

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ «ЖИТОМИРСЬКА ПОЛІТЕХНІКА»
ФАКУЛЬТЕТ ІНФОРМАЦІЙНО-КОМП'ЮТЕРНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

ПОЯСНЮВАЛЬНА ЗАПИСКА

до кваліфікаційної роботи освітнього ступеня «магістр»
за спеціальністю 121 «Інженерія програмного забезпечення»
(освітня програма «Інженерія програмного забезпечення»)

на тему:

**«Дослідження оптимізації роботи БД для
високонавантажених платформ»**

Виконав студент групи ПЗм-23-1
Туровський Ярослав Сергійович

Керівник роботи:
Чижмотря Олена Геннадіївна

Рецензент:
Вакалюк Тетяна Анатоліївна

Житомир – 2024

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ «ЖИТОМИРСЬКА ПОЛІТЕХНІКА»
ФАКУЛЬТЕТ ІНФОРМАЦІЙНО-КОМП'ЮТЕРНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

«ЗАТВЕРДЖУЮ»

Зав. кафедри інженерії
програмного забезпечення
Тетяна ВАКАЛЮК
«01» жовтня 2024 р.

ЗАВДАННЯ
на кваліфікаційну роботу

Здобувач вищої освіти: **ТУРОВСЬКИЙ Ярослав Сергійович**

Керівник роботи: **ЧИЖМОТРЕ Олена Геннадіївна**

Тема роботи: **«Дослідження оптимізації роботи БД для високонавантажених платформ»**,

затверджена Наказом закладу вищої освіти від **«01» жовтня 2024 р., №520/с**

Вихідні дані для роботи: бази даних, що використовуються у високонавантажених платформах, методи та підходи до оптимізації роботи баз даних для підвищення їхньої продуктивності в умовах інтенсивного навантаження.

Керівники / консультанти з магістерської кваліфікаційної роботи із зазначенням розділів, що їх стосуються:

Розділ	Консультант	Завдання видав	Завдання прийняв
1	Чижмотря О.Г.	01.10.2024р.	01.10.2024р.
2	Чижмотря О.Г.	01.10.2024р.	01.10.2024р.
3	Чижмотря О.Г.	01.10.2024р.	01.10.2024р.

РЕФЕРАТ

Кваліфікаційна магістерська робота присвячена дослідженню методів оптимізації роботи баз даних у високонавантажених платформах. Основною метою роботи є аналіз сучасних підходів до оптимізації баз даних та оцінка їх ефективності для покращення продуктивності, надійності та масштабованості цифрових систем.

У роботі проведено аналіз предметної галузі високонавантажених платформ, розглянуто ключові характеристики та проблеми, що виникають у їхній роботі. Досліджено сучасні методи оптимізації баз даних, зокрема реплікацію, шардинг, індексацію таблиць і пакетні оновлення. Реалізовано тестові сценарії для оцінки ефективності зазначених методів та запропоновано рекомендації для їх інтеграції в реальні системи. Пояснювальна записка до кваліфікаційної магістерської роботи складається зі вступу, трьох розділів, списку використаних джерел з 42 найменування та 33 рисунки. Повний обсяг роботи становить 81 сторінку.

КЛЮЧОВІ СЛОВА: БАЗА ДАНИХ, ОПТИМІЗАЦІЯ, РЕПЛІКАЦІЯ, ШАРДИНГ, ІНДЕКСАЦІЯ, ПАКЕТНІ ОНОВЛЕННЯ.

ABSTRACT

The qualification master's thesis is dedicated to the study of database optimization methods for high-load platforms. The primary goal of the work is to analyze modern approaches to database optimization and evaluate their effectiveness in improving the performance, reliability, and scalability of digital systems.

The thesis analyzes the subject area of high-load platforms, considers the key characteristics and problems that arise in their operation. It explores modern methods of database optimization, including replication, sharding, table indexing, and batch updates. Test scenarios were implemented to evaluate the efficiency of these methods, and recommendations were proposed for their integration into real systems. The explanatory note to the qualification master's thesis consists of an introduction, three chapters, a list of references with 42 titles, and 33 figures. The total volume of the work is 81 pages.

KEYWORDS: DATABASE, OPTIMIZATION, REPLICATION, SHARDING, INDEXING, BATCH UPDATES.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ	6
ВСТУП.....	7
РОЗДІЛ 1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ.....	10
1.1 Аналіз особливостей високонавантажених платформ та їх характеристик	10
1.2 Огляд типових прикладів високонавантажених платформ.....	12
1.3 Постановка задачі дослідження.....	15
Висновки до першого розділу	18
РОЗДІЛ 2 МЕТОДИ ОПТИМІЗАЦІЇ РОБОТИ БД ДЛЯ ВИСОКОНАВАНТАЖЕНИХ ПЛАТФОРМ	19
2.1 Аналіз реляційних баз даних у контексті високонавантажених платформ	19
2.2 Реплікація баз даних	22
2.3 Шардинг.....	24
2.4 Пакетні оновлення.....	27
2.5 Індексація таблиць	30
Висновки до другого розділу	34
РОЗДІЛ 3 РЕАЛІЗАЦІЯ ТА АНАЛІЗ ЕФЕКТИВНОСТІ МЕТОДІВ ОПТИМІЗАЦІЇ	36
3.1 Вибір інструментальних засобів та технологій реалізації	36
3.2 Реалізація та аналіз ефективності реплікації баз даних	37
3.3 Реалізація та аналіз ефективності шардингу	48
3.4 Реалізація та аналіз ефективності пакетних оновлень	53
3.5 Реалізація та аналіз ефективності індексації таблиць	58
Висновки до третього розділу	66
ВИСНОВКИ	68
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ.....	70
ДОДАТКИ	74

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

СКБД – система керування базами даних – набір взаємопов'язаних даних (база даних) і програм для доступу до цих даних.

SQL – Structured Query Language – мова структурованих запитів для роботи з базами даних.

ACID – Atomicity, Consistency, Isolation, Durability – набір властивостей для забезпечення надійності транзакцій у базах даних.

B-Tree – Balanced Tree – структура даних для ефективного зберігання та пошуку.

BRIN – Block Range Index – індекс для великих таблиць, що зберігає мінімальні та максимальні значення блоків.

ID – Identifier – унікальний ідентифікатор запису.

E-commerce – Electronic Commerce – електронна комерція, онлайн-торгівля товарами та послугами через Інтернет.

ВСТУП

Сучасний світ неможливо уявити без високонавантажених платформ, які щодня обслуговують мільйони користувачів і працюють зі значними обсягами даних. Соціальні мережі, інтернет-магазини, фінансові сервіси та інші подібні системи стали невід'ємною частиною нашого життя, забезпечуючи безперервний доступ до інформації, різних послуг чи розваг. Ці платформи мають критичне значення, оскільки підтримують соціальні та економічні процеси, забезпечуючи стабільність і ефективність функціонування навіть при інтенсивному навантаженні.

В умовах постійного зростання кількості користувачів та обсягів транзакцій, виникають великі виклики до продуктивності та надійності таких платформ. Стабільна робота бази даних, що забезпечує швидку обробку запитів, надійне зберігання та швидкий доступ до даних, є ключовим елементом у підтримці загальної ефективності системи. Підвищення навантаження вимагає гнучких підходів до використання баз даних, їх проектування та управління.

Актуальність дослідження зумовлена стрімким розвитком високонавантажених платформ та їхньою інтеграцією у більшість сфер людської діяльності. Забезпечення стабільності, масштабованості та ефективності баз даних у цих системах є необхідною умовою для подальшого розвитку сучасних цифрових технологій. Вивчення та впровадження сучасних методів оптимізації баз даних сприяють підвищенню продуктивності, зниженню витрат на ресурси та покращенню користувацького досвіду.

Метою роботи є аналіз особливостей роботи баз даних у високонавантажених платформах та визначення ключових факторів, що впливають на їхню продуктивність і стабільність. У рамках дослідження буде проаналізовано та реалізовано сучасні підходи до оптимізації роботи баз даних в умовах інтенсивного навантаження.

Для досягнення поставленої мети визначено такі основні завдання:

- Провести аналіз предметної галузі високонавантажених платформ та їхніх ключових характеристик
- Визначити типові проблеми, що виникають у роботі баз даних у таких системах.
- Дослідити та порівняти сучасні методи оптимізації баз даних: реплікацію, шардинг, індексацію, пакетні оновлення тощо.
- Оцінити ефективність впроваджених підходів на прикладі тестових сценаріїв.

Об'єктом дослідження є бази даних, що використовуються у високонавантажених платформах.

Предметом дослідження є методи та підходи до оптимізації роботи баз даних для підвищення їхньої продуктивності в умовах інтенсивного навантаження.

Наукова новизна дослідження полягає у вивченні, аналізі й практичній реалізації сучасних методів оптимізації баз даних. У роботі буде розглянуто різний спектр методів оптимізації баз даних. Ці методи будуть проаналізовані, впроваджені у тестові сценарії та оцінені з точки зору їхньої ефективності для високонавантажених платформ. Додатково буде розроблено рекомендації щодо покращення їхньої реалізації з метою підвищення стабільності й швидкодії баз даних.

Практичне значення роботи полягає в оцінці реальної ефективності методів оптимізації на прикладі моделювання типових сценаріїв навантаження. Результати дослідження можуть бути використані для розробки нових підходів до управління базами даних, інтеграції сучасних методів оптимізації у реальні платформи, а також зниження часу виконання запитів і забезпечення стабільності систем навіть за умов значного навантаження.

У роботі використовуються методи системного аналізу, математичного моделювання, а також експериментальні дослідження ефективності алгоритмів реплікації, шардингу, індексації та пакетних оновлень.

За темою кваліфікаційної магістерської роботи було опубліковано 2 тези конференції [1, 2].

РОЗДІЛ 1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ

1.1 Аналіз особливостей високонавантажених платформ та їх характеристик

У сучасному світі, розвиток цифрових технологій кардинально змінив підхід до обробки та зберігання даних. Зростання кількості користувачів та їх потреб стали ключовими факторами формування високонавантажених платформ (high-load platforms [3]). Ці платформи забезпечують швидкий доступ до послуг і даних, які щоденно використовуються мільйонами користувачів. Високонавантажені платформи мають попит у багатьох сферах сучасного життя: від соціальних мереж до фінансових сервісів, що є критично важливими для забезпечення стабільності глобальної економіки.

Високонавантажені платформи – це системи, які здатні підтримувати одночасну роботу великої кількості користувачів, забезпечуючи стабільну обробку величезного обсягу запитів і зберігання даних. Основною метою таких платформ є забезпечення продуктивності та безперебійної роботи незалежно від рівня навантаження.

Поняття «високе навантаження» характеризується кількома ключовими аспектами, які визначають ефективність роботи системи. Насамперед, це трафік, що відображає кількість одночасних користувачів, які взаємодіють із системою. Також важливо враховувати об'єм даних, що охоплює інформацію, яка передається, зберігається та обробляється додатком у режимі реального часу. Окрім цього, значущим показником є швидкість роботи, яка визначає час, необхідний для обробки запиту користувача або завершення транзакції.

Зважаючи на це, головним завданням таких додатків є забезпечення стабільної та безперебійної роботи, навіть за умов інтенсивного навантаження. Це можливо завдяки спеціальним підходам до архітектури системи, які включають розподіл навантаження, горизонтальне та вертикальне масштабування, а також налаштування СКБД, що підтримують зростаючі потреби користувачів.

У таких системах особливу увагу приділяють забезпеченню продуктивності, стабільності та безпеки, адже перебої в роботі можуть призвести до значних фінансових витрат, компенсацій або ж втрати довіри клієнтів.

Високонавантажені платформи мають ряд відмінних характеристик, які визначають їх функціональність. По-перше, це їх здатність адаптуватися до змін у навантаженні. Уявіть платформу електронної комерції підчас «чорної п'ятниці», коли кількість одночасних користувачів може зростати в десятки разів. Система повинна забезпечувати роботу навіть при таких умовах. Для цього впроваджуються спеціальні рішення, які дозволяють динамічно розширювати ресурси шляхом автоматизованого масштабування, моніторингу продуктивності та перенаправлення запитів на менш навантажені сервери.

Ще однією важливою рисою є надійність системи. У сучасних реаліях, коли кожна секунда простою може коштувати бізнесу мільйони, системи повинні забезпечувати безперервну роботу навіть за екстремальних обставин. Це досягається шляхом впровадження відмовостійких механізмів, таких як резервування даних, реплікація серверів тощо.

Продуктивність також відіграє критичну роль у функціонуванні високонавантажених платформ. Висока швидкість обробки запитів і мінімальні затримки є основними факторами, які визначають якість користувацького досвіду. Для цього широко застосовуються такі технології, як кешування даних і балансування навантаження між серверами.

Інфраструктура високонавантажених платформ має бути добре спланована для забезпечення стабільності роботи. Наприклад використання розподілених дата-центрів, розташованих у різних регіонах, дозволяє знизити затримки та забезпечити високу доступність сервісів для користувачів у всьому світі. Компанія Google [4] або Amazon [5], розробили свої системи таким чином, щоб кожен користувач, незалежно від географічного розташування, міг отримати доступ до сервісу з мінімальною затримкою.

Оскільки високонавантажені платформи часто працюють із чутливими даними, такими як особиста інформація чи платіжні дані, питання безпеки є

одним із найважливіших. Витік даних чи втручання третьої сторони у роботу системи можуть призвести до значних втрат репутації або фінансових збитків. Для мінімізації таких ризиків впроваджуються багаторівневі системи захисту, включаючи шифрування даних, багатофакторну аутентифікацію та регулярний моніторинг безпеки.

Також важливо зазначити що у сучасному світі персоналізація стала невід'ємною частиною високонавантажених платформ. Для забезпечення індивідуального підходу до кожного користувача використовуються алгоритми машинного навчання та аналізу великих даних. Такі системи дозволяють аналізувати поведінку користувачів у реальному часі та надавати релевантні рекомендації. Наприклад, у сфері електронної комерції персоналізовані рекомендації часто призводять до збільшення продажів, а в соціальних мережах підвищують інтерес аудиторії користувачів.

1.2 Огляд типових прикладів високонавантажених платформ

Застосування високонавантажених додатків охоплює численні галузі, у яких виникає необхідність обробки великих обсягів даних. Основним завданням таких додатків є забезпечення швидкого й стабільного доступу до інформації навіть при інтенсивному навантаженні. У цьому контексті бази даних відіграють ключову роль у підтримці продуктивності, забезпечуючи своєчасну обробку запитів, управління транзакціями та зберігання великих масивів інформації. Розглянемо специфіку високонавантажених додатків, функціонал яких тісно пов'язаний із роботою баз даних, у найбільш популярних галузях.

Фінансові платформи, включаючи банки, платіжні системи та брокерські сервіси, потребують високоякісних сервісів, які будуть забезпечувати максимальну надійність і швидкість операцій. Це формує особливий вид навантаження, оскільки кількість транзакцій, що обробляються одночасно, є дуже високою.

Платформи для торгівлі цінними паперами, акціями та іншими фінансовими інструментами повинні підтримувати низьку затримку обробки

операцій, оскільки навіть незначне запізнення у відповідях бази даних може вплинути на успіх угоди. Наприклад, при великих обсягах торгівлі на біржах обробка замовлень повинна відбуватися в режимі реального часу, і база даних має забезпечувати запис даних про операції та оновлення ціни активів майже миттєво.

Також можна виділити процес кредитного скорингу у банкових установах. Цей процес здійснюється в режимі реального часу, коли клієнти подають заявки на кредити. База даних повинна швидко надавати інформацію про платоспроможність клієнтів, їхні попередні фінансові записи та результати скорингу. Це забезпечує високоточний і надійний процес оцінки, який працює навіть при великих обсягах заявок, наприклад, у пікові періоди попиту на кредитні послуги.

Платформи електронної комерції (e-commerce [6]) сьогодні стикаються з величезним обсягом запитів від користувачів, які включають пошук товарів, обробку замовлень та відстеження доставок. База даних таких додатків повинна швидко обробляти запити, щоб користувачі могли отримати актуальну інформацію щодо продуктів та здійснити покупку без затримок.

На платформах e-commerce ціни можуть змінюватися в залежності від попиту або поточних акцій, що потребує швидкої обробки ціни в базі даних. При великій кількості товарів динамічне оновлення цін і застосування знижок створює додаткове навантаження, оскільки база даних повинна підтримувати облік і відображення коректних цін у реальному часі.

Слід також виділити важливість відгуків користувачів на комерційних платформах, де відгуки клієнтів відіграють важливу роль у прийнятті рішень про покупку наступних користувачів. Додатки повинні швидко обробляти інформацію про відгуки, рейтинги та вподобання користувачів, щоб забезпечити максимально прозорий досвід купівлі. База даних повинна зберігати відгуки й рейтинги та забезпечувати їхню актуальність для відображення на сторінках товарів.

Соціальні мережі є високонавантаженими платформами для обслуговування мільйонів користувачів, які щоденно публікують контент, діляться ним та проводять спілкування. Бази даних відіграють основну роль у забезпеченні швидкої роботи цих функцій, обробляючи запити з різних куточків світу та зберігаючи дані користувачів у реальному часі.

Додатки соціальних мереж потребують швидкої модерації контенту для забезпечення безпеки платформи та дотримання правил спільноти. Системи на основі баз даних повинні забезпечити аналіз та маркування потенційно неприйняттого контенту, зберігати історію користувацьких скарг і коментарів, що допоможе підтримувати норми контенту на платформі.

Алгоритми рекомендацій в соціальних мережах використовують великі обсяги даних для надання персоналізованих наборів контенту. Ці алгоритми зчитують та аналізують дії користувачів, що зберігаються у базі даних, надаючи таргетовану рекламу та релевантні рекомендації, які підтримують високу зацікавленість користувачів у платформі.

Стримінгові сервіси створюють інтенсивне навантаження на бази даних, оскільки одночасний доступ великої кількості користувачів до відео або аудіоконтенту потребує зберігання й обробки великих обсягів інформації. Такі сервіси не лише забезпечують трансляцію, а й активно аналізують дії користувачів, щоб краще адаптувати сервіс до інтересів аудиторії.

Платформи, як наприклад Netflix [7] або Spotify [8], аналізують історію переглядів та прослуховувань для персоналізації рекомендацій. Вони повинні мати великий архів даних, які постійно оновлюються, забезпечуючи при цьому миттєвий доступ до контенту.

Щоб вдосконалити роботу рекомендаційної системи, платформи відстежують тривалість перегляду або прослуховування контенту. Цей трекінг вимагає обробки тисяч запитів щосекунди, а бази даних повинні забезпечувати швидкий запис і читання даних для точного відображення дій користувачів.

Онлайн-ігри є також одним з видів високонавантажених платформ для обслуговування мільйонів гравців. Бази даних у цій сфері зберігають прогрес

гравців, їхні досягнення та ігрову валюту, що вимагає швидкої обробки даних та надійності.

У багатьох онлайн-іграх існують внутрішні економіки, де гравці можуть обмінювати ігрові ресурси, купувати предмети чи здійснювати торгівлю. Це вимагає складної системи управління транзакціями у реальному часі, що підтримує цілісність та безперервність ігрового процесу. Бази даних повинні швидко обробляти операції купівлі, продажу або обміну, щоб запобігти помилкам або некоректним обчисленням.

Онлайн-ігри часто включають систему реєстрації подій, де кожна дія гравця (наприклад перемога, програш чи знищення ворога) зберігається для подальшого аналізу або відображення в ігровій статистиці. База даних повинна мати високу пропускну здатність для запису подій в режимі реального часу, що дозволяє оперативно оновлювати статистику і рейтинги.

1.3 Постановка задачі дослідження

Високонавантажені платформи відіграють ключову роль у сучасному цифровому середовищі, забезпечуючи стабільний доступ до продуктів, послуг і даних. Однак зростаючі обсяги даних, інтенсивність транзакцій та кількість користувачів створюють значні виклики. Проблеми, що виникають у роботі таких платформ, часто мають прямий вплив на продукт: від зниження швидкості обслуговування до втрати клієнтів через затримки у виконанні запитів.

Однією з головних продуктових проблем є зниження швидкості обробки запитів у системі. Для користувача це може виглядати як повільне завантаження сторінок на сайті, тривалий час очікування при оформленні замовлення або затримки статусу оплати транзакції. Наприклад, платформа електронної комерції під час акцій може не встигати обробляти одночасно тисячі запитів на оновлення статусу товарів і кошика. Це веде до погіршення користувацького досвіду та втрати довіри до бізнесу. У свою чергу, такий негативний досвід може відлякати постійних користувачів, змушуючи їх обирати альтернативні сервіси.

Крім того високонавантажені платформи часто стикаються з тим, що пікове навантаження може перевищити можливості інфраструктури, призводячи до перебоїв у роботі. Для фінансових платформ це може означати неможливість проведення важливих транзакцій, а для соціальних мереж – перебої у публікаціях контенту або затримки в оновленні стрічки. Ненадійна робота продукту безпосередньо впливає на лояльність клієнтів. Більше того, часті перебої можуть створювати репутаційні ризики для компанії, особливо якщо такі ситуації повторюються у критично важливий час.

Слід зазначити що зростання бази користувачів платформи, розширення асортименту продуктів або збільшення обсягу транзакцій створюють навантаження, до якого система може бути не готова. Відсутність ефективних механізмів масштабування призводить до того, що нові функції або розширення послуг можуть бути обмежені або не реалізовані зовсім. Це позбавляє бізнес можливості ефективно реагувати на ринкові зміни. Як наслідок, компанія ризикує втратити конкурентні переваги, відстаючи від інших гравців у галузі.

На додачу до всього вищезгаданого, для сучасних користувачів персоналізований досвід є важливим фактором вибору платформи. Якщо система не здатна швидко аналізувати дані користувачів і пропонувати персоналізовані рекомендації, це знижує її привабливість. Наприклад, у стримінгових сервісах затримка в оновленні рекомендаційного списку може виглядати як «байдужість» платформи до уподобань клієнта. У бізнес-контексті це може означати втрату можливостей для підвищення конверсії та утримання користувачів.

Також важливо зазначити, що користувачі очікують, що їхні дії будуть миттєво відображатися у всіх частинах системи: оформлений замовлення в кошику має оновити залишок товарів у базі, а транзакція має одразу відображатися в історії покупок. Проблеми синхронізації призводять до плутанини, що негативно впливає на довіру користувачів до сервісу. Системні помилки або затримки у синхронізації можуть створити ілюзію втрати даних, що особливо критично для фінансових чи юридичних платформ.

Завдяки аналізу проблематики, можемо сформулювати наступне завдання. По-перше, це дослідження реплікації баз даних [9], адже реплікація баз даних є одним із ключових механізмів, що забезпечують доступність даних, підвищують стійкість до відмов і сприяють оптимізації продуктивності платформ. У межах цього завдання потрібно дослідити принципи роботи реплікації, зокрема синхронізацію даних між основними та резервними базами. Окрім цього, важливо змодельовати сценарії, пов'язані зі змінами у структурі та змісті бази даних, щоб оцінити стабільність системи та її здатність до відновлення у разі збоїв. Додатково, необхідно розглянути механізми перемикання вузлів, які підтримують доступність даних у випадку втрати основного сервера. Результати такого дослідження дозволять визначити оптимальні стратегії налаштування реплікації залежно від вимог до продуктивності та доступності платформи.

Наступним завданням є розробка механізмів шардингу [10], який виступає важливим підходом для масштабування баз даних і оптимізації роботи платформ, що обробляють великі обсяги даних. У межах цього завдання передбачено створення моделі розподілу даних між шардовими вузлами на основі географічного критерію. Наприклад, дані про користувачів можуть бути розподілені за країнами, що дозволить зменшити навантаження на кожен вузол і підвищити ефективність запитів. Також дослідження включатиме порівняльний аналіз продуктивності операцій читання та запису у випадках використання шардингу та без нього.

Крім того, необхідно проаналізувати пакетне оновлення [11], яке дозволяє знижувати навантаження на систему за рахунок обробки змін великими блоками. У цьому контексті потрібно розглянути, як різні схеми оновлення впливають на швидкість відображення змін для користувача. Наприклад, у комерційних платформах це може забезпечити ефективне оновлення цін або додавання акцій.

Ще одним важливим завданням є дослідження індексації [12]. Це передбачає розробку стратегій створення індексів для швидкого пошуку та аналізу даних користувачів. Особливу увагу слід приділити вибору типів індексів, які забезпечують збалансований доступ до великих обсягів даних,

уникаючи перевантаження системи. Це є надзвичайно важливим для динамічних платформ.

Наприкінці, завдання передбачає формулювання рекомендацій щодо інтеграції оптимізацій. Зокрема, це може включати розробку чітких рекомендацій для бізнесу стосовно використання запропонованих підходів. Наприклад, для фінансових платформ можуть бути розроблені специфічні стратегії, спрямовані на зменшення часу обробки транзакцій. Окрім цього, такі рекомендації допоможуть уникнути типових помилок у проектуванні оптимізаційних рішень.

Висновки до першого розділу

У розділі розглянуто ключові характеристики високонавантажених платформ, які забезпечують швидкий доступ до послуг і стабільну обробку великих обсягів даних у режимі реального часу. Висвітлено проблеми, з якими стикаються такі системи, зокрема зниження продуктивності, перебої у доступності сервісів і складнощі з масштабуванням. Особливу увагу приділено аналізу їх застосування у фінансовій сфері, електронній комерції, соціальних мережах і стримінгових сервісах, де бази даних відіграють ключову роль у підтримці стабільності й швидкості роботи.

На основі проведеного аналізу було сформовано завдання дослідження, спрямоване на підвищення ефективності платформ, включаючи впровадження шардингу, оптимізацію пакетних оновлень і розробку індексації для персоналізації. Це дозволить не лише забезпечити високу продуктивність і надійність платформ, але й створити рекомендації для їх подальшого вдосконалення відповідно до потреб користувачів і бізнесу.

РОЗДІЛ 2 МЕТОДИ ОПТИМІЗАЦІЇ РОБОТИ БД ДЛЯ ВИСОКОНАВАНТАЖЕНИХ ПЛАТФОРМ

2.1 Аналіз реляційних баз даних у контексті високонавантажених платформ

У високонавантажених платформах, де кожна секунда затримки чи помилка у транзакціях може призвести до втрати користувачів і фінансових збитків, вибір правильної системи управління базами даних є критичним завданням. Серед великої кількості варіантів реляційні бази даних залишаються найпоширенішим вибором, завдяки своїй здатності забезпечувати структуроване зберігання даних, транзакційну цілісність і гнучкість у виконанні запитів. У цьому дослідженні розглядаються особливості таких СКБД як MySQL [13] та PostgreSQL [14], їх переваги та недоліки в умовах високонавантажених платформ, а також аргументи на користь їх використання замість NoSQL [15].

Реляційні бази даних спираються на реляційну модель, яка забезпечує цілісність даних через суворі правила, такі як первинні та зовнішні ключі, а також гарантує транзакційну послідовність завдяки ACID-властивостям [16]. Це робить їх незамінними для систем, де будь-яке порушення консистентності або втрата транзакцій може мати катастрофічні наслідки..

У контексті високонавантажених платформ реляційні бази часто стикаються з викликами, такими як масштабування, але вони компенсують це своїми функціональними перевагами. Гнучкість SQL, можливість виконання складних запитів і підтримка багатофакторного аналізу забезпечують реляційним системам стійкість навіть за умов інтенсивного навантаження.

MySQL є однією з найпопулярніших реляційних баз даних, відомою своєю простотою, стабільністю та продуктивністю. Ця система часто використовується у високонавантажених платформах завдяки її швидкодії, особливо при виконанні операцій читання.

MySQL має кілька важливих переваг, які роблять її придатною для високонавантажених платформ. Однією з головних переваг є швидке читання та

проста реплікація. Завдяки механізму майстер-слейв реплікації MySQL дозволяє ефективно виконувати операції читання, розподіляючи навантаження між кількома вузлами. Це сприяє зменшенню затримок під час обробки запитів. Наприклад, у платформ електронної комерції це забезпечує миттєвий доступ до інформації про товар або швидке оновлення кошика користувача.

Ще одна важлива перевага — це оптимізація для потокових запитів. MySQL ефективно працює у сценаріях, де часто виконуються операції вибірки, наприклад, при перегляді стрічки новин або сторінок із продуктами. Окрім цього, механізм кешування зменшує кількість звернень до основної бази даних, що позитивно впливає на продуктивність.

Щодо викликів MySQL, є кілька аспектів, які потребують уваги. По-перше, це обмежена підтримка складних транзакцій. Хоча MySQL підтримує ACID через InnoDB [17], великі транзакції або запити, що інтегрують кілька таблиць, можуть спричиняти затримки. Це є критичним для фінансових платформ або CRM-систем.

По-друге, важливим викликом є масштабування. MySQL зазвичай покладається на вертикальне масштабування, яке має свої обмеження для платформ із різким збільшенням кількості користувачів. Горизонтальне масштабування через шардинг потребує значних зусиль у налаштуванні, що ускладнює використання MySQL для таких платформ.

PostgreSQL часто називають «найбільш просунутою» відкритою реляційною базою даних. Вона ідеально підходить для платформ, які потребують не лише швидкості, а й гнучкості в обробці складних запитів і підтримки різноманітних форматів даних.

Серед переваг PostgreSQL можна виділити перш за все можливість роботи зі складними запитами та аналітикою. PostgreSQL пропонує широкий набір інструментів для виконання складних запитів, включаючи підтримку аналітичних функцій і спеціалізованих типів даних. Наприклад, в системах рекомендацій контенту можна використовувати вкладені запити або спеціальні індекси для швидкого пошуку інформації.

Ще однією важливою перевагою є розширюваність. PostgreSQL дозволяє додавати власні функції та типи даних, що особливо корисно для платформ із нестандартними вимогами. Наприклад, геолокаційні сервіси можуть використовувати вбудовану підтримку геометричних типів для зберігання й обробки просторових даних, забезпечуючи точність і зручність у роботі.

Крім цього, PostgreSQL має надійну транзакційну систему. На відміну від MySQL, вона краще справляється з великими транзакціями, що є критично важливим для платформ із високим рівнем взаємодії, таких як біржі чи платіжні системи.

Втім, PostgreSQL має і свої виклики. Наприклад, вона може виявитися менш ефективною у випадках, коли система працює із простими запитами, які виконуються з високою частотою. Це може бути проблемою для соціальних мереж або інших платформ, де домінують потокові операції.

Ще один виклик — це складність конфігурації. PostgreSQL потребує значних знань для налаштування оптимальної роботи, особливо в умовах великого навантаження. Наприклад, створення правильних індексів або партиціонування таблиць потребує чималих зусиль і часу.

NoSQL-системи, такі як MongoDB [18] або Cassandra [19], часто використовуються для високонавантажених платформ через їхню здатність до горизонтального масштабування та роботу з нефіксованими структурами даних. Однак, реляційні бази даних мають низку переваг, які роблять їх більш привабливими для таких платформ.

Однією з ключових переваг є консистентність та транзакції. Для фінансових платформ чи систем обробки замовлень гарантування консистентності даних є обов'язковим. Реляційні бази забезпечують транзакційну послідовність навіть у найскладніших сценаріях, на відміну від NoSQL-систем, які часто жертвують цілісністю даних заради продуктивності.

Ще однією перевагою є структурованість даних. У випадках, коли структура даних є складною, реляційні бази надають більшу гнучкість. Наприклад, в платформах електронної комерції вони дозволяють ефективно

об'єднувати інформацію про користувачів, замовлення та продукти, що значно спрощує управління та обробку даних.

Також важливо зазначити, що SQL як стандарт є ще однією перевагою реляційних систем. Використання стандартизованої мови для роботи з даними значно спрощує інтеграцію, забезпечуючи доступ до великої кількості інструментів аналітики. Це є особливо важливим для платформ, де потрібні складний аналіз даних або створення звітності.

2.2 Реплікація баз даних

Реплікація баз даних є основним механізмом забезпечення високої доступності та стійкості сучасних інформаційних систем. Вона дозволяє створювати та підтримувати кілька копій даних у різних фізичних або логічних місцях, забезпечуючи безперервний доступ до інформації навіть у випадках збоїв у системі. Здатність забезпечити швидкий і надійний доступ до даних під час високого навантаження робить її критично важливою для цифрових платформ, які обслуговують тисячі або навіть мільйони користувачів одночасно.

Реплікація баз даних полягає у копіюванні даних з одного головного джерела (мастера) до одного або кількох цільових місць (реплік), підтримуючи їхню актуальність та узгодженість. Цей процес включає три важливі етапи, кожен із яких має значний вплив на ефективність роботи системи. Спершу всі зміни, які відбуваються у головній базі даних, фіксуються у спеціальному журналі змін [20]. Цей журнал виконує функцію відстеження дій користувачів і системи, гарантуючи, що жодна транзакція не залишиться непоміченою.

Далі ці зміни передаються до реплік через мережеві з'єднання. У випадку синхронної реплікації цей процес відбувається миттєво: всі репліки мають підтвердити прийняття змін, перш ніж транзакція буде завершена. Натомість у асинхронній реплікації дані передаються із певною затримкою, що може варіюватися залежно від налаштувань системи. На завершальному етапі отримані дані інтегруються у відповідні репліки, забезпечуючи актуальність усіх

копій бази. У сучасних архітектурах цей етап часто виконується у фоновому режимі, щоб мінімізувати вплив на продуктивність головного сервера.

Ця складна взаємодія між вузлами дозволяє забезпечити швидкий доступ до даних та їхню доступність навіть у разі збою одного із компонентів системи. Водночас реалізація кожного з етапів потребує ретельного налаштування, щоб уникнути помилок і затримок.

Реплікація забезпечує численні переваги для високонавантажених платформ, які стикаються з великим обсягом транзакцій і запитів. Ці переваги не лише підвищують продуктивність, а й забезпечують стабільність системи за умов інтенсивного використання.

Підвищена доступність даних є однією з ключових переваг реплікації. У випадку збою головного сервера система може автоматично переключитися на одну з реплік, забезпечуючи безперервність обслуговування користувачів. Наприклад, для платформи електронної комерції це означає, що клієнти продовжать отримувати доступ до каталогу товарів, навіть якщо один із серверів перестав відповідати. Ця властивість особливо важлива для систем із високими вимогами до часу безвідмовної роботи.

Реплікація також сприяє розподілу навантаження між кількома серверами, що знижує навантаження на кожен окремий вузол. Наприклад, соціальна мережа може направляти запити на читання до однієї репліки, а запити на запис – до іншої. Це не лише знижує ризик перевантаження, але й забезпечує стабільний досвід користувачів навіть у пікові періоди.

Додатково, розміщення реплік ближче до кінцевих користувачів зменшує затримки під час передачі даних через мережу. У глобально розподілених системах, таких як стрімінгові платформи, це може суттєво вплинути на швидкість завантаження контенту. Географічно розташовані репліки також дозволяють уникнути проблем із пропускнуою здатністю мережі, що часто виникають у глобальних системах.

Попри всі переваги, реплікація також створює певні виклики, які можуть ускладнити її впровадження. Розуміння цих недоліків дозволяє правильно оцінити ресурси та складність під час налаштування системи.

Одним із основних викликів є складність підтримки узгодженості даних між репліками, особливо у випадку асинхронної реплікації [21]. Якщо зміни передаються із затримкою, дані на репліках можуть відрізнитися від головного сервера. Наприклад, у системах онлайн-замовлень це може призвести до некоректного відображення кількості доступного товару, що викликає непорозуміння серед клієнтів.

Додатково, реплікація потребує значних ресурсів. Зберігання кількох копій бази даних вимагає додаткового місця на диску, а передача даних через мережу створює навантаження на пропускну здатність. Це може стати обмеженням для організацій із невеликим.

Процес управління реплікацією також є досить складним. Адміністрування кількох реплік вимагає ретельного моніторингу та налаштування, щоб уникнути помилок у синхронізації. Наприклад, помилка в налаштуваннях може призвести до дублювання даних або втрати транзакцій.

Нарешті, синхронна реплікація може впливати на продуктивність системи через необхідність підтвердження змін на всіх репліках. Це особливо відчутно у системах із великим обсягом записів, де затримки через синхронізацію можуть значно вплинути на час виконання транзакцій.

2.3 Шардинг

Шардинг баз даних – це процес розподілу даних між кількома вузлами або серверами, що дозволяє підвищити масштабованість та продуктивність системи. У сучасних високонавантажених платформах, таких як соціальні мережі, електронна комерція чи фінансові сервіси, обсяги даних можуть сягати терабайтів. У таких умовах традиційні підходи до зберігання даних можуть стати «проблемним місцем» системи, тому шардинг виступає як один із найефективніших методів оптимізації.

Шардинг (sharding) – це метод горизонтального розподілу даних, який передбачає розбиття однієї великої таблиці на кілька менших частин, що називаються шардами. Кожен шард містить підмножину даних і функціонує як окрема база даних. На відміну від вертикального масштабування, яке передбачає додавання потужностей до одного сервера, шардинг дозволяє розподіляти дані між кількома серверами, зменшуючи навантаження на кожен окремий вузол.

Принцип роботи шардингу базується на алгоритмі, який визначає, до якого шарду буде розподілено кожен запис. Одним із методів є розподіл за діапазоном значень. У цьому випадку дані розподіляються відповідно до певного діапазону, наприклад, за ID користувача. Користувачі з ID у межах 1–1000 можуть бути віднесені до одного шарду, тоді як ті, у кого ID 1001–2000, — до іншого. Такий підхід є досить простим і зрозумілим, але у випадках, коли дані розподіляються нерівномірно, може виникнути дисбаланс навантаження.

Інший поширений підхід — це використання хешування. У цьому випадку застосовується хеш-функція, яка створює унікальний ключ для кожного запису та визначає його місце в конкретному шарді. Хешування забезпечує рівномірний розподіл даних, проте додає складнощів при необхідності додавання нових шардів, оскільки це може вимагати перерозподілу вже наявних даних.

Ще одним методом є географічний розподіл, за якого дані класифікуються залежно від регіону. Наприклад, користувачі з Європи можуть зберігатися в одному шарді, тоді як користувачі з Північної Америки — в іншому. Такий підхід дозволяє зменшити затримки доступу, однак водночас потребує більш складної логіки для маршрутизації запитів.

Завдяки цим підходам шардинг баз даних стає одним із найефективніших методів оптимізації високонавантажених систем, оскільки кожен із методів забезпечує конкретний функціональний або бізнес-результат.

Серед переваг можна виділити те, що шардинг дозволяє досягти горизонтальної масштабованості [22]. У той час як вертикальне масштабування [23] обмежене фізичними параметрами одного сервера (обсяг оперативної пам'яті, кількість процесорів), горизонтальне масштабування через шардинг

дозволяє додавати нові вузли без істотних змін у загальній архітектурі. Наприклад, у разі зростання бази користувачів достатньо додати новий сервер і перенаправити до нього частину даних, що забезпечує стабільну продуктивність системи.

По-друге, цей метод зменшує затримки при доступі до даних. Оскільки кожен шард містить лише частину загальних даних, операції вибірки та запису виконуються швидше, ніж у випадку, коли всі дані зберігаються на одному сервері. Це особливо важливо для систем, які обробляють мільйони запитів одночасно. Наприклад, у соціальних мережах шардинг дозволяє забезпечити миттєве завантаження стрічки новин або сторінок профілів навіть за умов пікового навантаження.

Ще однією перевагою є підвищення стійкості до збоїв. У разі виходу з ладу одного шарду, функціональність системи майже не зазнає втрат. Це знижує ризик повної зупинки роботи платформи, забезпечуючи безперервність обслуговування користувачів. Наприклад, якщо один із серверів, на якому розташований певний шард, зазнає збою, доступ до інших даних залишиться незмінним.

Попри очевидні переваги, шардинг має й певні недоліки, які можуть ускладнити його впровадження та управління. Кожен із цих недоліків вимагає ретельного планування та глибокого розуміння архітектури системи.

Найбільшим викликом є складність управління шардованою базою даних. Процес налаштування шардів вимагає аналізу структури даних та вибору оптимальної стратегії розподілу. Наприклад, неправильний вибір критерію шардингу може призвести до «гарячих шардів», коли один із серверів обробляє більше запитів, ніж інші, що створює дисбаланс навантаження.

Ще однією проблемою є складність виконання запитів, які потребують доступу до даних із кількох шардів. Наприклад, якщо транзакція охоплює дані, розподілені між різними шардами, час виконання запиту може значно збільшитися через необхідність об'єднання результатів. Це може стати

серйозною проблемою для аналітичних платформ, де складні звіти генеруються з даних, що розташовані у різних шард.

Процес перешардування також є складним і ресурсоємним. Якщо структура даних змінюється або з'являється потреба у додаванні нових шардів, необхідно перерозподілити існуючі дані. Це може створювати значне навантаження на систему та викликати тимчасові затримки у роботі платформи.

Шардинг часто плутають із партиціюванням [24], оскільки обидва методи передбачають поділ даних на частини для підвищення продуктивності баз даних. Однак між цими підходами існують суттєві відмінності. Партиціювання — це процес розділення таблиці на частини в межах однієї бази даних, що дозволяє зменшити час виконання запитів за рахунок оптимізації роботи з окремими сегментами даних. Наприклад, таблиця може бути поділена за роками, регіонами чи іншими критеріями, щоб запити оброблялися лише у відповідному сегменті, а не сканували всю таблицю. Однак у партиціюванні всі дані залишаються в межах одного сервера, що створює обмеження для масштабованості.

Шардинг, на відміну від партиціювання, передбачає розподіл даних між кількома серверами або базами даних. Таким чином, шардинг пропонує ширші можливості для масштабування, забезпечує кращу стійкість до збоїв і підходить для глобально розподілених платформ, що робить його значно ефективнішим порівняно з партиціюванням для високонавантажених систем.

2.4 Пакетні оновлення

Пакетні оновлення, є одним із методів оптимізації роботи баз даних, який полягає у виконанні великих операцій оновлення даних невеликими частинами (пакетами) замість обробки всього обсягу записів однією транзакцією. Це дозволяє уникнути перевантаження системи, забезпечити стабільність її роботи та підвищити продуктивність. Наприклад, замість одночасного оновлення мільйона записів таблиці, їх обробляють у пакетах по 50 000 чи 100 000 рядків за раз. Такий підхід допомагає уникнути блокування ресурсів та переповнення журналу транзакцій.

Основний принцип роботи пакетних оновлень полягає у поділі великих операцій оновлення на менші частини, які виконуються послідовно. Система обирає певну кількість записів, оновлює їх і переходить до наступного блоку. Наприклад, під час обробки даних у базі SQL система формує запит, який оновлює першу групу записів, наприклад 50 000 рядків. Після завершення цієї операції система переходить до наступної групи. Такий підхід дозволяє зменшити навантаження на сервер і уникнути блокування всієї таблиці.

Під час кожної ітерації пакетних оновлень відбувається оновлення частини даних, що також гарантує стабільність роботи інших запитів до цієї таблиці. Наприклад, у разі помилки втрати даних будуть мінімальними, оскільки зміни вже застосовані до завершених пакетів.

Пакетні оновлення мають низку переваг, які роблять їх важливим інструментом для високонавантажених платформ. По-перше, вони допомагають знизити навантаження на систему. Поділ великих транзакцій на менші пакети дозволяє уникнути перевантаження обчислювальних ресурсів, таких як процесор, пам'ять і дискові операції. Наприклад, обробка мільйона записів одним запитом може заблокувати всю базу даних, тоді як поступова обробка невеликими пакетами дозволяє зберігати стабільність системи.

Крім того, пакетні оновлення сприяють оптимізації журналу транзакцій. Однією з основних проблем великих оновлень є ризик переповнення цього журналу. Робота з невеликими пакетами дозволяє виконувати кожну групу записів в межах окремої транзакції, що допомагає уникнути переповнення і мінімізує ризики збоїв.

Ще однією перевагою є уникнення блокувань таблиць [25]. Використання пакетних оновлень дозволяє зменшити час блокування таблиць або рядків під час змін, що є особливо важливим для систем, де безперервний доступ до даних є критичною вимогою.

Важливо зазначити, що такий підхід забезпечує масштабованість. Система стає більш гнучкою та адаптивною до зростання обсягів даних, дозволяючи обробляти навіть великі набори даних без зниження загальної продуктивності.

Окрім цього, пакетні оновлення знижують ризик відкату транзакцій. У разі помилки лише поточний пакет повертається у попередній стан, тоді як усі попередні пакети залишаються незмінними. Це значно зменшує обсяг роботи, необхідної для відновлення даних у разі збою.

Попри численні переваги, пакетні оновлення мають і свої недоліки, які важливо враховувати при їх впровадженні. По-перше, це затримка в оновленні даних. Використання пакетного підходу може призводити до ситуацій, коли частина даних вже оновлена, а решта ще очікує своєї черги. У реальних умовах, це може створювати труднощі для систем, яким потрібна актуальність даних у режимі реального часу.

Також, реалізація пакетних оновлень може бути складною. Для цього часто потрібні значні зміни у системній архітектурі або в роботі додатків. Розробники мають враховувати такі фактори, як обмеження на розмір пакета, доступні ресурси системи та специфіку конфігурації бази даних.

Ще однією проблемою є навантаження на мережу. У розподілених системах передача даних кількома пакетами може збільшити загальний обсяг мережевого трафіку, особливо якщо кожен пакет потребує додаткових запитів для підтвердження свого виконання.

Окрім того, важким завданням є налаштування оптимального розміру пакета. Невірно обраний розмір може негативно вплинути на ефективність. Наприклад, надто великі пакети можуть спричиняти блокування ресурсів, тоді як надто малі — збільшують кількість транзакцій і відповідно витрати на їх обробку.

Пакетні оновлення активно використовуються у таких сферах, як електронна комерція, аналітика великих даних і банківські сервіси. Наприклад, в інтернет-магазинах пакетні оновлення застосовуються для оновлення інформації про наявність товарів після великих розпродажів, таких як «Чорна п'ятниця». У банківській сфері цей підхід використовується для обробки масових транзакцій, наприклад, розрахунку відсотків або обробки списання коштів. Крім того, у системах бізнес-аналітики пакетні оновлення допомагають швидко обробляти

дані перед генерацією звітів, забезпечуючи швидкість та стабільність роботи системи.

2.5 Індксація таблиць

Індксація таблиць є одним із ключових інструментів для оптимізації продуктивності реляційних баз даних. Індкси забезпечують ефективний пошук і вибірку даних, скорочуючи час виконання запитів. Вони працюють подібно до покажчика в книзі: дозволяють швидко знайти потрібну інформацію, уникнувши перегляду всієї таблиці. У сучасних високонавантажених системах індксація є основним способом забезпечення швидкості доступу до даних, зокрема у фінансових сервісах, платформах електронної комерції та соціальних мережах.

Індекс у базі даних – це спеціальна структура, яка створюється на основі одного або декількох стовпців таблиці. Його мета полягає у прискоренні пошуку та зменшенні кількості необхідних операцій зчитування. Наприклад, у таблиці з мільйоном записів запит на пошук конкретного значення без індексу змушує систему переглянути всі рядки. За допомогою індексу база даних звертається до структури, яка містить значення стовпця разом із посиланням на фізичне розташування відповідного запису. Це дозволяє знайти потрібну інформацію за лічені мілісекунди.

Коли створюється індекс, база даних будує структуру, яка організує значення стовпців за логічним порядком. Найчастіше це дерево В-типу (B-Tree), що дозволяє ефективно виконувати пошук, вставку та видалення. Наприклад, у разі запиту, який фільтрує дані за стовпцем "Дата", індекс дозволяє швидко знайти діапазон значень, замість того щоб сканувати всю таблицю. Ця структура оптимізована для пошуку за ключами, діапазонами та сортування.

Однак індксація також додає складності до операцій запису. Наприклад, при додаванні нового запису всі пов'язані індекси мають бути оновлені, щоб відобразити зміни. Це може уповільнити операції вставки або оновлення, особливо у таблицях із великою кількістю індексів.

Існує кілька типів індексів, які відрізняються за структурою, принципом роботи та областю застосування. Одним із найпоширеніших є B-tree індекси [26]. Вони базуються на структурі збалансованого дерева, що дозволяє ефективно виконувати операції пошуку, вставки та видалення елементів. B-tree індекси добре підходять як для точкових запитів, так і для запитів з діапазонами значень. Наприклад, при пошуку записів за стовпцем «Дата», де значення потрапляють у певний діапазон, B-tree індекс дає змогу швидко знайти початкову точку і сканувати лише необхідну частину даних. Ця властивість робить їх універсальними для більшості сценаріїв використання, оскільки вони забезпечують баланс між швидкістю доступу та ефективністю зберігання.

Ще одним типом індексів є хеш-індекси [27], які використовують хеш-функції для перетворення значень індексованого стовпця у позиції в хеш-таблиці. Цей підхід особливо ефективний для точкового пошуку, коли потрібно знайти запис з конкретним значенням. Проте хеш-індекси не підтримують діапазонні запити, оскільки хеш-функція не зберігає порядок значень. Використання хеш-індексів доцільне в ситуаціях, коли запити переважно здійснюються за точним співпадінням значень, і де порядок значень не має значення.

BRIN (Block Range Index) індекси [28] призначені для роботи з дуже великими таблицями, які містять впорядковані або майже впорядковані дані. Вони зберігають узагальнену інформацію про блоки даних, такі як мінімальні та максимальні значення індексованого стовпця в кожному блоці. Це дозволяє швидко відкидати блоки, що не можуть містити шуканих значень, значно зменшуючи обсяг даних для сканування. BRIN індекси займають менше місця порівняно з традиційними індексами і є ефективними при запитах, що охоплюють великі діапазони значень у впорядкованих даних. Таким чином, вони особливо корисні для оптимізації запитів у великих базах даних, де використання інших типів індексів може бути неефективним через розмір.

Часткові індекси [29] створюються на підмножині рядків таблиці, які відповідають певній умові. Це дозволяє зменшити розмір індексу та прискорити

запити, які часто використовують цю умову. Наприклад, якщо в таблиці є стовпець «balance», і запити часто фільтруються за умовою «balance >= 5000», створення часткового індексу лише для цих записів покращить продуктивність відповідних запитів. Часткові індекси ефективно використовують простір та ресурси системи, оскільки індексують тільки релевантні дані. Це робить їх особливо корисними в ситуаціях, коли певні значення або діапазони значень використовуються значно частіше за інші.

Окрім цього, існують композитні індекси [30], які охоплюють кілька стовпців одночасно і корисні для запитів, що фільтрують дані за декількома критеріями. Порядок стовпців у композитному індексі має значення, оскільки він впливає на те, які запити можуть ефективно використовувати індекс. Наприклад, індекс на стовпцях «Клієнт» і «Дата» буде ефективним для запитів, що спочатку фільтрують за «Клієнт», а потім за «Дата». Якщо ж запит фільтрує лише за другим стовпцем, ефективність індексу може знизитися. Композитні індекси дозволяють оптимізувати складні запити і є незамінними в багатьох випадках, де необхідно швидко отримати дані за декількома параметрами одночасно.

Індексація є важливим інструментом оптимізації роботи баз даних і надає низку переваг, які роблять її незамінною для ефективного функціонування систем. Насамперед, індексація значно прискорює виконання запитів. Завдяки індексам, пошук даних у таблицях виконується значно швидше, оскільки вони дозволяють уникнути повного сканування таблиці, скорочуючи кількість операцій читання. Наприклад, у таблиці з мільйонами записів запит на пошук конкретного рядка без індексу може тривати кілька секунд, тоді як за наявності індексу результат можна отримати за мілісекунди. Це особливо важливо для систем, що працюють у режимі реального часу, таких як фінансові платформи чи онлайн-магазини, де швидкий доступ до даних має критичне значення для користувачів.

Крім цього, індекси оптимізують операції сортування й групування даних. Наприклад, у звітах, які потребують сортування за датою або угруповання продажів за регіонами, індексація дозволяє значно пришвидшити обробку

запитів. Для систем бізнес-аналітики це означає можливість створювати складні звіти без надмірного навантаження на сервер, що сприяє стабільній роботі платформи.

Окрім прискорення роботи, ефективна індексація допомагає зменшити загальне навантаження на сервер. Це досягається завдяки зменшенню кількості операцій читання, що, своєю чергою, знижує витрати серверних ресурсів.

Ще однією важливою перевагою є підтримка унікальності даних. Унікальні індекси автоматично запобігають дублюванню значень у ключових стовпцях. Це критично важливо для таблиць, які містять унікальні атрибути, такі як електронна пошта або ідентифікатор користувача. Завдяки цьому система забезпечує точність даних без необхідності додаткових перевірок, що спрощує управління базою даних.

Хоча індексація має значні переваги, неправильне її використання може створювати низку проблем, які впливають на ефективність роботи бази даних. Однією з цих проблем є витрати дискового простору. Кожен індекс займає додаткове місце в пам'яті, що може значно збільшити загальний обсяг бази даних. Наприклад, у великих таблицях із сотнями мільйонів записів, де створено багато індексів, розмір індексів може навіть перевищувати обсяг основних даних. Це може стати критичним в системах із обмеженими ресурсами.

Ще одним викликом є уповільнення операцій запису. Індекси впливають на швидкість вставки, оновлення та видалення, оскільки під час кожної зміни даних усі пов'язані індекси потребують оновлення. Це особливо помітно у систем із великим потоком записів, як-от логістичні платформи або аналітичні сервіси, що працюють із постійним оновленням інформації.

Також важливо враховувати, що надмірна кількість індексів може негативно вплинути на продуктивність системи. Наприклад, якщо таблиця містить десятки індексів, кожна операція запису буде вимагати оновлення цих структур, що створить затримки. До того ж, не всі індекси використовуються однаково часто: ті, що рідко застосовуються в умовах запитів, лише витрачають ресурси без реальної користі.

Окрім цього, управління індексами може бути складним і вимагати значних зусиль. Неправильний вибір типу індексу або його місця створення може знижувати ефективність роботи бази даних замість її оптимізації. Наприклад, створення індексу для поля, яке часто оновлюється, може більше нашкодити, ніж допомогти. Адміністратори баз даних мають ретельно аналізувати, які індекси потрібні, щоб уникнути перевантаження системи.

Нарешті, індекси можуть створювати проблеми зі складними запитами. У деяких випадках, коли запит потребує доступу до багатьох стовпців або виконує об'єднання кількох таблиць, база даних може ігнорувати індекси й виконувати повне сканування таблиці. Це призводить до додаткових витрат на ресурси, що впливає на загальну продуктивність системи.

Висновки до другого розділу

В другому розділі було розглянуто та досліджено ключові підходи до оптимізації роботи реляційних баз даних, зокрема в контексті їхньої ефективності у високонавантажених платформах. Було проаналізовано основні властивості реляційних баз даних, таких як MySQL та PostgreSQL, включаючи їх здатність до забезпечення консистентності даних і високої надійності транзакцій навіть за умов інтенсивного навантаження.

У межах дослідження було детально розглянуто реплікацію як важливий механізм підвищення доступності та стійкості даних. Було вивчено як синхронну, так і асинхронну реплікацію, їхні переваги для обслуговування великої кількості запитів та забезпечення безперервного доступу до даних навіть у разі збоїв.

Особливу увагу приділено шардингу як методу горизонтального масштабування баз даних, що дозволяє розподіляти дані між кількома вузлами для підвищення продуктивності. Також були досліджені пакетні оновлення як спосіб уникнення блокувань при масових змінах даних, а індексація таблиць розглянута як інструмент, що забезпечує значне прискорення виконання запитів та оптимізацію доступу до великих обсягів інформації.

Таким чином, другий розділ надав комплексне уявлення про основні інструменти та методи, що дозволяють підвищити ефективність і масштабованість реляційних баз даних, які є критично важливими для сучасних високонавантажених платформ.

РОЗДІЛ 3 РЕАЛІЗАЦІЯ ТА АНАЛІЗ ЕФЕКТИВНОСТІ МЕТОДІВ ОПТИМІЗАЦІЇ

3.1 Вибір інструментальних засобів та технологій реалізації

Для реалізації високонавантажених платформ критично важливо обрати інструменти, які забезпечують продуктивність, масштабованість і стабільність системи. У цьому дослідженні використовується комбінація сучасних технологій, що включає Docker [31], Docker Compose [32], PostgreSQL, MySQL та ProxySQL [33]. Кожен із цих інструментів був обраний через його ключові переваги та відповідність поставленим вимогам.

Docker є основним інструментом для контейнеризації додатків. Контейнери, створені за допомогою Docker, дозволяють ізолювати середовище виконання, забезпечуючи портативність і узгодженість між різними системами. Наприклад, використання контейнерів виключає ризик конфліктів між залежностями під час переміщення додатків із середовища розробки у прод середовище. Крім того, Docker полегшує процеси розгортання та масштабування, що є критично важливим для систем із великим обсягом запитів. Завдяки підтримці новітніх версій Docker, можна скористатися найновішими функціями та вдосконаленнями, які підвищують ефективність роботи.

Для управління кількома контейнерами одночасно використовується Docker Compose. Цей інструмент дозволяє описувати конфігурацію багатоконтейнерних додатків у зручному форматі YAML-файлів [34]. Це спрощує процес налаштування складних систем, що включають кілька взаємозалежних сервісів, таких як бази даних і проксі-сервери. Використання Docker Compose також дозволяє легко керувати залежностями між сервісами, запускати їх одночасно та швидко розгортати середовища тестування. Така автоматизація значно скорочує час на налаштування системи та знижує ризики помилок.

Для роботи з базою даних обрано PostgreSQL та MySQL, дві потужні системи керування базами даних, які зарекомендували себе як надійні рішення

для високонавантажених систем. PostgreSQL забезпечує розширені можливості для обробки складних запитів і роботи з великими обсягами даних. Завдяки підтримці типів даних, процедур і тригерів, PostgreSQL є ідеальним вибором для додатків із високими вимогами до складності запитів. Крім того, PostgreSQL активно підтримується спільнотою розробників, що гарантує стабільність та регулярні оновлення.

MySQL, у свою чергу, забезпечує високу швидкість обробки запитів і простоту у використанні. Завдяки своїй масштабованості MySQL підходить для застосунків, таких як платформи електронної комерції чи соціальні мережі.

Для оптимізації роботи з MySQL використовується ProxySQL, високопродуктивний проксі-сервер, який забезпечує балансування навантаження, кешування запитів і розподіл трафіку між кількома вузлами баз даних. Використання ProxySQL дозволяє знизити навантаження на сервери баз даних, підвищуючи продуктивність системи в цілому. Завдяки його гнучкості та підтримці складних конфігурацій ProxySQL легко інтегрується з іншими компонентами інфраструктури.

Обрані інструменти й технології були підібрані з урахуванням їхньої сучасності, активної підтримки розробників і відповідності високим вимогам стабільності та продуктивності. Використання останніх версій цих технологій дозволяє скористатися найновішими оновленнями, що підвищують ефективність роботи. Поєднання Docker і Docker Compose для управління контейнерами, PostgreSQL і MySQL для роботи з даними, а також ProxySQL для оптимізації трафіку створює надійну й масштабовану основу для реалізації системи, здатної впоратися з великими обсягами навантаження.

3.2 Реалізація та аналіз ефективності реплікації баз даних

У сучасних високонавантажених системах стабільність, узгодженість даних і масштабованість є ключовими аспектами, що визначають ефективність роботи баз даних. Логічна реплікація PostgreSQL дозволяє досягти цих цілей, забезпечуючи синхронізацію даних між різними базами даних у реальному часі.

У цьому підрозділі ми детально розглянемо процес налаштування та управління середовищем логічної реплікації за допомогою Docker.

Основною метою є створення практичного середовища, що моделює реальний сценарій використання. Для цього будуть розгорнуті три екземпляри PostgreSQL, налаштовані на логічну реплікацію. Один із серверів працюватиме як головний (publisher), тоді як інші виконуватимуть роль реплік (subscribers). Такий підхід дозволить нам продемонструвати механізми узгодженості даних, управління масштабуванням і реакцію на збої. Це дослідження дозволить отримати глибоке розуміння логічної реплікації, охоплюючи аспекти її налаштування, використання у випадках збою та забезпечення стабільності даних.

Перше завдання – створити основне середовище для роботи. За допомогою Docker Compose ми розгортаємо три PostgreSQL-контейнери: один головний сервер і дві репліки. Для симуляції реальних умов ми використовуємо різні версії PostgreSQL для кожного екземпляра. Такий підхід перевіряє сумісність логічної реплікації між версіями та демонструє, як можна мінімізувати простій під час оновлення бази даних.

Ця конфігурація забезпечує:

- Використання різних PostgreSQL версій 16.1 (головний сервер), 15.5 та 14.10 (репліки).
- Легкі та оптимізовані образи PostgreSQL на основі Alpine для зменшення обчислювальних ресурсів.
- Вольюми для кожного контейнера, що забезпечать збереження даних навіть після перезапуску.

Наступний файл `docker-compose.yml` описує нашу конфігурацію (рис. 3.1):

```
#Containers for replication testing
master:
  image: postgres:16.1-alpine3.19
  container_name: pg_master
  volumes:
    - pg_master-data:/var/lib/postgresql/data
  environment:
    POSTGRES_USER: postgres
    POSTGRES_PASSWORD: postgres
    POSTGRES_DB: postgres
  restart: unless-stopped

replica1:
  image: postgres:15.5-alpine3.19
  container_name: pg_replica1
  volumes:
    - pg_replica1-data:/var/lib/postgresql/data
  environment:
    POSTGRES_USER: postgres
    POSTGRES_PASSWORD: postgres
    POSTGRES_DB: postgres
  restart: unless-stopped

replica2:
  image: postgres:14.10-alpine3.19
  container_name: pg_replica2
  volumes:
    - pg_replica2-data:/var/lib/postgresql/data
  environment:
    POSTGRES_USER: postgres
    POSTGRES_PASSWORD: postgres
    POSTGRES_DB: postgres
  restart: unless-stopped
```

Рисунок. 3.1 – Налаштування docker-compose.yml

Наступний етап – створення єдиної схеми для всіх екземплярів бази даних (рис. 3.2). Початкова схема включає три таблиці: users, posts і comments. Вона моделює реальну структуру бази даних для платформи блогів, демонструючи зв'язки між користувачами, постами та коментарями.

```
CREATE TABLE users (
  user_id SERIAL PRIMARY KEY,
  username VARCHAR(255) NOT NULL,
  email VARCHAR(255) UNIQUE NOT NULL,
  created_at TIMESTAMP WITHOUT TIME ZONE DEFAULT (now() at time zone 'utc')
);

CREATE TABLE posts (
  post_id SERIAL PRIMARY KEY,
  user_id INT NOT NULL,
  title VARCHAR(255) NOT NULL,
  content TEXT NOT NULL,
  created_at TIMESTAMP WITHOUT TIME ZONE DEFAULT (now() at time zone 'utc'),
  updated_at TIMESTAMP WITHOUT TIME ZONE,
  FOREIGN KEY (user_id) REFERENCES users(user_id)
);

CREATE TABLE comments (
  comment_id SERIAL PRIMARY KEY,
  post_id INT NOT NULL,
  user_id INT NOT NULL,
  content TEXT NOT NULL,
  created_at TIMESTAMP WITHOUT TIME ZONE DEFAULT (now() at time zone 'utc'),
  FOREIGN KEY (post_id) REFERENCES posts(post_id),
  FOREIGN KEY (user_id) REFERENCES users(user_id)
);
```

Рисунок. 3.2 – Схема екземплярів баз даних

Останнім етапом підготовки є налаштування PostgreSQL для підтримки логічної реплікації. Для цього необхідно внести певні зміни до конфігурації бази даних. Перш за все, параметр `wal_level` потрібно встановити на значення `logical`, щоб забезпечити підтримку рівня транзакційної реплікації. Крім того, слід збільшити значення параметра `max_replication_slots`, що дозволить підтримувати необхідну кількість реплікаційних з'єднань. Також важливо налаштувати параметр `max_wal_senders`, який забезпечує потокову передачу даних між вузлами.

Для автоматизації цього процесу використовується скрипт `db-setup.sh`, який забезпечує однорідність параметрів на всіх екземплярах.

Скрипт виглядає наступним чином:

```
databases=("pg_master" "pg_replica1" "pg_replica2")

max_number_of_replicas=4
max_wal_senders=8

for db in "${databases[@]"; do
    docker exec -i "$db" bash -c "sed -i 's/^##wal_level .*$/wal_level = logical/'
    /var/lib/postgresql/data/postgresql.conf"
    docker exec -i "$db" bash -c "sed -i 's/^##max_replication_slots
    .*$/max_replication_slots = $max_number_of_replicas/'
    /var/lib/postgresql/data/postgresql.conf"
    docker exec -i "$db" bash -c "sed -i 's/^##max_wal_senders .*$/max_wal_senders
    = $max_wal_senders/' /var/lib/postgresql/data/postgresql.conf"
    docker exec -i "$db" bash -c "grep -qxF 'host replication all all md5'
    /var/lib/postgresql/data/pg_hba.conf || echo 'host replication all all md5'
    >> /var/lib/postgresql/data/pg_hba.conf"
    docker restart "$db"
done
```

Після підготовки наших екземплярів PostgreSQL для логічної реплікації наступним етапом є налаштування самих механізмів реплікації. Ми визначимо один екземпляр бази даних як головний сервер (master або publisher), а два інших – як репліки (subscribers). Цей процес автоматизується за допомогою скрипта `replication-setup.sh`.

Скрипт починається з налаштування головного сервера, визначаючи публікацію, яка включає всі таблиці в базі даних. Ця публікація виступає джерелом змін, які будуть реплікуватися до підписників. Для кожної репліки

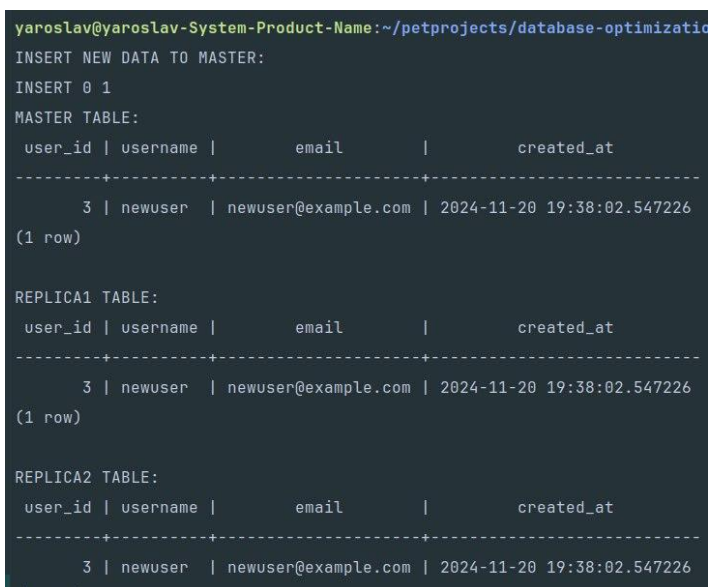
скрипт створює підписку на цю публікацію [35], встановлюючи зв'язок між реплікою та головним сервером.

Скрипт виглядає так:

```
master="pg_master"
replicas_array=("pg_replica1" "pg_replica2")
docker exec -i "$master" psql -U postgres -d postgres -c "DROP PUBLICATION IF
    EXISTS my_publication;"
docker exec -i "$master" psql -U postgres -d postgres -c "CREATE PUBLICATION
    my_publication FOR ALL TABLES;"

for replica in "${replicas_array[@]"; do
    connection="host=$master port=5432 user=postgres password=postgres
        dbname=postgres"
    docker exec -i "$replica" psql -U postgres -d postgres -c "DROP SUBSCRIPTION IF
        EXISTS ${replica}_subscription;"
    docker exec -i "$replica" psql -U postgres -d postgres -c "CREATE SUBSCRIPTION
        ${replica}_subscription CONNECTION 'dbname=postgres host=$master
            user=postgres password=postgres' PUBLICATION my_publication;"
done
```

Після налаштування механізмів реплікації можна перевірити її роботу. Додамо новий запис у таблицю users на головному сервері. Щоб переконатися, що зміни синхронізувалися, перевіримо вміст таблиці users на усіх базах даних (рис. 3.3).



```
yaroslav@yaroslav-System-Product-Name:~/petprojects/database-optimizatio
INSERT NEW DATA TO MASTER:
INSERT 0 1
MASTER TABLE:
 user_id | username | email | created_at
-----+-----+-----+-----
      3 | newuser | newuser@example.com | 2024-11-20 19:38:02.547226
(1 row)

REPLICA1 TABLE:
 user_id | username | email | created_at
-----+-----+-----+-----
      3 | newuser | newuser@example.com | 2024-11-20 19:38:02.547226
(1 row)

REPLICA2 TABLE:
 user_id | username | email | created_at
-----+-----+-----+-----
      3 | newuser | newuser@example.com | 2024-11-20 19:38:02.547226
```

Рисунок. 3.3 – Вміст таблиць users на усіх базах даних

Результати підтвердили очікування: новий запис було синхронізовано в реальному часі між головним сервером і репліками. Така синхронізація демонструє узгодженість і оперативність роботи логічної реплікації PostgreSQL.

Для тестування стійкості системи було виконано симуляцію збою шляхом відключення однієї з реплік. Під час збою головний сервер продовжував надсилати зміни до іншої активної репліки, підтверджуючи стабільність системи навіть за умов неповної доступності (рис. 3.4).

```

yaroslav@yaroslav-System-Product-Name:~/petprojects/database-optimization-research/replication$ /bin/bash /home/yarosl
STOP REPLICAS:
pg_replica2
INSERT NEW DATA TO MASTER:
INSERT 0 1
MASTER TABLE:
 user_id | username | email | created_at
-----+-----+-----+-----
      3 | newuser | newuser@example.com | 2024-11-20 19:38:02.547226
      4 | newuser2 | newuser2@example.com | 2024-11-20 19:38:51.036098
(2 rows)

REPLICA1 TABLE:
 user_id | username | email | created_at
-----+-----+-----+-----
      3 | newuser | newuser@example.com | 2024-11-20 19:38:02.547226
      4 | newuser2 | newuser2@example.com | 2024-11-20 19:38:51.036098
(2 rows)

REPLICA2 TABLE:
Error response from daemon: container 098dbff6974e32e6f0b1057e4fa3bf3daad52631c58bce13221c463be4c96a16 is not running

```

Рисунок. 3.4 – Результат стійкості системи після збою однієї з реплік

Такий підхід дозволить нам продемонструвати механізми узгодженості даних, управління масштабуванням і реакцію на збої (рис. 3.5). Це дослідження дозволить отримати глибоке розуміння логічної реплікації, охоплюючи аспекти її налаштування, використання у випадках збою та забезпечення стабільності даних.

```

yaroslav@yaroslav-System-Product-Name:~/petprojects/database-optimization-research/replication$ /bin/bash /home/yarosl
RESTART REPLICAS:
pg_replica2
MASTER TABLE:
 user_id | username | email | created_at
-----+-----+-----+-----
      3 | newuser | newuser@example.com | 2024-11-20 19:38:02.547226
      4 | newuser2 | newuser2@example.com | 2024-11-20 19:38:51.036098
(2 rows)

REPLICA1 TABLE:
 user_id | username | email | created_at
-----+-----+-----+-----
      3 | newuser | newuser@example.com | 2024-11-20 19:38:02.547226
      4 | newuser2 | newuser2@example.com | 2024-11-20 19:38:51.036098
(2 rows)

REPLICA2 TABLE:
 user_id | username | email | created_at
-----+-----+-----+-----
      3 | newuser | newuser@example.com | 2024-11-20 19:38:02.547226
      4 | newuser2 | newuser2@example.com | 2024-11-20 19:38:51.036098
(2 rows)

```

Рисунок. 3.5 – Результат стійкості системи після збою однієї з реплік

Далі було додано нового користувача (newuser3) до таблиці users на головному сервері (рис. 3.6). Ця операція мала підтвердити, що всі бази даних, включаючи ту, яка раніше була вимкнена, синхронізуються в реальному часі.

```

yaroslav@yaroslav-System-Product-Name:~/petprojects/database-optimization-
INSERT NEW DATA TO MASTER:
INSERT 0 1
MASTER TABLE:
 user_id | username | email | created_at
-----+-----+-----+-----
 3 | newuser | newuser@example.com | 2024-11-20 19:38:02.547226
 4 | newuser2 | newuser2@example.com | 2024-11-20 19:38:51.036098
 5 | newuser3 | newuser3@example.com | 2024-11-20 19:40:15.060125
(3 rows)

REPLICA1 TABLE:
 user_id | username | email | created_at
-----+-----+-----+-----
 3 | newuser | newuser@example.com | 2024-11-20 19:38:02.547226
 4 | newuser2 | newuser2@example.com | 2024-11-20 19:38:51.036098
 5 | newuser3 | newuser3@example.com | 2024-11-20 19:40:15.060125
(3 rows)

REPLICA2 TABLE:
 user_id | username | email | created_at
-----+-----+-----+-----
 3 | newuser | newuser@example.com | 2024-11-20 19:38:02.547226
 4 | newuser2 | newuser2@example.com | 2024-11-20 19:38:51.036098
 5 | newuser3 | newuser3@example.com | 2024-11-20 19:40:15.060125
(3 rows)

```

Рисунок. 3.6 – Успішна синхронізація даних для всіх реплік

Результати показали, що доданий запис автоматично з'явився на всіх екземплярах, включаючи раніше відключену репліку. Це доводить, що механізм логічної реплікації PostgreSQL здатен ефективно відновлювати цілісність даних після збою. Така стабільність і узгодженість є ключовими для забезпечення роботи у розподілених системах.

Другим важливим експериментом стало тестування сценарію переходу ролі головного сервера на одну з реплік. Такий сценарій є критично важливим у реальних умовах, наприклад, під час запланованого технічного обслуговування або у випадку аварійного збою головного сервера.

Процес переходу складався з кількох ключових етапів. Першим кроком було відключення головного сервера. Цей сервер було навмисно вимкнено, після чого роль нового головного було передано репліці pg_replica1.

Наступним етапом стало налаштування нового головного сервера. Репліка pg_replica1 була сконфігурована як джерело даних (publisher), а всі її локальні

послідовності (sequences) оновлено для уникнення можливих конфліктів із уже існуючими даними.

Останній крок включав скидання стану іншої репліки. Репліка `pg_replica2` була очищена й налаштована для синхронізації з новим головним сервером, щоб забезпечити стабільну роботу системи.

Особливу увагу було приділено автоматичному оновленню послідовностей у таблицях на новому головному сервері (рис. 3.7). Це забезпечило унікальність ідентифікаторів при вставці нових записів і виключило можливість конфліктів.

```

yaroslav@yaroslav-System-Product-Name:~/petprojects/database-optimization-research/replication$ /bin/bash /home/yaros
INSERT NEW DATA TO NEW MASTER(REPLICA1):
INSERT 0 1
MASTER TABLE:
Error response from daemon: container 13671830286a3160fd3bf7fea133cd1a57302c9e6886a1d41941408a7237efd0 is not running
REPLICA1 TABLE:
 user_id | username | email | created_at
-----+-----+-----+-----
      3 | newuser | newuser@example.com | 2024-11-20 19:38:02.547226
      4 | newuser2 | newuser2@example.com | 2024-11-20 19:38:51.036098
      5 | newuser3 | newuser3@example.com | 2024-11-20 19:40:15.060125
      6 | newuser4 | newuser4@example.com | 2024-11-20 19:41:43.414564
(4 rows)

REPLICA2 TABLE:
 user_id | username | email | created_at
-----+-----+-----+-----
      3 | newuser | newuser@example.com | 2024-11-20 19:38:02.547226
      4 | newuser2 | newuser2@example.com | 2024-11-20 19:38:51.036098
      5 | newuser3 | newuser3@example.com | 2024-11-20 19:40:15.060125
      6 | newuser4 | newuser4@example.com | 2024-11-20 19:41:43.414564
(4 rows)

```

Рисунок. 3.7 – Успішний перехід головного серверу на одну з реплік

Перевірка роботи після переходу ролей показала, що новий головний сервер (`pg_replica1`) успішно взяв на себе функції джерела даних. Вставка нового запису до таблиці `users` на цьому сервері продемонструвала, що система продовжує працювати узгоджено, а дані синхронізуються з іншою реплікою (`pg_replica2`).

Цей результат підтвердив кілька важливих характеристик системи. По-перше, було забезпечено безперервність роботи: перехід ролі не вплинув на доступність системи, що є критично важливим для високонавантажених платформ. По-друге, система продемонструвала стабільність узгодженості — дані залишалися консистентними навіть після внесення значних змін у

конфігурацію кластеру. Окрім цього, механізм реплікації виявився надзвичайно гнучким: система успішно адаптувалася до нового стану без втрати даних чи потреби в ручному втручанні.

У процесі реінтеграції первинного головного сервера (`pg_master`) до налаштованого середовища логічної реплікації було виявлено ряд несподіваних труднощів. Зокрема, дані не синхронізувалися після підключення старого головного сервера як репліки до нового головного сервера (`pg_replica1`). Цей виклик підкреслив складність у зміні ролей і необхідність чітких стратегій у таких сценаріях.

Після налаштування `pg_master` як репліки до нового головного сервера дані з останнього не передавалися назад. Аналіз логів показав, що конфлікти виникли через дублювання первинних ключів у таблиці `users`. Ця проблема була спричинена спробою передати повну базу даних зі стану нового головного сервера до старого, коли дані вже існували на обох екземплярах.

Конфлікти дублювання стали показовими для проблеми синхронізації в умовах, коли старий головний сервер тривалий час перебував офлайн. Під час повторного підключення система намагалася відтворити зміни, створені новим головним сервером, але не враховувала вже наявні записи в базі даних старого головного.

Для усунення цієї проблеми було прийнято рішення повністю відновити `pg_master` до початкового стану. Процес включав кілька етапів. Спочатку було видалено контейнер разом із вольюмами даних, щоб забезпечити очищення від усіх старих даних і конфігурацій.

Наступним кроком стало повторне створення бази даних. Було розгорнуто новий екземпляр `pg_master`, використовуючи вихідну схему, щоб відновити початковий стан системи.

Останній етап полягав у налаштуванні старого головного сервера як репліки. Його було сконфігуровано для підписки на публікацію нового головного сервера, що забезпечило синхронізацію даних і стабільну роботу системи.

Цей метод, хоч і вимагав значних змін, дозволив усунути конфлікти даних і забезпечити узгодженість між усіма екземплярами бази даних.

Після повторного налаштування pg_master як репліки було додано новий запис на новому головному сервері (pg_replica1). Усі три екземпляри (новий головний сервер, друга репліка та старий головний сервер як репліка) успішно синхронізували дані, включаючи новий запис (рис. 3.8).

```

yaroslav@yaroslav-System-Product-Name:~/petprojects/database-optimization
INSERT NEW DATA TO NEW MASTER(REPLICA1):
INSERT 0 1
MASTER TABLE:
 user_id | username | email | created_at
-----+-----+-----+-----
 3 | newuser | newuser@example.com | 2024-11-20 19:38:02.547226
 4 | newuser2 | newuser2@example.com | 2024-11-20 19:38:51.036098
 5 | newuser3 | newuser3@example.com | 2024-11-20 19:40:15.060125
 6 | newuser4 | newuser4@example.com | 2024-11-20 19:41:43.414564
 7 | newuser5 | newuser5@example.com | 2024-11-20 19:43:11.361952
(5 rows)

REPLICA1 TABLE:
 user_id | username | email | created_at
-----+-----+-----+-----
 3 | newuser | newuser@example.com | 2024-11-20 19:38:02.547226
 4 | newuser2 | newuser2@example.com | 2024-11-20 19:38:51.036098
 5 | newuser3 | newuser3@example.com | 2024-11-20 19:40:15.060125
 6 | newuser4 | newuser4@example.com | 2024-11-20 19:41:43.414564
 7 | newuser5 | newuser5@example.com | 2024-11-20 19:43:11.361952
(5 rows)

REPLICA2 TABLE:
 user_id | username | email | created_at
-----+-----+-----+-----
 3 | newuser | newuser@example.com | 2024-11-20 19:38:02.547226
 4 | newuser2 | newuser2@example.com | 2024-11-20 19:38:51.036098
 5 | newuser3 | newuser3@example.com | 2024-11-20 19:40:15.060125
 6 | newuser4 | newuser4@example.com | 2024-11-20 19:41:43.414564
 7 | newuser5 | newuser5@example.com | 2024-11-20 19:43:11.361952

```

Рисунок. 3.8 – Успішна синхронізація даних між усіма базами даних

Цей результат підтвердив високу ефективність повного відновлення. Система повернулася до стабільного стану після усунення конфліктів. А також можливість гнучкого управління ролями. Старий головний сервер успішно запрацював як репліка без втрати узгодженості даних.

Наше дослідження PostgreSQL логічної реплікації, реалізоване в Docker-середовищі, висвітлює ключові аспекти управління базами даних, зокрема, узгодженість даних, їх доступність і масштабованість. Практичний підхід до вивчення реплікації підтвердив її значущість у забезпеченні стійкості та адаптивності систем, особливо в сценаріях аварійного відновлення та оновлень із мінімальним часом простою.

Логічна реплікація забезпечує деталізований контроль, дозволяючи вибірково синхронізацію баз даних, таблиць або окремих записів. Це надає адміністраторам можливість розробляти стратегії, адаптовані до конкретних операційних потреб. Використання Docker спростило процес розгортання та управління складними розподіленими системами, що зробило такі функції, як логічна реплікація, більш доступними й зрозумілими.

Стійкість системи до простоїв реплік і її здатність плавно виконувати рольові переходи в умовах збоїв підкреслили важливість передбачення і ретельного планування. У процесі управління реплікацією, збереження узгодженості даних між усіма екземплярами, особливо під час і після несподіваних збоїв, стало найважливішим аспектом. Це продемонструвало необхідність систематичного підходу до управління переходами ролей і оновленнями системи.

Це дослідження також висвітлило нові можливості для вдосконалення системи. Зокрема, було визначено потребу в оптимізації процесів переходу ролей. Проблеми, що виникали під час реінтеграції первинного головного сервера як репліки, підкреслили важливість розробки більш ефективних і менш трудомістких стратегій для таких операцій.

Ще одним напрямом удосконалення є зменшення часу простою системи. Замість повної реплікації бази даних з нуля, дослідження альтернативних підходів до заміни головного сервера може суттєво підвищити операційну ефективність і скоротити витрати часу.

Крім того, розширення підходів до реплікації може принести додаткові переваги. Зокрема, це може включати вивчення можливостей часткової реплікації або впровадження більш автоматизованих механізмів синхронізації даних між екземплярами, що дозволить зробити процеси більш гнучкими й надійними.

3.3 Реалізація та аналіз ефективності шардингу

У цьому підрозділі ми розглянемо практичну реалізацію шардингу в середовищі MySQL за допомогою ProxySQL – потужного SQL-проксі, який виконує функції балансування навантаження, маршрутизації запитів і автоматичного перемикання між вузлами. Зосереджуємося на розподілі даних таблиці за країнами, оцінюючи оптимізацію та ефективність, досягнуті завдяки цьому підходу. Такий метод дозволяє підвищити масштабованість системи, забезпечити високу продуктивність і мінімізувати затримки обробки запитів.

Шардинг, який реалізується через ProxySQL, передбачає поділ бази даних на незалежні частини – шарди, на основі певного критерію, наприклад, географічного розташування. У цьому випадку ProxySQL виступає як проміжний шар, що оптимізує управління запитами, спрощує маршрутизацію до відповідного шардованого сервера і покращує моніторинг продуктивності.

Першим етапом реалізації шардингу було налаштування трьох окремих екземплярів баз даних MySQL, які функціонуватимуть у межах однієї віртуальної мережі Docker. Для цього використано файл `docker-compose.yml`, що забезпечує одночасне створення й запуск усіх контейнерів з чітко визначеними параметрами. Такий підхід є більш оптимізованим, ніж покрокове налаштування контейнерів вручну, оскільки дозволяє централізовано управляти конфігураціями.

Docker compose гарантує, що кожен контейнер бази даних отримає унікальну IP-адресу в межах мережі `multi-tenant-network`. Це спрощує зв'язок між ними, оскільки доступ до контейнерів можливий за їхніми іменами або статично призначеними IP-адресами. У файлі `docker-compose.yml` визначено три бази даних. Кожна з них працює на останній версії MySQL, а доступ до них здійснюється через порти 3307, 3308 і 3309 відповідно.

Ця реалізація також включає налаштування мережі `multi-tenant-network`, що використовує драйвер `bridge`. Завдяки цьому всі контейнери мають ізольоване середовище для безпечного обміну даними.

Наступним кроком було створення користувачів у кожному екземплярі бази даних. У кожному контейнері додано двох користувачів: `user_shard`, що використовується для запитів, та `monitor`, який забезпечує моніторинг стану бази даних. Це налаштування гарантує, що ProxySQL зможе ефективно взаємодіяти з базами даних, виконуючи маршрутизацію запитів до відповідного шарду.

Ми створили базову інфраструктуру для подальшого налаштування ProxySQL і реалізації шардингу, що є ключовим компонентом оптимізації роботи розподіленої системи баз даних.

Далі було реалізовано підключення ProxySQL до середовища для управління шардингом у MySQL. ProxySQL виконує функцію посередника між клієнтськими запитами та шардованими базами даних, оптимізуючи трафік, забезпечуючи балансування навантаження і маршрутизацію запитів на відповідні MySQL-сервери.

Контейнер ProxySQL був налаштований за допомогою файлу конфігурації `proxysql.cnf`, що містить необхідні параметри для підключення до MySQL-серверів і визначення правил маршрутизації. Зокрема, було вказано IP-адреси серверів, адміністративні облікові дані та інформацію про користувачів бази даних. Ця конфігурація дозволяє ProxySQL ефективно управляти запитами, спрямовуючи їх до відповідних серверів залежно від визначених критеріїв.

Після запуску контейнера ProxySQL надає доступ до адміністративного інтерфейсу через порт 6032, що дозволяє в реальному часі змінювати конфігурації та відстежувати стан серверів. Порт 6033 використовується для маршрутизації SQL-запитів, гарантуючи їх доставку на відповідний шард. Такий підхід значно спрощує управління шардованими базами даних і забезпечує їх оптимальну роботу в умовах високого навантаження.

Наступним кроком після налаштування контейнера ProxySQL стало отримання доступу до його адміністративної консолі для перевірки підключення до шардованих MySQL-серверів і моніторингу їхнього стану (рис. 3.9).

```
MySQL [(none)]> SELECT * FROM mysql_servers;
```

hostgroup_id	hostname	port	gtid_port	status	weight	compression	max_connections	max_replication_lag	use_ssl	max_latency_ms	comment
10	172.19.0.3	3306	0	ONLINE	1	0	100	0	0	0	
20	172.19.0.4	3306	0	ONLINE	1	0	100	0	0	0	
30	172.19.0.5	3306	0	ONLINE	1	0	100	0	0	0	

3 rows in set (0.001 sec)

Рисунок. 3.9 – Перелік підключених MySQL-серверів

Цей крок дозволяє переконаватися, що всі MySQL-сервери належним чином підключені до ProxySQL. Адміністративний доступ також відкриває можливості для подальшого налаштування маршрутизації запитів і оптимізації роботи системи шардингу.

Для аналізу ефективності шардингу було створено додатковий MySQL-контейнер, що не входить в мережу шардингу. Він використовується для тестування продуктивності запитів на таблиці, яка об'єднує всі записи шардів. У той же час, у шардованих базах даних дані розділено за країнами, що дозволяє безпосередньо порівняти результати.

Структура таблиці, використаної для тестування, була визначена наступним чином (рис. 3.10):

```
CREATE TABLE addresses (
    id INT AUTO_INCREMENT PRIMARY KEY,
    street_name VARCHAR(100) NOT NULL,
    city VARCHAR(100) NOT NULL,
    state VARCHAR(100) NOT NULL,
    postal_code VARCHAR(20) NOT NULL,
    country VARCHAR(50) NOT NULL
);
```

Рисунок. 3.10 – Структура таблиці шардування

У базі даних без шардингу таблиця містила записи для трьох країн, тоді як у шардованих базах даних записи розподілялися між трьома окремими MySQL-серверами відповідно до значення поля country. Такий підхід дозволив моделювати сценарій, коли дані фізично розподілені між різними серверами, а також зберігати однаковий загальний обсяг даних для порівняння.

Кількість записів у кожній базі була ретельно узгоджена. У базі без шардингу таблиця містила близько 780 тисяч записів (по 260 тисяч для кожної

країни), тоді як у кожному шарді було збережено по 260 тисяч записів для відповідної країни. Розподіл даних дозволяє оцінити, як шардована структура впливає на час виконання запитів і продуктивність у порівнянні з єдиною великою таблицею.

Наступним етапом стало додавання правил маршрутизації запитів для реалізації шардингу. Ці правила визначають, до якого MySQL-сервера буде спрямовано запит залежно від значення певного параметра в SQL-запиті. У нашому випадку маршрутизація здійснюється на основі зазначеного атрибуту country, який використовується для ідентифікації відповідного шарду.

У базі ProxySQL були створені правила маршрутизації для шардів (рис. 3.11), що відповідають країнам США, Канаді та Мексиці, приклад такого правила виглядає наступним чином:

```
INSERT INTO mysql_query_rules (rule_id, active, username, match_pattern, destination_hostgroup, apply)
VALUES (10, 1, 'user_shard', "\s*country\s*=\s*.*?(USA).*?\s*", 10, 0);
```

Рисунок. 3.11 – Правило маршрутизації для шарду USA

Ці правила використовують регулярні вирази для пошуку в запитах відповідності значення country. Після виявлення відповідного значення запит перенаправляється до визначеної групи хостів (hostgroup), що відповідає конкретному шарду. Завдяки цьому підходу кожен SQL-запит обробляється відповідним сервером, що підвищує ефективність системи та знижує навантаження на кожен окремий сервер.

Наступним кроком стало тестування роботи маршрутизації та аналіз її впливу на продуктивність системи. У адміністративній консолі було здійснено SQL-запити до таблиці addresses для трьох країн з однаковим id (рис. 3.12).

```

MySQL [shop]> SELECT * FROM addresses WHERE country='Canada' and id=123;
+-----+-----+-----+-----+-----+-----+
| id | street_name | city | state | postal_code | country |
+-----+-----+-----+-----+-----+-----+
| 123 | Street 897 | City 21 | Province 39 | A18829 | Canada |
+-----+-----+-----+-----+-----+-----+
1 row in set (0.000 sec)

MySQL [shop]> SELECT * FROM addresses WHERE country='USA' and id=123;
+-----+-----+-----+-----+-----+-----+
| id | street_name | city | state | postal_code | country |
+-----+-----+-----+-----+-----+-----+
| 123 | Street 3902 | City 75 | State 29 | 62168 | USA |
+-----+-----+-----+-----+-----+-----+
1 row in set (0.001 sec)

MySQL [shop]> SELECT * FROM addresses WHERE country='Mexico' and id=123;
+-----+-----+-----+-----+-----+-----+
| id | street_name | city | state | postal_code | country |
+-----+-----+-----+-----+-----+-----+
| 123 | Calle 3541 | Ciudad 42 | Estado 2 | 91168 | Mexico |
+-----+-----+-----+-----+-----+-----+
1 row in set (0.000 sec)

```

Рисунок. 3.12 – Отримання даних addresses по різним шардам

Результати виконаних запитів чітко демонструють, що ProxySQL успішно перенаправляє запити до потрібних серверів, відповідно до налаштованих правил маршрутизації. Це підтверджує коректність роботи шардингу та налаштувань у ProxySQL.

Далі для оцінки ефективності шардингу було виконано складний SQL-запит, який вимагав значних обчислювальних ресурсів. Він включав агреговані функції, вкладені підзапити та умови групування, що робило його ресурсоємним. Запит виконували як у базі даних без шардингу, так і в шардованих репліках, щоб оцінити різницю у продуктивності.

У базі даних без шардингу таблиця містила всі записи для трьох країн, що значно збільшувало обсяг даних для обробки (рис. 3.14). У шардованій системі дані були розподілені між трьома репліками, кожна з яких обробляла лише частину загального обсягу, відповідно до налаштованих правил маршрутизації. Це дозволило зменшити навантаження на кожен окремий сервер (рис. 3.13).

використовувався скрипт, що генерує записи з такими атрибутами, як ідентифікатор, ім'я, статус активності, штат і країна. Загалом було створено 10 мільйонів записів, що зберігались у форматі CSV [36].

Згенеровані файли об'єднали в один великий файл, який був підготовлений для завантаження в базу даних MySQL. Для цього було використано контейнер MySQL у середовищі Docker. Контейнер забезпечив ізоляцію робочого середовища, що дозволило уникнути впливу зовнішніх факторів на експеримент. Завантаження даних в базу було виконано за допомогою команди імпорту, яка зчитує CSV-файл і вставляє дані в таблицю бази.

У базі даних була створена таблиця з відповідними стовпцями для кожного атрибуту: ідентифікатор користувача, ім'я, статус активності, штат і країна. Було також визначено первинний ключ, що гарантує унікальність кожного запису. Загальний обсяг даних у таблиці після завантаження склав 10 мільйонів записів. Час, необхідний для повного завантаження даних, підтвердив високу ефективність операції в межах MySQL.

Після успішного завантаження даних таблиця стала основою для подальшого тестування методів пакетного оновлення. Це дозволило створити реалістичний сценарій для оцінки продуктивності системи та впливу оновлень на доступність даних для інших операцій.

Уявімо, що ми працюємо з базою даних e-commerce-платформи. Через баг у системі, в полі st (state) для деяких користувачів встановлено неправильне значення. Нам потрібно оновити 5 мільйонів записів із загальних 10 мільйонів, щоб у полі st було встановлено значення «NY». Основна умова – мінімізувати вплив оновлення на доступність системи для користувачів. Якщо запис користувача заблокований під час оновлення, він не зможе змінювати свої дані.

Для прикладу було виконано масове оновлення (рис. 3.15):

```
updates> update user set st = 'SL' where user_id between 3000000 and 8000000
[2024-12-15 15:29:10] 4,958,154 rows affected in 14 s 869 ms
```

Рисунок. 3.15 – Результат виконання масового оновлення звичайним update

Цей запит тривав понад 14 секунд, протягом яких заблоковані записи були недоступні для інших операцій. В контексті нашого дослідження це не є великою тривалістю, проте у реальній системі таблиці мають набагато складнішу структуру та більше записів і там ця цифра зростає до десятків хвилин. Далі було штучно сповільнено виконання цього запиту для того щоб паралельному терміналі продемонструвати спробу змінити запис із `user_id = 102`, який заблокований масовим оновленням (рис. 3.16):

```
> update updates.user set name = 'Yaroslav Testing3' where user_id = 3000400
[2024-12-15 15:32:33] [40001][1205] Lock wait timeout exceeded; try restarting transaction
```

Рисунок. 3.16 – Демонстрація блокування запису з `user_id = 102`

Ця помилка виникла, оскільки великий запит оновлення заблокував запис, і будь-які інші операції, що намагалися модифікувати цей запис, автоматично очікували звільнення блокування до завершення основного запиту. Значення параметра `innodb_lock_wait_timeout` [37], що визначає максимальний час очікування блокування, можна корегувати та за замовчуванням воно становить 50 секунд (рис. 3.7). Це можна перевірити запитом і отримати результат:

	Variable_name	Value
1	<code>innodb_lock_wait_timeout</code>	50

Рисунок. 3.17 – Значення змінної `innodb_lock_wait_timeout`

Тривале очікування суттєво погіршує користувацький досвід, оскільки зростає ймовірність таймаутів і збоїв транзакцій.

Замість того, щоб оновлювати всі 5 мільйонів записів одним запитом, можна використовувати підхід із пакетним оновленням. Такий метод дозволяє оновлювати лише невелику кількість записів за один раз, що зменшує кількість заблокованих записів і скорочує час очікування для інших транзакцій. Це забезпечує набагато менший вплив на доступність системи і підтримує її стабільність під час виконання оновлень великого обсягу.

Цей метод передбачає поділ даних на невеликі частини за допомогою ключового стовпця, в нашому випадку це `user_id`, що є монотонно зростаючим ідентифікатором. Оновлення виконувалося у пакетах по 50 000 записів, що дозволило значно зменшити час блокування окремих записів, уникнути тривалих блокувань і забезпечити доступність бази даних для інших транзакцій навіть під час виконання оновлення.

Логіка пакетного оновлення була реалізована у вигляді збереженої процедури `BatchUpdate` (рис. 3.18). Ця процедура включає ітераційний цикл, у якому стартовий і кінцевий ідентифікатори пакета (`batch_start_id` і `batch_end_id`) поступово зміщуються на розмір пакета після кожної ітерації. Вона також містить логічні оператори для відображення поточного статусу оновлення, що дозволяє легко моніторити процес.

```

DROP PROCEDURE IF EXISTS BatchUpdate;

DELIMITER $$
CREATE PROCEDURE BatchUpdate(
    startBoundary INT,
    finishBoundary INT,
    stepSize INT)
BEGIN
    DECLARE currentLowerBound INT DEFAULT startBoundary;
    DECLARE currentUpperBound INT DEFAULT startBoundary + stepSize;
    DECLARE iterationCount INT DEFAULT 0;

    WHILE currentUpperBound <= finishBoundary DO

        SELECT CONCAT('Processing Range: ', currentLowerBound, ' to ', currentUpperBound) AS process_log;

        UPDATE user
        SET st = 'NJ'
        WHERE user_id >= currentLowerBound AND user_id <= currentUpperBound;

        SET currentLowerBound = currentLowerBound + stepSize;
        SET currentUpperBound = currentUpperBound + stepSize;
        SET iterationCount = iterationCount + 1;

    END WHILE;

    SELECT CONCAT('Final Update from ', currentUpperBound - stepSize, ' to ', finishBoundary) AS final_log,
           iterationCount AS total_iterations;

    UPDATE user
    SET is_active = TRUE
    WHERE user_id >= (currentUpperBound - stepSize) AND user_id <= finishBoundary;

END$$
DELIMITER ;

call BatchUpdate(3000000, 8000000, 50000);

```

Рисунок. 3.18 – Створена процедура для пакетного оновлення

Процедура була викликана з параметрами 3000000 і 8000000, що охоплювало 5 мільйонів записів, які потребували оновлення. Завдяки поділу даних на частини, процедура виконувалась трохи більше 33 секунд, що є значно

довше за звичайний запит на оновлення всіх даних, але вона забезпечила можливість одночасного виконання інших транзакцій.

Наприклад, під час роботи процедури було виконано окреме оновлення для запису з `user_id = 3300000`, і цей запит завершився швидко без конфліктів (рис. 3.19).

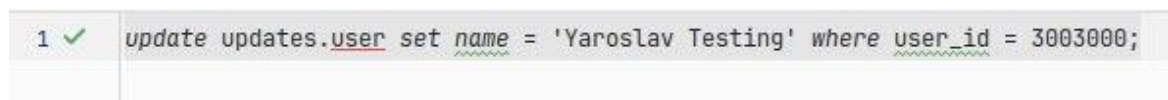


Рисунок. 3.19 – Безперешкодне та швидке виконання оновлення

Цей експеримент підтвердив ефективність пакетного підходу, оскільки кожен окремий пакет оновлень блокує лише невелику кількість записів, залишаючи решту таблиці доступною для інших транзакцій. У порівнянні з традиційним методом оновлення великих обсягів даних, цей підхід демонструє високу гнучкість і стійкість до навантаження.

Загалом, використання пакетного оновлення дозволило збалансувати потребу в оновленні великої кількості записів і підтриманні доступності бази даних для користувачів. Такий підхід особливо корисний у системах з високим навантаженням, де одночасні транзакції є важливими для безперебійного функціонування.

Після завершення експерименту з пакетними оновленнями варто підсумувати ключові аспекти та зробити кілька важливих висновків щодо ефективності цього методу. У процесі оновлення великого обсягу даних у базі виникають ризики блокування записів, що може призвести до суттєвих проблем у системі, таких як затримка або зупинка інших транзакцій. Це є особливо критичним у середовищах із високою кількістю одночасних запитів, де блокування даних на тривалий час може серйозно вплинути на користувацький досвід.

Пакетний підхід до оновлень демонструє себе як надійне рішення цієї проблеми. Завдяки поділу великого оновлення на менші частини система уникає тривалих блокувань. Пакети даних оновлюються поступово, що дозволяє іншим транзакціям продовжувати роботу з базою. Наприклад, під час експерименту

пакетний запит займав більше часу в цілому, проте кожен індивідуальний запит всередині пакета виконувався значно швидше, що мінімізувало вплив на інших користувачів. Одночасно була доведена можливість виконання окремих транзакцій, навіть коли пакетне оновлення все ще тривало, що підтвердило гнучкість цього методу.

Важливим аспектом, який варто зазначити, є вплив порядку оновлень на кінцеві результати. У випадку, коли пакетні оновлення та індивідуальні транзакції змінюють один і той самий стовпець, остаточне значення буде залежати від послідовності застосування змін. Це потребує ретельного планування та контролю для запобігання непередбачуваним результатам.

Крім того, варто враховувати можливість використання паралельних пакетних оновлень для прискорення процесу, проте це потребує додаткових механізмів запобігання конфліктам, таких як уникнення взаємних блокувань.

У випадках, коли логіка оновлень є складнішою, а таблиця має індекси, існує необхідність у додаткових стратегіях, таких як використання `SELECT ... FOR UPDATE` [38] із параметрами `SKIP LOCKED` або `NOWAIT` [39]. Ці інструменти дозволяють більш ефективно працювати з даними, уникаючи очікування на доступ до заблокованих записів.

Таким чином, пакетні оновлення є універсальним інструментом, який дозволяє значно покращити ефективність роботи бази даних та знизити вплив тривалих операцій на загальну продуктивність системи. Вони забезпечують оптимальний баланс між швидкістю виконання оновлень та доступністю бази для інших транзакцій, що робить цей підхід ідеальним вибором для систем з великим обсягом даних і високими вимогами до масштабованості та стабільності.

3.5 Реалізація та аналіз ефективності індексації таблиць

Одним із основних інструментів оптимізації запитів у реляційних базах даних є індекси. Вони дозволяють значно прискорити виконання запитів за рахунок швидкого доступу до потрібних рядків. Проте індекси також можуть

мати негативний вплив на продуктивність системи, особливо при частих операціях вставки та оновлення даних. Метою цього дослідження є детальний аналіз впливу різних типів індексів у MySQL на продуктивність запитів та операцій модифікації даних у великій таблиці, що моделює реальну високонавантажену платформу.

Для проведення експериментів була створена таблиця `user_data_no_opt` та її копія для оптимізації `user_data_with_opt`, яка містить інформацію про користувачів (рис. 3.20):

```
CREATE TABLE user_data_no_opt
(
    id                SERIAL PRIMARY KEY,
    user_id           BIGINT NOT NULL,
    full_name         VARCHAR(255),
    email             VARCHAR(255),
    created_at        TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    last_login        TIMESTAMP,
    status            VARCHAR(20),
    is_premium        BOOLEAN DEFAULT FALSE,
    age              INT,
    country           VARCHAR(100),
    balance           DECIMAL(15, 2),
    subscription_level INT,
    preferences       JSONB,
    activity_score    DECIMAL(10, 2),
    last_transaction_date TIMESTAMP
);
```

Рисунок. 3.20 – Структура таблиці `user_data_no_opt`

Обидві таблиці були заповнені 1 000 000 записів з випадковими даними для імітації реальної високонавантаженої платформи. Процедура `recreate_test_tables()` використовувалась для очищення та повторного створення таблиці перед кожним тестом, що забезпечувало чистоту експерименту та відсутність впливу попередніх операцій на результати.

Дослідження проводилось шляхом послідовного створення різних індексів на таблиці `user_data_with_opt` та виконання запитів до обох таблиць з використанням команди `EXPLAIN ANALYZE` [40]. Це дозволяло отримати детальну інформацію про план виконання запиту, використання індексів та час виконання. Для кожного типу індексу проводився аналіз його впливу на продуктивність вибіркового запиту, а також на прикладі одного з індексів було проведено аналіз впливу на операції вставки та оновлення даних. Особлива увага

приділялась ситуаціям, коли індекси можуть негативно впливати на продуктивність системи.

Першим кроком було дослідження впливу індексу на первинному ключі id. Оскільки ми визначили id як звичайне поле, а не первинний ключ, ми можемо створити індекс (рис. 3.21) і дослідити його вплив.

```
1 CREATE UNIQUE INDEX ON user_data_with_opt(id);
2
```

Рисунок. 3.21 – Створення індексу по ідентифікатору

Виконані запити для вибірки одного користувача за його id (рис. 3.22):

QUERY PLAN	
1	Seq Scan on user_data_no_opt (cost=0.00..29484.38 rows=694 width=1419) (actual time=0.025..79.830 rows=1 loops=1)
2	Filter: (id = 73)
3	Rows Removed by Filter: 999999
4	Planning Time: 0.107 ms
5	Execution Time: 79.842 ms
QUERY PLAN	
1	Index Scan using user_data_with_opt_id_idx on user_data_with_opt (cost=0.42..8.44 rows=1 width=1419) (actual time=0.025..0.025 rows=1 loops=1)
2	Index Cond: (id = 73)
3	Planning Time: 0.113 ms
4	Execution Time: 0.025 ms

Рисунок. 3.22 – Результати виконаних запитів для вибірки по id

Аналіз плану виконання показав, що використовується індексований пошук (Index Scan [41]), що дозволяє знайти потрібний запис дуже швидко. Час виконання запиту був мінімальним 0.025 ms, порівнюючи з 79.84 ms, що підтверджує ефективність індексу на первинному ключі для точкових запитів.

Наступним було досліджено вплив індексації стовпця country, за яким теоретично можуть часто виконуватись запити (рис. 3.23).

```
9 CALL recreate_test_tables();
10 CREATE INDEX ON user_data_with_opt(country);
```

Рисунок. 3.23 – Створення індексу по країні

Виконано запити для вибірки користувачів з певної країни (рис. 3.24):

1	Seq Scan on user_data_no_opt (cost=0.00..29485.44 rows=694 width=240) (actual time=0.101..153.577 rows=499312 loops=1)
2	Filter: ((country)::text = 'USA'::text)
3	Rows Removed by Filter: 500688
4	Planning Time: 0.126 ms
5	Execution Time: 165.370 ms

1	Bitmap Heap Scan on user_d QUERY PLAN: text =59.17..12875.00 rows=5000 width=240) (actual time=16.226..112.882 rows=500094 loops=1)
2	Recheck Cond: ((country)::text = 'USA'::text)
3	Heap Blocks: exact=27751
4	-> Bitmap Index Scan on user_data_with_opt_country_idx (cost=0.00..57.92 rows=5000 width=0) (actual time=12.804..12.804 rows=500094 loops=1)
5	Index Cond: ((country)::text = 'USA'::text)
6	Planning Time: 0.052 ms
7	Execution Time: 121.861 ms

Рисунок. 3.24 – Результати виконаних запитів для вибірки по країні

Результати показали, що при наявності індексу використовується Index Scan, що значно прискорює виконання запиту – 121 ms порівняно з повним скануванням таблиці (Seq Scan [42]) коли запит виконувався 165 ms. Проте, оскільки стовпець country може мати велику кількість повторюваних значень, ефективність індексу може бути нижчою порівняно зі стовпцями з високою унікальністю. Незважаючи на це, індекс суттєво покращив продуктивність запиту.

Для складніших запитів було вирішено створити композитний індекс на стовпцях age та country (рис. 3.25), щоб перевірити його вплив на продуктивність.

```

15 CALL recreate_test_tables();
16 CREATE INDEX ON user_data_with_opt(age, country);

```

Рисунок. 3.25 – Створення композитного індексу по віку та країні

Виконані запити для вибірки користувачів певного віку з певної країни (рис. 3.26):

1	Seq Scan on user_data_no_opt (cost=0.00..29832.33 rows=3 width=240) (actual time=0.020..125.540 rows=8293 loops=1)
2	Filter: (((country)::text = 'USA'::text) AND (age = 34))
3	Rows Removed by Filter: 991707
4	Planning Time: 0.082 ms
5	Execution Time: 125.731 ms

1	Bitmap Heap Scan on user_data_with_opt (cost=4.68..102.81 rows=25 width=240) (actual time=1.106..27.819 rows=8185 loops=1)
2	Recheck Cond: ((age = 34) AND ((country)::text = 'USA'::text))
3	Heap Blocks: exact=7051
4	-> Bitmap Index Scan on user_data_with_opt_age_country_idx (cost=0.00..4.67 rows=25 width=0) (actual time=0.427..0.428 rows=8185 loops=1)
5	Index Cond: ((age = 34) AND ((country)::text = 'USA'::text))
6	Planning Time: 0.110 ms
7	Execution Time: 28.067 ms

Рисунок. 3.26 – Результати виконаних запитів для вибірки по віку та країні

Аналіз показав, що запит використовує композитний індекс, що дозволяє набагато швидше знайти записи, які відповідають обом умовам. Час виконання запиту значно зменшився до 28 ms порівняно з тим, коли індекси на ці стовпці були відсутні та він виконувався 125 ms. Це демонструє ефективність композитних індексів для запитів з декількома умовами.

Було вирішено дослідити використання часткового індексу для специфічної умови. Створено індекс (рис. 3.27), який включає лише записи з балансом більше або рівним 5000 та парним значенням балансу:

```
21 CALL recreate_test_tables();
22 CREATE INDEX ON user_data_with_opt(balance) WHERE balance >= 5000 AND balance % 2 = 0;
```

Рисунок. 3.27 – Створений індекс по специфічній умові

Виконані запити (рис. 3.28):

QUERY PLAN
1 Gather (cost=1000.00..29785.86 rows=231 width=240) (actual time=0.362..77.410 rows=2496 loops=1)
2 Workers Planned: 2
3 Workers Launched: 2
4 -> Parallel Seq Scan on user_data_no_opt (cost=0.00..28762.76 rows=96 width=240) (actual time=0.133..73.611 rows=832 loops=3)
5 Filter: ((balance >= '5000'::numeric) AND ((balance % '2'::numeric) = '0'::numeric))
6 Rows Removed by Filter: 332501
7 Planning Time: 0.099 ms
8 Execution Time: 77.480 ms
QUERY PLAN
1 Bitmap Heap Scan on user_data: QUERY PLAN: text 3.03..5365.06 rows=1667 width=240) (actual time=0.400..7.988 rows=2542 loops=1)
2 Recheck Cond: ((balance >= '5000'::numeric) AND ((balance % '2'::numeric) = '0'::numeric))
3 Heap Blocks: exact=2447
4 -> Bitmap Index Scan on user_data_with_opt_balance_idx (cost=0.00..32.62 rows=1667 width=0) (actual time=0.168..0.168 rows=2542 loops=1)
5 Planning Time: 0.122 ms
6 Execution Time: 8.066 ms

Рисунок. 3.28 – Результати запитів з використанням специфічної умови

Результати показали, що частковий індекс майже в 10 разів покращив продуктивність запиту, оскільки індекс містить лише необхідні записи. Це зменшує розмір індексу та прискорює пошук. Такий підхід особливо ефективний для великих таблиць, де умова вибірки відбирає невелику частину даних.

Далі було проведено дослідження ефективності індексів з більш складними алгоритмами на стовпці activity_score (рис. 3.29). Було досліджено наступні алгоритми: btree, hash та brin.

```

CALL recreate_test_tables();
CREATE INDEX hash_test ON user_data_with_opt USING HASH (activity_score);
EXPLAIN ANALYZE SELECT country FROM user_data_with_opt WHERE activity_score>50;

CALL recreate_test_tables();
CREATE INDEX brin_test ON user_data_with_opt USING brin (activity_score);
EXPLAIN ANALYZE SELECT country FROM user_data_with_opt WHERE activity_score>50;

CALL recreate_test_tables();
CREATE INDEX btree_test ON user_data_with_opt USING btree (activity_score);
EXPLAIN ANALYZE SELECT country FROM user_data_with_opt WHERE activity_score>50;

```

Рисунок. 3.29 – Порівняння різних алгоритмів індексації

Результати ефективності індексації (рис. 3.30):

4	Planning Time: 0.393 ms
5	Execution Time: 164.098 ms
4	Planning Time: 0.096 ms
5	Execution Time: 138.313 ms
6	Planning Time: 0.037 ms
7	Execution Time: 134.866 ms

Рисунок. 3.30 – Результати порівняння різних алгоритмів індексації

Результати показали, що B-tree індекс є найбільш універсальним та ефективним для такого типу запитів, включаючи пошук за діапазоном значень. Hash індекс не підтримує пошук за діапазоном і в даному випадку був неефективним. BRIN індекс показав хорошу продуктивність на великій таблиці, особливо якщо дані в стовпці мають певну послідовність. Проте його ефективність залежить від фізичного розташування даних та розміру таблиці.

Було важливо оцінити, як наявність індексів впливає на операції вставки та оновлення даних, оскільки у високонавантажених системах ці операції можуть виконуватися дуже часто. Порівняння проваелося коли таблиця user_data_with_opt містила композитний індекс по age та country.

Виконано операцію вставки нового запису (рис. 3.31):

QUERY PLAN	
1	Insert on user_data_no_opt (cost=0.00..0.03 rows=0 width=0) (actual time=0.121..0.121 rows=0 loops=1)
2	-> Result (cost=0.00..0.03 rows=1 width=1419) (actual time=0.062..0.062 rows=1 loops=1)
3	Planning Time: 0.023 ms
4	Execution Time: 0.131 ms
QUERY PLAN	
1	Insert on user_data_with_opt (cost=0.00..0.03 rows=0 width=0) (actual time=0.170..0.171 rows=0 loops=1)
2	-> Result (cost=0.00..0.03 rows=1 width=1419) (actual time=0.025..0.026 rows=1 loops=1)
3	Planning Time: 0.031 ms
4	Execution Time: 0.185 ms

Рисунок. 3.31 – Результати ефективності виконання запиту вставки нового запису

Також виконано операцію оновлення існуючого запису (рис. 3.32):

QUERY PLAN	
1	Update on user_data_no_opt (cost=0.00..40251.01 rows=0 width=0) (actual time=47.306..47.307 rows=0 loops=1)
2	-> Seq Scan on user_data_no_opt (cost=0.00..40251.01 rows=1 width=40) (actual time=47.271..47.272 rows=1 loops=1)
3	Filter: (user_id = 100000)
4	Rows Removed by Filter: 999999
5	Planning Time: 0.100 ms
6	Execution Time: 47.322 ms
QUERY PLAN	
1	Update on user_data_with_opt (cost=0.00..40301.45 rows=0 width=0) (actual time=64.675..64.676 rows=0 loops=1)
2	-> Seq Scan on user_data_with_opt (cost=0.00..40301.45 rows=5000 width=40) (actual time=64.640..64.640 rows=1 loops=1)
3	Filter: (user_id = 100000)
4	Rows Removed by Filter: 1000000
5	Planning Time: 0.062 ms
6	Execution Time: 64.688 ms

Рисунок. 3.32 – Результати ефективності виконання запиту оновлення записів

Аналіз показав, що час виконання операцій вставки та оновлення збільшився у порівнянні з таблицею без індексів. Це пояснюється тим, що при кожній зміні даних база даних повинна оновлювати всі відповідні індекси, що додає додаткові витрати ресурсів. В системах з високою частотою операцій запису це може негативно впливати на загальну продуктивність.

Для демонстрації випадку, коли індекси можуть негативно впливати на продуктивність, було вирішено створити індекс на стовпці з низькою селективністю. Створено індекс на стовпці status, який може мати лише декілька різних значень (наприклад, «active», «inactive»):

Виконано запити для вибірки користувачів зі статусом «active» (рис. 3.33):

QUERY PLAN	
1	Seq Scan on user_data_no_opt (cost=0.00..40251.00 rows=502300 width=172) (actual time=0.063..83.579 rows=500168 loops=1)
2	Filter: ((status)::text = 'active'::text)
3	Rows Removed by Filter: 499832
4	Planning Time: 0.040 ms
5	Execution Time: 91.967 ms
QUERY PLAN	
1	Seq Scan on user_data_with_opt (cost=0.00..40250.00 rows=499733 width=172) (actual time=0.008..88.219 rows=501071 loops=1)
2	Filter: ((status)::text = 'active'::text)
3	Rows Removed by Filter: 498929
4	Planning Time: 0.042 ms
5	Execution Time: 96.953 ms

Рисунок. 3.33 – Результати виконання вибірки користувачів з певним статусом

Аналіз показав, що оптимізатор бази даних не використовує індекс і виконує повне сканування таблиці (Seq Scan). Це пов'язано з тим, що вибірка повертає велику кількість записів, і використання індексу неефективне. Наявність такого індексу не тільки не покращує продуктивність запитів, але й в

нашому випадку збільшує накладні витрати на операції вставки та оновлення, оскільки індекс потребує оновлення при кожній зміні даних у стовпці status. Це приклад того, коли індекс може шкодити загальній продуктивності системи.

Проведені експерименти показали, що індекси є потужним інструментом для оптимізації запитів, але їх ефективність залежить від багатьох факторів. Індеси на стовпцях з високою селективністю та унікальними значеннями забезпечують найбільше покращення продуктивності. Композитні індекси особливо корисні для запитів з умовами на декілька стовпців.

Водночас індекси на стовпцях з низькою селективністю можуть не тільки не приносити користі, але й негативно впливати на продуктивність через додаткові накладні витрати на підтримку індексів. Це підкреслює важливість ретельного аналізу перед створенням індексів.

Вплив індексів на операції модифікації даних є суттєвим фактором, який необхідно враховувати у високонавантажених системах. Велика кількість індексів може значно сповільнити операції вставки та оновлення, що може негативно позначитися на загальній продуктивності системи.

Порівняння різних типів індексів показало, що вибір типу індексу повинен базуватися на характері запитів та структурі даних. B-tree індекси є найбільш універсальними, але в деяких випадках спеціалізовані індекси, такі як BRIN, можуть бути більш ефективними.

Дослідження підтвердило, що індекси є необхідним інструментом для оптимізації роботи баз даних у високонавантажених платформах. Правильне використання індексів може значно покращити продуктивність запитів, але їх застосування повинно бути обґрунтованим та враховувати особливості даних та запитів.

Необхідно уникати індексації стовпців з низькою селективністю, оскільки це може призвести до зниження продуктивності. Також слід враховувати баланс між швидкістю вибірки даних та витратами на операції запису. У деяких випадках доцільно відмовитися від індексів на користь швидшого виконання операцій модифікації даних.

Рекомендується проводити регулярний аналіз виконуваних запитів та оптимізувати індекси відповідно до реальних потреб системи. Це дозволить забезпечити максимальну ефективність роботи бази даних та підтримувати високу продуктивність високонавантажених платформ.

Висновки до третього розділу

У третьому розділі магістерської роботи реалізовано та протестовано ключові методи оптимізації баз даних для високонавантажених платформ, зокрема реплікацію, шардинг, індексацію таблиць та пакетні оновлення. Проведений аналіз результатів дослідження підтвердив ефективність зазначених підходів для підвищення продуктивності, масштабованості та надійності систем в умовах інтенсивного навантаження.

Реалізація реплікації продемонструвала її здатність забезпечувати високу доступність даних та стійкість системи до збоїв. Застосування реплікації дозволило знизити затримки при виконанні транзакцій, водночас забезпечуючи розподіл навантаження між вузлами системи. Це особливо актуально для платформ електронної комерції, де важливим є безперервний доступ до інформації про товари. Шардинг показав свою ефективність для масштабування баз даних за рахунок рівномірного розподілу даних між серверами. Використання географічного критерію дозволило знизити затримки у запитах від користувачів із різних регіонів. Результати тестів підтвердили, що шардинг значно покращує продуктивність систем, які обробляють великі обсяги транзакцій. Індexсація таблиць суттєво пришвидшила виконання складних запитів. Найбільш ефективними виявилися В-Tree та часткові індекси, які дозволили зменшити кількість операцій читання для великих таблиць. Це критично для платформ, що працюють у режимі реального часу. Пакетні оновлення забезпечили зниження навантаження на систему при обробці великих обсягів даних. Розбиття оновлень на малі транзакції дозволило уникнути блокування таблиць і забезпечити стабільну роботу інших процесів у системі.

На основі отриманих результатів можна рекомендувати впроваджувати реплікацію для забезпечення доступності даних у системах з високими вимогами до часу безвідмовної роботи, а також оптимізувати затримки у передачі даних для покращення продуктивності. Шардинг доцільно використовувати для масштабування платформ з географічно розподіленою аудиторією, обираючи критерії розподілу даних відповідно до особливостей бізнесу. Індексацію варто впроваджувати у таблицях із великим обсягом записів, зокрема вибираючи типи індексів залежно від характеру запитів. Пакетні оновлення можуть бути корисними для періодичної обробки даних, зокрема оновлення великих таблиць або у періоди високого навантаження, наприклад під час акцій в електронній комерції.

Запропоновані підходи сприятимуть підвищенню продуктивності, стабільності та гнучкості цифрових платформ. Їх реалізація забезпечить покращення користувацького досвіду та ефективності бізнес-процесів, адаптуючи системи до високих навантажень і динамічних умов роботи.

ВИСНОВКИ

У межах виконаної кваліфікаційної роботи магістра на тему «Дослідження оптимізації роботи баз даних для високонавантажених платформ» було досягнуто поставлених цілей і виконано завдання, визначені у вступі. Проведені дослідження дозволили оцінити ефективність різних методів оптимізації баз даних, визначити їх вплив на продуктивність систем і запропонувати практичні рекомендації для їх інтеграції.

У першому розділі здійснено аналіз предметної галузі високонавантажених платформ, їхньої ролі у сучасних цифрових системах і ключових характеристик. Особливу увагу приділено проблемам, пов'язаним із великими обсягами даних і високою інтенсивністю запитів, які виникають у таких сферах, як фінансові сервіси, електронна комерція та стримінгові платформи. Цей аналіз заклав основу для вибору найбільш ефективних методів оптимізації.

Другий розділ зосереджено на дослідженні методів оптимізації баз даних, таких як реплікація, шардинг, індексація таблиць і пакетні оновлення. Виявлено, що кожен метод має свої переваги та обмеження, які необхідно враховувати при інтеграції в системи з високим навантаженням. Наприклад, реплікація забезпечує високу доступність даних, тоді як шардинг дозволяє досягти горизонтального масштабування. Проведений аналіз показав, що комбінація цих підходів є найбільш ефективною у багатьох сценаріях.

Третій розділ присвячено практичній реалізації зазначених методів оптимізації. За допомогою розроблених тестових сценаріїв проведено аналіз ефективності різних підходів у реальних умовах навантаження. Результати експериментів підтвердили доцільність використання методів для покращення продуктивності систем, зокрема у сфері фінансових платформ і електронної комерції.

На основі проведеного дослідження сформульовано рекомендації для інтеграції методів оптимізації баз даних у проєктування та розвиток

високонавантажених платформ. Рекомендується використовувати реплікацію як базовий підхід для забезпечення високої доступності даних і мінімізації ризику простоїв у випадках збою серверів. Для забезпечення масштабованості та розподілу навантаження між серверами доцільно впроваджувати шардинг, особливо для платформ, що обслуговують географічно розподілених користувачів. Інтеграція індексації таблиць рекомендована для пришвидшення виконання складних запитів, а використання пакетних оновлень — для оптимізації роботи з великими масивами даних.

Розроблені рекомендації є універсальними та можуть бути адаптовані для специфічних потреб окремих бізнесів. Їх впровадження сприятиме підвищенню продуктивності платформ, зниженню витрат на підтримку інфраструктури та покращенню користувацького досвіду. Проведене дослідження є внеском у розвиток підходів до оптимізації баз даних, що забезпечують стабільну роботу цифрових систем навіть за умов інтенсивного навантаження.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Туровський Я. С., Чижмотря О. Г. Проблеми використання індексів баз даних у високонавантажених додатках. Комп'ютерні технології: інновації, проблеми, рішення: тези VII Всеукраїнської науково-технічної конференції, м. Житомир, 02-03 грудня 2024 р. – Житомир: Державний університет «Житомирська політехніка», 2024.
2. Туровський Я. С., Чижмотря О. Г. Проблеми використання індексів баз даних у високонавантажених додатках. Комп'ютерні технології: інновації, проблеми, рішення: тези VII Всеукраїнської науково-технічної конференції, м. Житомир, 02-03 грудня 2024 р. – Житомир: Державний університет «Житомирська політехніка», 2024.
3. What Is HighLoad and When to Consider Developing a HighLoad System for Your Project? [Електронний ресурс] – Режим доступу до ресурсу: <https://geniusee.com/single-blog/introduction-to-high-load-what-is-it>.
4. About Google [Електронний ресурс] – Режим доступу до ресурсу: <https://about.google/>.
5. Who We Are [Електронний ресурс] – Режим доступу до ресурсу: <https://www.aboutamazon.com/about-us>.
6. Laudon, Kenneth C.; Traver, Carol Guercio (2014). E-commerce: Business, Technology, Society (10th ed.). Pearson plc. ISBN 9781292009094. Archived from the original on 5 May 2021. Retrieved 5 May 2021.
7. What is Netflix? [Електронний ресурс] – Режим доступу до ресурсу: <https://help.netflix.com/en/node/412>.
8. What is Spotify? [Електронний ресурс] – Режим доступу до ресурсу: <https://support.spotify.com/uk/article/what-is-spotify/>.
9. Database replication: Definition, types and setup [Електронний ресурс] – Режим доступу до ресурсу: <https://www.fivetran.com/learn/database-replication>.
10. What is Database Sharding? [Електронний ресурс] – Режим доступу до ресурсу: <https://aws.amazon.com/what-is/database-sharding/>.

11. How to Perform Batch Updates in SQL Server [Электронный ресурс] – Режим доступа до ресурсу: <https://www.geeksforgeeks.org/how-to-perform-batch-updates-in-sql-server/>.
12. Introduction To Database Indexing [Электронный ресурс] – Режим доступа до ресурсу: <https://medium.com/@rtawadrous/introduction-to-database-indexes-9b488e243cc1>.
13. MySQL: Understanding What It Is and How It's Used [Электронный ресурс] – Режим доступа до ресурсу: <https://www.oracle.com/mysql/what-is-mysql/>.
14. Krosing, Hannu; Roybal, Kirk (June 15, 2013). PostgreSQL Server Programming (second ed.). Packt Publishing. ISBN 978-1-84951-698-3.
15. McCreary, Dan; Kelly, Ann (2013). Making Sense of NoSQL: A guide for managers and the rest of us. Manning. ISBN 9781617291074.
16. ACID Transactions [Электронный ресурс] – Режим доступа до ресурсу: <https://www.databricks.com/glossary/acid-transactions>.
17. InnoDB – the best storage engine for MySQL? [Электронный ресурс] – Режим доступа до ресурсу: <https://www.ionos.com/digitalguide/hosting/technical-matters/what-is-innodb/>.
18. Chodorow, Kristina; Dirolf, Michael (September 23, 2010), MongoDB: The Definitive Guide (1st ed.), O'Reilly Media, pp. 216, ISBN 978-1-4493-8156-1
19. System Design Solutions: When to use Cassandra and when not to [Электронный ресурс] – Режим доступа до ресурсу: <https://medium.com/geekculture/system-design-solutions-when-to-use-cassandra-and-when-not-to-496ba51ef07a>.
20. Change Data Capture (CDC): What it is and How it Works [Электронный ресурс] – Режим доступа до ресурсу: <https://www.striim.com/blog/change-data-capture-cdc-what-it-is-and-how-it-works/>.
21. Asynchronous Data Replication Pattern [Электронный ресурс] – Режим доступа до ресурсу: <https://medium.com/@josueparra2892/asynchronous-data-replication-pattern-31973665ba5d>.
22. Horizontal Scaling [Электронный ресурс] – Режим доступа до ресурсу: <https://www.virtana.com/glossary/what-is-horizontal-scaling/>.

23. Vertical scaling [Электронный ресурс] – Режим доступа до ресурсу: <https://planetscale.com/learn/courses/database-scaling/scaling/vertical-scaling>.
24. DbSchema | SQL Server - How to Implement Partitioning? [Электронный ресурс] – Режим доступа до ресурсу: <https://dbschema.com/2023/07/01/sqlserver/partitioning/>.
25. Understand and resolve SQL Server blocking problems [Электронный ресурс] – Режим доступа до ресурсу: <https://learn.microsoft.com/en-us/troubleshoot/sql/database-engine/performance/understand-resolve-blocking>.
26. How Database B-Tree Indexing Works [Электронный ресурс] – Режим доступа до ресурсу: <https://builtin.com/data-science/b-tree-index> [Электронный ресурс] – Режим доступа до ресурсу: <https://builtin.com/data-science/b-tree-index>.
27. Hash Indexes [Электронный ресурс] – Режим доступа до ресурсу: <https://www.postgresql.org/docs/current/hash-index.html>.
28. Postgres Indexing: When Does BRIN Win? [Электронный ресурс] – Режим доступа до ресурсу: <https://www.crunchydata.com/blog/postgres-indexing-when-does-brin-win>.
29. Partial Indexes [Электронный ресурс] – Режим доступа до ресурсу: <https://www.postgresql.org/docs/current/indexes-partial.html>.
30. Multicolumn Indexes [Электронный ресурс] – Режим доступа до ресурсу: <https://www.atlassian.com/data/sql/multicolumn-indexes>.
31. What is Docker? [Электронный ресурс] – Режим доступа до ресурсу: <https://www.ibm.com/topics/docker>.
32. What is Docker Compose? How to Use it with an Example [Электронный ресурс] – Режим доступа до ресурсу: <https://www.freecodecamp.org/news/what-is-docker-compose-how-to-use-it/>.
33. ProxySQL [Электронный ресурс] – Режим доступа до ресурсу: <https://developers.clever-cloud.com/guides/proxysql/>.
34. YAML Tutorial: Everything You Need to Get Started in Minutes [Электронный ресурс] – Режим доступа до ресурсу: <https://www.cloudbees.com/blog/yaml-tutorial-everything-you-need-get-started>.

- 35.Publication [Электронный ресурс] – Режим доступа до ресурсу: <https://www.postgresql.org/docs/current/logical-replication-publication.html>.
- 36.What is a CSV file: A comprehensive guide [Электронный ресурс] – Режим доступа до ресурсу: <https://flatfile.com/blog/what-is-a-csv-file-guide-to-uses-and-benefits/>.
- 37.What Causes “Lock wait timeout exceeded” Error in MySQL? [Электронный ресурс] – Режим доступа до ресурсу: <https://www.baeldung.com/sql/mysql-lock-wait-timeout-error>.
- 38.What is SELECT FOR UPDATE in SQL (with examples)? [Электронный ресурс] – Режим доступа до ресурсу: <https://www.cockroachlabs.com/blog/select-for-update/>.
- 39.MySQL 8.0.1: Using SKIP LOCKED and NOWAIT to handle hot rows [Электронный ресурс] – Режим доступа до ресурсу: <https://dev.mysql.com/blog-archive/mysql-8-0-1-using-skip-locked-and-nowait-to-handle-hot-rows/>.
- 40.EXPLAIN ANALYZE [Электронный ресурс] – Режим доступа до ресурсу: <https://planetscale.com/learn/courses/mysql-for-developers/queries/explain-analyze>.
- 41.EXPLAIN - Index Scan [Электронный ресурс] – Режим доступа до ресурсу: <https://pganalyze.com/docs/explain/scan-nodes/index-scan>.
- 42.Queries in PostgreSQL: 3. Sequential scan [Электронный ресурс] – Режим доступа до ресурсу: <https://postgrespro.com/blog/pgsql/5969403>.

ДОДАТКИ

docker-compose.yml

```
services:
  #Main testing databases
  postgres:
    image: postgres:latest
    container_name: postgres-container
    ports:
      - "5432:5432"
    volumes:
      - pgdata:/var/lib/postgresql/data
    environment:
      POSTGRES_DB: test
      POSTGRES_USER: test
      POSTGRES_PASSWORD: test

  mysql_main:
    image: mysql:latest
    container_name: mysql_main
    environment:
      MYSQL_ROOT_PASSWORD: pass123
    ports:
      - "3310:3306"

  #Containers for sharding testing
  mysql_db_1:
    image: mysql:latest
    container_name: mysql_db_1
    networks:
      multi-tenant-network:
        ipv4_address: 172.19.0.3
    environment:
      MYSQL_ROOT_PASSWORD: pass123
    ports:
      - "3307:3306"

  mysql_db_2:
    image: mysql:latest
    container_name: mysql_db_2
    networks:
      multi-tenant-network:
        ipv4_address: 172.19.0.4
    environment:
      MYSQL_ROOT_PASSWORD: pass123
    ports:
      - "3308:3306"

  mysql_db_3:
    image: mysql:latest
```

```

container_name: mysql_db_3
networks:
  multi-tenant-network:
    ipv4_address: 172.19.0.5
environment:
  MYSQL_ROOT_PASSWORD: pass123
ports:
  - "3309:3306"

proxysql:
  image: proxysql/proxysql
  container_name: proxysql
  networks:
    - multi-tenant-network
  ports:
    - "6032:6032"
    - "6033:6033"
    - "6080:6080"
  volumes:
    - ./sharding/proxysql.cnf:/etc/proxysql.cnf
  environment:
    - DATADIR=/var/lib/proxysql

#Containers for replication testing
master:
  image: postgres:16.1-alpine3.19
  container_name: pg_master
  volumes:
    - pg_master-data:/var/lib/postgresql/data
  environment:
    POSTGRES_USER: postgres
    POSTGRES_PASSWORD: postgres
    POSTGRES_DB: postgres
  restart: unless-stopped

replica1:
  image: postgres:15.5-alpine3.19
  container_name: pg_replica1
  volumes:
    - pg_replica1-data:/var/lib/postgresql/data
  environment:
    POSTGRES_USER: postgres
    POSTGRES_PASSWORD: postgres
    POSTGRES_DB: postgres
  restart: unless-stopped

replica2:
  image: postgres:14.10-alpine3.19
  container_name: pg_replica2
  volumes:
    - pg_replica2-data:/var/lib/postgresql/data
  environment:

```

```
POSTGRES_USER: postgres
POSTGRES_PASSWORD: postgres
POSTGRES_DB: postgres
restart: unless-stopped
```

volumes:

```
pgdata:
  name: pgdata
pg_master-data:
  name: pg_master-data
pg_replica1-data:
  name: pg_replica1-data
pg_replica2-data:
  name: pg_replica2-data
```

networks:

```
multi-tenant-network:
  driver: bridge
  ipam:
    config:
      - subnet: 172.19.0.0/16
```

proxysql.cnf

```

datadir="/var/lib/proxysql"

admin_variables=
{
    admin_credentials="admin:admin;radmin:radmin"
    mysql_ifaces="0.0.0.0:6032"
    refresh_interval=2000
    hash_passwords=false
}

mysql_variables=
{
    threads=4
    max_connections=2048
    default_query_delay=0
    default_query_timeout=36000000
    have_compress=true
    poll_timeout=2000
    interfaces="0.0.0.0:6033;/tmp/proxysql.sock"
    default_schema="information_schema"
    stacksize=1048576
    server_version="5.1.30"
    connect_timeout_server=10000
    monitor_history=60000
    monitor_connect_interval=200000
    monitor_ping_interval=200000
    ping_interval_server_msec=10000
    ping_timeout_server=200
    commands_stats=true
    sessions_sort=true
    monitor_username="monitor"
    monitor_password="pass123"
}

mysql_servers =
(
    { address="172.19.0.3" , port=3306 , hostgroup=10, max_connections=100 },
    { address="172.19.0.4" , port=3306 , hostgroup=20, max_connections=100 },
    { address="172.19.0.5" , port=3306 , hostgroup=30, max_connections=100 }
)

mysql_users =
(
    { username = "user_shard" , password = "pass123" , default_hostgroup = 10 , active = 1 },
    { username = "user_shard" , password = "pass123" , default_hostgroup = 20 , active = 1 },
    { username = "user_shard" , password = "pass123" , default_hostgroup = 30 , active = 1 }
)

```

Процедура для пакетного оновлення

```

DROP PROCEDURE IF EXISTS BatchUpdate;

DELIMITER $$
CREATE PROCEDURE BatchUpdate(
    startBoundary INT,
    finishBoundary INT,
    stepSize INT)
BEGIN

    DECLARE currentLowerBound INT DEFAULT startBoundary;
    DECLARE currentUpperBound INT DEFAULT startBoundary + stepSize;
    DECLARE iterationCount INT DEFAULT 0;

    WHILE currentUpperBound <= finishBoundary DO

        SELECT CONCAT('Processing Range: ', currentLowerBound, ' to ', currentUpperBound) AS process_log;

        UPDATE user
        SET st = 'NJ'φ
        WHERE user_id >= currentLowerBound AND user_id <= currentUpperBound;

        SET currentLowerBound = currentLowerBound + stepSize;
        SET currentUpperBound = currentUpperBound + stepSize;
        SET iterationCount = iterationCount + 1;

    END WHILE;

    SELECT CONCAT('Final Update from ', currentUpperBound - stepSize, ' to ', finishBoundary) AS final_log,
           iterationCount AS total_iterations;

    UPDATE user
    SET is_active = TRUE
    WHERE user_id >= (currentUpperBound - stepSize) AND user_id <= finishBoundary;

END$$
DELIMITER ;

call BatchUpdate(3000000, 8000000, 50000);

```

Процедура перестворення таблиць для індексації

```

CREATE OR REPLACE PROCEDURE recreate_test_tables()
  LANGUAGE plpgsql
AS $$
BEGIN
  -- Видалення існуючих таблиць, якщо вони вже є
  DROP TABLE IF EXISTS user_data_no_opt;
  DROP TABLE IF EXISTS user_data_with_opt;

  -- Створення таблиці user_data_no_opt (без оптимізацій)
  CREATE TABLE user_data_no_opt
  (
    id                SERIAL PRIMARY KEY,
    user_id           BIGINT NOT NULL,
    full_name         VARCHAR(255),
    email             VARCHAR(255),
    created_at        TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    last_login        TIMESTAMP,
    status            VARCHAR(20),
    is_premium        BOOLEAN  DEFAULT FALSE,
    age              INT,
    country           VARCHAR(100),
    balance           DECIMAL(15, 2),
    subscription_level INT,
    preferences       JSONB,
    activity_score    DECIMAL(10, 2),
    last_transaction_date TIMESTAMP
  );

  -- Створення таблиці user_data_with_opt (для майбутніх оптимізацій)
  CREATE TABLE user_data_with_opt
  (
    id                SERIAL PRIMARY KEY,
    user_id           BIGINT NOT NULL,
    full_name         VARCHAR(255),
    email             VARCHAR(255),
    created_at        TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    last_login        TIMESTAMP,
    status            VARCHAR(20),
    is_premium        BOOLEAN  DEFAULT FALSE,
    age              INT,
    country           VARCHAR(100),
    balance           DECIMAL(15, 2),
    subscription_level INT,
    preferences       JSONB,
    activity_score    DECIMAL(10, 2),
    last_transaction_date TIMESTAMP
  );

```

```
);
```

```
-- Заповнення таблиць великим обсягом даних для симуляції навантаження
```

```
FOR i IN 1..1000000 LOOP
```

```
    INSERT INTO user_data_no_opt (
        user_id, full_name, email, created_at, last_login, status, is_premium, age, country,
        balance, subscription_level, preferences, activity_score, last_transaction_date
    ) VALUES (
        i,
        CONCAT('User_', i),
        CONCAT('user', i, '@example.com'),
        NOW() - (INTERVAL '1 day' * (RANDOM() * 365)),
        NOW() - (INTERVAL '1 day' * (RANDOM() * 90)),
        CASE WHEN RANDOM() > 0.5 THEN 'active' ELSE 'blocked' END,
        RANDOM() > 0.7,
        FLOOR(RANDOM() * 60) + 18,
        CASE WHEN RANDOM() > 0.5 THEN 'USA' ELSE 'Canada' END,
        RANDOM() * 10000,
        FLOOR(RANDOM() * 5),
        '{"preferences": {"theme": "dark", "notifications": "all"}}':jsonb,
        RANDOM() * 100,
        NOW() - (INTERVAL '1 day' * (RANDOM() * 180))
    );
```

```
    INSERT INTO user_data_with_opt (user_id, full_name, email, created_at, last_login, status,
is_premium, age, country,
                                balance, subscription_level, preferences, activity_score,
last_transaction_date)
```

```
    VALUES (i,
        CONCAT('User_', i),
        CONCAT('user', i, '@example.com'),
        NOW() - (INTERVAL '1 day' * (RANDOM() * 365)),
        NOW() - (INTERVAL '1 day' * (RANDOM() * 90)),
        CASE WHEN RANDOM() > 0.5 THEN 'active' ELSE 'blocked' END,
        RANDOM() > 0.7,
        FLOOR(RANDOM() * 60) + 18,
        CASE WHEN RANDOM() > 0.5 THEN 'USA' ELSE 'Canada' END,
        RANDOM() * 10000,
        FLOOR(RANDOM() * 5),
        '{
            "preferences": {
                "theme": "dark",
                "notifications": "all"
            }
        }':jsonb,
        RANDOM() * 100,
        NOW() - (INTERVAL '1 day' * (RANDOM() * 180)));
```

```
END LOOP;
```

```
END;
```

```
$$;
```