Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет Комп`ютерних наук
(повна назва)

Кафедра Комп'ютерного моделювання та інтелектуальних технологій
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти другий (магістерський)

Порівняльне дослідження стратегій кешування даних у серверній частині високонавантаженого веб-застосунку на платформі .NET
(тема)

Виконав:
здобувач 2 року навчання,
групи СПРм-24-1

Данііл Кієнко
(власне ім'я, прізвище)

Спеціальність 122 Комп`ютерні науки
(код і повна назва спеціальності)
Тип програми освітньо-наукова
(освітньо-професійна або освітньо-наукова)
Освітня програма Системне проектування
(повна назва освітньої програми)

Керівник проф. каф. КМІТ Вадим Саваневич
(посада, власне ім'я, прізвище)

Допускається до захисту

Зав. кафедри _____ Ігор Гребеннік
(підпис) (власне ім'я, прізвище)

2026 р.

Я, як студент ХНУРЕ, розумію і підтримую політику закладу із академічної доброчесності. Я не надавав і не одержував недозволену допомогу під час підготовки кваліфікаційної роботи. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело.

08.05.2026  *Кієнко Д. В.*

Кваліфікаційна робота не містить відомостей заборонених до відкритого опублікування.

Кваліфікаційна робота виконана у відповідності до стандартів, що діють в Україні.

Попередній захист проведено 8 травня 2026 р.

Керівник кваліфікаційної роботи  *Саваневич В.Є.*

Харківський національний університет радіоелектроніки

Факультет Комп`ютерних наук

Кафедра Комп'ютерного моделювання та інтелектуальних технологій

Рівень вищої освіти другий (магістерський)

Спеціальність 122 Комп`ютерні науки
(код і повна назва)

Тип програми освітньо-наукова
(освітньо-професійна або освітньо-наукова)

Освітня програма Системне проектування
(повна назва)

ЗАТВЕРДЖУЮ:

Зав. кафедри _____
(підпис)
«____»_____ 20 ___ р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Здобувачеві Кієнко Даніілу Вадимовичу
(прізвище, ім'я, по батькові)

1. Тема роботи Порівняльне дослідження стратегій кешування даних у серверній частині високонавантаженого веб-застосунку на платформі .NET
затверджена наказом університету від 6 квітня 2026 р. № 268Ст
2. Термін подання здобувачем роботи до екзаменаційної комісії 8 травня 2026 р.
3. Вихідні дані до роботи *Науково-технічна література з питань оптимізації продуктивності веб-застосунків та методів кешування даних у розподілених системах, матеріали щодо архітектури високонавантажених веб-застосунків, алгоритми та політики кешування даних, технології серверної розробки платформи ASP.NET Core (.NET 8), інструменти реалізації кешування IMemoryCache та Redis, система керування базами даних PostgreSQL, засоби розробки серверних веб-застосунків Visual Studio та .NET SDK, інструменти навантажувального тестування k6, метрики оцінювання продуктивності інформаційних систем (час відповіді, пропускна здатність),*
4. Перелік питань, що потрібно опрацювати в роботі _ *Вступ, обґрунтування актуальності теми дослідження, аналіз предметної області високонавантажених веб-застосунків, характеристика архітектури серверних веб-систем, дослідження теоретичних основ кешування даних у розподілених інформаційних системах, класифікація стратегій кешування та алгоритмів керування кешем, аналіз сучасних технологій кешування у веб-застосунках, дослідження засобів кешування платформи ASP.NET Core, проєктування архітектури експериментального веб-застосунку з каталогом товарів, розробка серверної частини веб-застосунку на платформі .NET, реалізація різних стратегій кешування (без кешування, локальне кешування IMemoryCache, розподілене кешування Redis, гібридна модель кешування), організація навантажувального тестування системи, дослідження продуктивності*

серверної частини веб-застосунку, порівняльний аналіз результатів використання стратегій кешування, формування висновків та рекомендацій щодо оптимізації продуктивності високонавантажених веб-застосунків.

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (слайдів) *1. Об'єкт, предмет та мета дослідження. 2. Актуальність дослідження. 3. Проблема повторних операцій. 4. Рішення: кешування даних. 5. Теоретична база дослідження. 6. Використані технології. 7. Архітектура системи. 8. Реалізовані стратегії кешування. 9. Організація експерименту. 10. Параметри тестування. 11. Метрики оцінювання продуктивності системи. 12. Аналіз продуктивності системи. 13. Аналіз середнього часу відповіді. 14. Аналіз медіанного часу відповіді. 15. Аналіз пропускної здатності системи. 16. Аналіз показників стабільності системи. 17. Аналіз максимального часу відповіді. 18. Підсумкове порівняння стратегій кешування. 19. Ключові висновки дослідження. 20. Апробація результатів дослідження.*

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Терміни виконання етапів роботи	Примітка
1.	*Отримання завдання на виконання кваліфікаційної роботи.*	*06.04.26*	*Виконано*
2.	*Аналіз завдання та предметної області*	*06.04.26 – 10.04.26*	*Виконано*
3.	*Опрацювання літератури та аналіз об'єкту дослідження*	*10.04.26 – 14.04.26*	*Виконано*
4.	*Формування постановки задачі*	*14.04.26 – 17.04.26*	*Виконано*
5.	*Проектування апаратного та програмного засобу*	*17.04.26 – 23.04.26*	*Виконано*
6.	*Розробка та тестування програмного засобу*	*23.04.26 – 29.04.26*	*Виконано*
7.	*Аналіз результатів дослідження*	*29.04.26 – 01.05.26*	*Виконано*
8.	*Оформлення пояснювальної записки та презентаційних матеріалів комп'ютерного захисту*	*01.05.26 – 08.05.26*	*Виконано*
9.	*Представлення роботи на рецензування*	*08.05.26*	*Виконано*

Дата видачі завдання 6 квітня 2026 р.

Здобувач _____
(підпис)

Керівник роботи _____ проф. каф. КМІТ Вадим САВАНЕВИЧ
(підпис) (посада, власне ім`я, прізвище)

РЕФЕРАТ

Пояснювальна записка до кваліфікаційної роботи містить: 137 стор., 25 рис., 11 табл., 1 дод., 24 джерел.

КЕШУВАННЯ ДАНИХ, ВИСОКОНАВАНТАЖЕНІ ВЕБ-ЗАСТОСУНКИ, ASP.NET CORE, REDIS, IMEMORYCACHE, ПРОДУКТИВНІСТЬ СИСТЕМ.

Об’єкт дослідження – серверна частина високонавантаженого веб-застосунку, що забезпечує обробку запитів до каталогу товарів.

Предмет дослідження – стратегії кешування даних у серверній частині веб-застосунку на платформі .NET та їх вплив на продуктивність системи.

Метою роботи є підвищення продуктивності серверної частини високонавантаженого веб-застосунку шляхом порівняльного дослідження стратегій кешування даних та визначення оптимального підходу до їх використання.

Методи дослідження включають системний аналіз предметної області високонавантажених веб-застосунків, методи проєктування серверних систем, моделювання клієнт-серверної архітектури (RESTful API), методи роботи з реляційними базами даних, а також методи навантажувального тестування з використанням інструменту k6.

Результати дослідження можуть бути застосовані у процесі проєктування та оптимізації високонавантажених веб-застосунків, серверних API та розподілених інформаційних систем, зокрема у сфері електронної комерції, з метою підвищення продуктивності шляхом впровадження ефективних стратегій кешування.

ABSTRACT

Explanatory note to the qualification work contains: 137 pages, 25 figures, 11 tables, 1 appendices, 24 references.

DATA CACHING, HIGH-LOAD WEB APPLICATIONS, ASP.NET CORE, REDIS, IMEMORYCACHE, SYSTEM PERFORMANCE.

Object of research – the server-side of a high-load web application that ensures processing of requests to a product catalog.

Subject of research – data caching strategies in the server-side of a web application built on the .NET platform and their impact on system performance.

The purpose of the work is to improve the performance of the server-side of a high-load web application through a comparative study of data caching strategies and determination of the optimal approach to their use.

Research methods include system analysis of the domain of high-load web applications, methods of server system design, modeling of client-server architecture (RESTful API), methods of working with relational databases, as well as load testing methods using the k6 tool.

The results of the research can be applied in the process of designing and optimizing high-load web applications, server APIs, and distributed information systems, in particular in the field of e-commerce, with the aim of improving performance through the implementation of effective caching strategies.

ЗМІСТ

ВСТУП

У сучасному цифровому середовищі веб-застосунки є основою функціонування значної кількості інформаційних систем, електронної комерції, соціальних платформ, фінансових сервісів та корпоративних інформаційних систем. Зі зростанням кількості користувачів та обсягів оброблюваних даних особливої актуальності набуває проблема забезпечення високої продуктивності та масштабованості серверної частини веб-застосунків. Такі системи повинні обробляти велику кількість запитів за короткий проміжок часу, забезпечуючи при цьому стабільну роботу та мінімальний час відповіді.

Одним із ключових підходів до підвищення продуктивності серверних систем є використання механізмів кешування даних. Застосування кешу дозволяє оптимізувати доступ до часто використовуваної інформації та зменшити навантаження на систему зберігання даних. У високонавантажених веб-застосунках, де значна частина запитів має повторюваний характер, використання кешування може суттєво скоротити час обробки запитів і підвищити загальну ефективність роботи системи.

Сучасні платформи розробки серверних застосунків, зокрема платформа .NET та фреймворк ASP.NET Core, надають широкий набір інструментів для реалізації різних стратегій кешування. До таких інструментів належать локальне кешування у пам’яті застосунку, розподілені системи кешування, а також багаторівневі архітектури кешу. Однією з найбільш поширених технологій розподіленого кешування є система Redis, яка широко застосовується у високонавантажених веб-застосунках.

Разом із тим ефективність використання кешування значною мірою залежить від вибору відповідної стратегії кешування, механізмів інвалідації даних, алгоритмів витіснення кешу та архітектури системи. У системах із великою кількістю користувачів і високою інтенсивністю запитів неправильний

вибір стратегії кешування може призвести до неефективного використання ресурсів або навіть до погіршення продуктивності системи.

У зв’язку з цим актуальним є проведення порівняльного дослідження різних стратегій кешування у серверній частині високонавантажених веб-застосунків з метою визначення їх ефективності у різних сценаріях використання.

Об’єктом дослідження є серверна частина високонавантаженого веб-застосунку.

Предметом дослідження є стратегії кешування даних у серверній частині веб-застосунку на платформі .NET.

Метою роботи є дослідження та порівняльний аналіз стратегій кешування даних у серверній частині високонавантаженого веб-застосунку з метою визначення їх впливу на продуктивність системи.

Для досягнення поставленої мети у роботі необхідно розв’язати такі завдання:

– проаналізувати особливості функціонування високонавантажених веб-застосунків;
– дослідити теоретичні основи кешування даних;
– розглянути основні стратегії та алгоритми кешування;
– проаналізувати сучасні технології кешування у веб-застосунках;
– спроєктувати архітектуру експериментального веб-застосунку;
– реалізувати різні стратегії кешування у серверній частині системи;
– провести навантажувальне тестування системи;
– виконати порівняльний аналіз ефективності використаних стратегій кешування.

Практичне значення отриманих результатів полягає у можливості використання результатів дослідження при розробці та оптимізації високонавантажених веб-застосунків на платформі .NET.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ

1.1 Особливості високонавантажених веб-застосунків

У сучасному цифровому середовищі веб-застосунки відіграють ключову роль у функціонуванні інформаційних систем різного призначення. Вони широко використовуються у сфері електронної комерції, фінансових сервісів, соціальних мереж, систем онлайн-бронювання, медіаплатформ та корпоративних інформаційних систем. Зі зростанням кількості користувачів та обсягів оброблюваних даних значно підвищуються вимоги до продуктивності, надійності та масштабованості таких систем.

Особливе місце серед сучасних інформаційних систем займають високонавантажені веб-застосунки. Під високонавантаженим веб-застосунком зазвичай розуміють програмну систему, здатну обробляти значну кількість одночасних запитів від користувачів, забезпечуючи при цьому стабільну роботу, прийнятний час відповіді та високу доступність сервісу. У таких системах серверна частина повинна ефективно обробляти великий потік запитів, виконувати бізнес-логіку застосунку та забезпечувати взаємодію з системами зберігання даних.

Характерною особливістю високонавантажених веб-застосунків є значна кількість одночасних користувачів, які можуть виконувати тисячі або навіть мільйони запитів протягом короткого проміжку часу. У таких умовах навіть незначні затримки в обробці запитів можуть призвести до помітного погіршення якості користувацького досвіду, а в окремих випадках – до часткової або повної недоступності сервісу.

Ще однією важливою характеристикою високонавантажених систем є високі вимоги до масштабованості. Масштабованість визначає здатність системи ефективно працювати при зростанні навантаження, зокрема при збільшенні кількості користувачів або обсягів оброблюваних даних. У сучасних веб-

застосунках масштабування зазвичай реалізується шляхом використання розподіленої серверної інфраструктури, балансування навантаження та використання декількох серверних екземплярів застосунку.

У більшості сучасних веб-систем значна частина операцій пов’язана з читанням даних. Наприклад, у системах електронної комерції користувачі регулярно переглядають каталоги товарів, характеристики продуктів, ціни, відгуки та іншу інформацію. У таких системах кількість операцій читання значно перевищує кількість операцій запису. Це створює додаткове навантаження на систему зберігання даних та може призводити до виникнення вузьких місць у серверній інфраструктурі.

Одним із ключових факторів, що впливають на ефективність роботи високонавантажених веб-застосунків, є архітектура серверної частини системи. Серверна частина відповідає за обробку HTTP-запитів, виконання бізнес-логіки, взаємодію з базою даних та формування відповіді для клієнтських застосунків. При великій кількості одночасних запитів саме серверна інфраструктура стає основним елементом, від якого залежить загальна продуктивність системи.

Однією з основних проблем при проєктуванні високонавантажених веб-застосунків є виникнення вузьких місць у системі. Найчастіше таким вузьким місцем виступає база даних, оскільки саме вона відповідає за зберігання та обробку великих обсягів інформації. При великій кількості одночасних запитів надмірне навантаження на базу даних може призвести до збільшення часу відповіді сервера та зниження загальної продуктивності системи.

З метою забезпечення стабільної роботи високонавантажених веб-застосунків використовуються різні архітектурні підходи та методи оптимізації продуктивності. До таких підходів належать горизонтальне масштабування серверної інфраструктури, використання балансування навантаження, оптимізація запитів до бази даних, а також застосування механізмів кешування даних.

Таким чином, високонавантажені веб-застосунки характеризуються великою кількістю одночасних користувачів, високою інтенсивністю запитів та підвищеними вимогами до продуктивності, масштабованості та доступності системи. Забезпечення ефективної роботи таких систем потребує використання сучасних архітектурних підходів та технологій оптимізації продуктивності, які дозволяють зменшити навантаження на серверну інфраструктуру та підвищити швидкість обробки запитів.

Для забезпечення ефективної роботи високонавантажених веб-застосунків важливу роль відіграє архітектура серверної частини системи, яка визначає спосіб обробки запитів, взаємодію між компонентами системи та механізми доступу до даних. У наступному підрозділі буде розглянуто архітектуру серверної частини сучасних веб-застосунків.

1.2 Архітектура серверної частини веб-застосунків

Архітектура серверної частини веб-застосунків відіграє ключову роль у забезпеченні ефективної роботи сучасних інформаційних систем. Саме серверна частина відповідає за приймання та обробку запитів користувачів, виконання бізнес-логіки, взаємодію з базами даних та формування відповідей для клієнтських застосунків. Від правильного проєктування серверної архітектури значною мірою залежать продуктивність, масштабованість і надійність веб-систем.

Більшість сучасних веб-застосунків побудовано на основі клієнт-серверної архітектури. У такій моделі система поділяється на дві основні частини: клієнтську та серверну. Клієнтська частина відповідає за взаємодію з користувачем і може бути реалізована у вигляді веб-інтерфейсу, мобільного застосунку або іншого клієнтського програмного забезпечення. Серверна частина виконує обробку запитів, реалізує бізнес-логіку застосунку та забезпечує доступ до даних.

У клієнт-серверній моделі клієнт формує запит до сервера через мережу, після чого сервер обробляє цей запит, виконує необхідні операції та повертає відповідь. Такий підхід дозволяє централізовано керувати даними та логікою системи, що спрощує підтримку, масштабування та розвиток програмного забезпечення.

У сучасних веб-застосунках серверна частина зазвичай реалізується у вигляді багаторівневої архітектури. Такий підхід передбачає поділ системи на декілька логічних рівнів, кожен із яких виконує власну функцію. Розділення функціональності на окремі рівні дозволяє підвищити модульність програмного забезпечення, спростити його підтримку та забезпечити можливість незалежного розвитку окремих компонентів системи.

Типова багаторівнева архітектура серверної частини веб-застосунку включає кілька основних рівнів. До них належать рівень обробки запитів, рівень бізнес-логіки та рівень доступу до даних.

Рівень обробки запитів відповідає за приймання HTTP-запитів від клієнтів, маршрутизацію запитів до відповідних компонентів системи та формування HTTP-відповідей. У сучасних веб-застосунках цей рівень зазвичай реалізується за допомогою веб-фреймворків або серверів веб-API. Наприклад, у застосунках, що розробляються на платформі .NET, для цього використовується фреймворк ASP.NET Core, який забезпечує обробку HTTP-запитів, маршрутизацію, перевірку вхідних даних та формування відповідей у форматах JSON або XML.

Рівень бізнес-логіки відповідає за реалізацію основної функціональності системи. На цьому рівні виконуються операції обробки даних, реалізуються бізнес-правила застосунку та координується взаємодія між різними компонентами системи. Саме на цьому рівні виконується більшість обчислювальних операцій, необхідних для обробки запитів користувачів.

Рівень доступу до даних забезпечує взаємодію застосунку із системами зберігання інформації, зокрема з базами даних. На цьому рівні виконуються операції читання, запису, оновлення та видалення даних. Для реалізації доступу

до даних можуть використовуватися системи керування базами даних, об’єктно-реляційні відображення (ORM) або спеціалізовані інтерфейси доступу до даних.

У більшості веб-застосунків значна частина запитів пов’язана з операціями читання даних. Наприклад, у системах електронної комерції користувачі постійно переглядають інформацію про товари, ціни, характеристики продукції або результати пошуку. У таких умовах серверна частина системи змушена виконувати велику кількість запитів до бази даних.

База даних часто стає одним із ключових вузьких місць у високонавантажених веб-застосунках. Кожен запит до бази даних потребує певного часу на виконання та використання обчислювальних ресурсів сервера. При великій кількості одночасних запитів це може призвести до збільшення часу відповіді системи та зниження загальної продуктивності веб-застосунку.

З метою підвищення ефективності роботи серверної частини веб-застосунків використовуються різні архітектурні підходи до оптимізації продуктивності. До таких підходів належать балансування навантаження між декількома серверами, використання розподілених систем зберігання даних, реплікація баз даних, а також застосування механізмів кешування.

Використання кешування у серверній архітектурі дозволяє зменшити кількість звернень до бази даних шляхом тимчасового зберігання результатів обробки запитів у швидкодоступному сховищі. Завдяки цьому система може швидше обробляти повторні запити до одних і тих самих даних та ефективніше використовувати обчислювальні ресурси серверної інфраструктури.

Таким чином, архітектура серверної частини веб-застосунків є важливим фактором, що визначає ефективність роботи інформаційної системи. Використання багаторівневої архітектури дозволяє розподілити функціональність системи між окремими компонентами, підвищити модульність програмного забезпечення та створити умови для ефективної оптимізації продуктивності.

У наступному підрозділі буде розглянуто основні проблеми продуктивності та масштабування веб-застосунків, які виникають у процесі обробки великої кількості запитів користувачів.

1.3 Проблеми продуктивності та масштабування веб-застосунків

Сучасні веб-застосунки функціонують в умовах постійного зростання кількості користувачів та обсягів оброблюваних даних. Це призводить до значного збільшення навантаження на серверну інфраструктуру та підвищує вимоги до продуктивності інформаційних систем. Забезпечення стабільної роботи веб-застосунків при високому навантаженні є одним із ключових завдань при проєктуванні та розробці сучасних програмних систем.

Продуктивність веб-застосунку визначається здатністю системи ефективно обробляти запити користувачів та забезпечувати мінімальний час відповіді сервера. У випадку, коли система не здатна швидко обробляти запити, користувачі можуть стикатися із затримками у роботі сервісу або навіть із повною недоступністю системи. Для веб-застосунків, що використовуються у сфері електронної комерції, фінансових сервісів або інформаційних платформ, такі ситуації можуть призвести до втрати користувачів та зниження ефективності роботи системи.

Однією з основних причин зниження продуктивності веб-застосунків є значна кількість одночасних запитів до серверної частини системи. У випадку великої кількості активних користувачів сервер повинен обробляти значну кількість запитів паралельно. Якщо архітектура системи не оптимізована для роботи в таких умовах, це може призвести до перевантаження серверних ресурсів та збільшення часу обробки запитів.

Важливим фактором, що впливає на продуктивність веб-застосунків, є ефективність роботи з базою даних. База даних є центральним компонентом більшості інформаційних систем, оскільки саме вона відповідає за зберігання та

обробку інформації. Однак операції доступу до бази даних зазвичай потребують значних обчислювальних ресурсів та часу на виконання. При великій кількості запитів до бази даних це може призвести до виникнення вузьких місць у системі.

У багатьох веб-застосунках значна частина запитів має повторюваний характер. Наприклад, користувачі можуть багаторазово переглядати одні й ті самі сторінки, списки товарів або результати пошуку. У таких випадках серверна частина системи виконує однакові операції доступу до бази даних для кожного нового запиту, що створює додаткове навантаження на систему зберігання даних.

Ще одним фактором, що впливає на продуктивність веб-застосунків, є складність виконання запитів до бази даних. У випадку використання складних запитів, які включають об’єднання декількох таблиць, сортування великих обсягів даних або агрегаційні операції, час виконання таких запитів може суттєво збільшуватися. Це призводить до збільшення часу відповіді всієї системи.

Крім операцій доступу до даних, важливу роль у забезпеченні продуктивності системи відіграє ефективне використання апаратних ресурсів сервера. До таких ресурсів належать процесор, оперативна пам’ять, дискова підсистема та мережеві ресурси. Неефективне використання цих ресурсів може призвести до зниження продуктивності системи навіть при відносно невеликому навантаженні.

Для забезпечення стабільної роботи веб-застосунків при зростанні навантаження використовуються різні підходи до масштабування системи. Масштабування дозволяє системі ефективно працювати при збільшенні кількості користувачів або обсягів оброблюваних даних.

Існує два основні підходи до масштабування інформаційних систем: вертикальне та горизонтальне масштабування. Вертикальне масштабування передбачає збільшення обчислювальних ресурсів одного сервера, наприклад шляхом використання більш потужного процесора, збільшення обсягу оперативної пам’яті або використання швидших накопичувачів. Такий підхід є

відносно простим у реалізації, однак має обмеження, пов’язані з фізичними можливостями обладнання.

Горизонтальне масштабування передбачає використання декількох серверів для обробки запитів. У цьому випадку навантаження розподіляється між декількома серверними екземплярами застосунку за допомогою механізмів балансування навантаження. Такий підхід дозволяє значно підвищити продуктивність системи, однак потребує більш складної архітектури та механізмів синхронізації даних між компонентами системи.

Незважаючи на використання механізмів масштабування, у багатьох випадках навантаження на систему зберігання даних залишається одним із головних факторів, що обмежує продуктивність веб-застосунків. Значна кількість повторних запитів до одних і тих самих даних може призводити до надмірного навантаження на базу даних і збільшення часу відповіді сервера.

У зв’язку з цим виникає необхідність використання додаткових методів оптимізації доступу до даних, які дозволяють зменшити навантаження на базу даних та прискорити обробку запитів користувачів. Одним із таких методів є використання механізмів кешування даних, що дозволяють тимчасово зберігати результати виконання запитів у швидкодоступному сховищі.

Таким чином, основними проблемами продуктивності сучасних веб-застосунків є високе навантаження на серверну інфраструктуру, велика кількість повторюваних запитів до даних, складність виконання запитів до бази даних та необхідність забезпечення масштабованості системи. Для ефективного вирішення цих проблем використовуються різні архітектурні підходи та технології оптимізації продуктивності.

У наступному підрозділі буде розглянуто роль кешування у підвищенні продуктивності веб-застосунків та зменшенні навантаження на систему зберігання даних.

1.4 Роль кешування у підвищенні продуктивності веб-застосунків

У попередньому підрозділі було розглянуто основні проблеми продуктивності веб-застосунків, які виникають у процесі обробки великої кількості запитів користувачів. Однією з ключових причин зниження продуктивності інформаційних систем є високе навантаження на систему зберігання даних. У таких умовах виникає необхідність використання ефективних підходів до оптимізації доступу до даних та зменшення навантаження на базу даних.

Одним із найбільш ефективних підходів до підвищення продуктивності веб-застосунків є використання механізмів кешування даних. Кешування передбачає тимчасове зберігання результатів виконання запитів або часто використовуваних даних у швидкодоступному сховищі, що дозволяє уникнути повторного звернення до основного джерела даних.

Основна ідея кешування полягає у повторному використанні результатів попередніх обчислень. Якщо певні дані запитуються користувачами багаторазово, їх можна зберегти у кеші та використовувати для обробки наступних запитів. У такому випадку система може обробляти запити значно швидше, оскільки доступ до кешу зазвичай виконується швидше, ніж доступ до бази даних або інших систем зберігання інформації.

У більшості веб-застосунків значна частина операцій пов’язана з читанням даних. Наприклад, у системах електронної комерції користувачі часто переглядають інформацію про популярні товари, результати пошуку або категорії продуктів. Такі запити можуть повторюватися тисячі разів протягом короткого проміжку часу. Якщо кожен запит обробляється шляхом безпосереднього звернення до бази даних, це може призвести до значного перевантаження системи.

Використання кешування дозволяє значно зменшити кількість звернень до бази даних. У випадку, коли необхідні дані вже знаходяться у кеші, сервер може

повернути відповідь користувачу без виконання додаткових операцій доступу до основного сховища даних. Це дозволяє суттєво скоротити час відповіді системи та підвищити її пропускну здатність.

Ще однією важливою перевагою використання кешування є ефективніше використання обчислювальних ресурсів серверної інфраструктури. Оскільки частина запитів обробляється без звернення до бази даних, зменшується навантаження на процесор, систему зберігання даних та мережеву інфраструктуру. Це дозволяє системі обробляти більшу кількість запитів при тих самих апаратних ресурсах.

У сучасних веб-застосунках кешування може використовуватися на різних рівнях архітектури системи. Наприклад, кеш може застосовуватися на стороні клієнта, у серверному застосунку або у спеціалізованих системах розподіленого кешування. Кожен із цих підходів має свої особливості та використовується залежно від архітектури системи та характеру навантаження.

Кешування на стороні клієнта дозволяє зменшити кількість запитів до сервера шляхом збереження певних ресурсів безпосередньо у браузері або клієнтському застосунку. Такий підхід часто використовується для кешування статичних ресурсів, таких як зображення, стилі або скрипти.

Серверне кешування передбачає збереження результатів виконання запитів безпосередньо у пам’яті серверного застосунку. У цьому випадку сервер може повторно використовувати результати попередніх обчислень для обробки нових запитів без звернення до бази даних.

У високонавантажених системах, що використовують горизонтальне масштабування, широко застосовується розподілене кешування. У такому випадку кешовані дані зберігаються у спеціалізованій системі кешування, яка доступна для декількох серверів застосунку одночасно. Це дозволяє забезпечити узгоджене використання кешованих даних у розподіленій серверній інфраструктурі.

Разом із перевагами використання кешування існують також певні складності, пов’язані з його реалізацією. Однією з основних проблем є забезпечення актуальності даних у кеші. Оскільки дані у базі даних можуть змінюватися, система повинна забезпечувати механізми оновлення або видалення застарілих даних із кешу.

Ще одним важливим аспектом використання кешування є вибір відповідної стратегії кешування та механізмів керування кешованими даними. Різні підходи до організації кешу можуть мати різну ефективність залежно від характеру навантаження, структури даних та архітектури системи.

Таким чином, кешування є одним із ключових механізмів оптимізації продуктивності веб-застосунків. Використання кешу дозволяє зменшити навантаження на базу даних, скоротити час відповіді системи та підвищити ефективність використання серверних ресурсів. У той же час ефективність використання кешування значною мірою залежить від правильного вибору архітектурних підходів та стратегій організації кешу.

1.5 Постановка задачі дослідження

Аналіз особливостей високонавантажених веб-застосунків показує, що однією з ключових проблем сучасних інформаційних систем є забезпечення високої продуктивності серверної частини при значній кількості одночасних запитів користувачів. У більшості веб-застосунків значна частина навантаження пов’язана з операціями доступу до даних, що може призводити до перевантаження бази даних та збільшення часу відповіді системи.

Одним із ефективних підходів до підвищення продуктивності веб-застосунків є використання механізмів кешування даних. Кешування дозволяє зменшити кількість звернень до бази даних шляхом збереження результатів виконання запитів у швидкодоступному сховищі. У сучасних веб-системах застосовуються різні стратегії кешування, зокрема локальне кешування у пам’яті

серверного застосунку, використання розподілених систем кешування, а також багаторівневі архітектури кешу.

Ефективність використання кешування значною мірою залежить від вибору відповідної стратегії організації кешу, архітектури системи та характеру навантаження. У зв’язку з цим виникає необхідність проведення порівняльного дослідження різних стратегій кешування з метою оцінювання їх впливу на продуктивність серверної частини веб-застосунків.

Метою даної роботи є підвищення продуктивності серверної частини високонавантаженого веб-застосунку шляхом проведення порівняльного дослідження різних стратегій кешування даних та визначення найбільш ефективного підходу до їх використання.

Для досягнення поставленої мети у роботі необхідно розв’язати такі задачі:

– проаналізувати особливості функціонування високонавантажених веб-застосунків та основні проблеми їх продуктивності;

– дослідити теоретичні основи та сучасні підходи до організації кешування даних у веб-застосунках;

– проаналізувати основні стратегії кешування та механізми керування кешованими даними;

– спроєктувати архітектуру експериментального веб-застосунку для дослідження ефективності різних стратегій кешування;

– реалізувати серверну частину веб-застосунку з використанням різних підходів до кешування даних;

– провести навантажувальне тестування системи для різних режимів роботи;

– виконати порівняльний аналіз отриманих результатів та визначити найбільш ефективну стратегію кешування.

Об’єктом дослідження є серверна частина високонавантаженого веб-застосунку, що забезпечує обробку запитів до каталогу товарів.

Предметом дослідження є стратегії кешування даних у серверній частині веб-застосунку на платформі .NET та їх вплив на продуктивність системи.

У процесі виконання роботи використовуються методи системного аналізу, методи проєктування програмних систем, методи експериментального дослідження продуктивності інформаційних систем, а також методи навантажувального тестування.

Практичне значення отриманих результатів полягає у можливості використання запропонованих підходів до організації кешування при проєктуванні та оптимізації високонавантажених веб-застосунків на платформі .NET.

Робота складається зі вступу, чотирьох розділів, висновків, переліку використаних джерел та додатків. У першому розділі проведено аналіз предметної області та сформульовано постановку задачі дослідження. У другому розділі розглянуто теоретичні основи та основні стратегії кешування даних. У третьому розділі описано реалізацію експериментального веб-застосунку та механізмів кешування. У четвертому розділі наведено результати експериментального дослідження ефективності різних стратегій кешування.

2 ТЕОРЕТИЧНІ ОСНОВИ ТА СТРАТЕГІЇ КЕШУВАННЯ ДАНИХ У ВЕБ-ЗАСТОСУНКАХ

2.1 Основні принципи кешування даних

Кешування є одним із ключових механізмів оптимізації продуктивності сучасних інформаційних систем. У високонавантажених веб-застосунках кешування використовується для зменшення часу доступу до даних, зниження навантаження на системи зберігання інформації та підвищення загальної пропускної здатності серверної інфраструктури.

У загальному випадку кешування передбачає використання спеціального проміжного сховища – кешу, у якому тимчасово зберігаються результати виконання запитів або часто використовувані дані. Таке сховище зазвичай характеризується значно меншою затримкою доступу порівняно з основним джерелом даних, яким у більшості веб-застосунків виступає база даних.

Основною метою використання кешу є мінімізація кількості дорогих операцій доступу до повільніших систем зберігання даних. Якщо результат виконання певного запиту вже був отриманий раніше, система може повторно використати цей результат без необхідності повторного виконання тих самих обчислень або звернення до бази даних. Це дозволяє значно скоротити час обробки запитів та підвищити ефективність використання серверних ресурсів.

Ефективність використання кешування значною мірою пояснюється властивостями доступу до даних у більшості інформаційних систем. У багатьох веб-застосунках характер доступу до даних є нерівномірним: невелика частина даних запитується значно частіше за інші. Таке явище часто описується правилом Парето або принципом 80/20, відповідно до якого приблизно 80 % запитів користувачів можуть бути спрямовані до відносно невеликої частини даних.

У таких умовах використання кешу дозволяє зберігати найбільш популярні дані у швидкодоступному сховищі та обслуговувати значну частину запитів без

звернення до бази даних. Це значно зменшує навантаження на систему зберігання даних і дозволяє системі обробляти більшу кількість запитів одночасно.

Ще одним важливим принципом ефективного використання кешування є принцип локальності доступу до даних. У теорії комп’ютерних систем виділяють два основні типи локальності: часову та просторову.

Часова локальність означає, що дані, до яких здійснювався доступ у недавньому минулому, з високою ймовірністю будуть запитуватися знову найближчим часом. Просторова локальність означає, що запит до певного елемента даних часто супроводжується зверненням до даних, розташованих поруч із ним.

Ці властивості доступу до даних роблять кешування особливо ефективним у системах, де значна частина запитів має повторюваний характер. Завдяки цьому система може обробляти більшість запитів без звернення до повільніших компонентів інфраструктури.

У контексті веб-застосунків кешування може використовуватися для зберігання різних типів даних. До них можуть належати результати виконання запитів до бази даних, об’єкти доменної моделі, результати складних обчислень або навіть готові відповіді HTTP-запитів. Вибір того, які саме дані доцільно кешувати, залежить від архітектури системи та характеру навантаження.

Разом із перевагами використання кешування існують і певні складності, пов’язані з його реалізацією. Однією з основних проблем є забезпечення актуальності даних у кеші. Оскільки дані у базі даних можуть змінюватися, необхідно використовувати механізми, які забезпечують оновлення або видалення застарілих даних із кешу.

Іншим важливим аспектом є обмежений обсяг пам’яті, що використовується для зберігання кешованих даних. Через це система повинна використовувати спеціальні алгоритми керування кешем, які визначають, які саме дані повинні залишатися у кеші, а які – видалятися для звільнення пам’яті.

Таким чином, кешування є важливим інструментом підвищення продуктивності високонавантажених веб-застосунків. Використання кешу дозволяє значно скоротити час обробки запитів, зменшити навантаження на базу даних та підвищити ефективність використання обчислювальних ресурсів серверної інфраструктури.

У наступному підрозділі буде розглянуто основні архітектурні підходи до організації кешування у веб-застосунках, зокрема локальне, розподілене та багаторівневе кешування.

2.2 Архітектурні підходи до кешування даних

У сучасних веб-застосунках кешування може реалізовуватися на різних рівнях архітектури системи. Вибір конкретного підходу залежить від архітектури серверного застосунку, характеру навантаження, вимог до масштабованості системи та обсягу оброблюваних даних.

З точки зору архітектури інформаційної системи кеш виступає проміжним рівнем між серверним застосунком та основним сховищем даних. Наявність такого проміжного шару дозволяє скоротити кількість звернень до бази даних та зменшити затримки під час обробки запитів користувачів. У випадку, коли необхідні дані вже присутні у кеші, сервер може отримати їх безпосередньо з кешу, що значно прискорює формування відповіді.

Архітектурні підходи до організації кешування визначають спосіб інтеграції кешу у структуру серверної частини веб-застосунку. У залежності від того, де саме розташовується кеш та яким чином він використовується серверним застосунком, виділяють декілька основних підходів до організації кешування.

Основні архітектурні підходи до організації кешування представлено на рисунку 2.1.

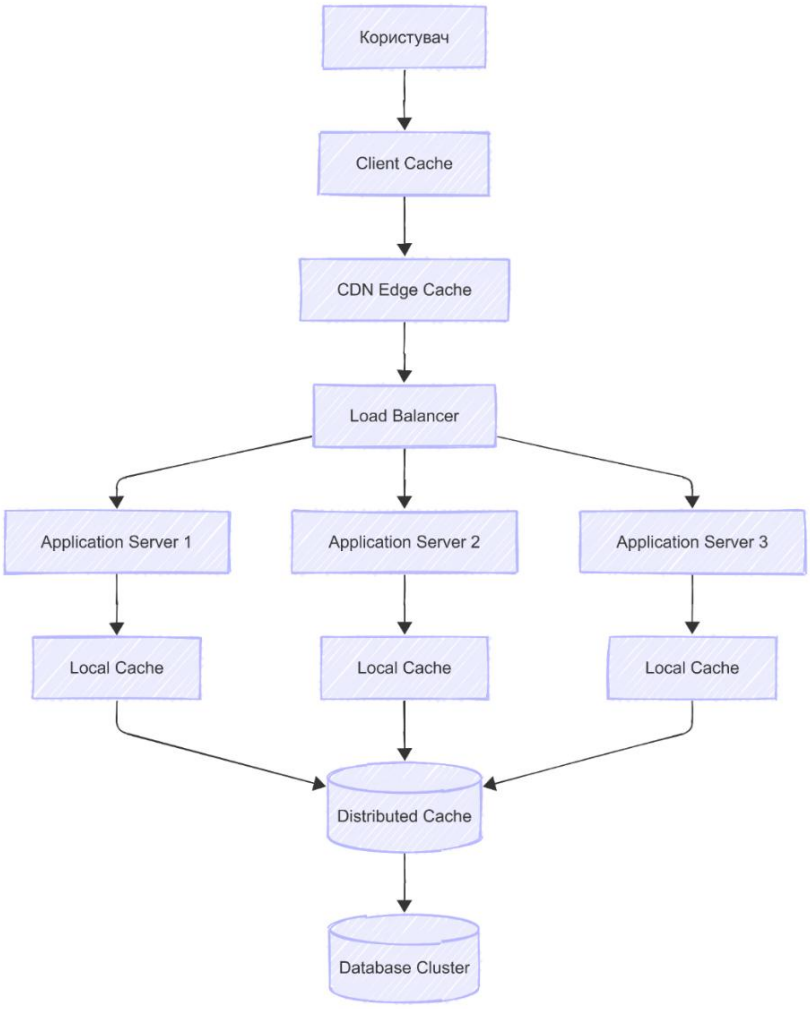

Рисунок 2.1 – Архітектурні підходи до організації кешування

Як показано на рисунку 2.1, кешування може реалізовуватися на різних рівнях системи, зокрема на рівні серверного застосунку або у вигляді окремого розподіленого сервісу кешування. У сучасних високонавантажених системах часто застосовуються також комбіновані підходи, що передбачають використання декількох рівнів кешу.

Найбільш поширеними архітектурними підходами до організації кешування у серверних веб-застосунках є локальне кешування у пам’яті застосунку, використання розподілених систем кешування та багаторівнева архітектура кешу.

У наступних підрозділах буде детально розглянуто особливості кожного з цих підходів, а саме локальне кешування у пам’яті застосунку, використання

розподілених систем кешування та організацію багаторівневого кешування у високонавантажених веб-системах.

2.2.1 Локальне кешування у пам’яті застосунку

Локальне кешування передбачає зберігання кешованих даних безпосередньо у оперативній пам’яті серверного застосунку. У цьому випадку кеш знаходиться всередині процесу застосунку і використовується для зберігання результатів виконання запитів або інших часто використовуваних даних.

Приклад роботи веб-застосунку, який використовує локальне кешування зображено на рисунку 2.2.

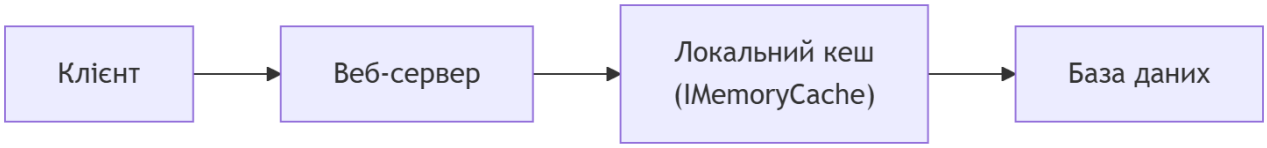

Рисунок 2.2 – Приклад роботи веб-застосунку, який використовує локальне кешування

Основною перевагою локального кешування є висока швидкість доступу до інформації. Оскільки дані зберігаються у пам’яті застосунку, звернення до кешу відбувається значно швидше, ніж доступ до зовнішніх систем зберігання даних або бази даних. Це дозволяє суттєво зменшити час обробки запитів та підвищити продуктивність системи.

Локальне кешування часто використовується у системах, де серверний застосунок працює на одному сервері або де не існує необхідності у синхронізації кешу між кількома екземплярами застосунку. У таких випадках кеш може використовуватися для зберігання результатів складних обчислень, конфігураційних параметрів або інших даних, які часто використовуються у процесі роботи системи.

У середовищі .NET локальне кешування може бути реалізоване за допомогою механізму IMemoryCache, який дозволяє зберігати об’єкти у пам’яті застосунку та керувати життєвим циклом кешованих даних. Такий підхід забезпечує просту інтеграцію кешування у серверну частину веб-застосунку та дозволяє ефективно використовувати оперативну пам’ять сервера.

Проте локальне кешування має певні обмеження. У системах, що використовують горизонтальне масштабування, серверний застосунок може працювати одночасно на декількох серверах. У такому випадку кожен сервер має власний локальний кеш, що може призводити до ситуацій, коли різні сервери використовують різні версії кешованих даних.

2.2.2 Розподілене кешування

Розподілене кешування використовується у системах, де серверний застосунок працює на декількох серверах або де необхідно забезпечити спільний доступ до кешованих даних для різних компонентів системи. У такому випадку кеш реалізується у вигляді окремої системи зберігання даних, яка доступна для всіх серверів застосунку.

У розподіленій архітектурі кеш розташовується поза межами серверного застосунку і може використовуватися одночасно кількома екземплярами системи. Це дозволяє забезпечити узгодженість кешованих даних та уникнути ситуацій, коли різні сервери використовують різні копії інформації.

Розподілені системи кешування зазвичай використовують оперативну пам’ять для зберігання даних, що забезпечує дуже швидкий доступ до інформації. Крім того, такі системи підтримують різні механізми управління кешем, включаючи алгоритми витіснення даних, налаштування часу життя кешованих об’єктів та інші механізми оптимізації продуктивності.

Приклад роботи веб-застосунку, який використовує розподілене кешування зображено на рисунку 2.3

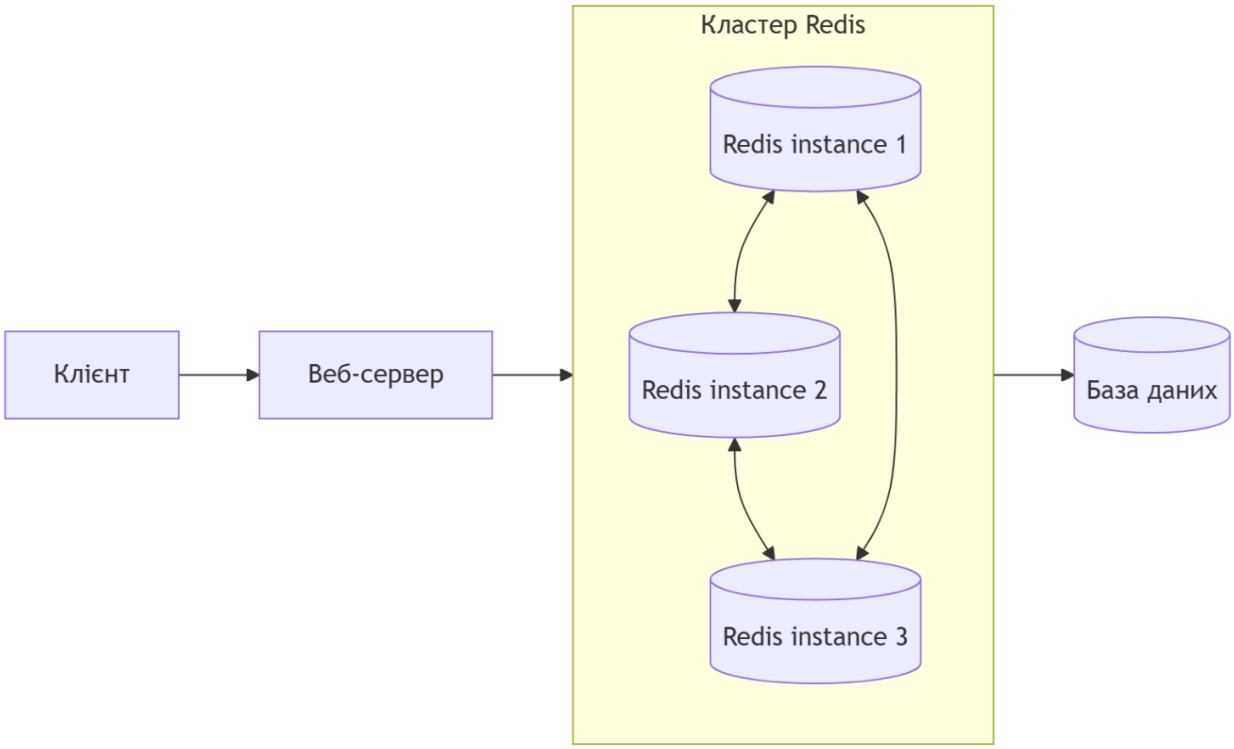

Рисунок 2.3 – Приклад роботи веб-застосунку, який використовує розподілене кешування

Однією з найбільш популярних систем розподіленого кешування є Redis. Redis є високопродуктивною системою зберігання даних у пам’яті, яка підтримує різні структури даних та забезпечує дуже швидкий доступ до інформації. Завдяки своїй продуктивності та масштабованості Redis широко використовується у високонавантажених веб-застосунках для реалізації механізмів кешування.

Ще одним прикладом системи розподіленого кешування є Memcached, яка також використовується для зберігання даних у пам’яті та оптимізації доступу до баз даних. Такі системи дозволяють ефективно масштабувати веб-застосунки та зменшувати навантаження на серверну інфраструктуру.

2.2.3 Багаторівневе кешування

У сучасних високонавантажених інформаційних системах кешування часто реалізується не у вигляді одного рівня кешу, а у вигляді багаторівневої

архітектури кешування. Такий підхід передбачає використання декількох рівнів кешу, які розташовуються на різних рівнях архітектури системи та забезпечують оптимізацію доступу до даних залежно від характеру навантаження.

Основна ідея багаторівневого кешування полягає у поєднанні високої швидкості доступу до локального кешу з можливостями масштабування, які забезпечують розподілені системи кешування. У типовій архітектурі веб-застосунку перший рівень кешу розташовується безпосередньо у серверному застосунку та використовує оперативну пам’ять процесу. Другий рівень кешу реалізується у вигляді розподіленої системи кешування, яка доступна для всіх екземплярів серверного застосунку.

У такій архітектурі процес отримання даних відбувається послідовно. Під час обробки запиту серверний застосунок спочатку перевіряє наявність необхідних даних у локальному кеші. Якщо дані відсутні у локальному кеші, виконується звернення до розподіленого кешу. Лише у випадку відсутності даних у обох рівнях кешу система виконує запит до бази даних. Після отримання результату дані можуть бути збережені у кеші для подальшого використання.

Перший рівень кешу зазвичай реалізується у межах серверного застосунку. Такий кеш використовує оперативну пам’ять процесу та забезпечує максимально швидкий доступ до даних. Основною перевагою локального кешу є дуже низький час доступу до інформації, оскільки звернення до даних відбувається без використання мережевих ресурсів. У серверних застосунках, розроблених на платформі .NET, локальне кешування може бути реалізоване за допомогою механізму IMemoryCache, який дозволяє зберігати об’єкти у пам’яті застосунку та керувати часом життя кешованих даних.

Разом із тим локальний кеш має певні обмеження. У системах, що використовують горизонтальне масштабування, кожен сервер має власний локальний кеш. У результаті різні екземпляри застосунку можуть містити різні версії одних і тих самих даних, що може призводити до проблем узгодженості інформації.

Другий рівень кешу зазвичай реалізується у вигляді розподіленої системи кешування. Такий кеш використовується спільно декількома екземплярами серверного застосунку та дозволяє забезпечити узгодженість даних між різними компонентами системи. Розподілений кеш зазвичай розташовується у окремому сервісі або кластері серверів та використовує оперативну пам’ять для зберігання даних, що забезпечує високу швидкість доступу до інформації навіть при великій кількості одночасних запитів.

Однією з найбільш поширених систем розподіленого кешування є Redis, який широко використовується у сучасних веб-застосунках. Redis забезпечує швидке зберігання даних у пам’яті, підтримує різні структури даних та дозволяє ефективно масштабувати систему. У серверних застосунках, розроблених на платформі .NET, взаємодія з Redis зазвичай реалізується за допомогою механізму IDistributedCache, що дозволяє використовувати розподілений кеш для зберігання спільних даних між різними екземплярами застосунку.

Приклад роботи веб-застосунку, який використовує багаторівневе кешування зображено на рисунку 2.4.

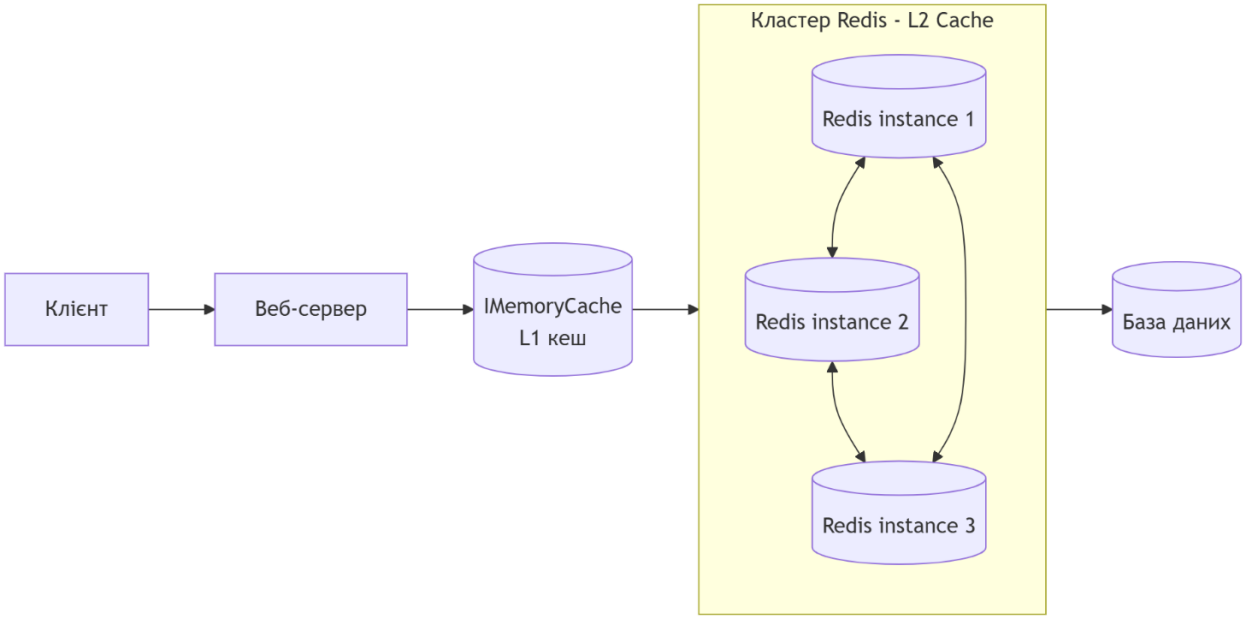

Рисунок 2.4 – Приклад роботи веб-застосунку, який використовує багаторівневе кешування

Таким чином, використання багаторівневого кешування дозволяє значно зменшити кількість звернень до бази даних, підвищити швидкість обробки запитів та знизити навантаження на основні компоненти системи. Завдяки поєднанню локального та розподіленого кешу багаторівнева архітектура кешування забезпечує високий рівень продуктивності та масштабованості, що є особливо важливим для сучасних високонавантажених веб-застосунків.

У наступному підрозділі буде розглянуто основні стратегії кешування даних, які визначають правила взаємодії між кешем, серверним застосунком та основним сховищем даних.

2.3 Основні стратегії кешування даних

Стратегії кешування визначають правила взаємодії між серверним застосунком, системою кешування та основним сховищем даних. Від вибору відповідної стратегії залежить ефективність використання кешу, швидкість обробки запитів та рівень навантаження на базу даних. Крім того, обрана стратегія безпосередньо впливає на масштабованість системи та її поведінку при зростанні навантаження.

Стратегії кешування визначають, у який момент дані повинні зчитуватися з кешу, коли вони повинні записуватися у кеш, а також яким чином відбувається синхронізація між кешем і базою даних. У сучасних веб-застосунках використовується декілька основних стратегій кешування, кожна з яких має власні особливості та сфери застосування. Вибір конкретної стратегії зазвичай залежить від характеру навантаження, співвідношення операцій читання і запису, а також вимог до консистентності даних.

До найбільш поширених стратегій кешування належать Cache-Aside, Read-Through, Write-Through, Write-Behind та Write-Around. У цьому підрозділі розглянуто основні принципи роботи та особливості цих стратегій.

2.3.1 Cache-Aside (Lazy Loading)

Стратегія Cache-Aside, яку також називають lazy loading, є однією з найбільш поширених стратегій кешування у сучасних веб-застосунках. У цій стратегії серверний застосунок самостійно керує процесом взаємодії з кешем і базою даних.

Основний принцип роботи Cache-Aside полягає у тому, що застосунок спочатку перевіряє наявність необхідних даних у кеші. Якщо дані присутні у кеші, вони повертаються без звернення до бази даних. У випадку відсутності даних у кеші застосунок виконує запит до бази даних, отримує необхідну інформацію та зберігає її у кеші для подальшого використання.

Таким чином, кеш заповнюється лише у тому випадку, коли дані дійсно запитуються системою. Це дозволяє ефективно використовувати пам’ять кешу та уникати зберігання зайвих даних.

Процес обробки запиту у стратегії Cache-Aside можна описати наступними етапами:

1. користувач надсилає запит до серверного застосунку;
2. серверний застосунок перевіряє наявність необхідних даних у кеші;
3. якщо дані знайдено у кеші, вони повертаються користувачу;
4. якщо дані відсутні у кеші, застосунок виконує запит до бази даних;
5. отримані дані зберігаються у кеші;
6. дані повертаються користувачу.

Такий механізм дозволяє значно зменшити кількість звернень до бази даних при повторних запитах до одних і тих самих даних. У системах, де більшість операцій пов’язана з читанням інформації, використання стратегії Cache-Aside може суттєво підвищити продуктивність системи

Процес обробки запиту з використанням стратегії Cache-Aside зображено на рисунку 2.5.

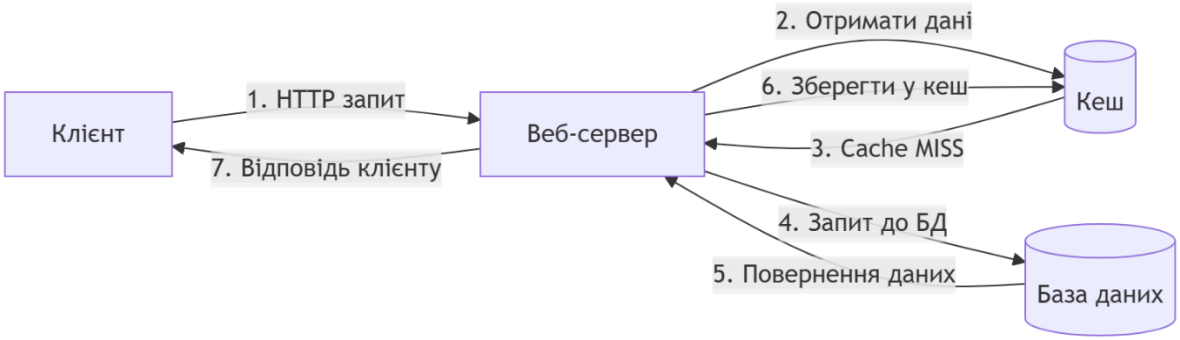

Рисунок 2.5 – Процес обробки запиту з використанням стратегії Cache-Aside

Такий механізм дозволяє значно зменшити кількість звернень до бази даних при повторних запитах до одних і тих самих даних. У системах, де більшість операцій пов’язана з читанням інформації, використання стратегії Cache-Aside може суттєво підвищити продуктивність системи.

Однією з головних переваг цієї стратегії є її простота та гнучкість. Оскільки застосунок самостійно керує процесом роботи з кешем, розробник може контролювати, які саме дані повинні кешуватися та як довго вони повинні зберігатися у кеші.

Крім того, стратегія Cache-Aside дозволяє легко інтегрувати кешування у існуючу архітектуру веб-застосунку. У багатьох сучасних фреймворках, зокрема у ASP.NET Core, передбачено спеціальні механізми для реалізації цієї стратегії за допомогою локального або розподіленого кешу.

Разом із тим стратегія Cache-Aside має певні обмеження. Однією з основних проблем є необхідність підтримки актуальності кешованих даних. Якщо дані у базі даних змінюються, відповідні записи у кеші можуть стати застарілими. Тому при використанні цієї стратегії необхідно реалізовувати механізми інвалідації кешу або обмеження часу життя кешованих даних.

Ще однією особливістю Cache-Aside є те, що перший запит до певних даних завжди потребує звернення до бази даних. Такий запит називають cache miss. Лише після отримання даних із бази вони зберігаються у кеші та можуть бути використані при наступних запитах.

Незважаючи на ці особливості, стратегія Cache-Aside широко використовується у сучасних веб-застосунках завдяки своїй простоті, гнучкості та ефективності. Вона добре підходить для систем, у яких переважають операції читання даних, а також для архітектур, де застосунок має прямий контроль над процесом роботи з кешем.

Саме стратегія Cache-Aside часто використовується у серверних застосунках, розроблених на платформі .NET, у поєднанні з такими системами кешування, як IMemoryCache або Redis.

У наступному підрозділі буде розглянуто стратегію Read-Through, яка передбачає інший підхід до взаємодії між серверним застосунком, кешем та основним сховищем даних.

2.3.2 Read-Through

Стратегія Read-Through є одним із підходів до організації кешування, при якому система кешування бере на себе відповідальність за отримання даних із основного сховища. На відміну від стратегії Cache-Aside, де серверний застосунок самостійно взаємодіє з кешем і базою даних, у випадку Read-Through логіка отримання даних із бази даних реалізується на рівні системи кешування.

Основний принцип роботи цієї стратегії полягає у тому, що серверний застосунок взаємодіє лише з кешем, не звертаючись безпосередньо до бази даних. Якщо необхідні дані присутні у кеші, вони одразу повертаються застосунку. У випадку відсутності даних у кеші система кешування автоматично виконує запит до основного сховища даних, отримує необхідну інформацію та зберігає її у кеші.

Після збереження даних у кеші результат повертається серверному застосунку. При наступних запитах до цих самих даних вони вже можуть бути отримані безпосередньо з кешу без повторного звернення до бази даних.

Процес обробки запиту у стратегії Read-Through можна описати наступними етапами:

1. користувач надсилає запит до серверного застосунку;
2. серверний застосунок звертається до системи кешування для отримання необхідних даних;
3. якщо дані присутні у кеші, вони повертаються застосунку;
4. якщо дані відсутні у кеші, система кешування виконує запит до бази даних;
5. отримані дані зберігаються у кеші;
6. дані повертаються серверному застосунку.

Процес обробки запиту у стратегії Read-Through зображено на рисунку 2.6.

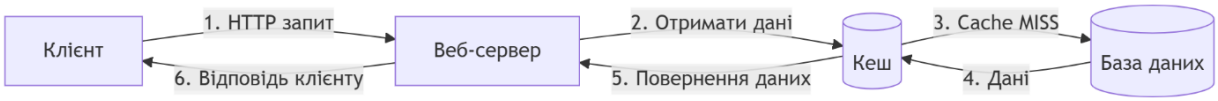

Рисунок 2.6 – Процес обробки запиту у стратегії Read-Through

Однією з головних переваг стратегії Read-Through є те, що вона дозволяє відокремити логіку роботи з кешем від серверного застосунку. Уся взаємодія з базою даних у випадку cache miss відбувається на рівні системи кешування. Це дозволяє спростити архітектуру застосунку та централізувати механізми керування кешованими даними.

Крім того, такий підхід дозволяє легше підтримувати та масштабувати систему, оскільки логіка кешування реалізується у спеціалізованому компоненті. У разі необхідності зміни параметрів кешування або механізмів отримання даних це може бути виконано без внесення змін у код серверного застосунку.

Разом із тим використання стратегії Read-Through має певні обмеження. Одним із них є залежність від функціональних можливостей системи кешування. Для реалізації цього підходу система кешування повинна підтримувати

механізми автоматичного завантаження даних із бази даних у випадку їх відсутності у кеші.

Ще однією особливістю є те, що серверний застосунок має обмежений контроль над процесом кешування. На відміну від стратегії Cache-Aside, де застосунок самостійно визначає, коли необхідно отримувати дані з бази даних та коли їх кешувати, у випадку Read-Through ці рішення приймаються на рівні системи кешування.

Незважаючи на ці особливості, стратегія Read-Through може бути ефективною у системах, де необхідно централізувати механізми роботи з кешем та зменшити складність серверного застосунку. Такий підхід часто використовується у великих інформаційних системах, де кешування реалізується як окремий сервіс або компонент інфраструктури.

У наступному підрозділі буде розглянуто стратегію Write-Through, яка визначає механізм обробки операцій запису даних у системах, що використовують кешування.

2.3.3 Write-Through

Стратегія Write-Through використовується для організації операцій запису даних у системах, що застосовують кешування. Основна ідея цього підходу полягає у тому, що всі операції запису виконуються одночасно як у кеші, так і в основному сховищі даних.

У стратегії Write-Through серверний застосунок під час запису нових або оновлених даних спочатку передає інформацію до системи кешування. Після цього система кешування негайно записує ці дані у базу даних. Таким чином, кеш і база даних завжди містять однакову інформацію.

Процес запису даних у стратегії Write-Through можна описати наступними етапами:

1. користувач надсилає запит на зміну або створення даних;
2. серверний застосунок передає дані у систему кешування;
3. дані записуються у кеш;
4. система кешування виконує запис тих самих даних у базу даних;
5. після успішного запису операція вважається завершеною.

Процес обробки запиту у стратегії Write-Through зображено на рисунку 2.7.

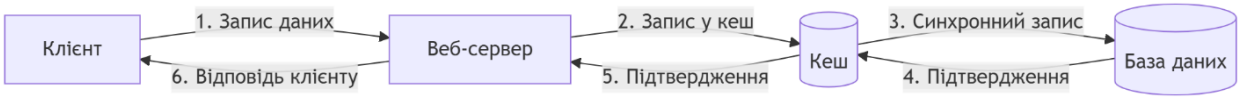

Рисунок 2.7 – Процес обробки запиту у стратегії Write-Through

Однією з головних переваг стратегії Write-Through є забезпечення високого рівня узгодженості даних між кешем і базою даних. Оскільки кожна операція запису виконується одночасно у двох системах, кеш завжди містить актуальні дані. Це дозволяє уникнути ситуацій, коли кеш містить застарілу інформацію.

Крім того, використання цієї стратегії дозволяє підвищити швидкість обробки запитів на читання. Оскільки після виконання операції запису дані вже знаходяться у кеші, наступні операції читання можуть виконуватися безпосередньо з кешу без звернення до бази даних.

Разом із тим стратегія Write-Through має певні недоліки. Основним із них є збільшення часу виконання операцій запису. Кожна операція запису потребує виконання двох операцій – запису у кеш та запису у базу даних. Це може призводити до збільшення затримок при обробці запитів на зміну даних.

Ще однією особливістю є те, що кеш може містити дані, які рідко використовуються. Оскільки кожна операція запису автоматично призводить до збереження даних у кеші, у ньому можуть накопичуватися записи, які не будуть використовуватися у подальших запитах.

Незважаючи на ці обмеження, стратегія Write-Through широко використовується у системах, де важливо забезпечити високу узгодженість

даних. Такий підхід дозволяє мінімізувати ризик використання застарілої інформації та забезпечити стабільну роботу системи.

У сучасних веб-застосунках стратегія Write-Through часто використовується у поєднанні зі стратегіями кешування для операцій читання, такими як Cache-Aside або Read-Through. Це дозволяє забезпечити баланс між продуктивністю системи та узгодженістю даних.

У наступному підрозділі буде розглянуто стратегію Write-Behind, яка передбачає інший підхід до синхронізації кешу з базою даних та дозволяє оптимізувати обробку операцій запису.

2.3.4 Write-Behind (Write-Back)

Стратегія Write-Behind, яку також називають write-back caching, використовується для оптимізації операцій запису у системах, що застосовують кешування. На відміну від стратегії Write-Through, де кожна операція запису одразу виконується як у кеші, так і у базі даних, у підході Write-Behind дані спочатку записуються лише у кеш, а оновлення основного сховища виконується із певною затримкою.

Основна ідея цієї стратегії полягає у тому, що кеш виступає як проміжний буфер для операцій запису. Коли серверний застосунок виконує операцію запису або оновлення даних, нове значення спочатку зберігається у кеші. Після цього система кешування накопичує зміни та періодично синхронізує їх із базою даних.

Такий підхід дозволяє зменшити кількість операцій запису до бази даних, оскільки декілька змін можуть бути об’єднані у одну операцію оновлення. Це особливо ефективно у системах, де виконується велика кількість операцій запису або оновлення даних.

Процес роботи стратегії Write-Behind можна описати наступними етапами:

1. користувач або система надсилає запит на зміну або створення даних;
2. серверний застосунок передає нові дані у систему кешування;

3. дані зберігаються у кеші;

4. кеш накопичує зміни та формує чергу операцій запису;

5. через певний інтервал часу система кешування синхронізує накопичені зміни з базою даних.

Процес обробки запиту у стратегії Write-Through зображено на рисунку 2.8.

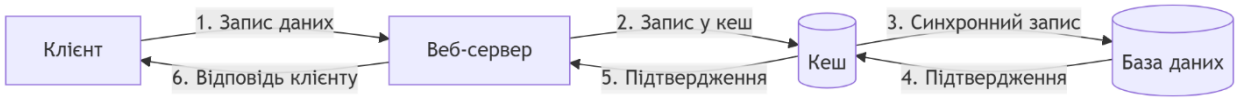

Рисунок 2.8 – Процес обробки запиту у стратегії Write-Through

Однією з головних переваг стратегії Write-Behind є підвищення продуктивності системи. Оскільки операції запису виконуються спочатку у кеші, серверний застосунок може швидко завершити обробку запиту без необхідності очікування завершення запису у базу даних. Це дозволяє зменшити затримки при обробці запитів та підвищити пропускну здатність системи.

Ще однією перевагою є можливість об’єднання декількох операцій запису у одну операцію оновлення бази даних. Такий підхід дозволяє значно зменшити навантаження на систему зберігання даних та оптимізувати використання ресурсів сервера.

Разом із тим стратегія Write-Behind має певні ризики. Основним із них є можливість втрати даних у випадку збою системи. Оскільки зміни певний час зберігаються лише у кеші, у випадку відмови системи до моменту синхронізації з базою даних частина даних може бути втрачена.

Ще однією особливістю є необхідність реалізації механізмів керування чергою записів та забезпечення надійної синхронізації кешу з основним сховищем даних. У складних розподілених системах це може потребувати додаткових механізмів контролю та обробки помилок.

Незважаючи на ці обмеження, стратегія Write-Behind широко використовується у системах, де необхідно забезпечити високу продуктивність

при великій кількості операцій запису. Такий підхід дозволяє ефективно оптимізувати роботу з базою даних та зменшити навантаження на серверну інфраструктуру.

Порівняно зі стратегією Write-Through, у підході Write-Behind операції запису виконуються асинхронно, що дозволяє підвищити швидкість обробки запитів. Однак це також створює додаткові вимоги до механізмів забезпечення надійності та узгодженості даних.

У наступному підрозділі буде розглянуто стратегію Write-Around, яка передбачає альтернативний підхід до обробки операцій запису у системах із використанням кешування.

2.3.5 Write-Around

Стратегія Write-Around є одним із підходів до організації операцій запису у системах із використанням кешування. Основна ідея цієї стратегії полягає у тому, що операції запису виконуються безпосередньо у базу даних, минаючи систему кешування. Кеш у такому випадку використовується лише для обробки операцій читання даних.

У стратегії Write-Around серверний застосунок під час виконання операції запису передає нові або змінені дані безпосередньо до бази даних. Кеш при цьому не оновлюється. Лише у випадку, коли дані запитуються у процесі виконання операцій читання, вони можуть бути збережені у кеші відповідно до обраної стратегії кешування, наприклад Cache-Aside або Read-Through.

Процес роботи стратегії Write-Around можна описати наступними етапами:

1. користувач або система надсилає запит на створення або зміну даних;
2. серверний застосунок передає дані безпосередньо у базу даних;
3. кеш не оновлюється під час виконання операції запису;
4. при наступному запиті на читання даних застосунок перевіряє їх наявність у кеші;

5. якщо дані відсутні у кеші, вони отримуються з бази даних та можуть бути збережені у кеші для подальшого використання.

Процес обробки запиту у стратегії Write-Around зображено на рисунку 2.9.

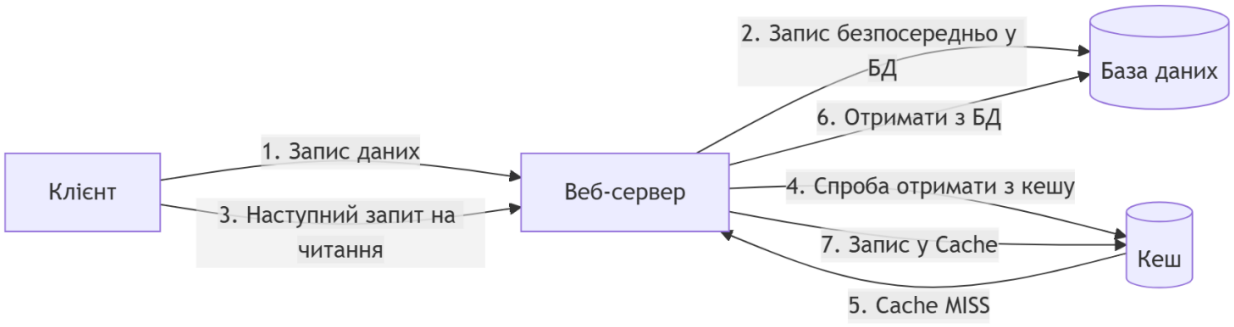

Рисунок 2.9 – Процес обробки запиту у стратегії Write-Around

Основною перевагою стратегії Write-Around є зменшення навантаження на кеш. Оскільки кеш не використовується для збереження кожної операції запису, у ньому зберігаються лише ті дані, які дійсно використовуються у процесі читання. Це дозволяє більш ефективно використовувати обмежений обсяг пам’яті кешу.

Такий підхід особливо ефективний у системах, де виконується велика кількість операцій запису, але лише частина даних активно використовується для читання. У цьому випадку кеш не перевантажується інформацією, яка може ніколи не знадобитися у подальших запитах.

Разом із тим стратегія Write-Around має певні недоліки. Одним із них є можливість виникнення cache miss при першому запиті до нещодавно записаних даних. Оскільки під час операції запису кеш не оновлюється, при першому зверненні до цих даних система буде змушена виконати запит до бази даних.

Ще однією особливістю є необхідність забезпечення актуальності кешованих даних. Якщо певні дані були раніше збережені у кеші, а потім змінені у базі даних через операцію Write-Around, відповідний запис у кеші може стати

застарілим. У таких випадках необхідно використовувати механізми інвалідації кешу або обмеження часу життя кешованих даних.

Незважаючи на ці обмеження, стратегія Write-Around може бути ефективною у системах із високою частотою операцій запису та нерівномірним використанням даних. Такий підхід дозволяє оптимізувати використання пам’яті кешу та зменшити кількість зайвих записів у кеш.

Порівняно зі стратегіями Write-Through та Write-Behind, у підході Write-Around кеш не бере участі у процесі запису даних, що спрощує механізм синхронізації між кешем та базою даних. Проте це також може призводити до збільшення кількості звернень до бази даних при наступних операціях читання.

Таким чином, стратегія Write-Around є альтернативним підходом до організації операцій запису у системах кешування. Її використання дозволяє оптимізувати роботу кешу у сценаріях, де значна частина записаних даних не використовується повторно.

У наступному підрозділі буде розглянуто методи інвалідації кешу та механізми підтримки актуальності кешованих даних, які відіграють важливу роль у забезпеченні коректної роботи систем із використанням кешування.

2.4 Методи інвалідації кешу та підтримка актуальності даних

Однією з основних проблем використання кешування у інформаційних системах є підтримка актуальності кешованих даних. Оскільки кеш зберігає копії даних, що знаходяться у базі даних або іншому основному сховищі інформації, з часом ці копії можуть втрачати актуальність у разі змін у вихідному джерелі даних.

Якщо кешовані дані не оновлюються своєчасно, система може використовувати застарілу інформацію. Це може призвести до некоректної роботи застосунку, порушення логіки бізнес-процесів або відображення

користувачам неактуальних даних. Тому у системах, що використовують кешування, важливу роль відіграють механізми інвалідації кешу.

Інвалідація кешу – це процес видалення або оновлення кешованих даних у випадку, коли вони більше не відповідають актуальному стану інформації у базі даних. Механізми інвалідації кешу дозволяють забезпечити баланс між високою продуктивністю системи та коректністю даних.

У сучасних інформаційних системах використовуються різні підходи до інвалідації кешу. Найбільш поширеними серед них є використання часу життя кешованих даних, інвалідація на основі подій та ручне очищення кешу.

2.4.1 TTL (Time-To-Live)

Одним із найпоширеніших методів інвалідації кешу є використання параметра Time-To-Live (TTL). TTL визначає максимальний час життя кешованих даних у системі. Після закінчення цього часу кешований запис автоматично видаляється або вважається неактуальним.

При використанні цього підходу кожному елементу кешу призначається певний часовий інтервал, протягом якого він може використовуватися. Після завершення цього інтервалу система повинна повторно отримати дані з основного сховища та зберегти їх у кеші.

Основною перевагою TTL є простота реалізації. Цей підхід не потребує складних механізмів синхронізації між кешем і базою даних. Завдяки цьому TTL широко використовується у багатьох системах кешування, зокрема у Redis та інших розподілених системах кешу.

Разом із тим використання TTL має певні обмеження. Основним із них є складність визначення оптимального часу життя кешованих даних. Якщо значення TTL занадто велике, система може використовувати застарілу інформацію. Якщо ж TTL занадто мале, кеш буде оновлюватися надто часто, що зменшить ефективність кешування.

2.4.2 Event-Based Invalidation

Іншим поширеним підходом до інвалідації кешу є інвалідація на основі подій. У цьому випадку кеш оновлюється або очищується у момент зміни даних у базі даних.

При використанні цього підходу система генерує спеціальні події у випадку створення, оновлення або видалення даних. Ці події можуть використовуватися для інвалідації відповідних записів у кеші. Таким чином забезпечується висока узгодженість між кешем і основним сховищем даних.

Основною перевагою цього підходу є висока точність оновлення кешу. Дані у кеші оновлюються саме у той момент, коли змінюється їх вихідне джерело. Це дозволяє мінімізувати ризик використання застарілої інформації.

Разом із тим реалізація інвалідації на основі подій є більш складною. Для цього необхідно реалізувати механізми генерації подій, їх передачі між компонентами системи та обробки відповідних змін у кеші.

2.4.3 Ручна інвалідація кешу

Ще одним підходом до підтримки актуальності кешованих даних є ручна інвалідація кешу. У цьому випадку очищення або оновлення кешу виконується безпосередньо серверним застосунком у процесі виконання операцій зміни даних.

Наприклад, після виконання операції оновлення певного запису у базі даних застосунок може видалити відповідний запис із кешу. При наступному зверненні до цих даних система отримає їх із бази даних та знову збереже у кеші.

Основною перевагою ручної інвалідації є повний контроль над процесом оновлення кешу. Розробник може точно визначити, які записи повинні бути видалені або оновлені у кеші після виконання певних операцій.

Проте цей підхід потребує ретельного контролю з боку розробників. Якщо певна операція оновлення даних не буде супроводжуватися відповідною інвалідацією кешу, система може продовжувати використовувати застарілі дані.

2.5 Алгоритми витіснення даних із кешу

У системах кешування обсяг пам’яті, що використовується для зберігання кешованих даних, є обмеженим. У процесі роботи системи кеш поступово заповнюється новими записами, що призводить до необхідності видалення частини існуючих даних для звільнення місця для нових елементів. Для вирішення цієї задачі використовуються спеціальні алгоритми витіснення кешу.

Алгоритми витіснення визначають правила, за якими система обирає записи для видалення у випадку досягнення максимально допустимого обсягу кешу. Основною метою таких алгоритмів є збереження у кеші найбільш корисних або часто використовуваних даних, що дозволяє підвищити ефективність роботи системи.

У сучасних інформаційних системах використовується велика кількість алгоритмів витіснення кешу. Кожен із них має власні особливості та може бути ефективним у різних сценаріях використання. Найбільш поширеними алгоритмами витіснення є FIFO, LRU, LFU, MRU, Random Replacement, Second Chance та Segmented LRU.

2.5.1 FIFO (First-In, First-Out)

Алгоритм FIFO є одним із найпростіших підходів до управління кешем. У цьому алгоритмі записи видаляються з кешу у тому порядку, в якому вони були додані. Іншими словами, першим видаляється той елемент, який знаходиться у кеші найдовше.

Принцип роботи FIFO базується на використанні структури даних типу черги. Коли новий елемент додається до кешу, він розміщується у кінці черги. Якщо кеш досягає максимального розміру, система видаляє елемент, що знаходиться на початку черги.

Принцип роботи алгоритму FIFO зображено на рисунку 2.10.

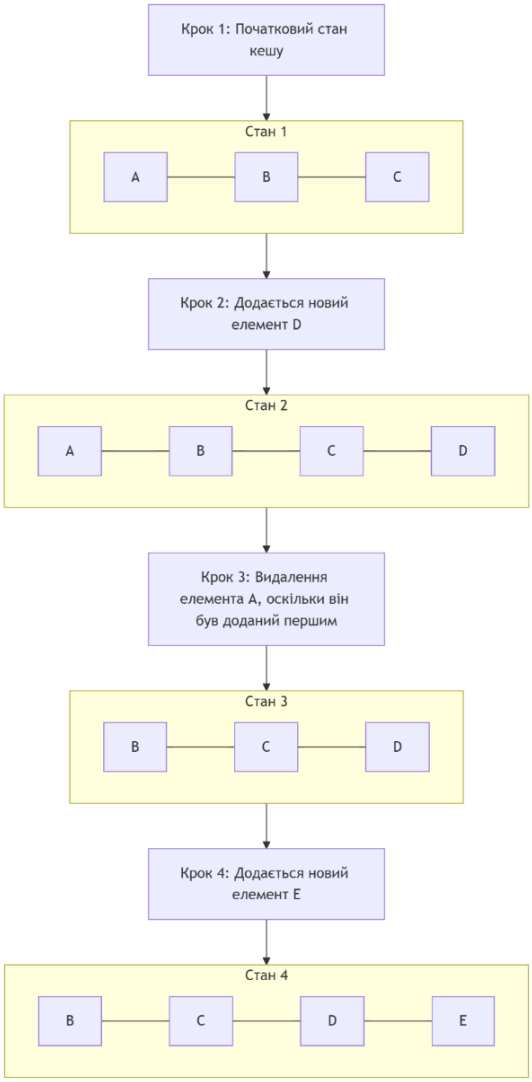

Рисунок 2.10 – Принцип роботи алгоритму FIFO

Основною перевагою алгоритму FIFO є простота реалізації та низькі витрати обчислювальних ресурсів. Проте цей алгоритм не враховує частоту або актуальність використання даних. У результаті можуть видалятися записи, які активно використовуються системою, що знижує ефективність кешування.

2.5.2 LRU (Least Recently Used)

Алгоритм LRU є одним із найпоширеніших методів управління кешем у сучасних інформаційних системах. Його основна ідея полягає у припущенні, що дані, які не використовувалися протягом тривалого часу, мають меншу ймовірність бути запитаними у майбутньому.

У цьому алгоритмі кожен доступ до кешованого елемента оновлює інформацію про час його останнього використання. Якщо кеш заповнений, система видаляє елемент, який використовувався найдавніше.

Принцип роботи алгоритму LRU зображено на рисунку 2.11.

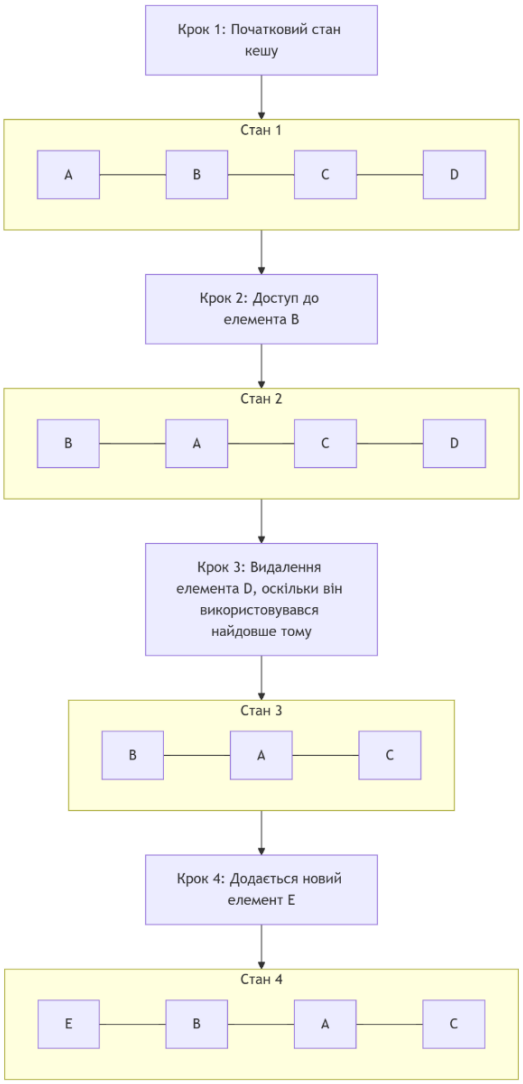

Рисунок 2.11 – Принцип роботи алгоритму LRU

Алгоритм LRU добре підходить для систем, де доступ до даних має локальність у часі. Це означає, що нещодавно використані дані з великою ймовірністю будуть використані повторно. Саме тому LRU широко застосовується у різних системах кешування.

У багатьох сучасних системах кешування, зокрема у Redis та Memcached, використовуються модифікації алгоритму LRU для ефективного управління кешованими даними. Основним недоліком цього алгоритму є необхідність відстеження часу доступу до кожного елемента, що може збільшувати складність реалізації.

2.5.3 LFU (Least Frequently Used)

Алгоритм LFU базується на аналізі частоти використання кешованих даних. У цьому підході кожен елемент кешу має спеціальний лічильник, який показує кількість звернень до цього елемента.

Коли кеш досягає максимального розміру, система видаляє той елемент, який має найменшу кількість звернень. Таким чином у кеші залишаються записи, що використовуються найчастіше. У разі однакових значень лічильників можуть застосовуватися додаткові критерії, наприклад час останнього доступу.

Перевагою алгоритму LFU є ефективне збереження популярних даних, які активно використовуються системою. Це дозволяє підвищити ефективність кешування у системах із стабільним набором часто використовуваних даних.

Разом із тим реалізація LFU потребує підтримки лічильників для кожного елемента кешу, що може збільшувати витрати пам’яті та обчислювальних ресурсів. Крім того, алгоритм може неефективно працювати у випадках, коли популярність даних змінюється з часом, оскільки елементи з високою історичною частотою можуть витісняти більш актуальні дані.

Принцип роботи алгоритму LFU зображено на рисунку 2.12.

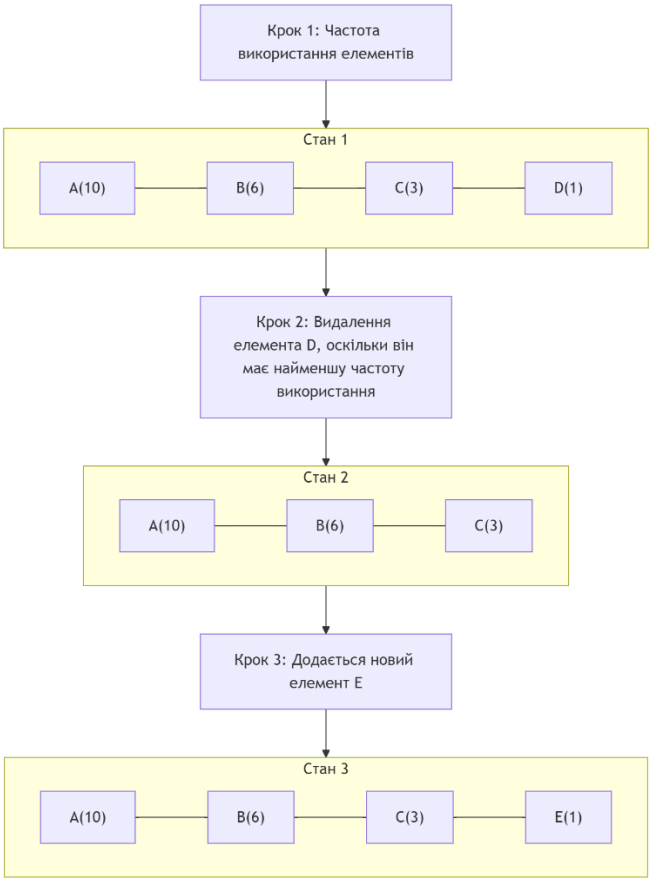

Рисунок 2.12 – Принцип роботи алгоритму LFU

2.5.4 MRU (Most Recently Used)

Алгоритм MRU працює за принципом, протилежним LRU. У цьому підході при необхідності звільнення місця у кеші система видаляє елемент, який використовувався останнім.

Такий підхід може бути ефективним у системах, де доступ до даних має послідовний характер. Наприклад, у випадках потокової обробки даних або послідовного перегляду великих наборів інформації нещодавно використані записи можуть мати низьку ймовірність повторного використання.

Проте у більшості систем кешування MRU використовується значно рідше, ніж LRU або LFU, оскільки його ефективність сильно залежить від характеру навантаження.

Принцип роботи алгоритму MRU зображено на рисунку 2.13.

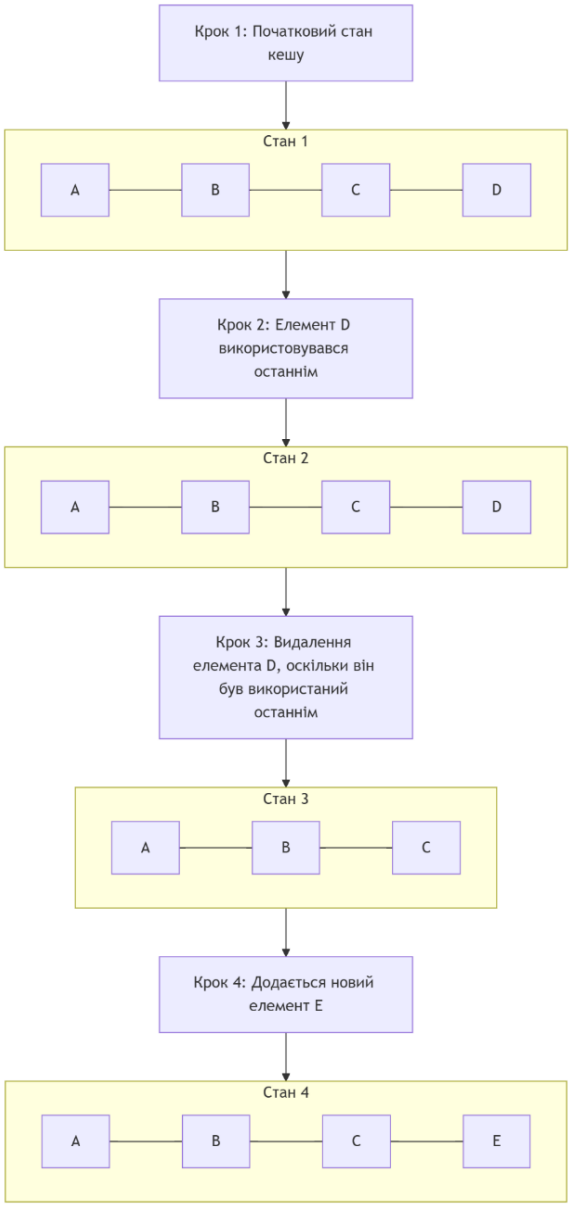

Рисунок 2.13 – Принцип роботи алгоритму MRU

2.5.5 Random Replacement

Алгоритм Random Replacement є одним із найпростіших підходів до управління кешем. У цьому випадку система випадковим чином обирає елемент для видалення з кешу.

Основною перевагою цього алгоритму є його простота та мінімальні витрати на обчислення. Система не потребує зберігання додаткової інформації про частоту або час використання елементів.

Разом із тим випадковий вибір елементів може призводити до видалення часто використовуваних записів, що знижує ефективність кешування. Тому цей алгоритм використовується переважно у простих або спеціалізованих системах.

Принцип роботи алгоритму Random Replacement зображено на рисунку 2.14.

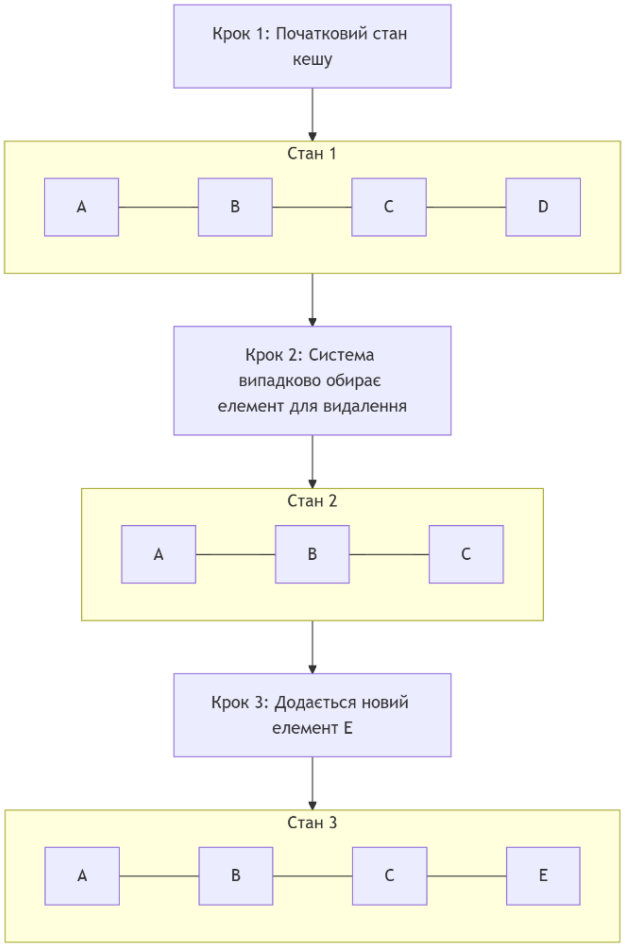

Рисунок 2.14 – Принцип роботи алгоритму Random Replacement

2.5.6 Second Chance

Алгоритм Second Chance є покращеною версією алгоритму FIFO. У цьому підході кожен елемент кешу має додатковий біт використання, який показує, чи був цей елемент використаний після додавання у кеш.

Коли система обирає елемент для видалення, вона перевіряє значення цього біта. Якщо елемент використовувався нещодавно, йому надається «другий

шанс», і він переміщується у кінець черги. Якщо ж елемент не використовувався, він видаляється з кешу.

Принцип роботи алгоритму Second Chance зображено на рисунку 2.15.

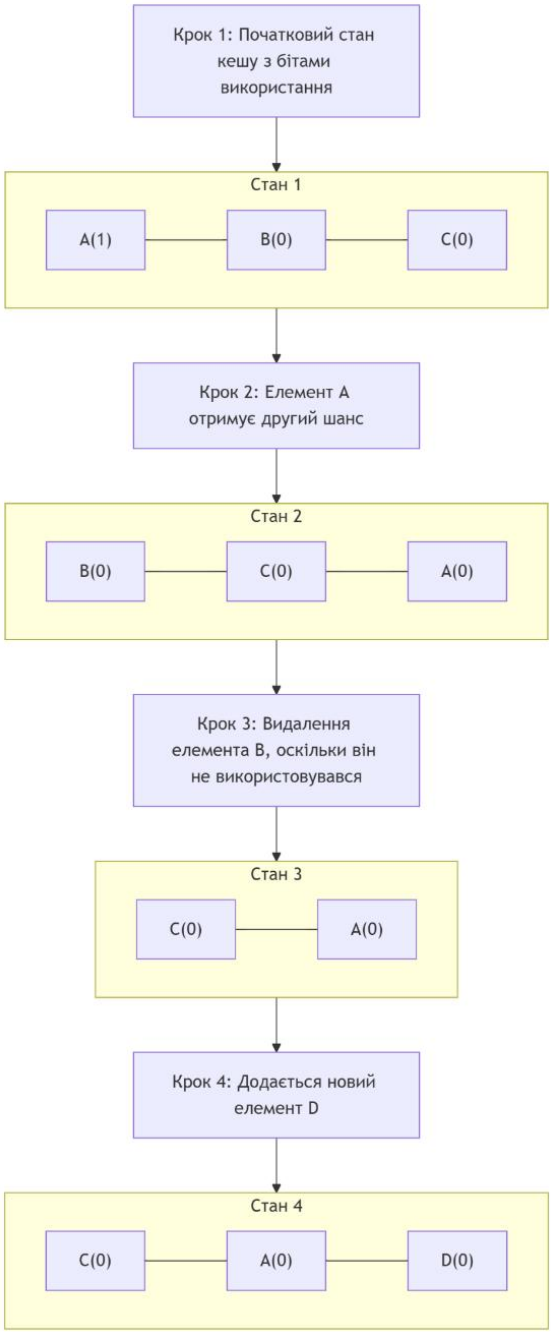

Рисунок 2.15 – Принцип роботи алгоритму Second Chance

Такий підхід дозволяє поєднати простоту алгоритму FIFO з частковим урахуванням частоти використання даних.

2.5.7 Segmented LRU

Алгоритм Segmented LRU є розширеною модифікацією класичного алгоритму LRU. У цьому підході кеш поділяється на декілька сегментів, які відповідають різним рівням використання даних.

Нові елементи спочатку потрапляють у сегмент для нещодавно доданих записів. Якщо елемент використовується повторно, він переміщується у сегмент для часто використовуваних даних.

Принцип роботи алгоритму Segmented LRU зображено на рисунку 2.16.

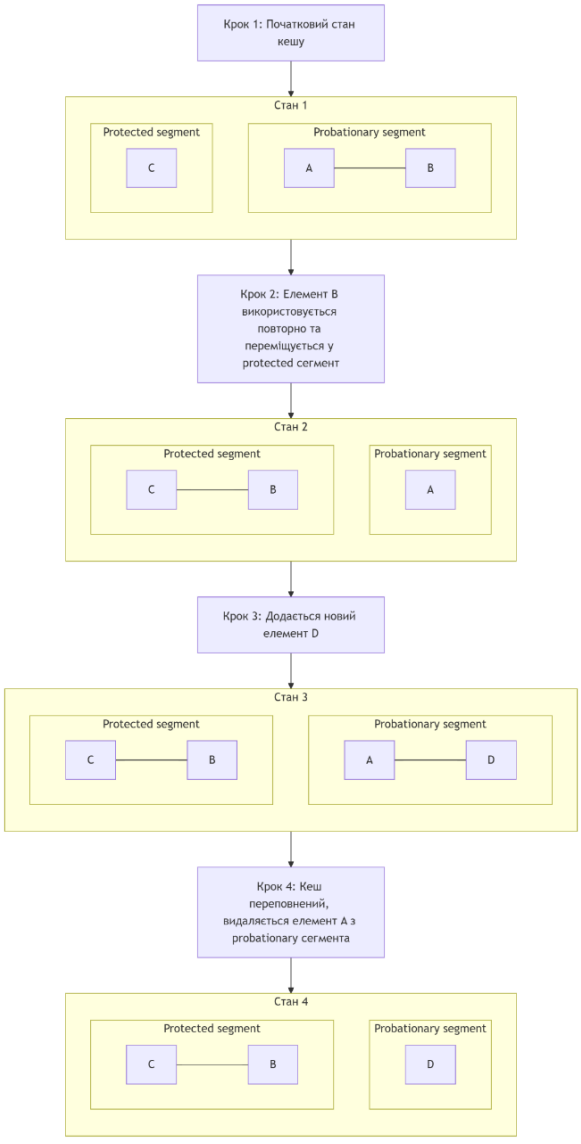

Рисунок 2.16 – Принцип роботи алгоритму Segmented LRU

Такий підхід дозволяє більш ефективно розподіляти пам’ять кешу між новими та популярними даними. У результаті система може краще адаптуватися до різних типів навантаження та підвищити ефективність кешування.

Таким чином, алгоритми витіснення відіграють важливу роль у роботі систем кешування. Вибір відповідного алгоритму залежить від характеру навантаження системи, структури даних та вимог до продуктивності. У сучасних системах кешування найчастіше використовуються різні модифікації алгоритмів LRU та LFU, які забезпечують баланс між ефективністю кешування та складністю реалізації.

2.6 Проблеми використання кешування у високонавантажених системах

Незважаючи на значні переваги використання кешування для підвищення продуктивності веб-застосунків, його застосування у високонавантажених системах пов’язане з рядом технічних викликів. У таких системах кеш виступає не лише механізмом оптимізації доступу до даних, але й важливим архітектурним компонентом, від правильного використання якого залежить стабільність та масштабованість всієї системи.

Однією з основних проблем у системах кешування є забезпечення узгодженості даних між кешем та основним сховищем інформації. Оскільки кеш зберігає копії даних, що знаходяться у базі даних, між цими двома джерелами інформації можуть виникати розбіжності. У простих системах, де використовується один серверний застосунок і один кеш, підтримка узгодженості даних є відносно простою задачею. Проте у сучасних високонавантажених веб-застосунках зазвичай використовується декілька екземплярів серверного застосунку, а також розподілені системи кешування, що значно ускладнює процес синхронізації даних.

Для опису різних аспектів синхронізації кешованих даних у розподілених системах використовуються два основні поняття: узгодженість кешу (cache coherency) та консистентність кешованих даних (cache consistency).

2.6.1 Узгодженість кешу (Cache Coherency)

Узгодженість кешу визначає, наскільки синхронізованими є копії даних, що зберігаються у різних кешах системи. У розподілених веб-застосунках може використовуватися декілька кешів, наприклад локальний кеш кожного серверного застосунку та спільний розподілений кеш.

У таких системах можуть виникати ситуації, коли різні кеші містять різні версії одного й того самого запису. Це відбувається у випадках, коли дані змінюються у базі даних, але відповідні записи у кеші не були своєчасно оновлені або видалені. У результаті різні користувачі можуть отримувати різні версії даних.

Для забезпечення узгодженості кешу використовуються різні механізми синхронізації. До таких механізмів належать інвалідація кешу після зміни даних, оновлення кешованих записів після виконання операцій запису, а також використання централізованих систем розподіленого кешування. Використання цих підходів дозволяє зменшити ймовірність виникнення конфліктів між різними копіями даних у системі.

2.6.2 Консистентність кешованих даних (Cache Consistency)

Консистентність кешованих даних визначає, наскільки дані у кеші відповідають актуальному стану інформації у базі даних. Іншими словами, консистентність характеризує ступінь відповідності кешованих даних їхній актуальній версії у системі зберігання.

У системах кешування можуть використовуватися різні моделі консистентності. У деяких випадках прагнуть забезпечити сильну консистентність, при якій кешовані дані завжди відповідають поточному стану бази даних. Проте реалізація такої моделі потребує додаткових операцій синхронізації та може негативно впливати на продуктивність системи.

У багатьох високонавантажених веб-застосунках використовується модель так званої eventual consistency. У цьому випадку допускається, що протягом певного часу кеш може містити застарілі дані, проте система гарантує, що з часом всі копії даних стануть узгодженими. Такий підхід дозволяє значно підвищити продуктивність системи та зменшити навантаження на базу даних.

2.6.3 Проблема Cache Stampede

Однією з поширених проблем у системах кешування є так звана проблема cache stampede, яка собливо гостро проявляється у системах із високим рівнем паралельності запитів. Вона виникає у ситуації, коли термін дії певного запису у кеші завершується, і велика кількість клієнтів одночасно намагається отримати ці дані.

У такому випадку замість одного запиту до бази даних система може генерувати десятки або навіть сотні однакових запитів. Це призводить до різкого збільшення навантаження на базу даних і може спричинити суттєве зниження продуктивності системи.

Для запобігання виникненню цієї проблеми використовуються різні механізми. Серед них можна виділити блокування доступу до кешу під час оновлення даних та попереднє оновлення кешованих записів. Такі підходи дозволяють зменшити ймовірність одночасного звернення великої кількості запитів до бази даних

Приклад проблеми Cache Stampede зображено на рисунку 2.17.

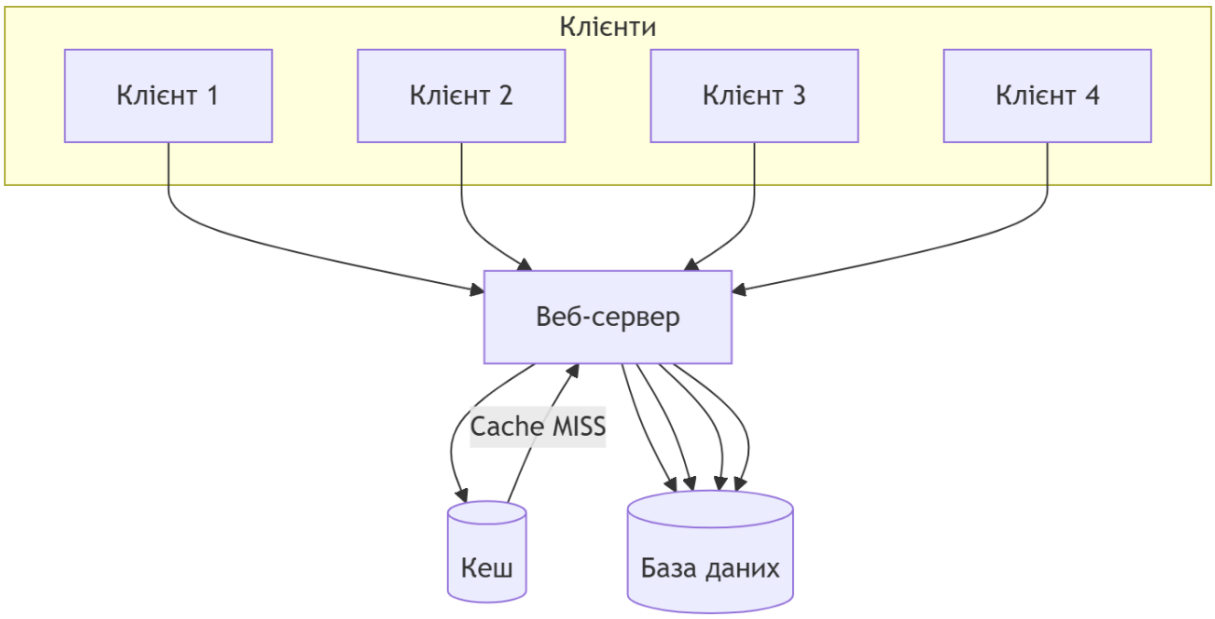

Рисунок 2.17 – Приклад проблеми Cache Stampede

2.6.4 Проблема «гарячих ключів» (Hot Keys)

Ще однією проблемою, що може виникати у системах кешування, є наявність так званих «гарячих ключів» (hot keys). Гарячими ключами називають записи у кеші, до яких звертається дуже велика кількість запитів протягом короткого проміжку часу.

У високонавантажених системах певні дані можуть бути значно популярнішими за інші. Наприклад, у системах електронної комерції такими даними можуть бути популярні товари або результати пошуку. Якщо велика кількість клієнтів одночасно звертається до одного й того самого запису у кеші, це може призвести до перевантаження кеш-сервера або збільшення затримок у системі.

Для зменшення впливу цієї проблеми використовуються різні підходи, зокрема реплікація кешованих даних, балансування навантаження між декількома вузлами кешу або використання багаторівневих архітектур кешування.

Приклад проблеми Hot Keys зображено на рисунку 2.18.

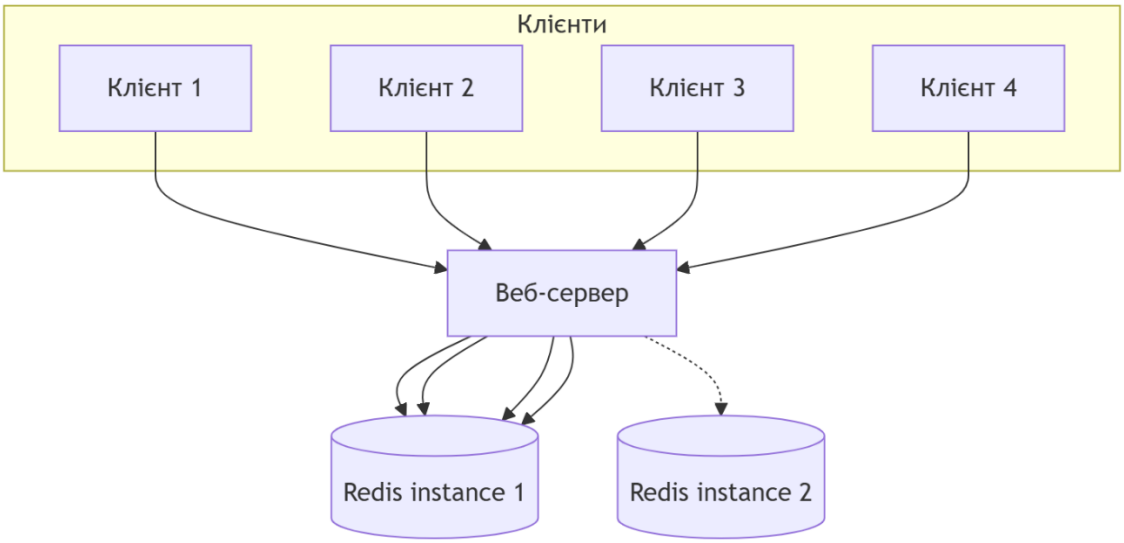

Рисунок 2.18 – Приклад проблеми Hot Keys

Таким чином, ефективне використання кешування у високонавантажених системах потребує врахування різних аспектів синхронізації даних та управління доступом до кешованої інформації. Правильне проєктування механізмів кешування дозволяє уникнути виникнення типових проблем та забезпечити стабільну роботу системи навіть при значному навантаженні.

2.7 Методи оптимізації ефективності кешування

Ефективність роботи систем кешування значною мірою залежить від того, наскільки правильно організовано управління вмістом кешу та яким чином система працює з популярними даними. У процесі функціонування інформаційної системи кеш може перебувати у різних станах залежно від характеру навантаження та структури запитів користувачів.

Для забезпечення максимальної ефективності кешування використовуються різні методи оптимізації. До таких методів належать підтримка кешу у «гарячому» стані, попереднє завантаження даних у кеш, а також ефективне управління популярними даними, які можуть створювати підвищене навантаження на систему.

Застосування відповідних механізмів оптимізації дозволяє зменшити кількість звернень до бази даних, підвищити швидкість обробки запитів та забезпечити стабільну роботу системи навіть при великій кількості одночасних користувачів.

2.7.1 Гарячий та холодний кеш

У процесі роботи інформаційної системи кеш може перебувати у різних станах залежно від того, наскільки ефективно він використовується. У цьому контексті часто використовуються поняття «гарячого» та «холодного» кешу. Перехід між цими станами відбувається динамічно під впливом характеру навантаження та частоти доступу до даних.

Гарячим кешем називають кеш, який уже містить значну кількість часто використовуваних даних. У такому стані більшість запитів може оброблятися без звернення до бази даних, що суттєво підвищує продуктивність системи.

Холодний кеш, навпаки, містить мало або зовсім не містить необхідних даних. У такій ситуації значна частина запитів змушена звертатися безпосередньо до бази даних, що може призводити до збільшення часу відповіді системи та підвищення навантаження на сервери баз даних. Такий стан характерний, зокрема, для початкового етапу роботи системи.

Ефективність роботи кешу часто оцінюється за допомогою показника cache hit ratio, який відображає частку запитів, що були оброблені безпосередньо кешем без звернення до основного сховища даних. Чим вищим є значення цього показника, тим ефективніше працює система кешування.

У високонавантажених веб-застосунках підтримка кешу у «гарячому» стані є важливою умовою забезпечення стабільної продуктивності системи, оскільки значна частина запитів користувачів може обслуговуватися без звернення до бази даних.

2.7.2 Прогрів кешу (Cache Warmup)

У системах кешування часто виникає проблема так званого «холодного кешу». Така ситуація виникає у випадку, коли кеш ще не містить необхідних даних, і значна частина запитів змушена звертатися безпосередньо до бази даних.

Для вирішення цієї проблеми використовується механізм прогріву кешу (cache warmup). Прогрів кешу передбачає попереднє завантаження найбільш часто використовуваних або критично важливих даних у кеш перед початком активної роботи системи.

Наприклад, під час запуску серверного застосунку система може автоматично завантажувати у кеш результати популярних запитів, конфігураційні параметри або інші часто використовувані дані. Це дозволяє уникнути різкого збільшення навантаження на базу даних у початковий період роботи системи.

Прогрів кешу особливо важливий для високонавантажених веб-застосунків. У випадку запуску нової версії застосунку або перезапуску сервера кеш може бути повністю порожнім. У такій ситуації велика кількість одночасних запитів може призвести до значного навантаження на базу даних. Використання механізмів попереднього завантаження даних дозволяє зменшити цей ефект та стабілізувати роботу системи.

3 РЕАЛІЗАЦІЯ ЕКСПЕРИМЕНТАЛЬНОЇ СИСТЕМИ ДОСЛІДЖЕННЯ СТРАТЕГІЙ КЕШУВАННЯ

У попередніх розділах було розглянуто теоретичні основи кешування даних, особливості високонавантажених веб-застосунків та сучасні підходи до оптимізації продуктивності серверних систем. На основі проведеного аналізу було визначено основні стратегії кешування, що використовуються у сучасних інформаційних системах.

З метою експериментального дослідження ефективності різних стратегій кешування у цій роботі було розроблено експериментальний веб-застосунок. Основним призначенням створеної системи є моделювання роботи серверної частини високонавантаженого веб-застосунку та проведення експериментального аналізу впливу різних підходів до кешування на продуктивність системи.

Експериментальний застосунок реалізовано у вигляді серверного веб-API, яке обробляє запити до каталогу товарів. Такий тип системи є типовим прикладом інформаційних систем електронної комерції, де значна частина запитів пов’язана з отриманням інформації про товари, їх характеристики та інші пов’язані дані. У подібних системах більшість операцій становлять операції читання даних, що робить використання кешування особливо ефективним.

Розроблена система дозволяє дослідити вплив різних стратегій кешування на продуктивність серверної частини веб-застосунку шляхом проведення навантажувального тестування. У рамках дослідження реалізовано декілька варіантів організації доступу до даних, зокрема роботу системи без використання кешування, використання локального кешування у пам’яті застосунку, використання розподіленого кешу, а також багаторівневе кешування.

У даному розділі описано архітектуру експериментального веб-застосунку, структуру серверної частини системи, реалізацію механізмів кешування та

інструменти, що використовуються для проведення навантажувального тестування.

3.1 Архітектура системи та конфігурація експериментального середовища

Для проведення експериментального дослідження ефективності різних стратегій кешування було розроблено спеціалізований серверний веб-застосунок, який моделює роботу високонавантаженої системи доступу до даних. Основною метою реалізації системи є створення контрольованого середовища, у якому можна досліджувати вплив різних механізмів кешування на продуктивність серверної частини веб-застосунку.

Архітектура системи побудована за принципами багаторівневої серверної архітектури, що передбачає чітке розділення відповідальності між окремими компонентами застосунку. Такий підхід дозволяє забезпечити модульність системи, спростити її розширення та забезпечити можливість незалежної модифікації окремих компонентів. Крім того, розділення системи на логічні рівні дозволяє більш чітко реалізувати механізми кешування та дослідити їхній вплив на різні етапи обробки запитів.

Серверна частина застосунку реалізована у вигляді REST API з використанням платформи ASP.NET Core (.NET 8). Вибір цієї технології обумовлений її високою продуктивністю, широкими можливостями інтеграції з іншими компонентами екосистеми .NET та підтримкою сучасних архітектурних підходів до побудови серверних застосунків. Крім того, платформа ASP.NET Core надає вбудовані механізми роботи з кешуванням, зокрема підтримку локального кешу у пам’яті застосунку.

У якості основного сховища даних використовується реляційна база даних PostgreSQL. Ця система управління базами даних характеризується високою стабільністю, підтримкою складних запитів та можливістю ефективної роботи з великими обсягами даних. У межах дослідження база даних використовується як

основне джерело інформації, доступ до якого здійснюється через рівень доступу до даних серверного застосунку.

Для реалізації розподіленого кешування використовується система Redis, яка є однією з найбільш поширених платформ для організації високопродуктивного кешу у сучасних веб-застосунках. Redis забезпечує дуже швидкий доступ до даних завдяки зберіганню інформації у пам’яті та підтримує різні механізми управління часом життя кешованих записів.

У рамках експериментального дослідження Redis використовується як розподілений кеш другого рівня, що дозволяє дослідити ефективність використання багаторівневих архітектур кешування.

Для локального кешування у пам’яті серверного застосунку використовується механізм IMemoryCache, який входить до складу платформи .NET.

Загальна архітектура експериментальної системи прдеставлена на рисунку 3.1.

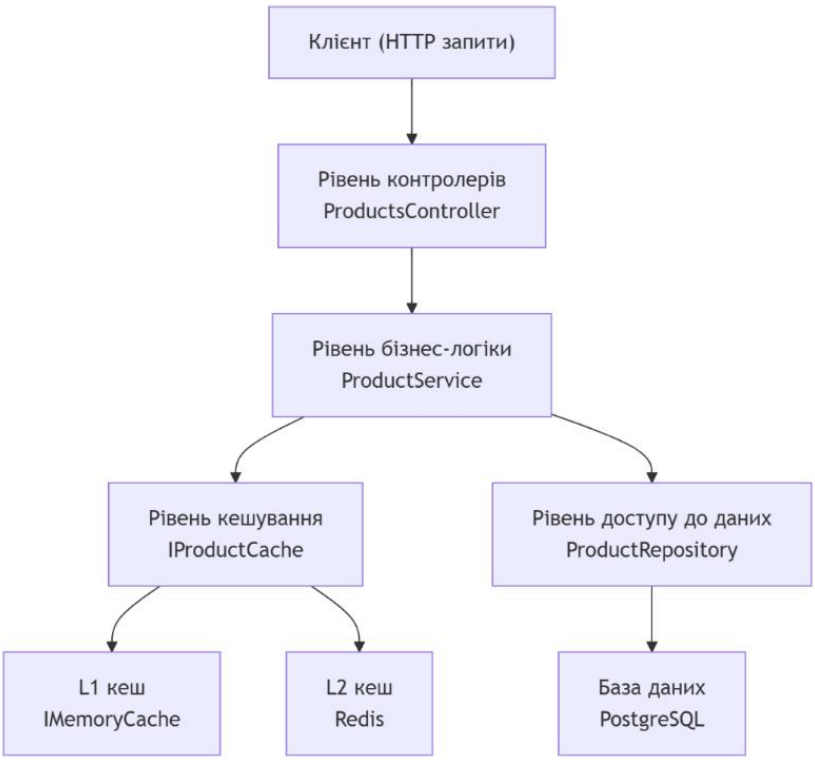

Рисунок 3.1 – Загальна архітектура експериментальної системи

Конфігурація параметрів кешування виконується за допомогою конфігураційного файлу застосунку. У цьому файлі визначаються параметри підключення до бази даних та кеш-сервера, а також режим роботи системи кешування.

Фрагмент конфігураційного файлу наведено у лістингу 3.1.

Лістинг 3.1 – Фрагмент конфігураційного файлу застосунку

```
"Caching": {
  "Mode": "Redis", // None, Memory, Redis, Hybrid
  "MemoryTtlSeconds": 60,
  "RedisTtlSeconds": 180
}
```

Параметр Mode визначає режим роботи системи кешування. У межах експерименту реалізовано декілька режимів функціонування системи, що відповідають різним стратегіям кешування. Зокрема, система може працювати без використання кешування, використовувати локальний кеш у пам’яті застосунку, розподілений кеш Redis або комбінований підхід, що поєднує обидва механізми.

Використання конфігураційного параметра для вибору режиму кешування дозволяє змінювати стратегію роботи системи без необхідності модифікації програмного коду. Це значно спрощує проведення експериментів та забезпечує можливість швидкого перемикання між різними режимами тестування.

Для розгортання інфраструктури системи використовується контейнеризація за допомогою платформи Docker. У контейнерах запускаються сервер бази даних PostgreSQL та кеш-сервер Redis. Такий підхід дозволяє забезпечити ізольоване середовище виконання, спростити розгортання системи та гарантувати відтворюваність експериментальних результатів.

Контейнеризація також дозволяє легко відтворювати однакові умови експерименту під час проведення навантажувального тестування, що є важливим фактором для отримання коректних результатів дослідження.

3.2 Архітектура серверної частини застосунку

Серверна частина розробленого веб-застосунку побудована відповідно до принципів багаторівневої архітектури, що передбачає розділення системи на окремі логічні рівні, кожен з яких відповідає за виконання певних функцій. Такий підхід дозволяє підвищити модульність програмної системи, спростити її підтримку та забезпечити можливість незалежної модифікації окремих компонентів.

У розробленій системі виділено декілька основних рівнів архітектури: рівень контролерів, рівень бізнес-логіки, рівень доступу до даних та рівень кешування. Кожен із цих рівнів виконує окрему роль у процесі обробки запитів клієнтів.

Розділення системи на окремі рівні дозволяє реалізувати механізми кешування таким чином, щоб вони були ізольовані від інших компонентів системи. Це, у свою чергу, забезпечує можливість дослідження впливу різних стратегій кешування без необхідності зміни логіки роботи інших частин застосунку.

3.2.1 Рівень контролерів

Рівень контролерів відповідає за обробку HTTP-запитів клієнтів та формування відповідей серверного застосунку. Контролери є точкою входу до системи та забезпечують взаємодію між клієнтською частиною застосунку і внутрішньою логікою серверної частини.

У розробленій системі контролери реалізовані з використанням механізмів ASP.NET Core Web API. Контролер приймає HTTP-запити від клієнтів, викликає відповідні методи рівня бізнес-логіки та повертає результати у вигляді HTTP-відповідей.

Контролер не містить складної логіки обробки даних. Його основною задачею є передача запиту до відповідного сервісу та формування відповіді у форматі, що відповідає вимогам REST-архітектури.

Фрагмент реалізації контролера наведено у лістингу 3.2.

Лістинг 3.2 – Реалізація контролера отримання продукту

```
[HttpGet("{id:int}")]
[ProducesResponseType(typeof(ProductDto), StatusCodes.Status200OK)]
[ProducesResponseType(StatusCodes.Status404NotFound)]
public async Task<IActionResult> GetById(int id, CancellationToken
cancellationToken)
{
    var product = await _productService.GetByIdAsync(id,
cancellationToken);

    return product is null ? NotFound() : Ok(product);
}
```

У наведеному фрагменті контролер обробляє HTTP-запит на отримання інформації про продукт за його ідентифікатором. Після отримання запиту контролер викликає метод GetByIdAsync сервісу ProductService, який виконує подальшу обробку запиту.

Якщо продукт з відповідним ідентифікатором не знайдено, контролер повертає HTTP-відповідь з кодом 404 Not Found. У разі успішного отримання даних формується відповідь з кодом 200 OK, що містить інформацію про продукт у форматі JSON.

Таким чином, рівень контролерів забезпечує взаємодію між клієнтами системи та серверною частиною застосунку, делегуючи виконання основної логіки обробки запитів відповідним сервісам.

3.2.2 Рівень бізнес-логіки

Рівень бізнес-логіки відповідає за реалізацію основних операцій обробки даних у системі. Саме на цьому рівні виконуються операції отримання, створення та обробки інформації, а також взаємодія з рівнем кешування та рівнем доступу до даних.

У розробленій системі бізнес-логіка реалізована у вигляді сервісів. Сервіси інкапсулюють логіку роботи з доменними об’єктами та забезпечують централізовану точку обробки запитів, що надходять від контролерів.

Основним компонентом цього рівня є сервіс ProductService, який відповідає за отримання інформації про продукти, створення нових записів та взаємодію з механізмами кешування.

Фрагмент реалізації методу отримання продукту наведено у лістингу 3.3.

Лістинг 3.3 – Реалізація методу отримання продукту у сервісі

```
public async Task<ProductDto?> GetByIdAsync(int id, CancellationToken
cancellationToken = default)
{
    _logger.LogInformation("Retrieving Product. ProductId: {ProductId}",
id);

    var cachedProduct = await _cache.GetProductAsync(id,
cancellationToken);
    if (cachedProduct is not null)
    {
        return CreateProductDto(cachedProduct);
```

```
    }

    var product = await _repository.GetByIdAsync(id, cancellationToken);
    if (product is null)
    {
        return null;
    }

    await _cache.SetProductAsync(product, cancellationToken);
    return CreateProductDto(product);
}
```

У цьому методі реалізовано основну логіку отримання даних про продукт. Спочатку виконується спроба отримати дані з кешу. Якщо відповідний запис присутній у кеші, система одразу повертає результат без звернення до бази даних.

У випадку відсутності даних у кеші виконується звернення до рівня доступу до даних, після чого отриманий результат зберігається у кеші для подальшого використання.

Такий підхід дозволяє суттєво зменшити кількість звернень до бази даних у випадку повторних запитів до одних і тих самих даних.

3.2.3 Рівень доступу до даних

Рівень доступу до даних відповідає за взаємодію серверного застосунку з основним сховищем інформації. Основною задачею цього рівня є виконання операцій читання та запису даних у базі даних, а також ізоляція логіки роботи з базою даних від інших компонентів системи.

У розробленій системі доступ до бази даних реалізовано з використанням бібліотеки Entity Framework Core, яка є об’єктно-реляційним відображувачем (ORM) для платформи .NET. Використання ORM дозволяє працювати з даними

у вигляді об’єктів доменної моделі, що значно спрощує розробку та підтримку застосунку.

Для організації доступу до даних використовується патерн Repository, який дозволяє інкапсулювати логіку взаємодії з базою даних та забезпечує чітке розділення відповідальності між різними рівнями архітектури. Репозиторій виступає як посередник між рівнем бізнес-логіки та базою даних.

У межах системи реалізовано репозиторій ProductRepository, який відповідає за виконання операцій отримання та збереження інформації про продукти.

Фрагмент реалізації методу отримання продукту наведено у лістингу 3.4.

Лістинг 3.4 – Реалізація методу отримання продукту з бази даних

```
public async Task<Product?> GetByIdAsync(int id, CancellationToken
cancellationToken = default)
{
    _logger.LogInformation("Querying database");

    await Task.Delay(2000);

    return await _dbContext.Products
        .AsNoTracking()
        .FirstOrDefaultAsync(p => p.Id == id, cancellationToken);
}
```

У наведеному фрагменті коду реалізовано отримання продукту з бази даних за його ідентифікатором. Для виконання запиту використовується контекст бази даних DbContext, який забезпечує доступ до таблиці Products.

Метод AsNoTracking() використовується для підвищення продуктивності читання даних, оскільки у цьому випадку Entity Framework не виконує

відстеження змін об’єктів. Такий підхід є доцільним у ситуаціях, коли отримані дані використовуються лише для читання.

У методі також використано штучну затримку виконання (Task.Delay), яка імітує повільний доступ до бази даних. Такий підхід дозволяє більш наочно продемонструвати вплив механізмів кешування на продуктивність системи під час проведення експериментального дослідження.

Таким чином, рівень доступу до даних забезпечує централізований механізм взаємодії із базою даних та дозволяє ізолювати логіку роботи з базою даних від інших компонентів серверної частини застосунку.

3.2.4 Рівень кешування

Рівень кешування відповідає за тимчасове зберігання часто використовуваних даних з метою зменшення кількості звернень до бази даних та підвищення швидкості обробки запитів.

У розробленій системі механізми кешування реалізовані у вигляді окремого шару архітектури, який розташований між рівнем бізнес-логіки та рівнем доступу до даних. Такий підхід дозволяє ізолювати логіку кешування від інших компонентів системи та забезпечити можливість гнучкої зміни стратегії кешування.

Для взаємодії з кешем використовується інтерфейс IProductCache, який визначає основні операції доступу до кешованих даних. Використання інтерфейсу дозволяє реалізувати декілька різних механізмів кешування, які можуть використовуватися залежно від конфігурації системи. Такий підхід відповідає принципам інверсії залежностей та сприяє підвищенню модульності системи.

У межах дослідження реалізовано декілька варіантів реалізації цього інтерфейсу, що відповідають різним стратегіям кешування:

– відсутність кешування;
– локальне кешування у пам’яті застосунку;
– використання розподіленого кешу Redis;
– багаторівневе кешування, що поєднує локальний кеш та Redis.

Завдяки використанню інтерфейсу доступу до кешу серверний застосунок може використовувати різні реалізації механізмів кешування без зміни логіки роботи рівня бізнес-логіки.

Конкретна реалізація кешування визначається конфігурацією застосунку. Такий підхід дозволяє легко змінювати режим роботи системи під час проведення експериментів та досліджувати вплив різних стратегій кешування на продуктивність системи.

Детальна реалізація кожної зі стратегій кешування буде розглянута у наступному підрозділі.

3.3 Реалізація механізмів кешування

Однією з основних задач даного дослідження є експериментальна оцінка ефективності різних стратегій кешування у серверній частині веб-застосунку. Для забезпечення можливості проведення такого дослідження у системі реалізовано декілька альтернативних механізмів кешування, які можуть використовуватися залежно від конфігурації застосунку.

Архітектура системи побудована таким чином, щоб механізми кешування були ізольовані від інших компонентів серверної частини застосунку. Це дозволяє змінювати стратегію кешування без модифікації логіки роботи контролерів або сервісів.

Для реалізації такого підходу використовується інтерфейс IProductCache, який визначає базові операції доступу до кешованих даних. Різні реалізації цього інтерфейсу відповідають різним стратегіям кешування.

У межах розробленої системи реалізовано чотири режими роботи:

– система без використання кешування;
– використання локального кешу у пам’яті застосунку;
– використання розподіленого кешу Redis;
– використання багаторівневого кешування.

Конкретний режим роботи системи визначається конфігурацією застосунку та може змінюватися без необхідності модифікації програмного коду.

3.3.1 Система без використання кешування

Базовим режимом роботи системи є режим без використання кешування. У цьому випадку кожен запит до серверного застосунку призводить до безпосереднього звернення до бази даних.

Такий режим використовується як контрольний сценарій під час проведення експериментального дослідження. Результати вимірювання продуктивності у цьому режимі дозволяють оцінити базовий час обробки запитів без використання механізмів кешування.

Для реалізації цього режиму у системі використовується клас NoProductCache, який реалізує інтерфейс IProductCache, але не виконує жодного збереження або отримання даних із кешу.

Фрагмент реалізації методу отримання продукту за допомогою класу NoProductCache наведено у лістингу 3.5.

Лістинг 3.5 – Реалізація механізму роботи без кешування

```
public Task<Product?> GetProductAsync(int id, CancellationToken
cancellationToken = default)
{
    _logger.LogInformation("Cache MISS");
    return Task.FromResult<Product?>(null);
}
```

У цьому випадку метод отримання продукту завжди повертає значення null, що сигналізує сервісному рівню про відсутність даних у кеші. У результаті система змушена виконувати звернення до бази даних для кожного запиту.

Такий підхід дозволяє отримати еталонні результати продуктивності системи без використання кешування, що є необхідним для подальшого порівняльного аналізу ефективності різних стратегій кешування.

3.3.2 Використання локального кешування IMemoryCache

Одним із найбільш поширених підходів до оптимізації продуктивності серверних застосунків є використання локального кешування у пам’яті процесу застосунку. У платформі .NET для реалізації такого механізму використовується компонент IMemoryCache, який дозволяє зберігати дані безпосередньо у пам’яті серверного застосунку.

Основною перевагою локального кешування є дуже низька затримка доступу до даних, оскільки операції читання та запису виконуються без використання мережевих з’єднань. Це дозволяє значно зменшити час обробки повторних запитів до одних і тих самих даних.

У розробленій системі локальне кешування реалізовано у класі MemoryProductCache, який використовує інтерфейс IMemoryCache для збереження та отримання кешованих записів.

Фрагмент реалізації методу отримання даних з кешу наведено у лістингу 3.6.

Лістинг 3.6 – Отримання даних із локального кешу

```
public Task<Product?> GetProductAsync(int id, CancellationToken
cancellationToken = default)
{
```

```
    if (_memoryCache.TryGetValue<Product>(CacheKeys.Product(id), out var
product))
    {
        _logger.LogInformation("Cache HIT (L1)");
        return Task.FromResult<Product?>(product);
    }

    _logger.LogInformation("Cache MISS");
    return Task.FromResult<Product?>(null);
}
```

У наведеному фрагменті коду використовується метод TryGetValue, який перевіряє наявність запису у кеші. У випадку, якщо відповідний запис присутній у кеші, система повертає кешовані дані без звернення до бази даних.

Якщо запис відсутній у кеші, метод повертає значення null, що сигналізує сервісному рівню про необхідність отримання даних із бази даних.

Збереження даних у кеші виконується після отримання інформації з бази даних. При цьому для кожного запису встановлюється час життя кешу, який визначає, як довго дані можуть зберігатися у пам’яті застосунку.

3.3.3 Використання розподіленого кешу Redis

У системах з високим навантаженням використання лише локального кешування у пам’яті застосунку може бути недостатнім. У випадку, коли серверний застосунок розгорнуто у вигляді декількох екземплярів, локальний кеш кожного процесу буде містити власну копію даних. Це може призводити до неузгодженості кешованих записів та зменшення ефективності кешування.

Для вирішення цієї проблеми у сучасних веб-застосунках широко використовуються розподілені системи кешування, які дозволяють зберігати

кешовані дані у спільному зовнішньому сховищі, доступному для всіх екземплярів серверного застосунку.

У межах даного дослідження у якості розподіленого кешу використовується система Redis, яка є однією з найбільш поширених платформ для організації високопродуктивного кешування. Redis зберігає дані у оперативній пам’яті та забезпечує дуже швидкий доступ до кешованих записів навіть при великій кількості одночасних запитів.

У розробленій системі взаємодія з Redis реалізована через інтерфейс IDistributedCache, який входить до складу платформи .NET. Такий підхід дозволяє використовувати стандартний механізм доступу до розподіленого кешу та спрощує інтеграцію з серверним застосунком.

Для роботи з Redis у системі реалізовано клас RedisProductCache, який відповідає за отримання та збереження кешованих записів.

Фрагмент реалізації методу отримання даних із Redis наведено у лістингу 3.7.

Лістинг 3.7 – Отримання даних із розподіленого кешу Redis

```
public async Task<Product?> GetProductAsync(int id, CancellationToken
cancellationToken = default)
{
    var payload = await
_distributedCache.GetStringAsync(CacheKeys.Product(id),
cancellationToken);
    if (!string.IsNullOrWhiteSpace(payload))
    {
        _logger.LogInformation("Cache HIT (Redis)");
        return JsonSerializer.Deserialize<Product>(payload);
    }

    _logger.LogInformation("Cache MISS");
```

```
    return null;
}
```

У наведеному фрагменті коду виконується спроба отримати кешований запис із Redis за допомогою методу GetStringAsync. Якщо відповідний запис присутній у кеші, він десеріалізується у об’єкт доменної моделі та повертається сервісному рівню.

У випадку відсутності даних у кеші система повертає значення null, що сигналізує про необхідність виконання запиту до бази даних.

Після отримання даних із бази даних система зберігає їх у Redis із використанням часу життя кешу, який визначається у конфігурації застосунку.

Використання розподіленого кешу дозволяє значно зменшити навантаження на базу даних та забезпечити спільне використання кешованих даних між різними екземплярами серверного застосунку.

3.3.4 Гібридне кешування

Для підвищення ефективності роботи систем кешування у сучасних веб-застосунках часто використовується багаторівнева архітектура кешу, яка поєднує декілька механізмів кешування. Такий підхід дозволяє використовувати переваги різних типів кешу та мінімізувати затримки доступу до даних.

У межах даного дослідження реалізовано гібридну модель кешування, яка поєднує локальний кеш у пам’яті застосунку та розподілений кеш Redis. У такій архітектурі кешування організовано у вигляді двох рівнів:

– L1 кеш – локальний кеш у пам’яті серверного застосунку;
– L2 кеш – розподілений кеш Redis.

Під час обробки запиту система спочатку перевіряє наявність необхідних даних у локальному кеші (L1). Якщо дані знайдено, вони одразу повертаються сервісному рівню без додаткових звернень до зовнішніх систем.

У випадку відсутності даних у локальному кеші виконується звернення до розподіленого кешу Redis (L2). Якщо запис знайдено у Redis, він зберігається у локальному кеші та повертається клієнту.

Лише у випадку відсутності даних у обох рівнях кешу система виконує запит до бази даних.

Фрагмент реалізації методу отримання даних у гібридному кеші наведено у лістингу 3.8.

Лістинг 3.8 – Реалізація гібридного механізму кешування

```
public async Task<Product?> GetProductAsync(int id, CancellationToken
cancellationToken = default)
{
    var key = CacheKeys.Product(id);

    if (_memoryCache.TryGetValue<Product>(key, out var memoryProduct))
    {
        _logger.LogInformation("Cache HIT (L1)");
        return memoryProduct;
    }

    var payload = await _distributedCache.GetStringAsync(key,
cancellationToken);
    if (!string.IsNullOrWhiteSpace(payload))
    {
        var redisProduct = JsonSerializer.Deserialize<Product>(payload);
        _logger.LogInformation("Cache HIT (Redis)");

        var memoryCacheEntryOptions = new MemoryCacheEntryOptions
        {
            AbsoluteExpirationRelativeToNow =
TimeSpan.FromSeconds(_cachingOptions.MemoryTtlSeconds)
        };
```

```
            _memoryCache.Set(key, redisProduct, memoryCacheEntryOptions);

            return redisProduct;
        }

        _logger.LogInformation("Cache MISS");
        return null;
    }
```

У цьому механізмі локальний кеш виконує роль першого рівня доступу до даних, забезпечуючи мінімальні затримки під час обробки запитів. Розподілений кеш Redis використовується як другий рівень кешування, що дозволяє зберігати дані у спільному кеш-сховищі.

Такий підхід дозволяє поєднати високу швидкість локального кешу із можливістю спільного використання кешованих даних між різними екземплярами серверного застосунку.

У результаті використання гібридного кешування дозволяє значно підвищити ефективність роботи системи та зменшити навантаження на базу даних при великій кількості повторних запитів.

3.4 Механізми інвалідації кешу

Ефективність систем кешування значною мірою залежить від правильного управління життєвим циклом кешованих даних. Оскільки кеш зберігає копії інформації, що міститься у базі даних, необхідно забезпечити механізми оновлення або видалення застарілих записів. Без таких механізмів система може повертати користувачам неактуальні дані.

Реалізація механізмів інвалідації кешу є необхідною умовою коректного функціонування системи кешування. Саме ці механізми забезпечують

актуальність кешованих даних та дозволяють проводити експериментальне дослідження ефективності різних стратегій кешування у контрольованих умовах.

У розробленій системі реалізовано декілька механізмів інвалідації кешу, які забезпечують підтримку актуальності кешованих даних. До них належать інвалідація на основі часу життя кешованих записів та оновлення кешу після виконання операцій запису.

3.4.1 Інвалідація на основі часу життя даних

Одним із найбільш поширених підходів до інвалідації кешу є використання часу життя кешованих даних (TTL – Time To Live). У цьому випадку кожному запису у кеші призначається певний інтервал часу, протягом якого дані вважаються актуальними. Після завершення цього інтервалу запис автоматично видаляється з кешу.

У розробленій системі механізм TTL використовується як для локального кешу у пам’яті застосунку, так і для розподіленого кешу Redis. При цьому для різних рівнів кешу можуть використовуватися різні значення часу життя.

У гібридній моделі кешування локальний кеш використовується як перший рівень доступу до даних (L1), а Redis – як другий рівень кешу (L2). Для локального кешу встановлено менший час життя, що дозволяє швидше оновлювати дані у пам’яті застосунку.

Фрагмент реалізації встановлення часу життя кешованих записів наведено у лістингу 3.9.

Лістинг 3.9 – Встановлення часу життя запису у локальному кеші

```
public Task SetProductAsync(Product product, CancellationToken
cancellationToken = default)
{
    var memoryCacheEntryOptions = new MemoryCacheEntryOptions
```

```
    {
        AbsoluteExpirationRelativeToNow =
TimeSpan.FromSeconds(_cachingOptions.MemoryTtlSeconds)
    };

    _memoryCache.Set(CacheKeys.Product(product.Id), product,
memoryCacheEntryOptions);

    return Task.CompletedTask;
}
```

У наведеному фрагменті коду використовується клас MemoryCacheEntryOptions, який дозволяє задати параметри зберігання запису у кеші. Параметр AbsoluteExpirationRelativeToNow визначає час, через який запис буде автоматично видалений із кешу.

Аналогічний механізм використовується для розподіленого кешу Redis. Фрагмент відповідної реалізації наведено у лістингу 3.10.

Лістинг 3.10 – Встановлення часу життя запису у Redis

```
public Task SetProductAsync(Product product, CancellationToken
cancellationToken = default)
{
    var distributedCacheEntryOptions = new DistributedCacheEntryOptions
    {
        AbsoluteExpirationRelativeToNow =
TimeSpan.FromSeconds(_cachingOptions.RedisTtlSeconds)
    };
    var payload = JsonSerializer.Serialize(product);
    return
_distributedCache.SetStringAsync(CacheKeys.Product(product.Id), payload,
distributedCacheEntryOptions, cancellationToken);
}
```

Використання TTL дозволяє автоматично видаляти застарілі записи з кешу та зменшувати ймовірність повернення неактуальних даних користувачам.

3.4.2 Оновлення кешу після зміни дани

Окрім інвалідації на основі часу життя записів, у системі реалізовано механізм оновлення кешу після виконання операцій запису до бази даних. Такий підхід дозволяє забезпечити актуальність кешованих даних одразу після їх зміни.

У розробленій системі цей механізм реалізовано на рівні бізнес-логіки. Після створення нового запису у базі даних відповідний об’єкт одразу зберігається у кеші.

Фрагмент реалізації створення нового продукту у сервісі наведено у лістингу 3.11.

Лістинг 3.11 – Оновлення кешу після створення нового запису

```
public async Task<ProductDto> CreateAsync(CreateProductRequest request,
CancellationToken cancellationToken = default)
{
    var product = new Product
    {
        Name = request.Name.Trim(),
        Description = request.Description.Trim(),
        Price = request.Price,
        Category = request.Category.Trim(),
        CreatedAt = DateTime.UtcNow
    };

    var created = await _repository.AddAsync(product, cancellationToken);

    await _cache.SetProductAsync(created, cancellationToken);
```

```
    return CreateProductDto(created);
}
```

Після успішного збереження нового продукту у базі даних викликається метод SetProductAsync, який додає новий запис до кешу. Такий підхід дозволяє уникнути ситуації, коли одразу після створення об’єкта система буде змушена повторно звертатися до бази даних.

Запит на створення нового продукту надходить до системи через відповідний HTTP-метод контролера. Фрагмент реалізації цього методу наведено у лістингу 3.12.

Лістинг 3.12 – Контролер створення нового продукту

```
[HttpPost]
[ProducesResponseType(typeof(ProductDto), StatusCodes.Status201Created)]
public async Task<IActionResult> Create([FromBody] CreateProductRequest
request, CancellationToken cancellationToken)
{
    var created = await _productService.CreateAsync(request,
cancellationToken);

    return CreatedAtAction(nameof(GetById), new { id = created.Id },
created);
}
```

У цьому випадку контролер приймає HTTP-запит на створення нового продукту та передає його до сервісного рівня, де виконується логіка створення запису та оновлення кешу.

Застосування механізму оновлення кешу після зміни даних дозволяє підтримувати актуальність кешованої інформації та зменшити кількість повторних звернень до бази даних.

3.5 Організація навантажувального тестування

Для експериментального дослідження ефективності різних механізмів кешування у серверній частині веб-застосунку було проведено навантажувальне тестування системи. Основною метою такого тестування є оцінка впливу використання кешування на продуктивність системи при великій кількості одночасних запитів.

Навантажувальне тестування дозволяє змоделювати роботу реального веб-застосунку в умовах значної кількості одночасних користувачів та визначити ключові показники продуктивності системи. У межах даного дослідження тестування проводиться для різних стратегій кешування, що реалізовані у серверній частині застосунку.

Експеримент передбачає виконання серії тестів, під час яких система працює у різних режимах кешування. Для кожного режиму виконуються однакові сценарії навантаження, що дозволяє провести коректне порівняння результатів.

У ході навантажувального тестування вимірюються ключові показники продуктивності системи, зокрема середній час відповіді запиту, медіанний час відповіді, значення 95-го перцентиля часу відповіді, пропускна здатність системи, показники стабільності часу відповіді, а також максимальний час відповіді.

3.5.1 Інструмент навантажувального тестування k6

Для проведення навантажувального тестування у межах даного дослідження використовується інструмент k6. Цей інструмент є сучасною платформою для тестування продуктивності веб-застосунків та широко використовується для оцінювання роботи серверних систем під навантаженням.

Однією з основних переваг k6 є можливість опису сценаріїв навантаження за допомогою скриптів мовою JavaScript, що дозволяє гнучко моделювати різні

типи поведінки користувачів. Крім того, інструмент забезпечує збір детальної статистики щодо виконання HTTP-запитів та дозволяє аналізувати ключові показники продуктивності системи.

У межах експерименту k6 використовується для генерації HTTP-запитів до REST API розробленого веб-застосунку. Кожен віртуальний користувач (Virtual User, VU) періодично виконує запити до API, що дозволяє моделювати поведінку реальних користувачів системи.

Сценарії навантаження описуються у вигляді JavaScript-скриптів, які визначають кількість віртуальних користувачів, тривалість тестування, а також логіку формування HTTP-запитів до серверного застосунку.

Фрагмент коду, що виконує HTTP-запит до API продуктів, наведено у лістингу 3.13.

Лістинг 3.13 – Фрагмент скрипта навантажувального тестування

```
const res = http.get(`${BASE_URL}/${productId}`);

check(res, {
    'status is 200': (r) => r.status === 200,
});
```

У наведеному фрагменті коду виконується HTTP-запит до серверного застосунку з метою отримання інформації про продукт за його ідентифікатором. Після виконання запиту перевіряється коректність відповіді сервера шляхом контролю HTTP-статусу відповіді.

Для більш реалістичного моделювання поведінки користувачів між окремими запитами використовується невелика затримка, що дозволяє уникнути надмірно агресивної генерації запитів.

Таким чином, використання інструменту k6 дозволяє відтворити різні сценарії роботи користувачів та отримати достовірні дані щодо продуктивності серверної частини веб-застосунку при різних механізмах кешування.

3.5.2 Сценарії навантажувального тестування

Для оцінювання ефективності різних стратегій кешування у системі використовується два типи сценаріїв навантаження, які моделюють різні варіанти доступу користувачів до даних. Використання декількох сценаріїв дозволяє дослідити поведінку механізмів кешування в умовах різних моделей використання системи.

У межах дослідження використовуються два сценарії навантаження:

– random workload – випадковий доступ до даних;

– hot workload – доступ до популярних даних із нерівномірним розподілом запитів.

Застосування цих двох сценаріїв дозволяє оцінити ефективність кешування як у випадку рівномірного розподілу запитів, так і у випадку, коли значна частина запитів спрямована до обмеженої групи популярних даних.

Сценарій random workload моделює ситуацію, коли користувачі звертаються до випадкових записів у системі. У такому випадку кожен запит вибирає випадковий ідентифікатор продукту з усього набору даних.

У межах експерименту база даних містить 100000 записів продуктів, тому кожен запит може звертатися до будь-якого з цих записів з однаковою ймовірністю.

У цьому сценарії кожен запит вибирає випадковий ідентифікатор продукту, що призводить до рівномірного розподілу звернень до бази даних.

Такий тип навантаження є складним для механізмів кешування, оскільки ймовірність повторного звернення до одного й того самого запису є відносно

низькою. У результаті значна частина запитів може призводити до звернення до бази даних навіть у разі використання кешування.

Фрагмент реалізації генерації випадкового запиту наведено у лістингу 3.14.

Лістинг 3.14 – Генерація випадкового запиту до API

```
const productId = Math.floor(Math.random() * TOTAL_PRODUCTS) + 1;

const res = http.get(`${BASE_URL}/${productId}`);
```

Сценарій random workload використовується для оцінювання ефективності кешування у випадку рівномірного доступу до даних.

Другий сценарій навантаження – hot workload – моделює ситуацію, коли значна частина запитів користувачів спрямована до обмеженої групи популярних даних.

У реальних веб-застосунках часто спостерігається нерівномірний розподіл доступу до інформації, коли невелика кількість записів використовується значно частіше за інші. Такий розподіл часто описується правилом Парето (80/20), відповідно до якого приблизно 80% запитів припадає на 20% або меншу частину даних.

У даному дослідженні для моделювання такого сценарію використовується наступна схема:

– 80% запитів спрямовані до групи з 500 популярних продуктів;

– 20% запитів звертаються до випадкових продуктів із повного набору даних.

У цьому випадку більшість запитів спрямована до обмеженої групи популярних продуктів, що призводить до частого повторного доступу до одних і тих самих даних.

Фрагмент реалізації вибору ідентифікатора продукту наведено у лістингу 3.15.

Лістинг 3.15 – Реалізація сценарію hot workload

```
function getProductId() {
    const r = Math.random();

    if (r < 0.8) {
        return Math.floor(Math.random() * HOT_PRODUCTS) + 1;
    }

    return Math.floor(Math.random() * TOTAL_PRODUCTS) + 1;
}
```

Такий сценарій є сприятливим для механізмів кешування, оскільки повторні звернення до тих самих записів дозволяють системі ефективно використовувати кешовані дані.

Використання сценарію hot workload дозволяє оцінити, наскільки різні стратегії кешування здатні зменшувати навантаження на базу даних у ситуаціях, коли значна частина запитів спрямована до популярних даних.

3.5.3 Параметри навантаження

Для забезпечення коректного порівняння ефективності різних стратегій кешування всі експерименти проводилися з використанням однакових параметрів навантаження. Це дозволяє виключити вплив зовнішніх факторів та забезпечити об’єктивність отриманих результатів.

Навантажувальне тестування проводиться із використанням моделі ramping-vus, яка передбачає поступове збільшення кількості віртуальних користувачів (Virtual Users, VU) протягом виконання тесту. Такий підхід дозволяє уникнути різкого пікового навантаження на систему та забезпечує більш стабільні результати вимірювань.

Сценарій навантаження складається з декількох послідовних фаз:

– фаза поступового збільшення навантаження (warm-up);
– фаза основного вимірювання продуктивності;
– фаза поступового зменшення навантаження.

Фрагмент конфігурації сценарію навантаження наведено у лістингу 3.16.

Лістинг 3.16 – Конфігурація сценарію навантаження у k6

```
stages: [
    { duration: '20s', target: 20 },
    { duration: '20s', target: 50 },
    { duration: '2m', target: 100 },
    { duration: '10s', target: 0 },
]
```

На початкових етапах тесту кількість віртуальних користувачів поступово збільшується до 100. Це дозволяє системі перейти у стабільний робочий стан перед початком основної фази вимірювання.

Основна фаза тестування триває 2 хвилини та виконується при 100 одночасних віртуальних користувачах. Саме під час цієї фази здійснюється збір основних показників продуктивності системи.

Після завершення фази вимірювання навантаження поступово зменшується до нуля, що дозволяє коректно завершити виконання тесту.

Для більш реалістичного моделювання поведінки користувачів між окремими HTTP-запитами використовується невелика затримка. Фрагмент відповідного коду наведено у лістингу 3.17.

Лістинг 3.17 – Затримка між запитами

```
sleep(0.05);
```

Ця затримка становить 50 мілісекунд та імітує невеликий інтервал між діями користувача у реальному веб-застосунку.

Основні параметри навантажувального тестування наведено у таблиці 3.1.

Таблиця 3.1 – Основні параметри навантажувального тестування

Параметр	Значення
Кількість продуктів у базі даних	100000
Кількість популярних продуктів (hot workload)	500
Максимальна кількість віртуальних користувачів	100
Тривалість основної фази вимірювання	2 хв
Затримка між запитами	50 мс

Застосування однакових параметрів навантаження для всіх експериментів дозволяє коректно порівнювати результати для різних стратегій кешування.

3.5.4 Метрики оцінювання продуктивності системи

Для оцінювання ефективності різних механізмів кешування у процесі навантажувального тестування вимірюється набір ключових показників продуктивності системи.

Основною метрикою є час відповіді HTTP-запиту, який характеризує швидкість обробки запиту серверною частиною застосунку. Для більш повного аналізу продуктивності використовуються декілька статистичних показників часу відповіді.

У межах дослідження аналізуються наступні метрики:

– average latency – середній час відповіді системи;
– median latency – медіанний час відповіді;
– p95 latency – 95-й перцентиль часу відповіді;
– throughput – пропускна здатність системи;
– показник стабільності роботи системи;
– maximum latency – максимальний час відповіді.

Зазначені метрики доповнюють одна одну: показники часу відповіді характеризують швидкість обробки запитів, пропускна здатність відображає здатність системи працювати під навантаженням, а показники стабільності дозволяють оцінити передбачуваність її поведінки.

Аналіз зазначених метрик дозволяє комплексно оцінити вплив використання кешування на продуктивність серверної частини веб-застосунку.

Середній час відповіді визначається як середнє арифметичне часу обробки всіх HTTP-запитів:

$$\overline{T} = \frac{1}{N}\sum_{i=1}^{N} T_i \tag{3.1}$$

де:

– T_i – час обробки і-го запиту;

– N – загальна кількість запитів.

Цей показник дозволяє отримати загальне уявлення про швидкість роботи системи, однак є чутливим до аномально повільних запитів.

Медіанний час відповіді визначається як значення, що ділить впорядкований набір значень часу відповіді навпіл:

$$T_{50} = \text{median}(T) \tag{3.2}$$

Ця метрика є стійкою до викидів та дозволяє оцінити типовий час відповіді системи.

95-й перцентиль визначає значення часу відповіді, нижче якого знаходяться 95% усіх запитів:

$$P_{95} = T_{\lceil 0.95 \cdot N \rceil} \tag{3.3}$$

Дана метрика використовується для оцінювання продуктивності системи у гірших, але типових умовах роботи.

Пропускна здатність системи визначається як кількість запитів, оброблених за одиницю часу:

$$RPS = \frac{N}{T_{total}} \tag{3.4}$$

де:

– N – кількість запитів;

– T_{total} – загальний час виконання тесту.

Дана метрика дозволяє оцінити здатність системи обробляти навантаження та характеризує її загальну продуктивність. Збільшення значення пропускної здатності свідчить про ефективніше використання ресурсів системи та зменшення часу обробки запитів.

Окрім середніх значень продуктивності, важливим аспектом є стабільність роботи системи, яка характеризує варіативність та передбачуваність часу відповіді.

Для оцінювання стабільності використовується співвідношення між значенням 95-го перцентиля та медіаною часу відповіді:

$$S_1 = \frac{P_{95}}{T_{50}} \tag{3.5}$$

Збільшення значення цього показника свідчить про зростання розкиду значень часу відповіді та зниження стабільності системи.

Окрім статистичних показників, у роботі враховується наявність аномально високих затримок (latency spikes), які можуть суттєво впливати на якість роботи системи.

Під latency spike розуміються окремі запити, час відповіді яких значно перевищує значення p95 та виходить за межі типового розподілу часу відповіді.

Для кількісної оцінки таких затримок використовується максимальне значення часу відповіді:

$$T_{max} = \max(T_i) \tag{3.6}$$

де:

– T_i – час обробки і-го запиту.

Наявність значних значень T_{max} свідчить про потенційні проблеми стабільності системи та може бути пов’язана із перевантаженням окремих компонентів або мережевими затримками.

Аналіз latency spikes дозволяє виявити ситуації, коли окремі запити виконуються значно довше за типові значення, навіть якщо середні показники продуктивності залишаються прийнятними.

4 ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ СТРАТЕГІЙ КЕШУВАННЯ

У цьому розділі представлено результати експериментального дослідження ефективності різних стратегій кешування, реалізованих у серверній частині веб-застосунку. Дослідження проводиться шляхом виконання серії навантажувальних тестів для різних режимів роботи системи, що дозволяє оцінити вплив використання кешування на продуктивність обробки запитів.

Під час експерименту аналізується робота системи у декількох конфігураціях кешування, що включають режим без використання кешу, локальне кешування у пам’яті застосунку, використання розподіленого кешу Redis, а також гібридну модель кешування, яка поєднує локальний кеш та Redis.

Для кожної конфігурації виконуються два сценарії навантаження, які моделюють різні моделі доступу користувачів до даних. Такий підхід дозволяє дослідити ефективність кешування як у випадку рівномірного доступу до інформації, так і у випадку, коли значна частина запитів спрямована до обмеженої групи популярних даних.

Результати проведених експериментів наведено у вигляді таблиць та графіків, що дозволяє наочно порівняти ефективність різних стратегій кешування. Детальний аналіз отриманих результатів представлено у наступних підрозділах.

4.1 Методика проведення експерименту

Для дослідження впливу різних стратегій кешування на продуктивність серверної частини веб-застосунку було проведено серію навантажувальних експериментів. Метою експерименту є оцінювання ефективності використання кешування при обробці запитів користувачів у різних сценаріях доступу до даних.

Експериментальне дослідження проводилося на основі розробленого веб-застосунку, який реалізує механізми кешування. Під час проведення експериментів система послідовно запускалася у різних режимах роботи, що дозволяло оцінити вплив кожної стратегії кешування на продуктивність системи.

У межах дослідження було розглянуто чотири конфігурації серверного застосунку:

– режим роботи без використання кешування;
– використання локального кешування у пам’яті застосунку;
– використання розподіленого кешу Redis;
– використання гібридного кешування, що поєднує локальний кеш та Redis.

Для кожної з конфігурацій було виконано два сценарії навантаження, що моделюють різні моделі доступу користувачів до даних. У першому сценарії використовується випадковий доступ до записів бази даних, тоді як у другому сценарії моделюється нерівномірний розподіл запитів, при якому значна частина звернень припадає на обмежену групу популярних даних.

Таким чином, загалом було проведено вісім окремих експериментів, що відповідають різним комбінаціям стратегій кешування та сценаріїв навантаження. Перелік проведених експериментів наведено у таблиці 4.1.

Таблиця 4.1 – Перелік проведених експериментів

Номер експерименту	Стратегія кешування	Тип навантаження
1	Без кешування	Random workload
2	Без кешування	Hot workload
3	IMemoryCache	Random workload
4	IMemoryCache	Hot workload
5	Redis	Random workload
6	Redis	Hot workload

Продовження таблиці 4.1

Номер експерименту	Стратегія кешування	Тип навантаження
7	Hybrid	Random workload
8	Hybrid	Hot workload

Для формалізації впливу параметрів на продуктивність системи у межах дослідження введемо узагальнену модель часу відповіді:

$$T_{response} = f(strategy, workload)$$

де:

– *strategy* – стратегія кешування;

– *workload* – тип навантаження.

У межах проведеного дослідження зазначені параметри виступають основними факторами експерименту, що визначають поведінку системи під навантаженням.

Тип навантаження безпосередньо впливає на характер доступу до даних, що, у свою чергу, визначає ефективність використання кешування. Зокрема, у сценарії random workload спостерігається низька ймовірність повторного доступу до даних, тоді як у сценарії hot workload значна частина запитів спрямована до обмеженої множини записів.

Інші параметри системи, зокрема час життя кешу (TTL), конфігурація інфраструктури та параметри навантаження, залишаються фіксованими протягом усіх експериментів з метою забезпечення коректності порівняння результатів.

Під час проведення експериментів використовувався набір тестових даних, що містить 100000 записів продуктів. Такий обсяг даних дозволяє змоделювати роботу системи в умовах реального веб-застосунку та забезпечити достатній рівень варіативності під час виконання запитів.

Для забезпечення наочності впливу кешування у шарі доступу до даних було реалізовано штучну затримку виконання запитів до бази даних. Такий підхід дозволяє чітко продемонструвати різницю між обробкою запитів без використання кешу та обробкою запитів із використанням кешованих даних.

Під час кожного експерименту вимірювалися ключові показники продуктивності системи, зокрема:

– середній час відповіді HTTP-запитів;
– медіанний час відповіді;
– значення 95-го перцентиля часу відповіді;
– пропускна здатність системи;
– показник стабільності часу відповіді;
– максимальний час відповіді.

Отримані результати використовуються для подальшого аналізу ефективності різних стратегій кешування та їх впливу на продуктивність серверної частини веб-застосунку.

4.2 Результати тестування системи без використання кешування

Першим етапом експериментального дослідження було проведення навантажувального тестування системи у режимі роботи без використання механізмів кешування. У цьому режимі кожен HTTP-запит до серверного застосунку призводить до безпосереднього звернення до бази даних для отримання необхідної інформації.

Метою проведення цього експерименту є визначення базових показників продуктивності системи, які використовуються як еталон для подальшого порівняння з результатами використання різних стратегій кешування.

Для оцінювання продуктивності системи виконано два сценарії навантаження, що відповідають різним моделям доступу до даних: випадковий

доступ до записів бази даних (random workload) та доступ до обмеженої групи популярних записів (hot workload).

4.2.1 Random workload

Результати навантажувального тестування системи без використання кешування для сценарію random workload наведено у таблиці 4.2.

Таблиця 4.2 – Результати тестування системи без кешування (random workload)

Метрика	Значення
Average latency	2.02 s
Median latency	2.02 s
p95 latency	2.04 s
Throughput	29.23 req/s
Stability	1.01
Max latency	2.16 s

Отримані результати свідчать, що середній час відповіді системи становить 2 секунди, що пов’язано з необхідністю виконання запиту до бази даних для кожного HTTP-запиту.

Значення медіанного часу відповіді практично збігається із середнім значенням, що свідчить про стабільний характер виконання запитів. Також значення 95-го перцентиля знаходиться близько до середнього часу відповіді, що означає відсутність значних відхилень у часі обробки запитів.

Пропускна здатність системи у цьому сценарії становить 29 запитів за секунду, що визначає базовий рівень продуктивності системи без використання механізмів кешування.

Аналіз показників стабільності показує, що значення p95 практично збігається з медіанним часом відповіді, що підтверджується значенням stability, близьким до 1, та свідчить про рівномірний характер обробки запитів.

Максимальний час відповіді також незначно відрізняється від типових значень, що вказує на відсутність аномальних затримок у системі.

4.2.2 Hot workload

Результати навантажувального тестування системи без використання кешування для сценарію hot workload наведено у таблиці 4.3.

Таблиця 4.3 – Результати тестування системи без кешування (hot workload)

Метрика	Значення
Average latency	2.02 s
Median latency	2.02 s
p95 latency	2.04 s
Throughput	29.34 req/s
Stability	1.01
Max latency	2.20 s

Аналіз результатів показує, що використання сценарію hot workload у цьому режимі не призводить до помітного покращення продуктивності системи. Середній та медіанний час відповіді залишаються на рівні 2 секунд, що практично співпадає з результатами сценарію random workload.

Це пояснюється тим, що у відсутності кешування система не може використовувати повторні звернення до одних і тих самих даних для прискорення обробки запитів. Навіть якщо запити часто звертаються до одних і тих самих записів, кожен запит все одно виконує повний цикл доступу до бази даних.

Пропускна здатність системи у цьому випадку також залишається на рівні 29 запитів за секунду, що підтверджує відсутність впливу сценарію доступу до даних на продуктивність системи без використання кешування.

Аналіз показників стабільності демонструє, що система працює рівномірно, оскільки значення p95 та медіанного часу відповіді практично співпадають. Значення stability, близьке до 1, підтверджує відсутність значного розкиду часу відповіді.

Максимальний час відповіді також знаходиться в межах типових значень, що свідчить про відсутність аномальних затримок.

Таким чином, отримані результати демонструють базовий рівень продуктивності системи та можуть використовуватися як еталон для подальшого порівняння з результатами використання різних механізмів кешування.

4.3 Результати використання локального кешування (IMemoryCache)

Наступним етапом експериментального дослідження було тестування системи з використанням локального кешування у пам’яті застосунку. У цьому режимі серверний застосунок використовує механізм IMemoryCache для збереження раніше отриманих даних у пам’яті процесу, що дозволяє зменшити кількість звернень до зовнішніх ресурсів та скоротити час обробки запитів..

Основною метою цього експерименту є оцінювання впливу локального кешування на продуктивність системи при обробці запитів користувачів. На відміну від режиму без кешування, у цьому випадку система може повертати дані без звернення до бази даних, якщо відповідний запис вже знаходиться у кеші, що особливо ефективно при повторному доступі до однакових даних.

Як і у попередньому експерименті, тестування проводилося для двох сценаріїв навантаження: random workload та hot workload.

4.3.1 Random workload

Результати навантажувального тестування системи з використанням локального кешування для сценарію random workload наведено у таблиці 4.4.

Таблиця 4.4 – Результати тестування з використанням IMemoryCache (random workload)

Метрика	Значення
Average latency	1.98 s
Median latency	2.02 s
p95 latency	2.04 s
Throughput	29.90 req/s
Stability	1.01
Max latency	2.19 s

Аналіз результатів демонструє, що використання локального кешування у цьому сценарії практично не призводить до суттєвого покращення продуктивності системи. Середній час відповіді зменшується лише незначно у порівнянні з режимом без кешування.

Причиною цього є те, що при випадковому доступі до даних ймовірність повторного звернення до одного й того самого запису є відносно низькою. У результаті значна частина запитів продовжує звертатися безпосередньо до бази даних, що обмежує ефективність використання кешування.

Пропускна здатність системи також змінюється незначно та становить 29 запитів за секунду, що близько до результатів, отриманих у режимі без кешування.

Показники стабільності свідчать про відсутність значних відхилень у часі відповіді, оскільки значення p95 та медіанного часу відповіді практично

співпадають. Значення stability, близьке до 1, підтверджує рівномірний характер виконання запитів.

Максимальний час відповіді також не демонструє суттєвих відхилень від основної маси запитів.

4.3.2 Hot workload

Результати тестування для сценарію hot workload наведено у таблиці 4.5.

Таблиця 4.5 – Результати тестування з використанням IMemoryCache (hot workload)

Метрика	Значення
Average latency	592 ms
Median latency	5.22 ms
p95 latency	2.02 s
Throughput	94.71 req/s
Stability	387.0
Max latency	2.29 s

На відміну від сценарію random workload, у цьому випадку використання локального кешування призводить до значного покращення продуктивності системи. Середній час відповіді зменшується з 2 секунд до 592 мілісекунд, що свідчить про суттєве зниження кількості звернень до бази даних.

Особливо показовим є значення медіанного часу відповіді, яке становить 5 мілісекунд. Це означає, що значна частина запитів обробляється безпосередньо із локального кешу без необхідності виконання запиту до бази даних.

Значення 95-го перцентиля часу відповіді залишається близьким до 2 секунд, що пояснюється наявністю запитів, які не знаходять відповідного запису у кеші та змушені звертатися до бази даних.

Пропускна здатність системи у цьому режимі значно зростає та становить 95 запитів за секунду, що більш ніж утричі перевищує базовий рівень продуктивності системи без використання кешування.

Отримані результати демонструють, що локальне кешування може значно підвищити продуктивність системи у випадках, коли значна частина запитів звертається до популярних даних.

Разом з тим, аналіз показників стабільності демонструє значний розкид часу відповіді. Незважаючи на дуже низьке значення медіанного часу відповіді, значення p95 залишається на рівні часу доступу до бази даних, що призводить до високого значення stability. Це пояснюється наявністю двох режимів роботи системи: швидкого обслуговування запитів із кешу та повільного доступу до бази даних у випадку cache miss.

Максимальний час відповіді також підтверджує наявність окремих повільних запитів.

4.4 Результати використання розподіленого кешу Redis

Наступним етапом експериментального дослідження було проведення навантажувального тестування системи з використанням розподіленого кешу Redis. У цьому режимі кешовані дані зберігаються у зовнішньому кеш-сервері, доступ до якого здійснюється через мережеве з’єднання.

Використання розподіленого кешу дозволяє забезпечити спільний доступ до кешованих даних для декількох екземплярів серверного застосунку. На відміну від локального кешування, при якому дані зберігаються лише у пам’яті одного процесу, Redis може використовуватися як централізований рівень кешування.

Як і у попередніх експериментах, тестування проводилося для двох сценаріїв навантаження: random workload та hot workload.

4.4.1 Random workload

Результати тестування системи з використанням Redis для сценарію random workload наведено у таблиці 4.6.

Таблиця 4.6 – Результати тестування з використанням Redis (random workload)

Метрика	Значення
Average latency	1.97 s
Median latency	2.02 s
p95 latency	2.04 s
Throughput	30.02 req/s
Stability	1.01
Max latency	2.33 s

Отримані результати показують, що використання Redis у цьому сценарії не призводить до значного покращення продуктивності системи. Середній час відповіді та значення медіанного часу відповіді залишаються близькими до результатів, отриманих у режимі без кешування.

Це пояснюється тим, що у випадку випадкового доступу до даних більшість запитів не знаходить відповідного запису у кеші. У результаті система змушена виконувати звернення до бази даних, що суттєво обмежує ефективність кешування.

Пропускна здатність системи у цьому режимі становить 29 запитів за секунду, що практично відповідає результатам попередніх експериментів.

Аналіз стабільності показує, що система працює рівномірно, оскільки значення p95 та медіанного часу відповіді практично збігаються. Значення stability близьке до 1, що свідчить про відсутність значних відхилень.

Максимальний час відповіді незначно перевищує середні значення, що не вказує на наявність суттєвих аномальних затримок.

4.4.2 Hot workload

Результати навантажувального тестування для сценарію hot workload наведено у таблиці 4.7.

Таблиця 4.7 – Результати тестування з використанням Redis (hot workload)

Метрика	Значення
Average latency	632 ms
Median latency	146 ms
p95 latency	2.27 s
Throughput	89.07 req/s
Stability	15.46
Max latency	23.5 s

Результати експерименту демонструють суттєве покращення продуктивності системи у порівнянні з режимом без використання кешування. Середній час відповіді зменшується до 632 мілісекунд, що свідчить про значне скорочення кількості звернень до бази даних.

Медіанний час відповіді становить 146 мілісекунд, що значно менше за час відповіді при прямому зверненні до бази даних. Це означає, що значна частина запитів обробляється із використанням кешованих даних у Redis.

Разом з тим, значення 95-го перцентиля залишається на рівні часу доступу до бази даних. Це пояснюється наявністю запитів, для яких відповідні записи відсутні у кеші, що призводить до виконання повного циклу звернення до бази даних.

Пропускна здатність системи у цьому режимі становить 89 запитів за секунду, що більш ніж утричі перевищує базовий рівень продуктивності системи без використання кешування.

Отримані результати демонструють, що використання розподіленого кешу дозволяє суттєво зменшити навантаження на базу даних та підвищити загальну продуктивність системи у випадках, коли значна частина запитів звертається до популярних даних.

Разом з тим, аналіз стабільності виявляє наявність значного розкиду часу відповіді. Значення p95 суттєво перевищує медіанний час відповіді, що призводить до підвищеного значення stability.

Крім того, максимальний час відповіді значно перевищує типові значення, що свідчить про наявність аномальних затримок (latency spikes). Це може бути пов’язано з мережевими затримками, перевантаженням кеш-сервера, накопиченням запитів у черзі, конкуренцією за ресурси кеш-сервера або витісненням даних із кешу.

4.5 Результати використання гібридного кешування

Останнім етапом експериментального дослідження було проведення тестування системи з використанням гібридної моделі кешування, яка поєднує локальний кеш у пам’яті застосунку та розподілений кеш Redis. У такій архітектурі кешування організовано у вигляді двох рівнів: локальний кеш виступає як перший рівень доступу до даних, тоді як Redis використовується як другий рівень кешування.

Такий підхід дозволяє поєднати високу швидкість доступу до даних у локальному кеші із можливістю зберігання кешованих записів у розподіленому середовищі. У випадку відсутності даних у локальному кеші система виконує звернення до Redis, що дозволяє уникнути повторного звернення до бази даних.

Як і у попередніх експериментах, тестування проводилося для двох сценаріїв навантаження: random workload та hot workload.

4.5.1 Random workload

Результати навантажувального тестування системи з використанням гібридного кешування для сценарію random workload наведено у таблиці 4.8.

Таблиця 4.8 – Результати тестування з використанням гібридного кешування (random workload)

Метрика	Значення
Average latency	1.97 s
Median latency	2.02 s
p95 latency	2.05 s
Throughput	29.95 req/s
Stability	1.01
Max latency	2.37 s

Отримані результати свідчать, що використання гібридного кешування у цьому сценарії не призводить до суттєвого покращення продуктивності системи. Значення середнього та медіанного часу відповіді залишаються близькими до результатів, отриманих у попередніх експериментах.

Це зумовлено тим, що при випадковому доступі до великої кількості записів більшість запитів не знаходить відповідних даних у кеші. У результаті система змушена звертатися до бази даних, що обмежує ефективність використання кешування навіть при наявності декількох рівнів кешу.

Пропускна здатність системи у цьому режимі становить 29 запитів за секунду, що відповідає базовому рівню продуктивності системи.

Показники стабільності свідчать про рівномірний характер роботи системи, оскільки значення p95 та медіанного часу відповіді практично співпадають. Значення stability близьке до 1, що підтверджує відсутність значного розкиду.

Максимальний час відповіді також не демонструє наявності аномальних затримок.

4.5.2 Hot workload

Результати тестування системи з використанням гібридного кешування для сценарію hot workload наведено у таблиці 4.9.

Таблиця 4.9 – Результати тестування з використанням гібридного кешування (hot workload)

Метрика	Значення
Average latency	460 ms
Median latency	7.29 ms
p95 latency	2.06 s
Throughput	119.02 req/s
Stability	282.6
Max latency	2.70 s

Отримані результати демонструють найбільш високий рівень продуктивності серед усіх досліджених стратегій кешування. Середній час відповіді зменшується до 460 мілісекунд, що є найкращим показником серед усіх проведених експериментів.

Особливо показовим є значення медіанного часу відповіді, яке становить 7 мілісекунд. Це означає, що більшість запитів обробляється безпосередньо з локального кешу без необхідності звернення до Redis або бази даних.

Значення 95-го перцентиля часу відповіді залишається на рівні 2 секунд, що пояснюється наявністю окремих запитів, для яких відповідні записи відсутні у кеші та потребують звернення до бази даних.

Пропускна здатність системи у цьому режимі становить 119 запитів за секунду, що є найвищим показником серед усіх протестованих конфігурацій системи.

Аналіз показників стабільності демонструє наявність розкиду часу відповіді, що пов’язано із поєднанням швидкого доступу до локального кешу та повільніших операцій доступу до бази даних. Значення stability є високим, однак максимальний час відповіді залишається значно нижчим у порівнянні з використанням лише Redis, що свідчить про більш стабільну роботу системи.

Отримані результати демонструють, що використання гібридного кешування дозволяє максимально ефективно використовувати переваги як локального, так і розподіленого кешу, забезпечуючи високу швидкість обробки запитів та значне зменшення навантаження на базу даних.

4.6 Порівняльний аналіз стратегій кешування

Після проведення серії експериментів для різних стратегій кешування було виконано порівняльний аналіз отриманих результатів. Метою цього аналізу є комплексне оцінювання впливу різних механізмів кешування на продуктивність серверної частини веб-застосунку з урахуванням не лише середніх значень часу відповіді, але й стабільності роботи системи та наявності аномальних затримок.

У межах дослідження порівнюються результати тестування чотирьох конфігурацій системи:

– система без використання кешування;
– використання локального кешу IMemoryCache;
– використання розподіленого кешу Redis;
– використання гібридного кешування.

Аналіз проводиться окремо для двох сценаріїв навантаження, оскільки характер доступу до даних суттєво впливає на ефективність кешування.

Для узагальнення результатів експериментів було сформовано зведену таблицю показників продуктивності системи для всіх протестованих стратегій кешування. Відповідні результати наведено у таблиці 4.10.

Таблиця 4.10 – Порівняння результатів експериментів для різних стратегій кешування

Стратегія	Workload	Avg latency	Median	p95	Throughput	Stability	Max latency
None	Random	2.02 s	2.02 s	2.04 s	29.23	1.01	2.16 s
Memory	Random	1.98 s	2.02 s	2.04 s	29.90	1.01	2.19 s
Redis	Random	1.97 s	2.02 s	2.04 s	30.02	1.01	2.33 s
Hybrid	Random	1.97 s	2.02 s	2.05 s	29.95	1.01	2.37 s
None	Hot	2.02 s	2.02 s	2.04 s	29.34	1.01	2.20 s
Memory	Hot	592 ms	5.22 ms	2.02 s	94.71	387.0	2.29 s
Redis	Hot	632 ms	146 ms	2.27 s	89.07	15.46	23.5 s
Hybrid	Hot	460 ms	7.29 ms	2.06 s	119.02	282.6	2.70 s

Наведені результати демонструють суттєву різницю у продуктивності системи залежно від типу навантаження та використаної стратегії кешування. Для більш детального аналізу отримані результати розглянуто окремо для кожного сценарію навантаження.

4.6.1 Порівняння результатів random workload

У сценарії random workload кожен запит звертається до випадкового запису бази даних. Такий тип навантаження характеризується рівномірним розподілом

доступу до даних, що суттєво зменшує ймовірність повторного використання кешованих записів.

Результати експерименту показують, що використання різних стратегій кешування у цьому сценарії практично не призводить до покращення продуктивності системи. Середній час відповіді для всіх протестованих конфігурацій залишається близьким до 2 секунд, що відповідає часу виконання запиту до бази даних.

Для наочного порівняння результатів було побудовано графік середнього часу відповіді системи для різних стратегій кешування у сценарії random workload, який зображено на рисунку 4.1.

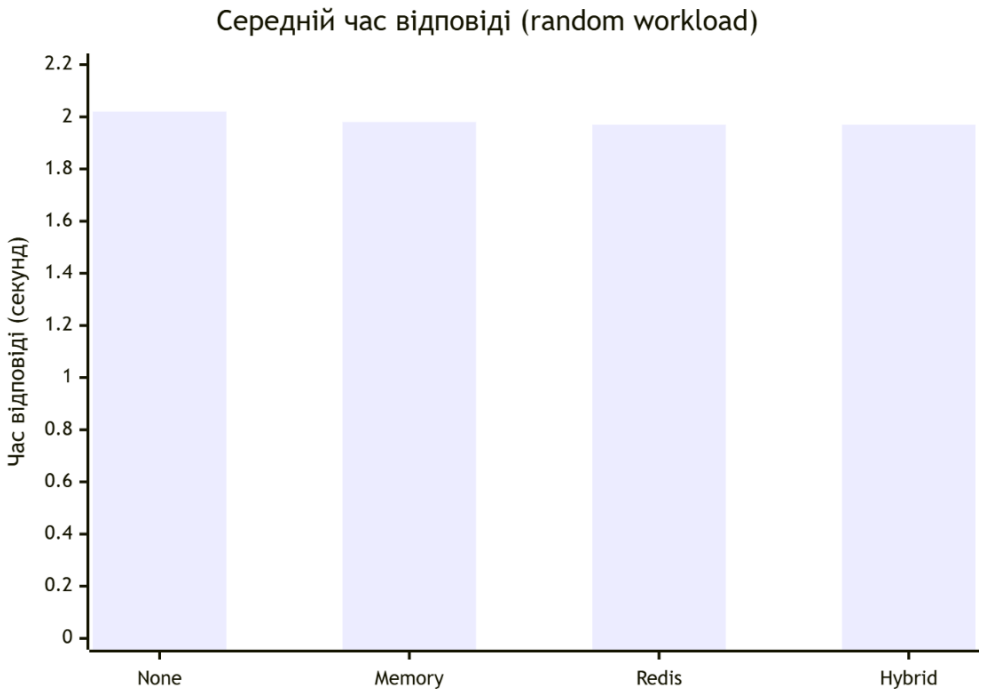

Рисунок 4.1 – Порівняння середнього часу відповіді для random workload

Як видно з графіка, значення середнього часу відповіді для всіх стратегій кешування відрізняються незначно. Аналогічна поведінка спостерігається і для інших показників продуктивності, зокрема медіанного часу відповіді та пропускної здатності системи, значення яких залишаються практично незмінними незалежно від використаної стратегії кешування.

Відсутність суттєвого ефекту від кешування у цьому сценарії пояснюється низьким рівнем повторного доступу до даних. Оскільки кожен запит звертається до випадкового запису, ймовірність cache hit є мінімальною, що призводить до необхідності виконання запиту до бази даних у більшості випадків незалежно від обраної стратегії кешування.

Аналіз показників стабільності підтверджує однорідний характер роботи системи. Для всіх стратегій кешування значення p95 практично співпадає з медіанним часом відповіді, що відповідає значенню stability, близькому до 1. Це свідчить про передбачуваний час обробки запитів без значних відхилень.

Аналіз максимального часу відповіді також не виявляє суттєвих відмінностей між різними конфігураціями системи. Значення максимального часу відповіді знаходиться у межах 2.1–2.3 секунд, що відповідає типовому часу доступу до бази даних. Відсутність значних відхилень свідчить про відсутність аномальних затримок (latency spikes).

Таким чином, результати експерименту показують, що у випадку рівномірного доступу до великої кількості даних використання кешування не забезпечує покращення продуктивності системи. Це обумовлено низькою ефективністю повторного використання даних, що фактично нівелює переваги кешування у даному сценарії.

4.6.2 Порівняння результатів hot workload

У сценарії hot workload значна частина запитів спрямована до обмеженої групи популярних записів. Такий розподіл доступу до даних створює сприятливі умови для ефективного використання механізмів кешування, оскільки одні й ті самі записи можуть багаторазово використовуватися різними запитами.

Результати експерименту показують, що у цьому випадку використання кешування суттєво впливає на продуктивність системи.

Порівняння середнього часу відповіді для різних стратегій кешування наведено на рисунку 4.2.

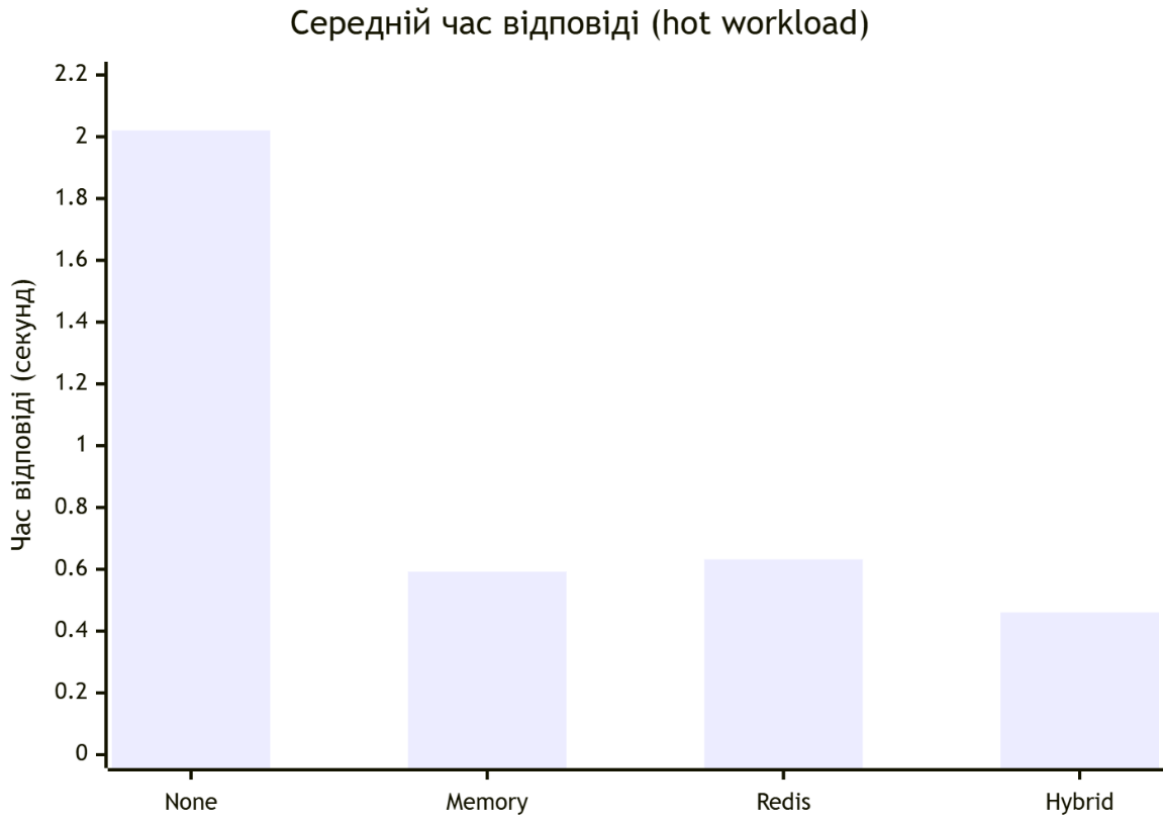

Рисунок 4.2 – Порівняння середнього часу відповіді для hot workload

Як видно з графіка, використання кешування дозволяє значно зменшити середній час відповіді системи. У режимі без використання кешування середній час відповіді становить 2 секунд, що відповідає часу виконання запиту до бази даних.

При використанні локального кешування середній час відповіді зменшується до 592 мілісекунд, тоді як при використанні Redis цей показник становить 632 мілісекунди. Найкращий результат демонструє гібридна модель кешування, у якій середній час відповіді зменшується до 460 мілісекунд.

Зниження середнього часу відповіді пояснюється високою ймовірністю повторного доступу до популярних даних, що дозволяє ефективно використовувати кешовані значення та уникати звернень до бази даних.

Особливо показовим є порівняння медіанного часу відповіді, яке наведено на рисунку 4.3.

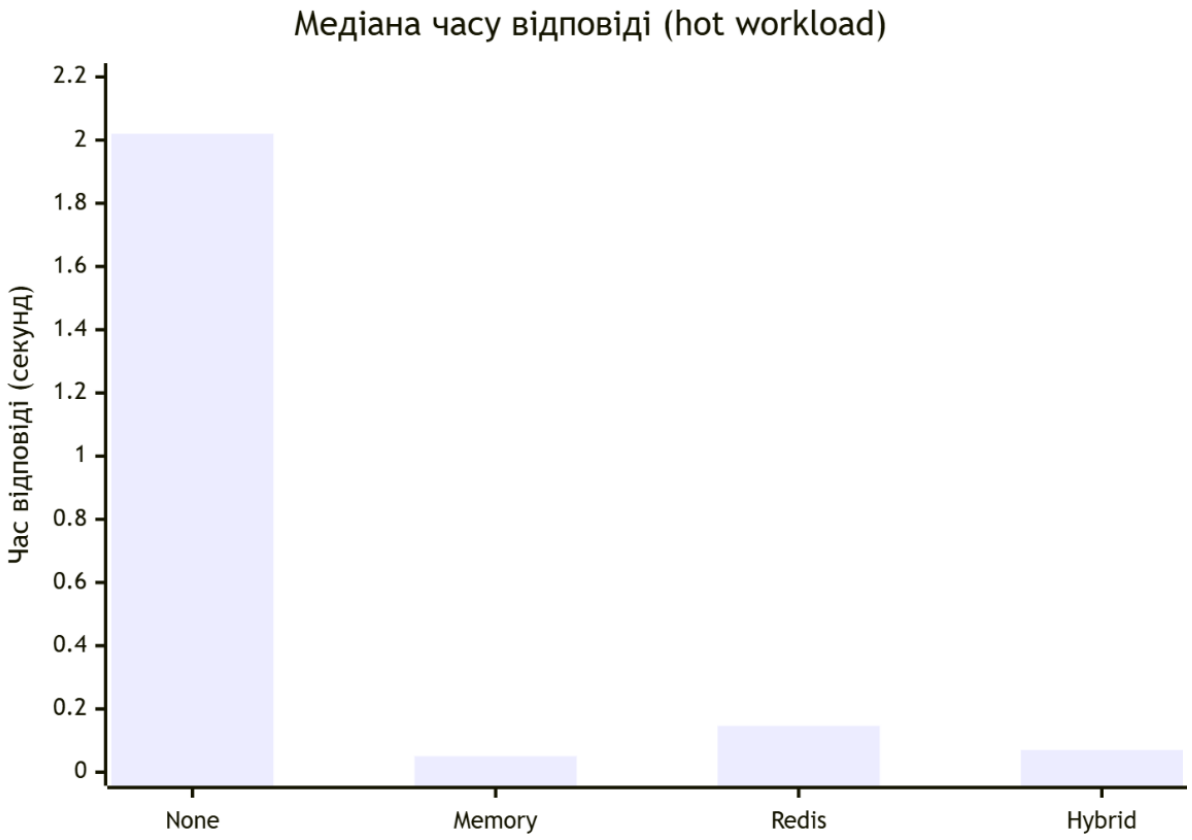

Рисунок 4.3 – Порівняння медіанного часу відповіді для hot workload

Медіанний час відповіді для системи без кешування становить 2 секунди, тоді як при використанні локального кешування він зменшується до 5 мілісекунд. Для Redis цей показник становить 146 мілісекунд, а для гібридної моделі кешування – 7 мілісекунд.

Таке суттєве зменшення медіанного часу відповіді свідчить про те, що більшість запитів у цьому сценарії обробляється безпосередньо із кешу. Водночас різниця між IMemoryCache та Redis пояснюється наявністю мережевих затримок при доступі до розподіленого кешу.

Порівняння пропускної здатності системи для різних стратегій кешування наведено на рисунку 4.4.

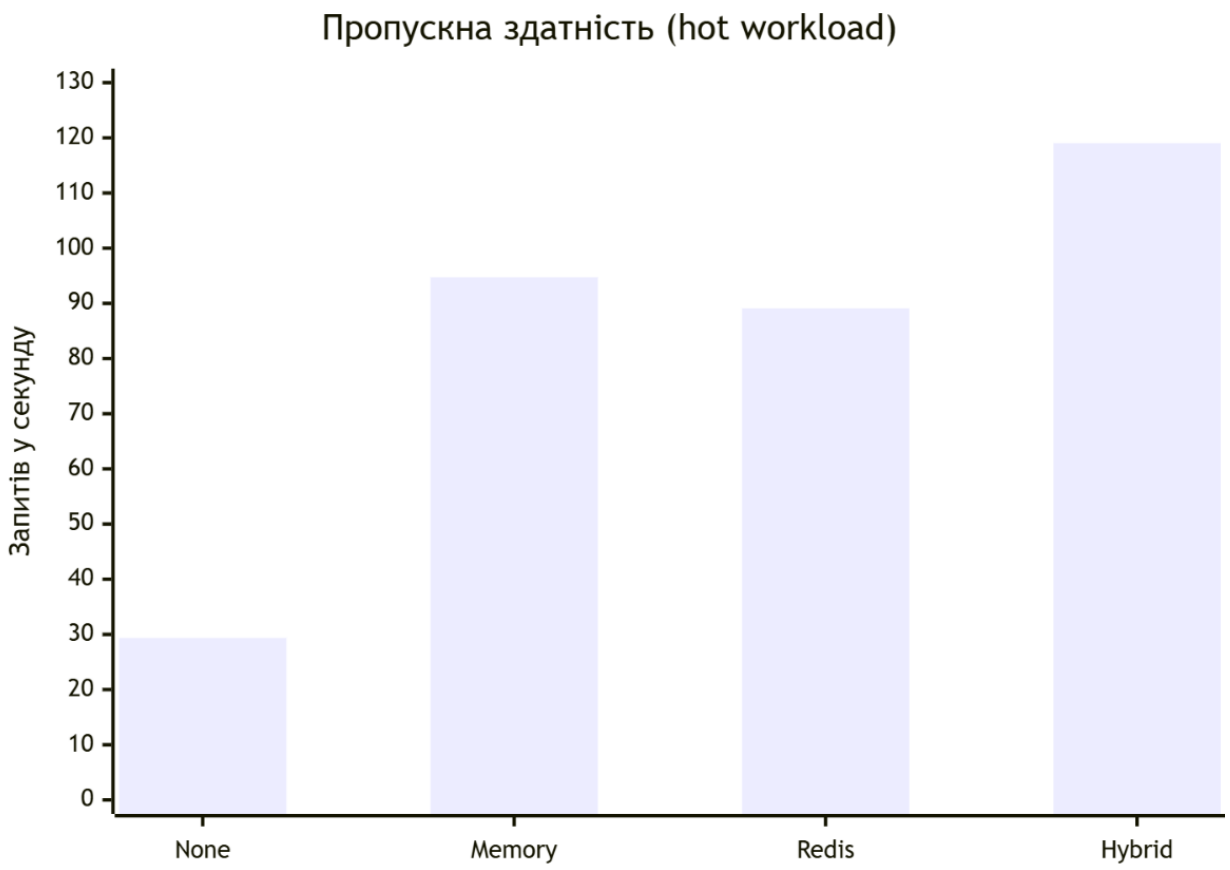

Рисунок 4.4 – Порівняння пропускної здатності системи для hot workload

Як показують результати експерименту, використання кешування дозволяє значно збільшити кількість запитів, які система може обробити за одиницю часу. У режимі без кешування пропускна здатність системи становить 29 запитів за секунду.

При використанні локального кешування пропускна здатність системи збільшується до 95 запитів за секунду, тоді як використання Redis забезпечує 89 запитів за секунду. Найвищу продуктивність демонструє гібридна модель кешування, у якій система здатна обробляти 119 запитів за секунду.

Разом із покращенням середніх показників продуктивності, важливим аспектом є аналіз стабільності роботи системи.

Для наочного порівняння показників стабільності було побудовано графік, представлений на рисунку 4.5.

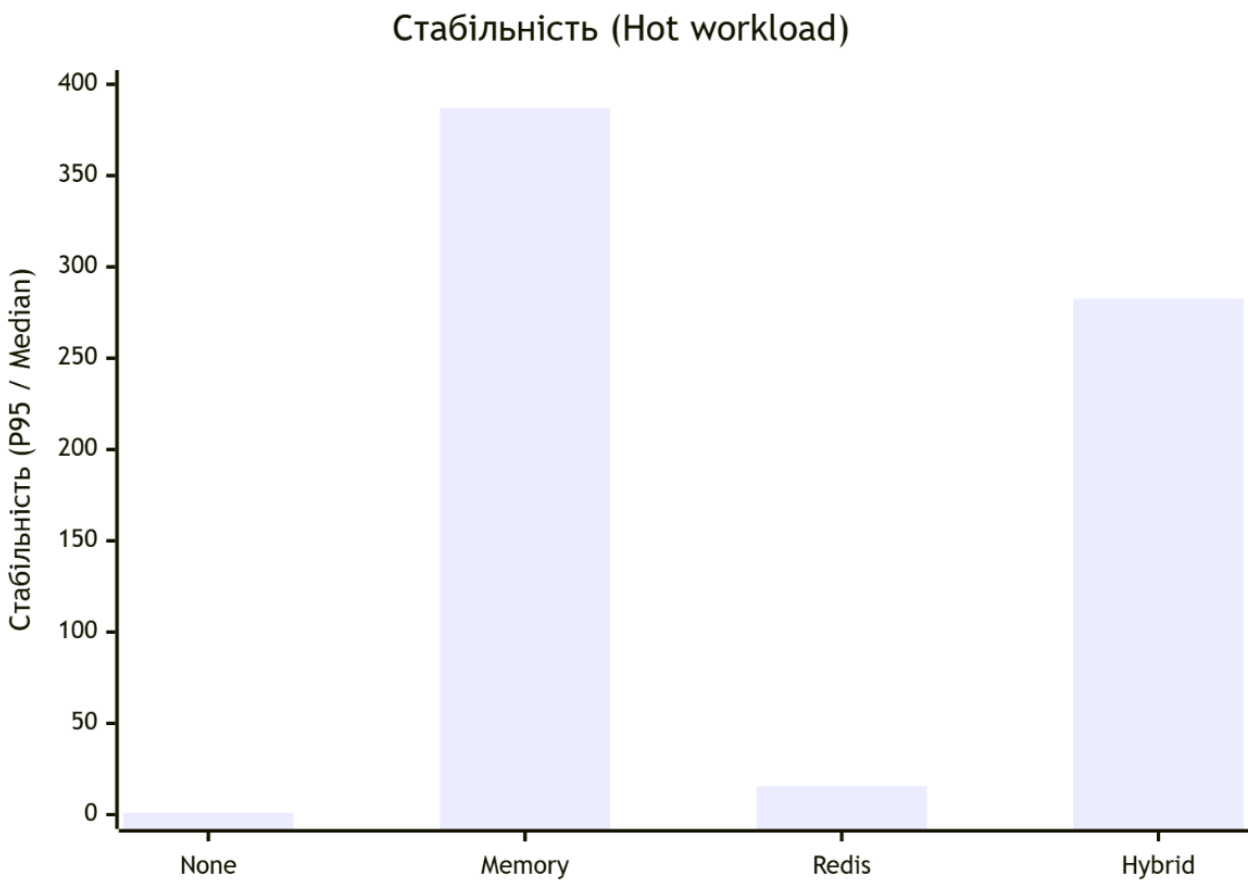

Рисунок 4.5 – Порівняння показників стабільності для hot workload

Аналіз показує, що у випадку використання кешування спостерігається суттєве зростання значення показника stability. Це зумовлено тим, що система працює у двох режимах: швидке обслуговування запитів із кешу та значно повільніша обробка запитів у випадку cache miss.

Найбільш виражений ефект спостерігається для локального кешування та гібридної моделі, де медіанний час відповіді є дуже малим, тоді як значення p95 залишається близьким до часу доступу до бази даних. Це призводить до значного розриву між типовими та гіршими значеннями часу відповіді.

Окремої уваги заслуговує аналіз максимального часу відповіді. Для наочного порівняння максимальних значень часу відповіді було побудовано графік, представлений на рисунку 4.6.

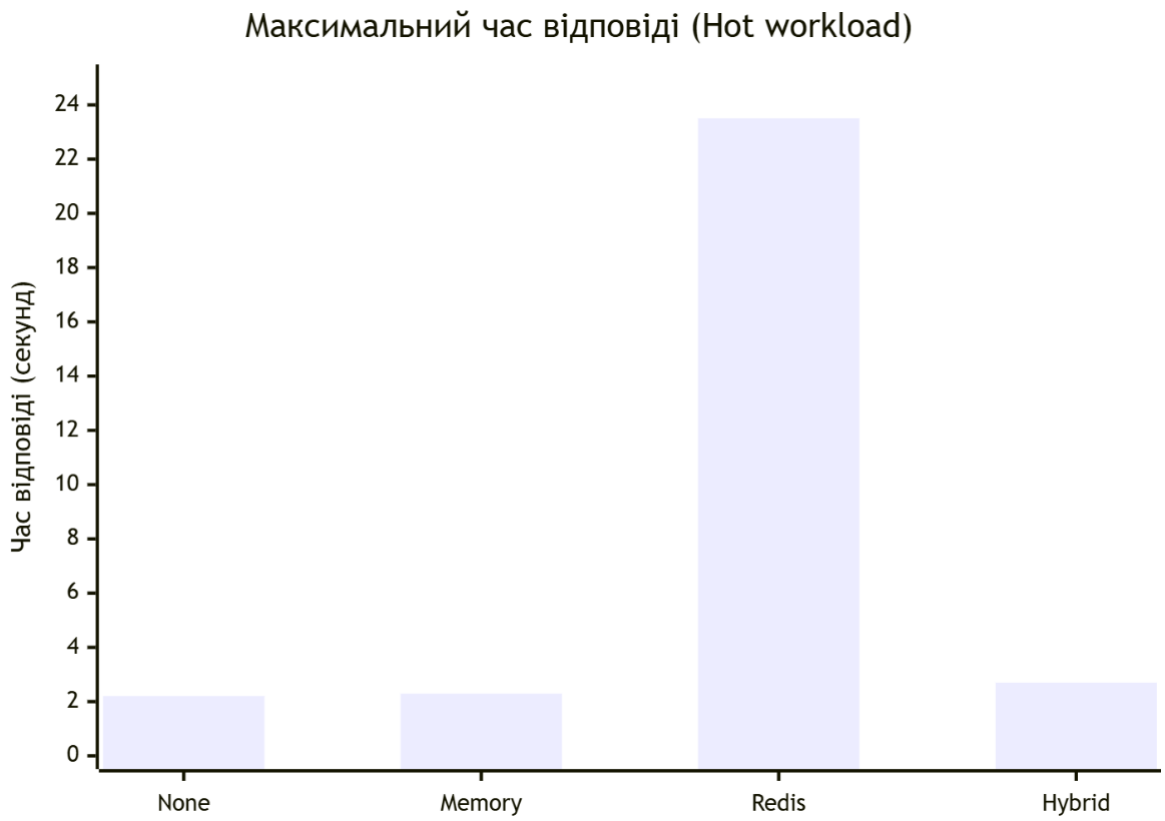

Рисунок 4.6 – Порівняння максимального часу відповіді для hot workload

Як видно з графіка, у випадку використання Redis максимальний час відповіді досягає значення 23.5 секунд, що суттєво перевищує типові значення та свідчить про наявність значних аномальних затримок.

Такі затримки можуть бути пов’язані з мережевими особливостями взаємодії із кеш-сервером, конкуренцією за ресурси або піковим навантаженням на окремі компоненти системи.

На відміну від Redis, гібридна модель кешування демонструє значно менший максимальний час відповіді (близько 2.7 секунд), що свідчить про більш стабільну роботу системи. Це пояснюється тим, що більшість запитів обробляється на рівні локального кешу, що дозволяє зменшити залежність від мережевих затримок.

Таким чином, гібридне кешування не лише забезпечує найкращі середні показники продуктивності, але й дозволяє зменшити вплив аномальних затримок у порівнянні з використанням лише розподіленого кешу.

4.7 Висновки за результатами експериментального дослідження

Проведене експериментальне дослідження дозволило комплексно оцінити вплив різних стратегій кешування на продуктивність серверної частини веб-застосунку з урахуванням як середніх показників часу відповіді, так і стабільності роботи системи та наявності аномальних затримок.

Отримані результати показують, що ефективність використання кешування визначається передусім характером доступу до даних. У сценарії random workload, який характеризується рівномірним розподілом запитів, кешування не забезпечує суттєвого покращення продуктивності. Усі досліджені стратегії демонструють близькі значення часу відповіді, пропускної здатності та стабільності, що пояснюється низькою ймовірністю повторного використання кешованих даних. У цьому випадку система фактично працює у режимі постійного звернення до бази даних, що нівелює переваги кешування.

У сценарії hot workload, навпаки, спостерігається значне підвищення ефективності кешування. Концентрація запитів на обмеженій множині даних забезпечує високий рівень повторного використання кешованих значень, що дозволяє суттєво зменшити середній та медіанний час відповіді, а також значно підвищити пропускну здатність системи. При цьому найбільший виграш спостерігається для стратегій, які забезпечують швидкий доступ до кешу, зокрема для локального та гібридного кешування.

Разом із покращенням середніх показників продуктивності було виявлено важливу особливість використання кешування – збільшення варіативності часу відповіді. Для стратегій кешування у сценарії hot workload характерним є поєднання двох режимів роботи: швидкої обробки запитів при cache hit та значно повільнішої обробки при cache miss. Це призводить до суттєвого зростання показника стабільності та появи значного розриву між медіанним значенням та значенням 95-го перцентиля.

Особливо помітним є цей ефект для локального кешування та гібридної моделі, де медіанний час відповіді є мінімальним, однак значення p95 залишається на рівні часу доступу до бази даних. Таким чином, при оцінюванні ефективності кешування важливо враховувати не лише середні значення, але й показники стабільності, які відображають передбачуваність роботи системи.

Окремої уваги заслуговує аналіз аномальних затримок. У випадку використання розподіленого кешу Redis було зафіксовано значні latency spikes, що проявляються у суттєвому збільшенні максимального часу відповіді. Це свідчить про можливу наявність додаткових накладних витрат, пов’язаних із мережевою взаємодією, конкуренцією за ресурси або перевантаженням кеш-сервера. У той же час гібридна модель кешування демонструє значно кращі показники у цьому аспекті, що пояснюється зменшенням кількості звернень до розподіленого кешу за рахунок використання локального рівня.

Порівняльний аналіз стратегій кешування показує, що:

– локальне кешування забезпечує мінімальний час доступу до даних, однак має обмеження щодо масштабованості;

– використання Redis дозволяє організувати спільний доступ до кешу, але супроводжується додатковими мережевими затримками та ризиком виникнення аномальних затримок;

– гібридна модель кешування поєднує переваги обох підходів та забезпечує найкращий баланс між продуктивністю, стабільністю та масштабованістю.

Таким чином, результати дослідження демонструють, що використання кешування може суттєво підвищити продуктивність системи у випадках, коли значна частина запитів звертається до популярних даних. При цьому найбільш ефективною є стратегія гібридного кешування, яка поєднує переваги локального кешу та розподіленого кешу, забезпечуючи баланс між швидкістю доступу та масштабованістю

ВИСНОВКИ

У магістерській роботі було проведено комплексне дослідження впливу стратегій кешування даних на продуктивність серверної частини високонавантаженого веб-застосунку, реалізованого на платформі .NET. Основну увагу приділено аналізу ефективності підходів до кешування в умовах змінного характеру доступу до даних та високого рівня конкурентного навантаження.

У межах роботи було проаналізовано сучасні підходи до організації кешування у серверних системах, зокрема локальне, розподілене та багаторівневе кешування. На основі проведеного аналізу було реалізовано експериментальний веб-застосунок, що дозволив дослідити поведінку системи при використанні різних стратегій кешування у контрольованих умовах.

Експериментальна частина дослідження базується на проведенні серії навантажувальних тестів із використанням інструменту k6 для двох принципово різних сценаріїв доступу до даних: рівномірного (random workload) та нерівномірного (hot workload). Такий підхід дозволив не лише оцінити середні показники продуктивності, але й дослідити вплив кешування на стабільність роботи системи та характер розподілу часу відповіді.

Результати дослідження показали, що ефективність кешування є критично залежною від характеру доступу до даних. У випадку рівномірного розподілу запитів використання кешування не забезпечує суттєвого виграшу в продуктивності, оскільки ймовірність повторного використання кешованих даних є низькою. У цьому сценарії система фактично працює у режимі постійного звернення до бази даних незалежно від обраної стратегії кешування.

У випадку нерівномірного доступу до даних (hot workload) кешування демонструє високу ефективність. Використання кешу дозволяє суттєво зменшити середній та медіанний час відповіді, а також значно підвищити пропускну здатність системи. Найкращі результати було досягнуто при використанні гібридної моделі кешування, де середній час відповіді зменшився

більш ніж у чотири рази, а пропускна здатність системи зросла більш ніж у три рази у порівнянні з режимом без кешування.

Важливим результатом роботи є виявлення впливу кешування на стабільність системи. Показано, що поряд зі зменшенням середнього часу відповіді використання кешування призводить до зростання варіативності часу обробки запитів. Це пояснюється наявністю двох режимів роботи системи – швидкого обслуговування запитів із кешу та повільнішої обробки у випадку cache miss. У результаті спостерігається значний розрив між медіанним значенням та високими перцентилями часу відповіді.

Окремо встановлено, що використання розподіленого кешу може супроводжуватися появою аномальних затримок (latency spikes), що проявляються у значному збільшенні максимального часу відповіді. Це пов’язано з додатковими мережевими витратами та можливим перевантаженням кеш-сервера. Водночас використання гібридної моделі кешування дозволяє суттєво зменшити вплив таких затримок за рахунок використання локального рівня кешу.

У результаті дослідження встановлено, що найбільш ефективною є багаторівнева стратегія кешування, яка поєднує високу швидкість доступу до даних у локальному кеші з можливістю масштабування, що забезпечується розподіленим кешем. Такий підхід дозволяє досягти оптимального балансу між продуктивністю, стабільністю та масштабованістю системи.

Практична цінність роботи полягає у розробці експериментальної методики оцінювання ефективності кешування, яка враховує не лише середні показники продуктивності, але й стабільність роботи системи та наявність аномальних затримок.

Дана робота була апробована на міжнародній науково-практичній конференції «Інформаційні технології в культурі, мистецтві, освіті, науці, економіці та бізнесі» в рамках доповіді на тему «Порівняльне дослідження стратегій кешування даних у серверній частині високонавантаженого веб-застосунку на платформі .NET».

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАНЬ

1. Kleppmann M. Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems. Sebastopol: O'Reilly Media, 2017. 616 p.

2. Newman S. Building Microservices: Designing Fine-Grained Systems. 2nd Edition. Sebastopol: O'Reilly Media, 2021. 620 p.

3. Richards M., Ford N. Fundamentals of Software Architecture: An Engineering Approach. Sebastopol: O'Reilly Media, 2020. 432 p.

4. Fowler M. Patterns of Enterprise Application Architecture. Boston: Addison-Wesley, 2003. 533 p.

5. Martin R. C. Clean Architecture: A Craftsman's Guide to Software Structure and Design. Boston: Prentice Hall, 2017. 432 p.

6. Nygard M. Release It! Design and Deploy Production-Ready Software. 2nd Edition. Raleigh: Pragmatic Bookshelf, 2018. 376 p.

7. Microsoft. ASP.NET Core Documentation. Microsoft Learn, 2024. URL: https://learn.microsoft.com/aspnet/core (дата звернення: 11.02.2026).

8. Microsoft. .NET Documentation. Microsoft Learn, 2024. URL: https://learn.microsoft.com/dotnet (дата звернення: 04.03.2026).

9. Microsoft. Performance Best Practices for ASP.NET Core. Microsoft Learn, 2024. URL: https://learn.microsoft.com/aspnet/core/performance (дата звернення: 27.02.2026).

10. Microsoft. IMemoryCache in ASP.NET Core. Microsoft Learn, 2024. URL: https://learn.microsoft.com/aspnet/core/performance/caching/memory (дата звернення: 21.02.2026).

11. Microsoft. Distributed caching in ASP.NET Core. Microsoft Learn, 2024. URL: https://learn.microsoft.com/aspnet/core/performance/caching/distributed (дата звернення: 03.03.2026).

12. Redis Ltd. Redis Documentation. 2024. URL: https://redis.io/docs (дата звернення: 14.02.2026).

13. Carlson J. Redis in Action. Greenwich: Manning Publications, 2013. 321 p.

14. Joshi A. Redis Essentials. Birmingham: Packt Publishing, 2016. 178 p.

15. PostgreSQL Global Development Group. PostgreSQL Documentation. 2024. URL: https://www.postgresql.org/docs (дата звернення: 18.02.2026).

16. Price M. C# 12 and .NET 8 – Modern Cross-Platform Development Fundamentals. Birmingham: Packt Publishing, 2024. 820 p.

17. Albahari J., Albahari B. C# 12 in a Nutshell. 9th Edition. Sebastopol: O'Reilly Media, 2024. 1100 p.

18. Freeman A. Pro ASP.NET Core 7. New York: Apress, 2023. 1100 p.

19. Esposito D. Architecting Applications for the Enterprise. Microsoft Press, 2019. 480 p.

20. Podlipnig S., Böszörmenyi L. A Survey of Web Cache Replacement Strategies // ACM Computing Surveys. 2003. Vol. 35, No. 4. P. 374–398.

21. Amazon Web Services. Caching Strategies and Best Practices. AWS Architecture Center. 2023. URL: https://aws.amazon.com/caching (дата звернення: 25.01.2026).

22. Google Cloud. Best Practices for Application Caching. 2023. URL: https://cloud.google.com/architecture/caching (дата звернення: 09.02.2026).

23. Microsoft. Distributed systems architecture patterns. Microsoft Learn. 2024. URL: https://learn.microsoft.com/azure/architecture/patterns (дата звернення: 16.02.2026).

24. Кієнко Д. В. Порівняльне дослідження стратегій кешування даних у серверній частині високонавантаженого веб-застосунку на платформі .NET: Міжнародна науково-практична конференція «Інформаційні технології в культурі, мистецтві, освіті, науці, економіці та бізнесі». Зб. матеріалів форуму. Т. 1., – Київ: КНУКіМ. 2026.