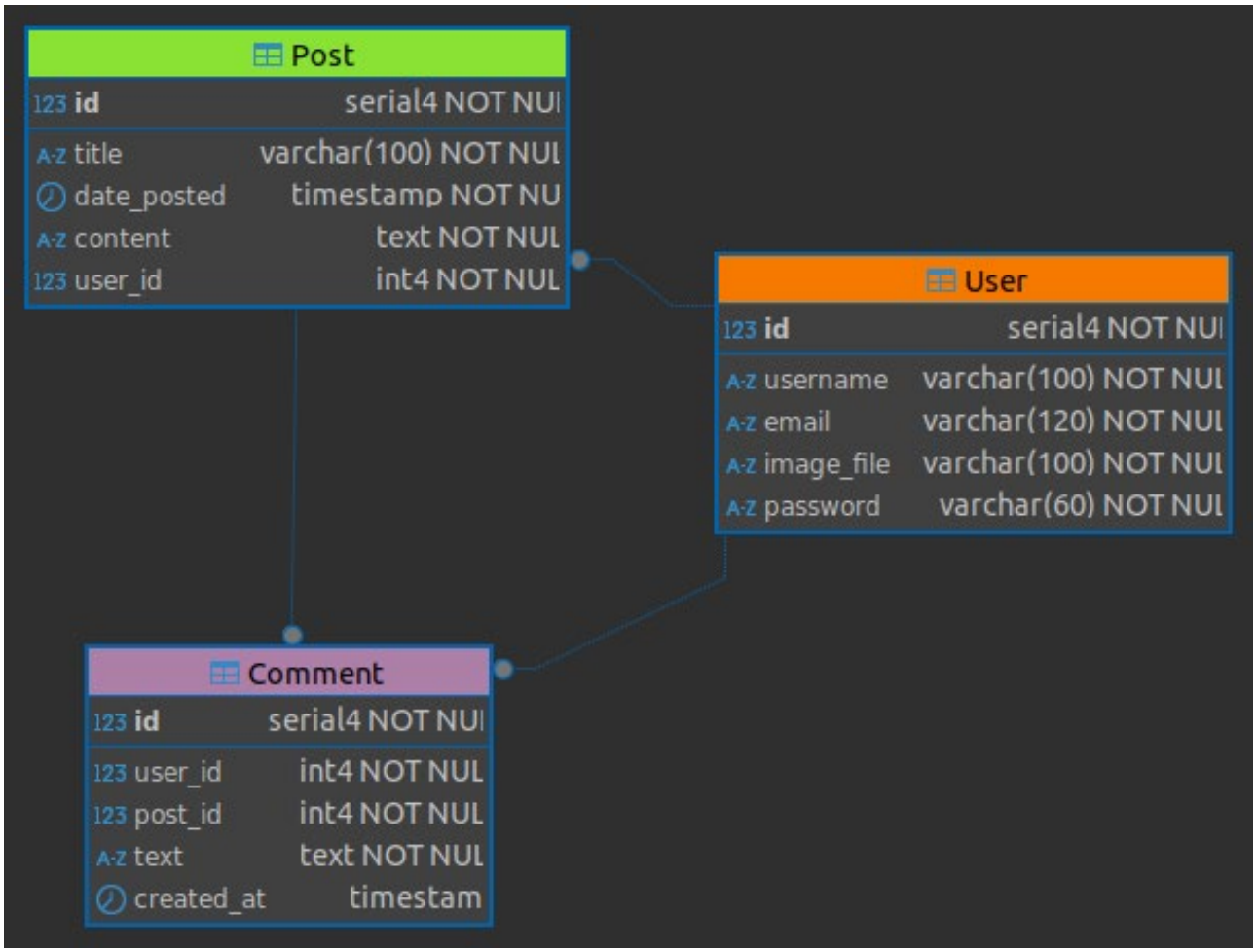

Рисунок 2.2 – ERD-діаграма в PostgreSQL.

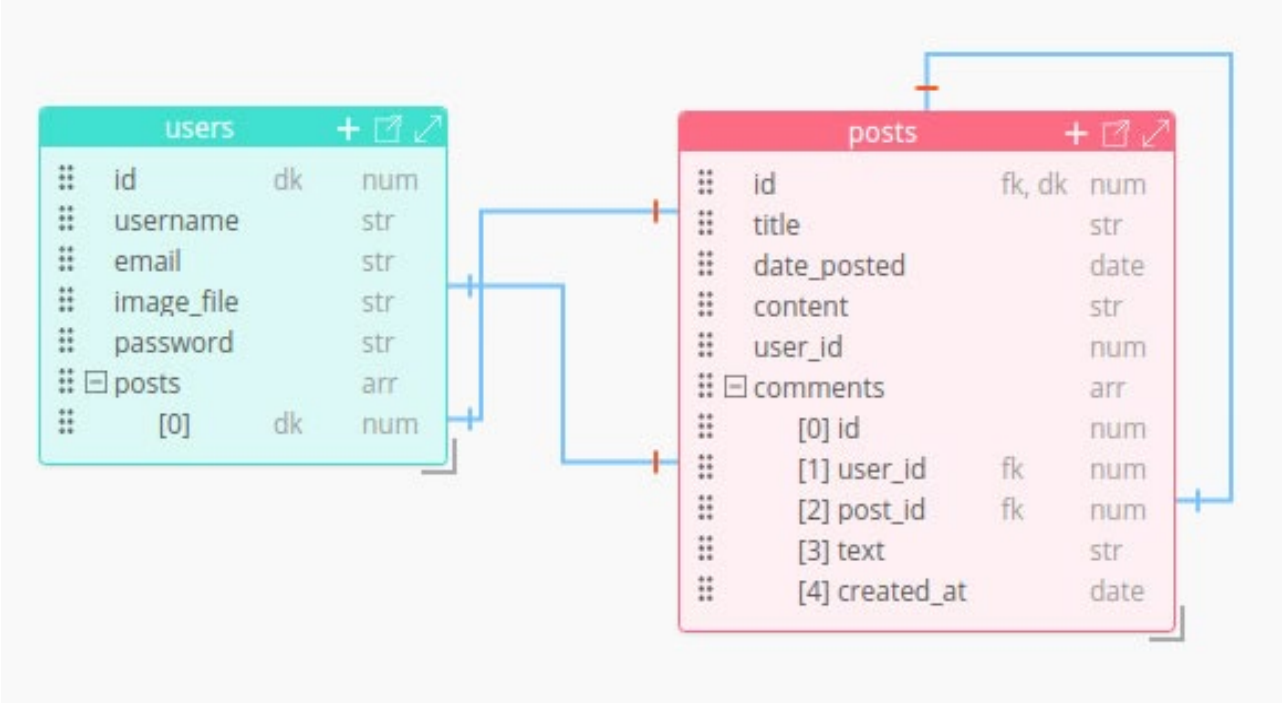

Рисунок 2.3 – ERD-діаграма в MongoDB для порівняльного аналізу.

З метою порівняльного аналізу було написано скрипт для генерації випадкових даних та заповнення ними реляційної бази даних (Додаток А, рис. А.1). У результаті, база даних PostgreSQL містить такі записи:

- 150 004 записів в таблиці «User»
- 10 200 009 записів в таблиці «Post»
- 10 000 005 записів в таблиці «Comment»

Наступним етапом було копіювання цих даних до нереляційної бази даних MongoDB за допомогою скрипта, що вивантажує дані з PostgresSQL та адаптує їх структуру для створення записів у MongoDB (Додаток А, рис. А.2; рис. А.3). У результаті, було згенеровано записи в нереляційній базі даних:

- 150 004 документів в колекції «users»
- 10 200 009 документів в колекції «posts» (10 000 005 вкладених об’єктів коментарів)

Таким чином, бази даних отримали ідентичне заповнення даними.

Для виконання вимірювання продуктивності були реалізовані спеціальні скрипти на Python із використанням бібліотек psycopg2 для PostgreSQL та pymongo для MongoDB (Додаток А, рис. А.4). Вимірювання часу виконання кожного запиту здійснювались за допомогою модуля time, а результати зберігались у текстові файли для подальшого аналізу. Після цього проводився аналіз цих даних та підрахунки середніх значень та візуалізація результатів у вигляді діаграм.

Для порівняння резервних копій баз даних був створений Bash-скрипт, що надсилає команди створення резервних копій у Docker контейнери кожної бази даних за допомогою утиліт pg_dump і mongodump, архівує обидві копії за допомогою утиліти tar та gzip з однаковими параметрами (Додаток А, рис. А.5). Таким чином, було отримано архіви резервних копій, що стиснені тим самим способом з метою об’єктивного порівняння їх розмірів.

Вебдодаток, для якого розроблялися моделі баз даних, складається з клієнтської та серверної частин. На рисунку 2.4 можна ознайомитися із UML-діаграмою проєкту. Клієнтська частина представлена у вигляді інтерфейсу, розробленого із застосуванням HTML, CSS і фреймворку Bootstrap, що забезпечує адаптивний дизайн та зручність використання на різних пристроях. Серверна частина проєкту включає в себе модульну архітектуру, реалізовану мовою Python із використанням фреймворку Flask [23]. Серверна частина взаємодіє з такими базами даних, як PostgreSQL та MongoDB.

Головна сторінка містить список останніх публікацій, кожна з яких має заголовок, короткий опис і посилання на повний текст. Також додаток дозволяє переглянути коментарі до окремої публікації, переглянути публікації, створені окремим користувачем. Користувач може зареєструватися або увійти до системи через форму авторизації, після чого йому відкривається можливість створення та редагування власних дописів.

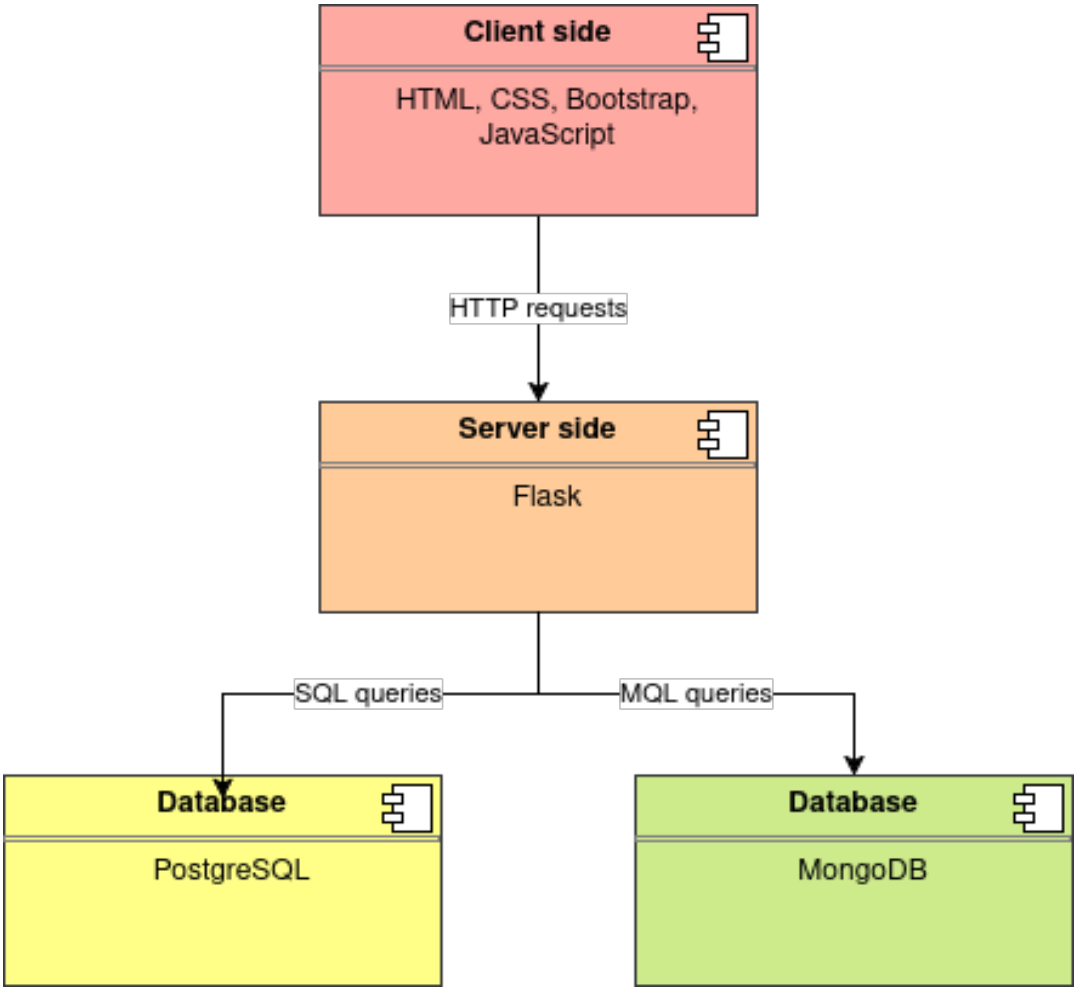

Рисунок 2.4 – UML-діаграма архітектури проєкту.

2.5 Аналіз отриманих результатів дослідження, рекомендації щодо використання та впровадження

2.5.1. Порівняння розміру бази даних та резервних копій з індексами і без них

У процесі порівняльного аналізу реляційної та нереляційної бази даних було проведено оцінку їх фактичного обсягу з урахуванням однакової структури та наповнення.

Для MongoDB, згідно з результатом db.stats(), загальний розмір бази даних (параметер totalSize) становив 3316 МБ у десятковій системі або 3163 МБ у

бінарній. Для PostgreSQL – загальний розмір бази даних становив приблизно 4760 МБ.

Після створення індексів у базах даних PostgreSQL та MongoDB було зафіксовано зміну розміру сховищ. Розмір бази даних PostgreSQL збільшився з 4760 МБ до 5474 МБ, що означає приріст приблизно на 15% (714 МБ). Після додавання аналогічних індексів у MongoDB, загальний об’єм збільшився на 628 МБ: з 3316 МБ до 3944 МБ (в десятковому представленні). А у бінарному представленні різнця склала 598 МБ: зафіксовано розмір даних у 3761 МБ. Збільшення склало приблизно 19%.

Для отримання об’єктивних даних щодо зберігання було створено резервні копії обох баз даних з використанням утиліти tar з параметрами -czf, що дозволяє створити стиснутий архів у форматі .tar.gz. Результати показали, що:

- розмір архівованого бекапу бази даних MongoDB склав 1,6 ГБ;
- розмір архівованого бекапу бази даних PostgreSQL склав 1,4 ГБ.

Незважаючи на менший загальний обсяг даних у MongoDB відповідно до статистики db.stats(), розмір її резервної копії виявився більшим, ніж у PostgreSQL. Це може бути зумовлено особливостями зберігання вкладених документів, відсутністю попередньої оптимізації схеми (на відміну від структурованих таблиць у PostgreSQL), а також різницею у внутрішньому представленні даних.

2.5.2. Порівняльний аналіз кількісних показників часу виконання запитів

У рамках дослідження було проведено порівняння запитів до реляційної та нереляційної баз даних для однакового набору функціональних операцій. В табл.2.1 наведено короткий опис основних сценаріїв використання баз даних та їх реалізації.

Таблиця 2. 1.

Основних сценарії використання баз даних MongoDB та PostgreSQL.

	Опис сценарію	**Запит MongoDB**	**Запит PostgreSQL**
1	Отримання 100 постів з пагінацією	db.posts.find().limit(100).skip(5447680)	SELECT * FROM "Post" limit 100 offset 5447680
2	Отримання постів одного користувача	db.posts.find(user_id: 54476)	SELECT * FROM "Post" WHERE user_id = 54476
3	Отримання 1 поста	db.posts.findOne({ _id: 5447680 })	SELECT * FROM "Post" WHERE id = 5447680
4	Отримання одного поста з пов’язаними коментарями	db.posts.findOne({ _id: 5447680 })	SELECT "Post".*, "Comment".* FROM "Post" JOIN "Comment" ON "Post".id = "Comment".post_id WHERE "Post".id = 5447680
5	Створити новий запис посту	db.posts.insertOne({ title: "Test title", content: "Test content", user_id: 54476 })	INSERT INTO "Post" (title, content, user_id, date_posted) VALUES (‘Test title’, ‘Test content’, 54476, NOW()) RETURNING id

6	Створити новий запис коментаря	db.posts.updateOne(
		{ _id: 5447680 },
		{$push: {
		comments: {
		user_id: 54476,
		text: "Test comment",
		id: 1
		}
		}
		}
		)
			INSERT INTO "Comment" (post_id, user_id, text, created_at)
			VALUES (5447680, 54476, 'Test comment', NOW())
			RETURNING id

7	Оновити запис посту	db.posts.updateOne({ _id: 5447680 }, { $set: { title: "'Updated title", content: "'Updated content" } })	UPDATE "Post" SET title = 'Updated title', content = 'Updated content' WHERE id = 5447680 RETURNING id
8	Оновити запис коментаря	db.posts.updateOne({ _id: 5447680, "comments.id": 1 }, { $set: { "comments.$.text": "Test comment" } })	UPDATE "Comment" SET text = ''Updated comment' WHERE id = 10000006 RETURNING id
9	Видалити запис коментаря	db.posts.updateOne({ _id: 5447680 }, { $pull: { comments: { id: 1 } } })	DELETE FROM "Comment" WHERE id = 10000006 RETURNING id

10	Видалити запис посту	db.posts.deleteOne({ _id: 5447680 })	DELETE FROM "Post" WHERE id = 5447680 RETURNING id

11	Комплексний запит	[{
		"$match": {
		"title": { "$regex": "a", "$options": "i" }
		# аналог ILIKE '%test%'
		}
		},
		{
		"$project": {
		"id": 1,
		"title": 1,
		"comment_count": { "$size": {
		"$ifNull": ["$comments", []] } }
		}
		},
		{
		"$match": {
		"comment_count": { "$gt": 2 }
		}
		},
		{
		"$sort": {
		"comment_count": -1
		}
		},
		{
		"$limit": 100
		}
		]
			SELECT p.id, p.title, COUNT(c.id) AS comment_count
			FROM "Post" p
			LEFT JOIN "Comment" c ON p.id = c.post_id
			WHERE p.title ILIKE '%test%'
			GROUP BY p.id
			HAVING COUNT(c.id) > 2
			ORDER BY comment_count DESC
			LIMIT 100;

12	Запит в транзакції	with session.start_transaction():
		  mongo_post_collection.insert_one({
		  "title": "Test title",
		  "content": "Test content",
		  "comments": []
		  }, session=session)
		
		  raise Exception("Force rollback")
			BEGIN
			INSERT INTO "Post" (title, content)
			VALUES (%s, %s)", ("Test title", "Test content")
			ROLLBACK

13	Запит з віконною функцією	[
		  {
		  "$setWindowFields": {
		  "sortBy": {"_id": 1},
		  "output": {
		  "rank": {
		  "$documentNumber": {}
		  }
		  }
		  }
		  },
		  {
		  "$match": {
		  "rank": {"$lte": 10}
		  }
		  }
		  ]
			WITH ranked_posts AS (
			  SELECT id, title,
			ROW_NUMBER() OVER (ORDER BY id) AS rank
			  FROM "Post"
			  )
			  SELECT * FROM ranked_posts
			WHERE rank <= 10;

MongoDB демонструє високу гнучкість під час виконання простих запитів, оскільки структура документів не зобов'язує використання додаткових зв'язків між колекціями. Наприклад, витяг поста разом з усіма його коментарями виконується одним запитом до документа, що містить вкладене поле comments. З іншого боку, PostgreSQL використовує класичну реляційну модель, де взаємозв'язки між сутностями реалізовані через зовнішні ключі. Це забезпечує більший контроль над цілісністю даних, але вимагає об'єднання таблиць для отримання повної інформації.

У таблиці порівняно низку типових операцій, включаючи вибірки з пагінацією, пошук постів за користувачем, CRUD-операції з постами та коментарями, транзакції, а також складні аналітичні запити з фільтрацією та агрегацією. В табл. 2.2. подано час виконання запитів для обох баз даних.

Таблиця. 2.2

Час виконання запитів поданих в таблиці 2.1. для обох баз даних

Опис сценарію	Час виконання запиту у MongoDB, с.	Час виконання запиту у PostgreSQL, с.
Отримання 100 постів з пагінацією	1.17676 ± 0.02576	1.03067 ± 0.04580
Отримання постів одного користувача	2.98306 ± 0.05252	1.42803 ± 0.31125
Отримання 1 поста	0.00069 ± 0.00011	0.00051 ± 0.00020
Отримання одного поста з пов'язаними коментарями	0.00064 ± 0.00004	1.59965± 0.23827
Створити новий запис посту	0.00981 ± 0.00401	0.00236 ± 0.00031
Створити новий запис коментаря	0.00207 ± 0.00028	0.00091 ± 0.00018
Оновити запис посту	0.00185 ± 0.00027	0.00091 ± 0.00009
Оновити запис коментаря	0.00191 ± 0.00027	0.00066 ± 0.00006

Видалити запис коментаря	0.00167 ± 0.00024	0.00059 ± 0.00005
Видалити запис посту	0.00183 ± 0.00030	1.19369 ± 0.38362
Комплексний запит	15.98925 ± 0.51079	26.36591 ± 2.06383
Запит в транзакції	0.00121 ± 0.00021	0.00194 ± 0.00052
Запит з віконною функцією	18.71325 ± 0.59879	0.03948 ± 0.00169

На рисунку 2.5 представлена візуальне зображення зібраних даних з метою порівняльного аналізу. Можемо побачити, що при відсутності індексів PostgreSQL має перевагу у більшості операцій, окрім отримання публікації з коментарями та видалення коментаря. Також спостерігається невелика перевага MongoDB у комплексних запитах та запиті з транзакцією. Враховуючи, що з точки зору бізнес-логіки коментарі та публікації тісно пов’язані між собою, є сенс тримати ці дані в одній колекції.

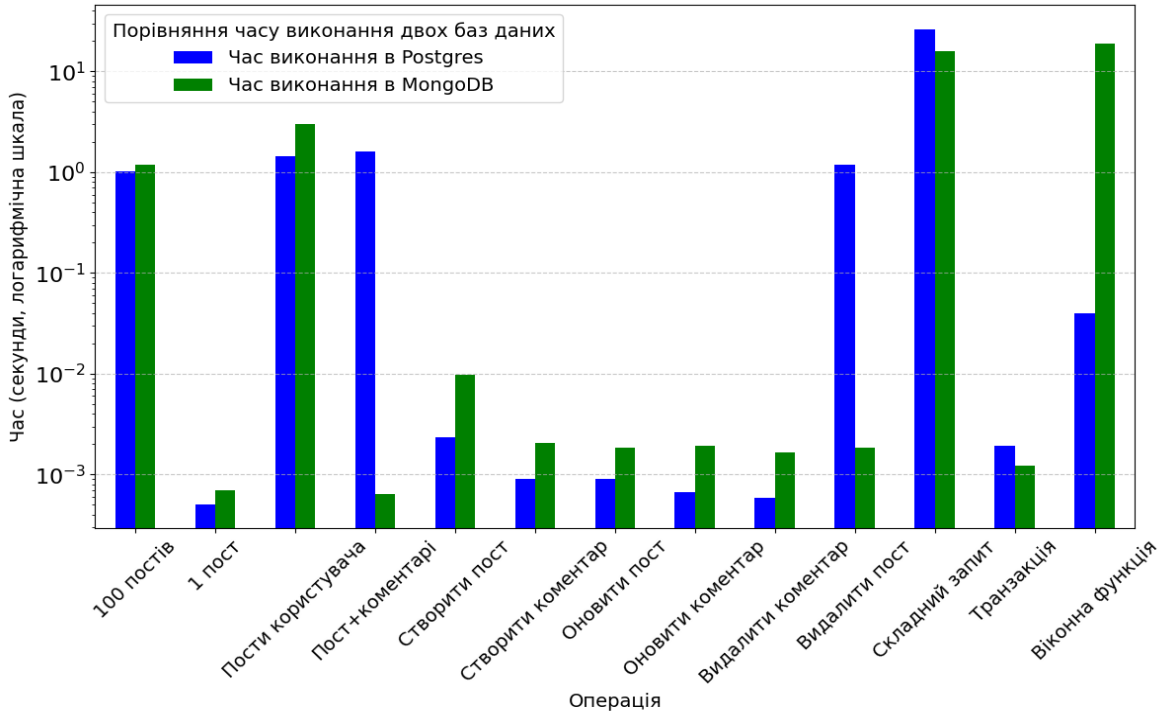

Рисунок 2.5 – Порівняння часу виконання запитів у двох базах даних без індексів.

2.5.3. Вплив індексації на швидкість виконання запитів

Для оцінки ефективності індексації у базах даних MongoDB та PostgreSQL було проведено серію замірів часу виконання ідентичних запитів, наведених у таблицях вище. Варто врахувати, що індекси для первинних ключів сутностей (користувачів, публікацій, та коментарів) були створені автоматично та вже існували під час додавання нових індексів. Усі тести виконувались після попереднього створення індексів над такими атрибутами (рис. 2.6):

- user_id, title для сутності публікацій;
- post_id для таблиці коментарів.

```
def add_indexes_to_postgres():
    with engine.connect() as connection:
        connection.execute("CREATE INDEX IF NOT EXISTS idx_post_user_id ON \"Post\"
        (user_id);")
        connection.execute("CREATE INDEX IF NOT EXISTS idx_comment_post_id ON \"Comment\"
        (post_id);")
        # Індекс для пошуку по заголовку постів з використанням pg_trgm
        connection.execute("CREATE EXTENSION IF NOT EXISTS pg_trgm;")
        connection.execute("CREATE INDEX IF NOT EXISTS idx_post_title ON \"Post\" USING gin
        (title gin_trgm_ops);")
```

Рисунок 2.6 – Python функція, в якій виконуються команди створення індексів в PostgreSQL.

Для того, щоб MongoDB отримала ідентичний набір індексів, для колекції posts було створено індекси над такими атрибутами: user_id, comments.id (так як він не був створений автоматично), а також title для текстового пошуку (рис. 2.7).

```
def add_indexes_to_mongo():
    mongo_post_collection.create_index([("user_id", 1)])
    mongo_post_collection.create_index([("comments.id", 1)])
    # Індекс для пошуку по заголовку постів
    mongo_post_collection.create_index([("title", "text")])
```

Рисунок 2.7 – Python функція, в якій виконуються команди створення індексів в MongoDB.

Окрім цього, запити для отримання публікацій з пагінацією зазнали змін, для того щоб використовувались проіндексовані дані. У табл 2.3. наведено оновлена логіка для обох баз даних

Таблиця 2.3.

Запити для отримання публікацій з пагінацією з використанням проіндексованих даних

Запит MongoDB	**Запит PostgreSQL**
db.posts.find({"_id": {"$gt": POST_ID}}).limit(100)	SELECT * FROM "Post" WHERE id >= 5447680 limit 100;

У табл. 2.4. представлені результати часу виконання запитів для баз даних після створення індексів.

Таблиця 2.4.

Результати часу виконання запитів для баз даних після створення індексів

Опис сценарію	**Час виконання запиту у MongoDB, с.**	**Час виконання запиту у Postgres, с.**
Отримання 100 постів з пагінацією	0.00168 ± 0.00015	0.00126 ± 0.00014
Отримання постів одного користувача	0.00052 ± 0.00003	0.00068 ± 0.00016
Отримання 1 поста	0.00038 ± 0.00004	0.00048 ± 0.00015
Отримання одного поста з пов’язаними коментарями	0.00024 ± 0.00003	0.00063 ± 0.00016
Створити новий запис посту	0.00309 ± 0.00105	0.00928 ± 0.00309
Створити новий запис коментаря	0.00311 ± 0.00091	0.00546 ± 0.00319

Оновити запис посту	0.00276 ± 0.00114	0.00190 ± 0.00054
Оновити запис коментаря	0.00282 ± 0.00100	0.00173 ± 0.00057
Видалити запис коментаря	0.00254 ± 0.00087	0.00121 ± 0.00065
Видалити запис посту	0.00274 ± 0.00101	0.00132 ± 0.00066
Комплексний запит	4.80482 ± 0.08026	0.39880 ± 0.01132
Запит в транзакції	0.00108 ± 0.00006	0.00041 ± 0.00014
Запит з віконною функцією	15.98996 ± 0.23344	0.03777 ± 0.00075

Додавши індекси в обидві бази даних, спостерігається помітне пришвидшення виконання деяких запитів, але й водночас отримано сповільнення інших, що пояснюється додатковою логікою оновлення індексів при операціях створення, оновлення та видалення (рис. 2.8; рис. 2.9).

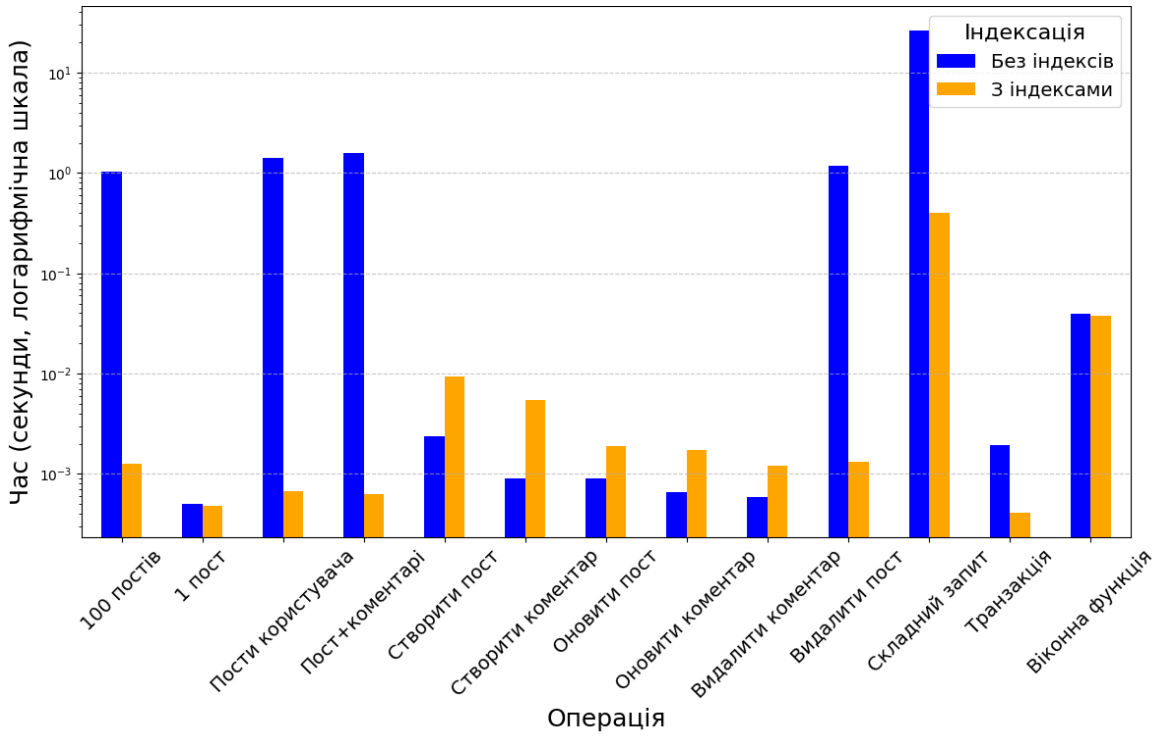

Рисунок 2.8 – Час виконання запитів в PostgreSQL до та після індексації.

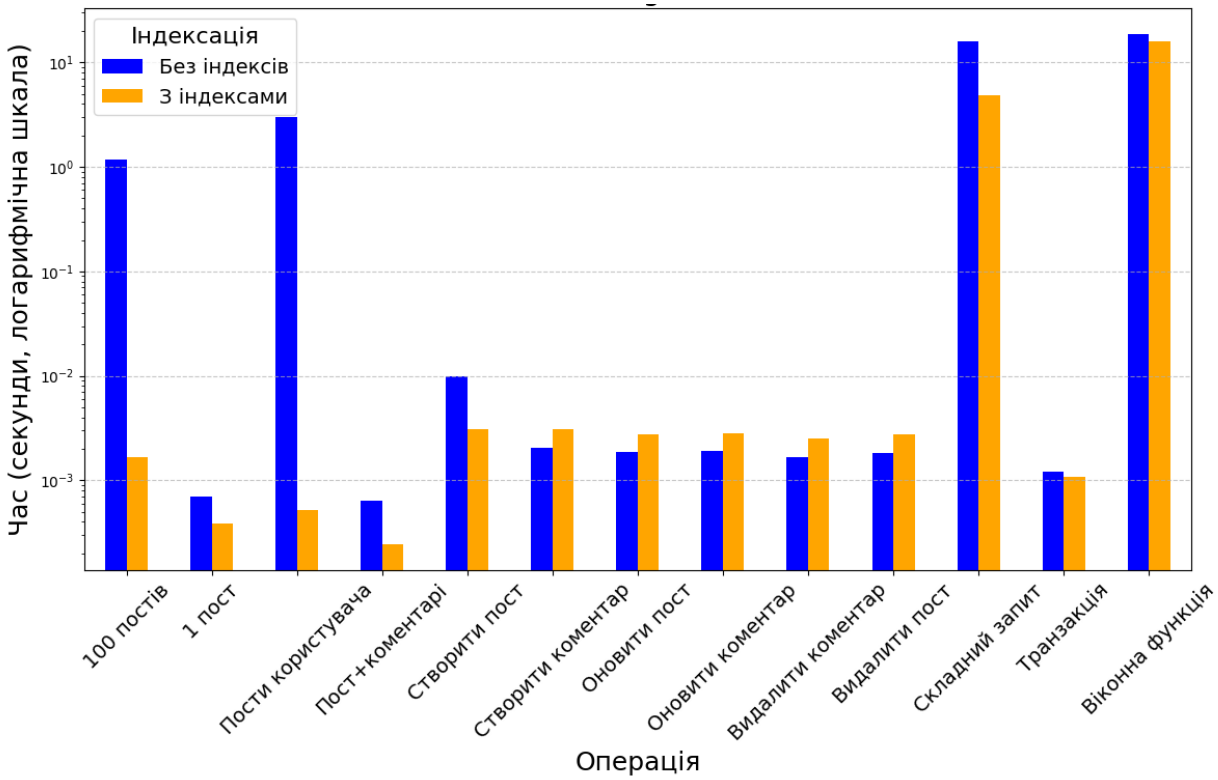

Рисунок 2.9 – Час виконання запитів в MongoDB до та після індексації.

Окрім цього, більшість операцій читання даних у MongoDB стали виконуватися швидше ніж в PostgreSQL (рис. 2.10), що знівелювало перевагу, яку мала PostgreSQL при відсутності індексів в обох базах. Якщо розглядати реальні сценарії користування додатком, то можна припустити, що користувачі будуть частіше робити запити на читання публікацій чи коментарів, ніж їх створення та редагування. Відповідно, важливо забезпечити максимальну ефективність отримання даних. Проте, реляційна база краще показала себе у виконанні комплексних і аналітичних запитів, що включають в себе транзакції, віконні функції та агрегації. Ці сценарії теж важливі при умові, що логіка додатку може ставати складнішою та потребуватиме більш комплексних запитів та постійну аналітику.

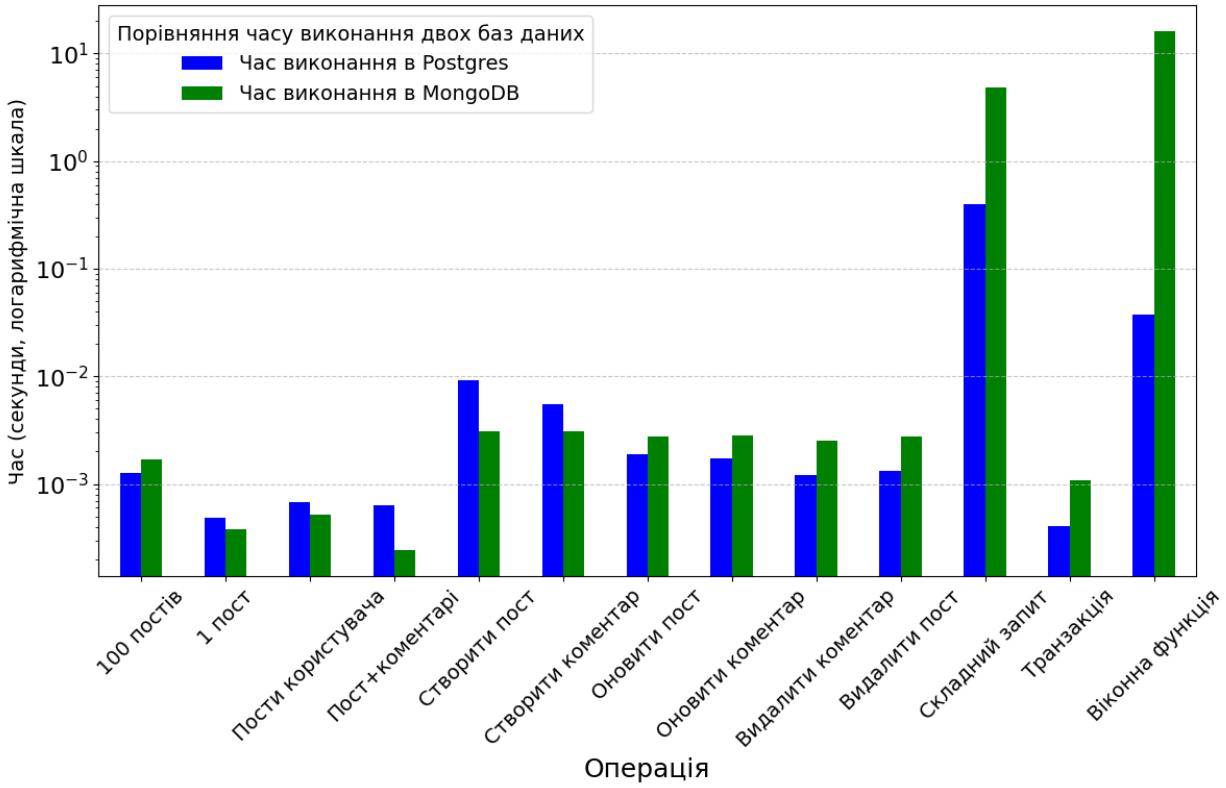

Рисунок 2.10 – Час виконання запитів в PostgreSQL та MongoDB з індексацією.

На рисунку 2.11 чітко видно, що індексація відіграє важливу роль у підвищенні продуктивності обох баз даних. Найбільше покращення помітне при виконанні запитів на вибірку великої кількості даних, таких як отримання вибірки публікацій або отримання дописів користувача. Це підтверджує важливість підбору правильної індексації, особливо при роботі з великими обсягами даних.

Порівняння часу виконання запитів в MongoDB та PostgreSQL показує, що MongoDB загалом краще працює з даними, що тісно між собою пов’язані, такі як публікації та коментарі. Це пов’язано з гнучкою документною структурою MongoDB, яка дозволяє працювати з вкладеними об’єктами без потреби у складних зв’язках між таблицями.

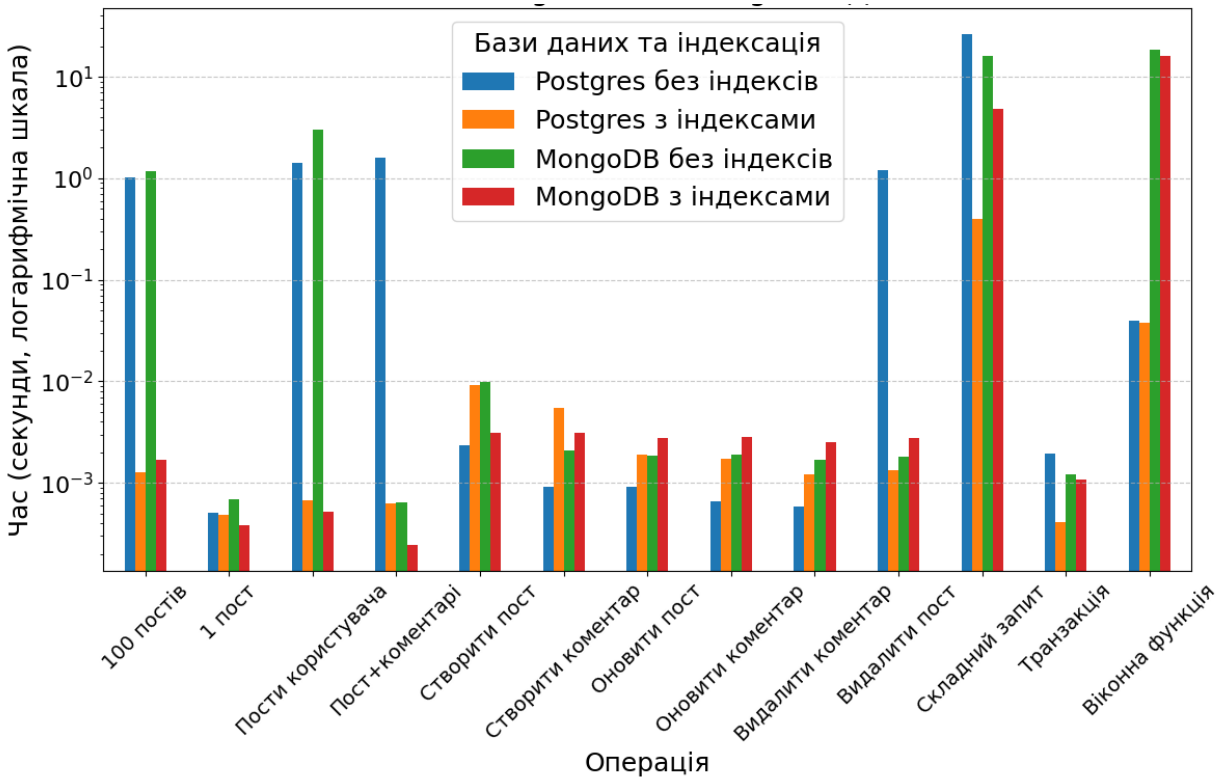

Рисунок 2.11 – Порівняння часу виконання операцій в двох базах даних при таких сценаріях: відсутність та наявність індексів.

Водночас PostgreSQL демонструє суттєву перевагу у виконанні складних запитів. Наприклад, у запиті з віконною функцією час виконання у PostgreSQL значно менший у порівнянні з MongoDB, навіть після індексації. Схожа ситуація спостерігається і у випадку комплексного запиту з агрегацією, фільтрацією та сортуванням, де реляційний підхід PostgreSQL демонструє більшу ефективність.

ВИСНОВКИ

У процесі дослідження було проведено порівняльний аналіз реляційної бази даних PostgreSQL та нереляційної бази MongoDB на прикладі взаємодії з вебдодатком для ведення персонального блогу. Досліджено ефективність обох підходів до зберігання та обробки даних.

Для кожної бази даних була розроблена архітектура для проведення тестування продуктивності стандартних CRUD-операцій, а також складних аналітичних запитів, включно з транзакціями та віконними функціями. Тестування проводилось як для баз без індексів, так і з попередньо створеними індексами. Дані збирались багаторазово, середній час та стандартне відхилення було розраховано для кожної операції, а потім результати порівнювались графічно та чисельно.

Результати аналізу підтверджують, що вибір бази даних має ґрунтуватися на особливості задачі. MongoDB є доцільним вибором для роботи з даними, що потребують гнучкої структури та міцно взаємопов’язані з точки зору бізнес-логіки. PostgreSQL, у свою чергу, забезпечує високу ефективність при виконанні складних аналітичних запитів і є оптимальним рішенням для систем, що потребують точності, надійності та складного механізму запитів. У контексті розробленого застосунку, доцільно є розглянути можливість використання MongoDB для роботи з публікаціями та коментарями, поступившись ефективністю складних та аналітичних запитів при умові, що даний функціонал не планується ускладнювати. Окрім цього, дана база даних дозволяє застосувати гнучке рішення із вкладеними коментарями, що набагато складніше реалізувати в реляційних базах даних. Натомість, PostgreSQL можна залишити для керування даними користувача та використати надійність та ефективність даної бази даних для розширення можливостей додатку, наприклад: інтеграція платіжних систем, інших соціальних мереж та застосунків.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Пєтух А.М. Бази даних. Мови запитів, управління транзакціями, розподілена обробка даних. URL: https://web.posibnyky.vntu.edu.ua/fitki/11petuh_bazdanyh_movy_zalitiv/31.htm (дата звернення: 01.05.2025).
2. 5 common web attacks: how to exploit and defend against them. *Hack The Box*. URL: https://www.hackthebox.com/blog/5-common-web-attacks (date of access: 06.03.2025).
3. 9.16. JSON functions and operators. *PostgreSQL Global Development Group*. URL: https://www.postgresql.org/docs/current/functions-json.html (date of access: 08.03.2025).
4. Bansal P. Kafka: powering real-time data streaming, its limitations, and learnin g resources. *Medium*. URL: https://medium.com/@prateekbansalind/kafka-powering-real-time-data-streaming-its-limitations-and-learning-resources-4e0973a5329 (date of access: 10.03.2025).
5. Columnar databases (use cases, examples, advantages & disadvantages). *DatabaseTown*. URL: https://databasetown.com/columnar-databases/ (date of access: 02.05.2025).
6. Database basics and types: relational vs non-relational. *Salesforce Trailhead*. URL: https://trailhead.salesforce.com/content/learn/modules/big-data-strategy/understand-the-basics-of-databases (date of access: 13.03.2025).
7. Distributed database (goals, types, advantages and disadvantages). *DatabaseTown*. URL: https://databasetown.com/distributed-database-goals-types-advantages-and-disadvantages/ (date of access: 01.05.2025).
8. Gillis A. S., Botelho B. What is MongoDB? Features and how it works. URL: https://www.techtarget.com/searchdatamanagement/definition/MongoDB (date of access: 08.03.2025).

9. Graph database (use cases, examples and properties). *DatabaseTown*. URL: https://databasetown.com/graph-database (date of access: 02.05.2025).
10. How nosql and SQL will work together (and why that matters). *Code With C*. URL: https://www.codewithc.com/how-nosql-and-sql-will-work-together-and-why-that-matters/ (date of access: 08.03.2025).
11. Key-Value database (use cases, list, pros & cons). *DatabaseTown*. URL: https://databasetown.com/key-value-database-use-cases (date of access: 01.05.2025).
12. Lee C. H. Understanding redis data persistence: ensuring data safety during fail ures. *Medium*. URL: https://medium.com/@hoon33710/understanding-redis-data-persistence-ensuring-data-safety-during-failures-f18394832c5d (date of access: 14.03.2025).
13. Miller A. Cloud databases vs. on-premises databases: which is right for you?. URL: https://www.code-beavers.com/blog/cloud-databases-vs-on-premises-databases/ (date of access: 08.04.2025).
14. Navigational DBMS. *DBMS Work*. URL: https://dbmswork.blogspot.com/2008/03/navigational-dbms.html (date of access: 08.03.2025).
15. Patra S. K. NoSQL databases overview: types, use cases, and key differences. URL: https://www.csinfo360.com/2024/09/nosql-databases-overview-types-use.html (date of access: 10.04.2025).
16. Roohitavaf M. In-memory vs. on-disk databases. *MyDistributed.Systems*. URL: https://www.mydistributed.systems/2020/07/an-overview-of-storage-engines.html (date of access: 01.05.2025).
17. Saxena S. System design of netflix. *Medium*. URL: https://saxenasanket.medium.com/system-design-of-netflix-part-1-4d65642ed738 (date of access: 08.05.2025).

18. Shubham S. Netflix subscribers statistics 2025 (by country & demographics). *DemandSage*. URL: https://www.demandsage.com/netflix-subscribers/ (date of access: 08.05.2025).
19. Shyshkov O. What do big websites like Facebook, Google, Twitter, and LinkedIn use for their database? – Oleksii Shyshkov's blog. *Oleksii Shyshkov's blog – Oleksii Shyshkov*. URL: https://oshyshkov.com/2019/01/05/what-do-big-websites-like-facebook-google-twitter-and-linkedin-use-for-their-database/ (date of access: 02.05.2025).
20. Skrgat A. In-memory vs. disk-based databases: why do you need a larger than memory architecture?. URL: https://memgraph.com/blog/in-memory-vs-disk-based-databases-larger-than-memory-architecture (date of access: 07.05.2025).
21. SQL vs nosql: which database is best for horizontal scaling, sharding, and replication?. *Infonikhil*. URL: https://medium.com/@infonikhil270/sql-vs-nosql-which-database-is-best-for-horizontal-scaling-sharding-and-replication-798947d8c1a (date of access: 14.03.2025).
22. TOPDB Top Database index. *PYPL*. URL: https://pypl.github.io/DB.html (date of access: 10.04.2025).
23. Welcome to flask – flask documentation (2.3.x). *Pallets Projects*. URL: https://flask-docs.readthedocs.io/en/latest/ (date of access: 10.04.2025).
24. What does ACID mean in Database Systems?. *Database.Guide*. URL: https://database.guide/what-is-acid-in-databases/ (date of access: 05.03.2025).
25. What is centralized database? | functions, advantages & disadvantages. *DatabaseTown*. URL: https://databasetown.com/centralized-database-functions-advantages (date of access: 19.04.2025).
26. What is cloud database? (advantages & disadvantages) 7 popular cloud databases. *DatabaseTown*. URL: https://databasetown.com/cloud-database-advantages-disadvantages/ (date of access: 19.04.2025).

27. What is document database? (document oriented database) uses cases, operations, model. *DatabaseTown*. URL: https://databasetown.com/what-is-document-database (date of access: 01.05.2025).
28. What is SQL. *SQL Tutorial*. URL: https://www.sqltutorial.org/what-is-sql/ (date of access: 10.04.2025).
29. Yusuf M. Redis Explained. URL: https://architecturenotes.co/p/redis (date of access: 15.05.2025).

ДОДАТКИ

Додаток А

```python
def create_users():
    # Створення користувачів в пакетах з метою оптимального використання ресурсів
    for offset in range(0, num_of_users, step):
        # Генерація користувачів з унікальними іменами та емейлами
        users = [User(username=f"{fake.user_name()}{num+last_user_id}",
                      email=f"{num+last_user_id}{fake.email()}",
                      password=fake.password(),
                      image_file=fake.file_name()
                     )
                 for num in range(num_of_users)]
        # Перевірка, чи є користувачі для додавання
        if not users:
            break
        # Додавання користувачів до сесії
        session.bulk_save_objects(users)


def create_posts(user_ids):
    # Створення постів в пакетах з метою оптимального використання ресурсів
    for offset in range(0, num_of_posts, step):
        # Генерація постів з випадковим текстом та випадковими користувачами, що вже існують в базі даних
        posts = [Post(user_id=random.choice(user_ids), title=fake.sentence(), content=fake.text())
                 for _ in range(offset, offset+step)]
        # Перевірка, чи є пости для додавання
        if not posts:
            break
        # Додавання постів до сесії
        session.bulk_save_objects(posts)
        print(f"Created batch of posts: {offset}")


def create_comments(post_ids, user_ids):
    # Створення коментарів для випадкової публікації та користувача, які вже існують в базі даних
    comments = [Comment(post_id=random.choice(post_ids), user_id=random.choice(user_ids),
                        text=fake.text()) for _ in range(num_of_comments)]
    # Перевірка, чи є коментарі для додавання
    if not comments:
        return
    # Додавання коментарів до сесії
    session.bulk_save_objects(comments)


if __name__ == "__main__":
    # Створення таблиць, якщо вони ще не існують
    create_users()
    session.commit()

    # Отримання всіх користувачів з бази даних
    user_ids = [user.id for user in session.query(User.id).all()]
    # Створення постів для існуючих користувачів
    create_posts(user_ids)
    session.commit()

    # Отримання всіх постів з бази даних
    post_ids = [post.id for post in session.query(Post.id).all()]
    # Створення коментарів для існуючих постів та користувачів
    create_comments(post_ids, user_ids)
    session.commit()
```

Рисунок А.1 – Python скрипт для створення випадкових даних в PostgreSQL.

```python
def insert_users(batch_size=1000):
    # Отримання кількості користувачів з PostgreSQL
    users_count = session.query(User).count()

    for offset in range(0, users_count, batch_size):
        mongo_users = []
        # Отримання даних користувачів з PostgreSQL
        users = session.query(User).offset(offset).limit(batch_size).all()
        for user in users:
            # Створення моделі користувача для MongoDB
            user_model = UserModel(
                id=user.id,
                username=user.username,
                email=user.email,
                image_file=user.image_file,
                password=user.password,
            )
            # Конвертація моделі в словник для створення в MongoDB
            mongo_users.append(user_model.model_dump())
        # Запит на створення користувачів в MongoDB
        mongo_user_collection.insert_many(mongo_users)
```

Рисунок А.2 – Python функція для копіювання даних користувачів з PostgreSQL в MongoDB.

```python
def insert_posts_with_comments(batch_size=1000000, start=0):
    # Отримання існуючих постів з MongoDB, щоб уникнути дублікатів
    existing_posts = mongo_post_collection.find({}, {"_id": 1})
    existing_posts_set = {post["_id"] for post in existing_posts}
    del existing_posts # звільнення пам'яті

    # Отримання кількості постів з PostgreSQL
    pg_posts_count = session.query(Post).count()

    for offset in range(start, pg_posts_count, batch_size):
        # Отримання даних постів з PostgreSQL
        posts = session.query(Post).order_by(Post.id.asc()).offset(offset).limit(batch_size).all()

        posts_to_create = defaultdict(dict)
        for post in posts:
            if post.id in existing_posts_set:
                continue
            post_model = PostModel(
                id=post.id,
                title=post.title,
                date_posted=post.date_posted,
                content=post.content,
                user_id=post.user_id
            )
            # додавання посту як словник для створення в MongoDB
            posts_to_create[post.id] = post_model.model_dump(by_alias=True)

        del posts

        if not posts_to_create:
            print(f"No new posts to insert. Size: {offset} - {offset + batch_size}")
            continue

        # Отримання коментарів для постів, які ми створюємо
        query = f"""
        SELECT c.id, c.user_id, c.post_id, c.text, c.created_at
        FROM "Comment" c
        WHERE c.post_id IN ({', '.join(str(post_id) for post_id in posts_to_create.keys())})
        """

        data = session.execute(query).fetchall()
        # Формування даних коментарів для постів
        for comment in data:
            comment_model = CommentModel(
                id=comment[0],
                user_id=comment[1],
                post_id=comment[2],
                text=comment[3],
                created_at=comment[4]
            )
            # Додавання коментаря до відповідного посту
            posts_to_create[comment_model.post_id]["comments"].append(comment_model.model_dump())

        del data # звільнення пам'яті

        if posts_to_create:
            # Додавання постів з коментарями до MongoDB
            posts_to_insert = list(posts_to_create.values())
            mongo_post_collection.insert_many(posts_to_insert)
            del posts_to_insert # звільнення пам'яті
            # Збережемо ідентифікатори постів, які ми додали,
            # щоб уникнути дублікатів в наступних запитах
            existing_posts_set.update(set(posts_to_create.keys()))
            del posts_to_create # звільнення пам'яті
```

Рисунок А.3 – Python функція для копіювання даних публікацій та коментарів з PostgreSQL в MongoDB.

```python
uuid_str = uuid.uuid4().hex
pg_result_file = f"operations_results/postgres_results_{uuid_str}.txt"
pg_f = open(pg_result_file, "w")

mongo_result_file = f"operations_results/mongo_results_{uuid_str}.txt"
mongo_f = open(mongo_result_file, "w")


def timing_decorator(message: str, output_file: io.TextIOWrapper):
    """Декоратор для вимірювання часу виконання функції та запису результатів у файл."""
    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            start = perf_counter()
            result = func(*args, **kwargs)
            end = perf_counter()
            # Формування рядка з результатами
            string = f"{message}: Execution time: {end - start:.10f} seconds"
            output_file.write(string + "\n") # Запис результату в файл
            return result
        return wrapper
    return decorator



def execute_postgres_query(query):
    """
    Надсилає запит до Postgres та повертає результати.
    """
    cursor.execute(query)
    return cursor.fetchall()


@timing_decorator("Retrieve all posts from Postgres paginated", pg_f)
def get_all_posts():
    get_posts_query = f"""SELECT * FROM "Post" limit 100 offset {POST_ID};"""
    return execute_postgres_query(get_posts_query)
```

Рисунок А.4 – Python скрипт для виконання запиту, вимірювання часу виконання та збереження результату у файл за допомогою декоратора.

```
#!/bin/bash

# Параметри PostgreSQL
POSTGRES_CONTAINER="postgres"
POSTGRES_DB="blog_db"
POSTGRES_USER="postgres"
POSTGRES_PASS="1234"
POSTGRES_BACKUP="postgres_backup.sql"

# Параметри MongoDB
MONGO_CONTAINER="mongo1"
MONGO_DB="blog_db"
MONGO_BACKUP_DIR="mongo_backup"

# Створення бекапу PostgreSQL
echo "Dumping PostgreSQL..."
docker exec -e PGPASSWORD="$POSTGRES_PASS" $POSTGRES_CONTAINER pg_dump \
  -U $POSTGRES_USER -d $POSTGRES_DB > $POSTGRES_BACKUP

# Створення бекапу MongoDB
echo "Dumping MongoDB..."
docker exec $MONGO_CONTAINER mongodump --db $MONGO_DB --archive > "$MONGO_BACKUP_DIR.archive"

# Архівування
echo "Compressing backups..."
tar -czf postgres_backup.tar.gz $POSTGRES_BACKUP
tar -czf mongo_backup.tar.gz $MONGO_BACKUP_DIR.archive

# Виведення розміру
POSTGRES_SIZE=$(du -sh postgres_backup.tar.gz | cut -f1)
MONGO_SIZE=$(du -sh mongo_backup.tar.gz | cut -f1)
echo "PostgreSQL backup size: $POSTGRES_SIZE"
echo "MongoDB backup size: $MONGO_SIZE"
echo "Done."
```

Рисунок А.5 – Bash скрипт для створення резервних копій PostgreSQL та MongoDB, їх архівування та виведення результату в термінал.

Анотація

Никитюк А.В. – Порівняльний аналіз ефективності реляційних та нереляційних баз даних у вебзастосунках.

Кваліфікаційна робота на здобуття освітнього ступеня «бакалавр» за спеціальністю 122 Комп'ютерні науки, освітньої програми Комп'ютерні науки та інформаційні технології. – Волинський національний університет імені Лесі Українки. – 2025 р.

У роботі проведено порівняльний аналіз реляційних та нереляційних баз даних на прикладі розробленого вебдодатку. Як приклад реляційної СУБД була обрана PostgreSQL, а MongoDB виступила прикладом нереляційного рішення. Також реалізовано вебдодаток для ведення персонального блогу, який має можливість одночасно взаємодіяти з двома базами даних з ідентичною логікою. Досліджено час виконання найпоширеніших запитів різної складності. Показники продуктивності обох баз даних було зібрано до та після додавання індексів. Також порівняно розмір баз даних та їх стиснених резервних копій, з метою оцінити ефективність використання пам'яті. PostgreSQL підтвердила свою надійність та ефективність, особливо у запитах із складною логікою, проте за допомогою індексів можна досягнути ефективного рішення й в MongoDB, що на додачу забезпечує роботу з вкладеними документами та гнучкість структури даних. Проте, дана база значно програє PostgreSQL в роботі із складними аналітичними задачами. Результати підтверджують доцільність використання гібридного підходу в розробці сучасних вебзастосунків, де одночасно можуть використовуватись як реляційні, так і нереляційні СУБД залежно від характеру даних.

Ключові слова: PostgreSQL, MongoDB, SQL, NoSQL, реляційна база даних, нереляційна база даних, вебдодаток, Python, гібридна архітектура