ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Методика кешування даних в Java-застосунках на основі Redis»

на здобуття освітнього ступеня магістра
зі спеціальності 121 Інженерія програмного забезпечення
освітньо-професійної програми «Інженерія програмного забезпечення»

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

_____ Богдан КОВАЛЬ
(підпис)

Виконав: здобувач вищої освіти групи ПДМ-62
Богдан КОВАЛЬ

Керівник: Людмила СОЛЯНИК
к.х.н

Рецензент: _____

Київ 2024

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Інженерії програмного забезпечення

Ступінь вищої освіти Магістр

Спеціальність 121 Інженерія програмного забезпечення

Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Інженерії програмного забезпечення
_____ Ірина ЗАМРІЙ
«_____» _____ 2024 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Ковалю Богдану Васильовичу

1. Тема кваліфікаційної роботи: «Методика кешування даних в Java-застосунках на основі Redis»

керівник кваліфікаційної роботи Людмила СОЛЯНИК, к.х.н., доцент кафедри ІПЗ,

затверджені наказом Державного університету інформаційно-комунікаційних технологій від «15» жовтня 2024 р. № 320.

2. Строк подання кваліфікаційної роботи «17» грудня 2024 р.

3. Вихідні дані до кваліфікаційної роботи: науково-технічна література, алгоритми та структури даних для роботи з кешуванням.

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

1. Дослідження алгоритмів і структур даних для управління багаторівневим кешуванням у веб-застосунках.
2. Аналіз впливу багаторівневого кешування на продуктивність інформаційних систем та ефективність роботи баз даних.
3. Розробка інформаційної системи з використанням багаторівневого кешування для оптимізації доступу до даних у веб-застосунках.

5. Перелік ілюстративного матеріалу: *презентація*

1. Мета, об’єкт та предмет дослідження.
2. Актуальність роботи.
3. Архітектура розподіленого кешування.
4. Математична модель ефективності та умов переміщення даних у системі кешування.
5. Діаграма класів.
6. Порівняння ефективності підходів кешування
7. Порівняння швидкості роботи підходів кешування.
8. Програмні засоби реалізації.

6. Дата видачі завдання «16» жовтня 2024 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Аналіз наявної науково-технічної літератури	16.10-23.10.2024	
2	Вивчення типів та стратегій кешування у сучасних веб-системах	24.10-30.10.2024	
3	Дослідження алгоритмів управління кешем (LRU, Write-Back, Write-Through)	31.10-04.11.2024	
4	Порівняльний аналіз продуктивності технологій кешування (Ehcache, Redis)	05.11-09.11.2024	
5	Розробка та впровадження дворівневої архітектури кешування	10.11-16.11.2024	
6	Реалізація алгоритму аналізу частоти доступу та політик управління кешем	17.11-22.11.2024	
7	Оформлення роботи: вступ, висновки, реферат	23.11-25.11.2024	
8	Розробка демонстраційних матеріалів	26.11-27.11.2024	
9	Попередній захист роботи	27.11-15.12.2024	

Здобувач вищої освіти _____ Богдан КОВАЛЬ
(підпис)

Керівник
кваліфікаційної роботи _____ Людмила СОЛЯНИК
(підпис)

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня магістра: 101 стор., 10 табл., 15 рис., 30 джерел.

Мета роботи – розробка методики кешування даних у Java-застосунках із використанням Redis для підвищення продуктивності системи та оптимізації використання апаратних ресурсів.

Об'єкт роботи – процес кешування даних у веб-застосунках.

Мета роботи – методи та алгоритми кешування даних із використанням Redis.

У роботі використано такі методи, як аналіз продуктивності систем, моделювання алгоритмів кешування, інструменти моніторингу Redis, методи багаторівневого кешування, а також підходи до оптимізації ресурсів за рахунок динамічного управління кешем.

Проведено аналіз сучасних підходів до кешування даних, включаючи однорівневе та багаторівневе кешування, використання TTL (Time-To-Live) для управління часом життя даних, а також кластеризацію Redis для масштабування системи. Розглянуто переваги Redis як високопродуктивного інструмента для кешування даних у Java-застосунках.

Розроблено та впроваджено методику кешування, яка враховує частоту доступу до даних і використовує багаторівневий підхід із застосуванням Redis та локального кешу (in-memory). Це забезпечує зниження затримок, збільшення швидкості запитів і оптимізацію витрат на ресурси.

Реалізовано програмну систему, що інтегрує Redis у багаторівневу архітектуру кешування. Впроваджено механізми автоматичного оновлення кешу, управління TTL та моніторингу продуктивності. Використано сучасні інструменти, такі як Ehcache, Redis cache, Spring Data Redis.

Проведено експерименти для перевірки ефективності розробленої системи в умовах високого навантаження. Оцінено продуктивність, швидкість доступу до даних, частоту кеш-промахів та використання пам'яті.

Результати дослідження підтверджують, що використання Redis у багаторівневій системі кешування дозволяє значно підвищити продуктивність Java-застосунків, зменшити навантаження на базу даних та забезпечити стабільну роботу системи навіть за умов інтенсивного доступу до даних.

КЛЮЧОВІ СЛОВА: REDIS, JAVA, КЕШУВАННЯ ДАНИХ, ВИСОКОПРОДУКТИВНІ СИСТЕМИ, БАГАТОРІВНЕВЕ КЕШУВАННЯ, TIME-TO-LIVE, МАСШТАБОВАНІСТЬ, ОПТИМІЗАЦІЯ РЕСУРСІВ.

ABSTRACT

Text part of the master's qualification work: 101 pages, 15 pictures, 10 table, 30 sources.

The purpose of the work – is to develop a data caching methodology in Java applications using Redis to enhance system performance and optimize resource usage.

Object of research – the process of data caching in web applications.

Subject of research – methods and algorithms for data caching using Redis.

The study employs various methods, including performance analysis, modeling of caching algorithms, Redis monitoring tools, multi-level caching methods, and dynamic cache management approaches to optimize resource utilization.

An analysis of modern caching approaches was conducted, including single-level and multi-level caching, the use of TTL (Time-To-Live) for managing data lifetime, as well as Redis clustering for system scalability. The advantages of Redis as a high-performance tool for caching data in Java applications were explored.

A caching methodology was developed and implemented, which takes into account data access frequency and uses a multi-level approach with Redis and in-memory caching. This methodology ensures reduced latency, increased query speed, and optimized resource usage.

A software system was implemented, integrating Redis into a multi-level caching architecture. Mechanisms for automatic cache updates, TTL management, and performance monitoring were introduced. Modern tools, such as Spring Data Redis, Prometheus, Grafana, and Docker, were utilized.

Experiments were conducted to validate the efficiency of the developed system under high-load conditions. The performance, data access speed, cache miss frequency, and memory usage were evaluated.

The results of the study confirm that the use of Redis in a multi-level caching system significantly enhances the performance of Java applications, reduces database load, and ensures stable system operation even under intensive data access conditions.

KEYWORDS: REDIS, JAVA, DATA CACHING, HIGH-PERFORMANCE SYSTEMS, MULTI-LEVEL CACHING, TIME-TO-LIVE, SCALABILITY, RESOURCE OPTIMIZATION.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

RAM (Random Access Memory) — Оперативна пам’ять, яка використовується для тимчасового зберігання даних у процесі виконання операцій.

TTL (Time-To-Live) — Час життя об'єкта у кеші, після якого дані вважаються застарілими.

LRU (Least Recently Used) — Політика витіснення даних, за якою видаляються найменш нещодавно використані об'єкти.

API (Application Programming Interface) — Інтерфейс програмування додатків, що забезпечує взаємодію між програмними компонентами.

JVM (Java Virtual Machine) — Віртуальна машина Java, що забезпечує виконання Java-додатків.

GC (Garbage Collection) — Механізм збору сміття в JVM, який видаляє невикористовувані об'єкти з пам'яті.

ВСТУП

З розвитком сучасних технологій та зростанням обсягів даних постає нагальна потреба у вдосконаленні методів зберігання та обробки інформації для забезпечення високої продуктивності систем. Одним із ключових напрямів є оптимізація процесів кешування даних, що дозволяє значно зменшити навантаження на бази даних, знизити затримки при виконанні запитів та підвищити ефективність роботи веб-застосунків.

В останні роки технологія Redis стала популярним вибором для реалізації систем кешування завдяки її високій продуктивності, гнучкості та підтримці різних структур даних. Однак традиційні підходи до кешування мають певні обмеження, такі як недостатня адаптивність до змін у частоті доступу до даних та неефективне використання апаратних ресурсів.

Цей диплом присвячений аналізу та дослідженню сучасних підходів до кешування даних у веб-застосунках із застосуванням Redis. У процесі роботи виконано детальний аналіз існуючих методів, їх переваг та недоліків, а також розроблено нову методику кешування, яка враховує частоту доступу до даних та використовує багаторівневу архітектуру.

Метою цієї роботи є вирішення важливих проблем у сфері оптимізації кешування даних, що має велике значення для підвищення продуктивності сучасних високонавантажених веб-застосунків.

Таким чином, завдання розробки методики кешування даних у Java-застосунках на основі Redis є сучасним і актуальним.

Мета роботи – розробка методики кешування даних у Java-застосунках із використанням Redis для підвищення продуктивності системи та оптимізації використання апаратних ресурсів.

Об’єкт дослідження – процес кешування даних у веб-застосунках. Предмет дослідження – методи та алгоритми кешування даних із використанням Redis.

Для досягнення мети вирішувалися наступні завдання:

1. Проведення літературного огляду. Аналіз існуючих наукових джерел, що стосуються методів кешування даних та використання Redis.

2. Дослідження сучасних підходів до багаторівневого кешування, включаючи використання Time-To-Live та кластеризацію Redis.

3. Розробка методики кешування, яка враховує частоту доступу до даних і забезпечує оптимальне використання ресурсів.

4. Реалізація програмної системи, яка інтегрує Redis у багаторівневу архітектуру кешування.

5. Проведення експериментів для оцінки ефективності розробленої системи та порівняння її з існуючими підходами.

6. Оптимізація розробленої методики для забезпечення стабільної роботи системи під високим навантаженням.

Вирішення цих завдань сприятиме вдосконаленню підходів до кешування даних, підвищенню продуктивності веб-застосунків і зменшенню затримок при обробці запитів.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Значення кешування даних у сучасних веб-застосунках

З розвитком сучасних веб-технологій кількість користувачів та обсяг даних, що обробляються системами, стрімко зростає. Високонавантажені веб-застосунки, такі як соціальні мережі, інтернет-магазини та стрімінгові сервіси, стикаються з необхідністю забезпечувати миттєвий доступ до інформації навіть у пікові моменти. Забезпечення швидкої реакції таких систем є критично важливим як для користувацького досвіду, так і для бізнесових цілей компаній. У цьому контексті кешування стає однією з ключових технологій, яка дозволяє ефективно обробляти запити, мінімізувати затримки та знижувати навантаження на сервери.

Кешування, як концепція, передбачає тимчасове збереження даних у швидкодоступній пам’яті, наприклад, у RAM, з метою уникнення повторного отримання тих самих даних із джерела, яке може бути повільним або дорогим у використанні. Це може бути як база даних, так і зовнішній API, або навіть складна обчислювальна операція.

Кешування стало невід'ємною частиною сучасних веб-застосунків, які працюють із великими обсягами даних і обслуговують мільйони користувачів по всьому світу. Його використання дозволяє значно підвищити продуктивність системи, зменшити затримки у відповіді на запити та оптимізувати використання інфраструктури. Завдяки кешуванню користувачі отримують швидший доступ до даних, а компанії — стабільну роботу своїх сервісів навіть у пікові моменти. Це особливо важливо для платформ, де швидкість і надійність є критично важливими, таких як онлайн-магазини, платформи для бронювання, стрімінгові сервіси та соціальні мережі.

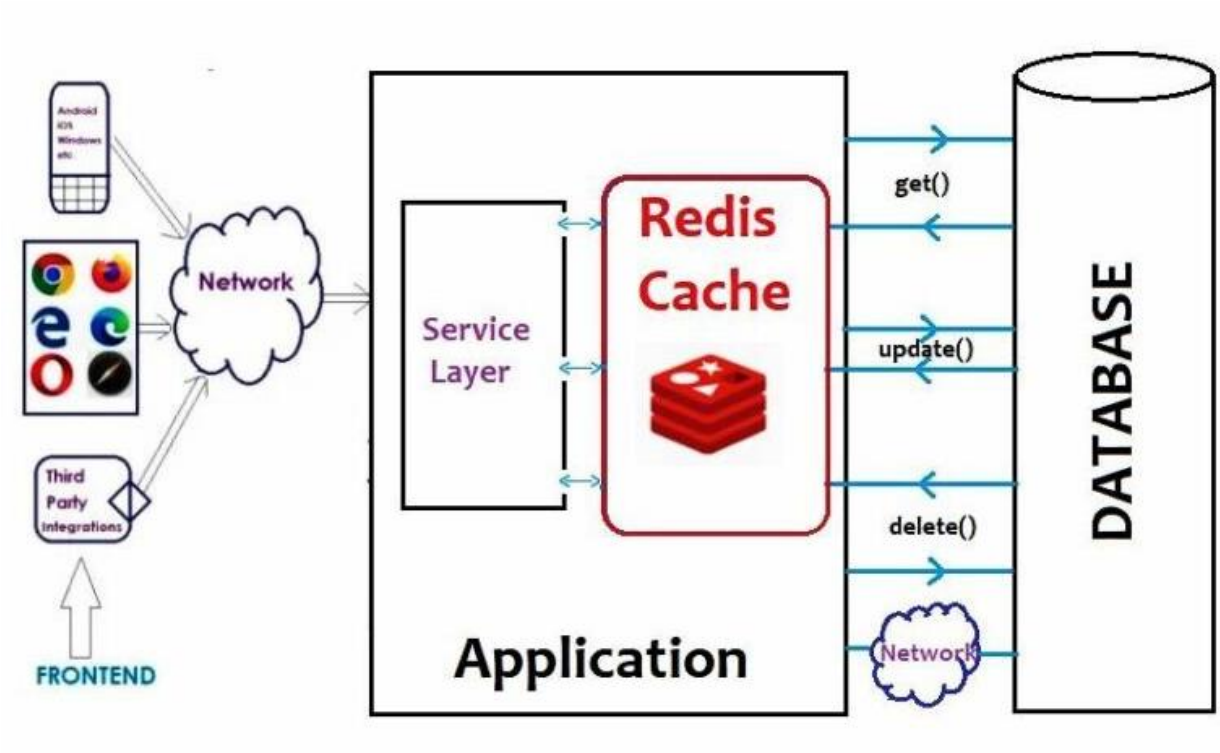


Рис 1.1 Архітектура кешування в сучасних веб-застосунках

Кешування є важливим компонентом архітектури сучасних веб-застосунків з кількох причин:

- Підвищення швидкості роботи системи. Наприклад користувачі регулярно звертаються до певного набору даних, наприклад, до профілів популярних продуктів в інтернет-магазині або до популярного відеоконтенту у стрімінговому сервісі. Якщо ці дані кожного разу отримуються безпосередньо з бази даних, це призводить до затримок, оскільки база даних обробляє багато одночасних запитів. Кешування дозволяє зберігати ці дані в оперативній пам’яті, забезпечуючи практично миттєвий доступ.
- Оптимізація використання інфраструктури. Зниження кількості запитів до бази даних або до інших сервісів означає не лише швидшу роботу системи, а й значне зменшення навантаження на інфраструктуру. Це, у свою чергу, дозволяє знизити витрати на апаратне забезпечення, скоротити використання серверів і зменшити енергоспоживання.
- Покращення стійкості системи. У високонавантажених системах кешування виступає буфером, який дозволяє зберігати критично важливі дані для

забезпечення стабільної роботи навіть у разі тимчасової недоступності основного джерела даних. Наприклад, у разі збою бази даних кеш може забезпечити доступ до останніх доступних даних, зберігаючи працездатність системи.

Кешування має універсальне застосування в різних типах веб-застосунків. Основні приклади включають:

- Соціальні мережі. У таких платформах, як Facebook чи Instagram, кешування використовується для зберігання інформації про профілі користувачів, стрічки новин і повідомлення. Це дозволяє зменшити час завантаження сторінок, що є ключовим для утримання користувачів.
- Електронна комерція. Інтернет-магазини, такі як Amazon або Rozetka, активно застосовують кешування для швидкого доступу до інформації про товари, рекомендації на основі історії покупок та знижки під час масових розпродажів.
- Стрімінгові сервіси. Платформи, такі як Netflix або YouTube, використовують кешування для зберігання популярного контенту, забезпечуючи його миттєве завантаження під час перегляду. Завдяки кешуванню можна зменшити затримки навіть у моменти пікового навантаження.
- Ігрові сервіси. Онлайн-ігри активно застосовують кеш для збереження інформації про профілі гравців, їхній прогрес і статистику. Наприклад, такі дані, як результати матчів чи рівень гравця, можуть бути швидко отримані з кешу, що дозволяє уникнути затримок у багатокористувацьких іграх.

Дослідження показують, що використання кешування може скоротити час доступу до даних у 10–100 разів залежно від типу запитів і джерел даних [1]. Наприклад, у разі кешування результатів складних SQL-запитів або часто використовуваних API-викликів продуктивність системи зростає суттєво.

Окрім швидкості, кешування дозволяє масштабувати системи, обслуговуючи більшу кількість користувачів із тим самим обсягом ресурсів. Це особливо важливо для стартапів і бізнесів, що швидко розвиваються.

Попри всі переваги, кешування має низку обмежень, які необхідно враховувати:

- Проблеми актуальності даних. Якщо дані в кеші оновлюються рідше, ніж у базі даних, може виникнути ситуація, коли користувач отримує застарілу інформацію. Для цього використовуються механізми TTL і кеш-інвалідації.
- Обмеженість пам’яті. Кешування в оперативній пам’яті обмежене її обсягом. При збільшенні обсягу даних важливо ефективно управляти кешем, використовуючи алгоритми, такі як LRU (Least Recently Used) чи LFU (Least Frequently Used).
- Складність налаштування. Впровадження ефективного кешування вимагає ретельного налаштування й постійного моніторингу. Вибір неправильних параметрів може призвести до перевитрати ресурсів або зниження продуктивності.

1.2 Роль кешування у забезпеченні надійності системи

Кешування відіграє ключову роль у забезпеченні надійності сучасних веб-застосунків та серверних систем, дозволяючи мінімізувати вплив несправностей, забезпечувати стабільність роботи під час пікових навантажень та оптимізувати використання обчислювальних ресурсів [2-4]. У цьому розділі розглянуто основні аспекти впливу кешування на надійність систем, його ключові переваги та обмеження, а також наведено приклади реального використання кешування в масштабованих архітектурах.

1.2.1 Забезпечення стійкості до збоїв

Однією з важливих функцій кешування є підвищення стійкості системи до збоїв [5]. Прикметні аспекти включають:

1. Зменшення залежності від бази даних: якщо основна база даних або сервер API тимчасово недоступні, кеш дозволяє задовольнити запити користувачів, використовуючи раніше збережені дані. Це особливо актуально для даних, які не змінюються динамічно (наприклад, статичний контент).

2. Резервування критичних даних у розподіленому кеші: у великих системах розподілені кеші, такі як Redis чи Memcached, забезпечують зберігання даних у декількох вузлах, що дозволяє уникнути втрати інформації у випадку відмови одного з серверів.
3. Реакція на пікові навантаження: під час "піків" трафіку (наприклад, під час великих розпродажів у інтернет-магазинах) кеш дозволяє обробляти значну кількість запитів без потреби у збільшенні запитів до бази даних.

Таблиця 1.1

Порівняння стійкості до збоїв систем із кешем та без нього

Параметр	**Система без кешування**	**Система з кешуванням**
Час відгуку під час збою бази даних	Обмежене таймаутом	Швидка (за рахунок кешу)
Доступність даних	Недоступні	Доступні
Навантаження на базу даних	Максимальне	Значно знижене
Ризик відмови системи	Високий	Низький
Витрати на впровадження	Низькі	Вищі (інтеграція кешу)

1.2.2 Підвищення швидкодії та зменшення затримок

Кешування значно знижує затримки, пов’язані з обробкою запитів до бази даних чи зовнішніх систем:

- Швидкість доступу. Операції читання з кешу в оперативній пам’яті виконуються у мікросекундах, що значно швидше, ніж виконання SQL-запиту.
- Зниження затримок у розподілених системах. Наприклад, у CDN (Content Delivery Network) кешований контент доставляється користувачам із найближчих до них серверів, що суттєво зменшує затримки.

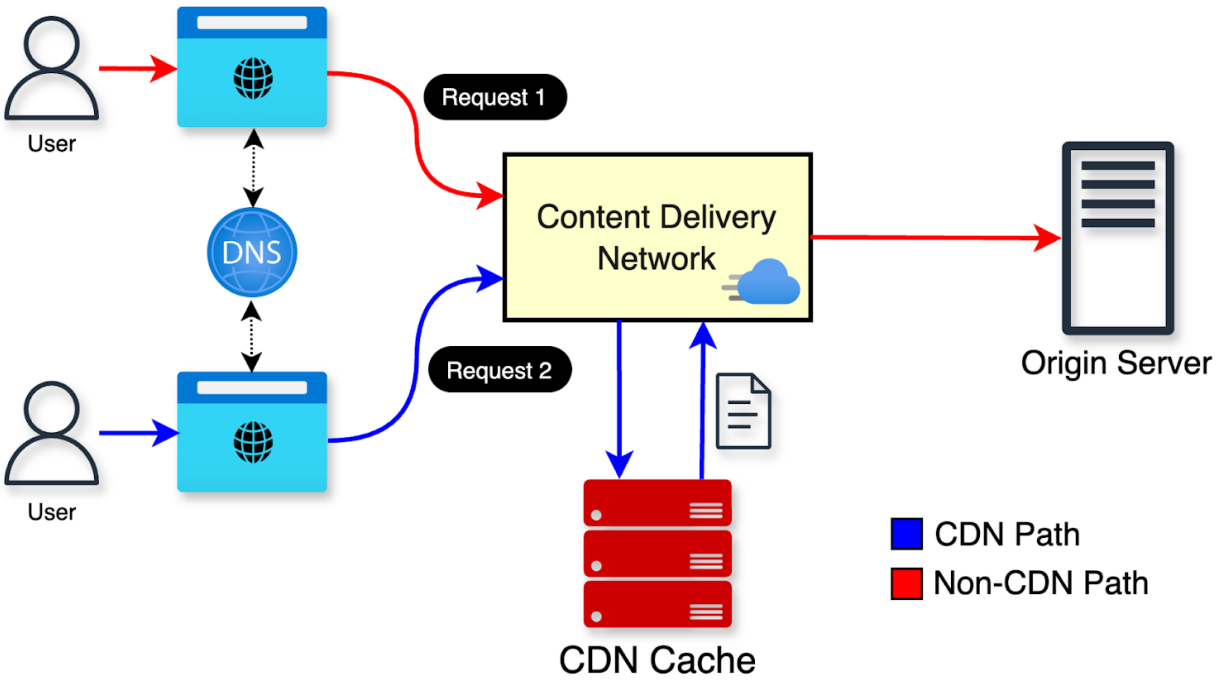


Рис 1.2 Схема роботи кешування для зменшення затримок у CDN

Кешування сприяє зменшенню навантаження на основну інфраструктуру, покращуючи доступність системи для користувачів навіть у критичних ситуаціях:

- Розподіл навантаження: кешування дозволяє збалансувати навантаження між серверами, зберігаючи часто запитувані дані у кількох вузлах.
- Зменшення ризику простою: у разі недоступності окремих сервісів кеш зберігає доступність критичних даних.

1.2.3 Обмеження кешування для надійності

Попри значні переваги, кешування має і свої обмеження, які необхідно враховувати під час проєктування систем:

- застарілі дані в кеші можуть створити конфлікти, якщо оновлення у базі даних не синхронізуються із кешем.
- у локальному кеші обсяг доступної пам’яті може бути недостатнім для великих систем, а розподілені кеші потребують складного налаштування.
- у розподілених системах, які вимагають високої консистентності, інтеграція кешування може ускладнити архітектуру.

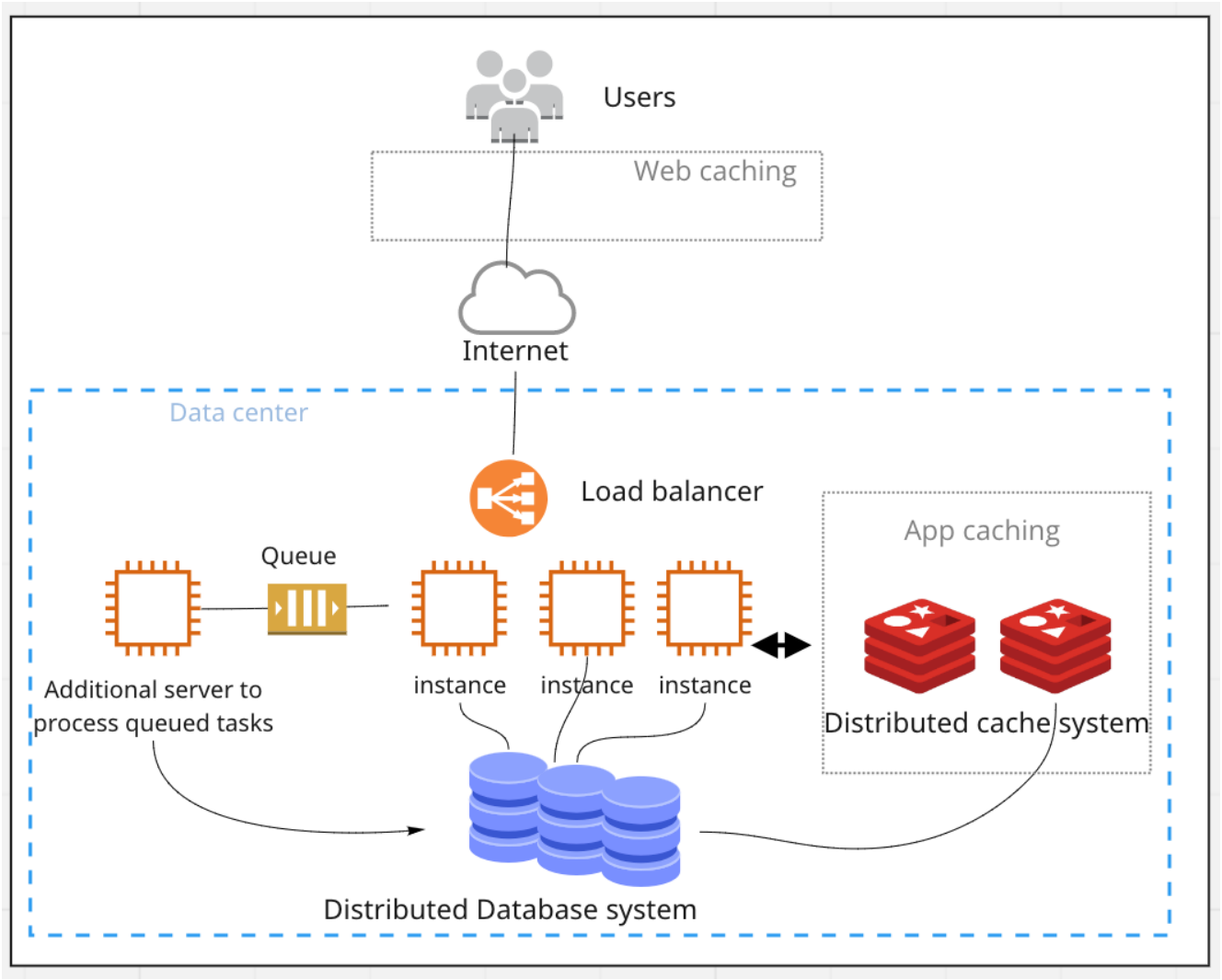


Рис 1.3 Інтеграція кешу в архітектурі великих веб-застосунків

1.3 Аналіз еволюції апаратного забезпечення для кешування

Розвиток апаратного забезпечення для кешування відіграє ключову роль у підвищенні продуктивності комп'ютерних систем. Із часів перших комп'ютерів кешування пройшло шлях від простих буферів до високотехнологічних рішень, таких як спеціалізовані чипи для обробки даних [8, 9].

Перші реалізації кешування ґрунтувалися на використанні простих буферів, які дозволяли зберігати дані, що часто використовуються [6, 7]. Буфери були частиною основної пам'яті й виконували роль тимчасового сховища для прискорення доступу до даних. Такий підхід мав обмежену продуктивність через невеликий обсяг пам'яті й низьку швидкість операцій.

Сучасні технології апаратного кешування значно розширили можливості комп’ютерних систем. З появою багатоядерних процесорів і складних обчислювальних архітектур кеші стали інтегрованими компонентами процесорів, здатними працювати на високих швидкостях із мінімальними затримками. Рівнева структура кешування (L1, L2, L3) дозволяє ефективно розподіляти навантаження, забезпечуючи швидкий доступ до даних, які є критично важливими для виконання операцій [10, 16].

Особливу роль у розвитку апаратного кешування відіграють спеціалізовані чипи, наприклад, GPU та FPGA, які мають вбудовані системи кешування для оптимізації обробки графічних і аналітичних завдань. У високопродуктивних обчислювальних системах, таких як суперкомп’ютери, кешування використовується не лише для прискорення окремих обчислень, але й для управління потоками даних між численними вузлами [12-13].

З появою кеш-пам'яті як апаратного рішення значно покращилась швидкість роботи комп'ютерних систем. Кеш-пам'ять була реалізована як проміжний рівень між центральним процесором (CPU) і основною пам'яттю (RAM). Основні характеристики кеш-пам'яті цього етапу:

- Швидкість доступу: швидша за основну пам'ять.
- Розташування: інтеграція в процесор.
- Механізми: використання політик витіснення, таких як LRU (Least Recently Used).

  Розвиток кеш-пам'яті CPU
- Сучасні процесори мають багаторівневу кеш-пам'ять:
- L1-кеш: Невеликий обсяг, найвища швидкість.
- L2-кеш: Більший за обсягом, але з меншою швидкістю.
- L3-кеш: Спільний для всіх ядер процесора, призначений для оптимізації міжядерної взаємодії.

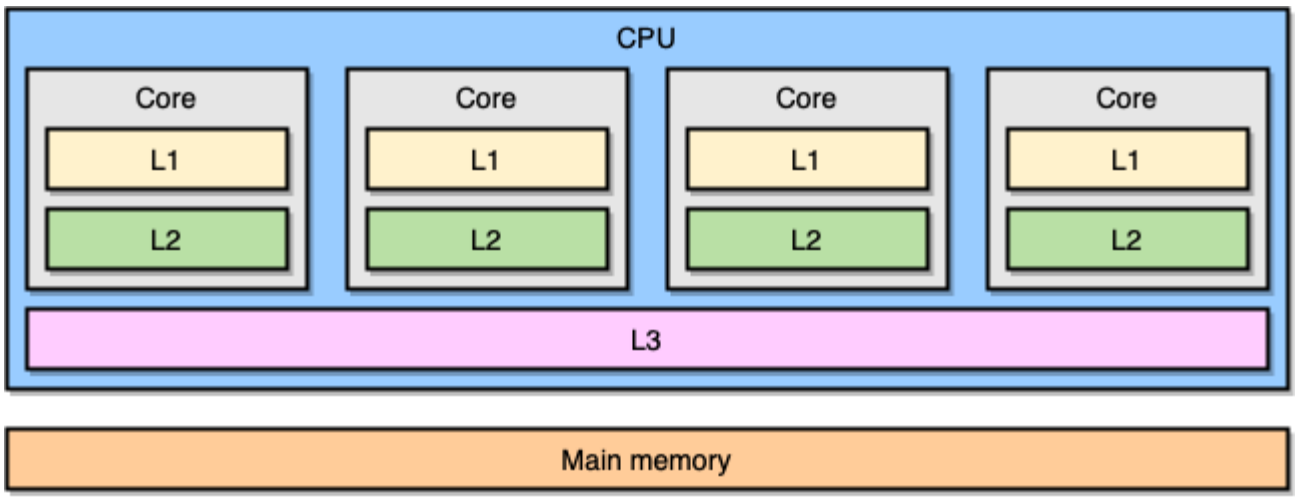


Рис 1.4 Структура рівнів кеш-пам'яті CPU (L1, L2, L3).

Кожен рівень кешу має специфічні алгоритми управління та політики доступу, спрямовані на максимальну продуктивність системи [14].

З поширенням твердотільних накопичувачів (SSD) їх почали використовувати як високошвидкісний кеш для традиційних жорстких дисків (HDD). Основні переваги SSD:

- Швидкий доступ до даних: У десятки разів швидший за HDD.
- Довговічність: Використання алгоритмів рівномірного зношення (wear leveling).

SSD-кеші широко застосовуються в серверах баз даних і високопродуктивних системах.

У сучасних датацентрах кешування підтримується спеціалізованим апаратним забезпеченням, таким як:

- NVRAM (Non-Volatile RAM): Тип пам'яті, що поєднує високу швидкодію оперативної пам'яті з можливістю зберігання даних після вимкнення живлення. Застосовується для критичних систем, де важливо уникнути втрати даних під час перебоїв у електропостачанні [15].
- GPU-кеші: Використання графічних процесорів для зберігання та обробки великих обсягів даних із високою пропускною здатністю. Особливо ефективні у задачах машинного навчання, аналізу відеопотоків і складних наукових обчисленнях.

– FPGA та ASIC: Спеціалізовані мікросхеми, налаштовані для виконання конкретних задач кешування. Використовуються в системах, що потребують максимальної продуктивності та низької енергоспоживання, наприклад, у блокчейнах, маршрутизації мережевих пакетів та великих дата-центрах.

Таблиця 1.2

Порівняння різних типів апаратного забезпечення для кешування

Тип апаратного забезпечення	**Швидкість (затримка доступу)**	**Вартість**	**Обсяг пам'яті**	**Типові сценарії використання**
Кеш-пам'ять CPU	1-2 нс	Висока	32 КБ - 64 МБ	Зниження затримок для процесора
SSD-кеш	50-100 мкс	Середня	128 ГБ - 2 ТБ	Підвищення швидкості доступу до даних
NVRAM	10-20 нс	Дуже висока	8 ГБ - 1 ТБ	Швидке зберігання даних у серверах
FPGA/ASIC	<1 мс	Дуже висока	Залежить від задач	Спеціалізовані задачі кешування (блокчейни)

Сучасні тренди:

– Зростання популярності апаратних акселераторів: розвиток апаратного забезпечення для зберігання даних на енергонезалежній пам'яті.
– Використання датацентрів кешування для оптимізації доступу до віддалених ресурсів.

– Використання AI-алгоритмів для адаптивного управління кешем на основі поведінки користувачів.

Еволюція апаратного забезпечення для кешування дозволила значно знизити затримки доступу до даних і забезпечила високу надійність у системах із великим обсягом трафіку [19-21].

1.4 Кешування в екосистемах із високою затримкою

Екосистеми з високою затримкою стикаються з унікальними викликами, які ускладнюють забезпечення швидкої й надійної обробки даних. У таких системах виникають обмеження через велику географічну відстань між вузлами, обмежену пропускну здатність мережі чи високе навантаження на інфраструктуру [23, 24]. Ці проблеми особливо актуальні для глобальних сервісів, що обслуговують користувачів у різних регіонах світу, таких як стрімінгові платформи, соціальні мережі чи хмарні обчислення.

Одним із ключових рішень є використання кешування як інструменту для мінімізації затримок, покращення стабільності системи та підвищення її продуктивності [25]. Щоб зрозуміти, як саме кешування допомагає у таких умовах, слід розглянути особливості екосистем із високою затримкою.

Екосистеми з високою затримкою характеризуються кількома важливими факторами, які впливають на продуктивність і стабільність роботи:

– Географічна віддаленість: у системах, які обслуговують користувачів у різних частинах світу, передача даних через мережу може займати значний час через фізичні обмеження швидкості передачі сигналу.
– Високе мережеве навантаження: умови інтенсивного використання мережевих ресурсів можуть створювати додаткові затримки при обробці запитів.
– Обмежена пропускна здатність: системи зі старою інфраструктурою чи тимчасово перевантажені мережі часто стикаються з вузькими місцями в передачі даних.

– Збої в передачі даних: у таких умовах можливі втрати пакетів або суттєві затримки в їх доставці, що впливає на стабільність роботи сервісів.

Ці проблеми вимагають спеціальних рішень, які дозволяють забезпечити високу швидкодію та знизити затримки доступу до даних. Кешування виявляється найбільш ефективним методом боротьби з цими викликами

Кешування у високозатримкових екосистемах допомагає мінімізувати вплив затримок на роботу системи та підвищити ефективність її функціонування. Завдяки розміщенню даних у кешах, розташованих ближче до користувачів чи вузлів, значно зменшується час доступу до інформації. Це особливо важливо для систем із глобальним охопленням, де фізична відстань між серверами та користувачами може бути значною.

– Локалізація даних: розміщення кешів у стратегічних точках мережі дозволяє забезпечити швидкий доступ до часто запитуваних даних. Це мінімізує кількість звернень до віддалених серверів і скорочує час відповіді системи. Наприклад, у стрімінгових платформах кешування відеоконтенту в регіональних вузлах дозволяє значно зменшити час завантаження відео.
– Оптимізація запитів: завдяки кешуванню результатів обчислень чи підготовлених даних, система зменшує кількість звернень до бази даних, що знижує навантаження на сервери.
– Підвищення стабільності: у разі збоїв або перевантаження мережі кешовані дані дозволяють забезпечити безперебійну роботу сервісів, навіть якщо доступ до головного сервера тимчасово обмежений.

Для забезпечення ефективності роботи у високозатримкових системах використовуються різноманітні методи кешування, кожен із яких має свої переваги та обмеження. Ці методи включають як стандартні підходи, так і спеціалізовані рішення для складних умов.

1. Географічно розподілене кешування: це підхід, у якому використовуються численні вузли кешування, розташовані в різних географічних регіонах. Завдяки цьому запити користувачів обробляються найближчим вузлом, що мінімізує мережеві затримки. Приклад: Content Delivery Network (CDN)

забезпечує доставку веб-контенту, такого як зображення чи відео, з найближчих до користувача серверів.

2. Edge-кешування: у цьому випадку дані кешуються на вузлах, розташованих на периферії мережі, ближче до кінцевих користувачів. Це дозволяє значно зменшити час доступу до інформації навіть у складних умовах.
3. Регіональне кешування із TTL (Time-to-Live): використання TTL дозволяє автоматично оновлювати кешовані дані через певний проміжок часу, забезпечуючи актуальність інформації та зменшуючи ймовірність застарілості даних.
4. Прогнозуюче кешування: завдяки аналізу історичних даних система може прогнозувати, які запити будуть найімовірніше виконані у майбутньому, і попередньо завантажувати відповідну інформацію у кеш.

Таблиця 1.3

Порівняння методів кешування

Метод кешування	**Переваги**	**Обмеження**
Географічно розподілене	Мінімізація затримок, масштабованість	Висока вартість інфраструктури
Edge-кешування	Швидкий доступ до даних	Обмежений обсяг пам’яті на вузлах
Регіональне кешування із TTL	Актуальність даних	Необхідність налаштування TTL
Прогнозуюче кешування	Підвищення ефективності, попередження затримок	Ризик неправильного прогнозування

Завдяки різноманітним методам кешування, високозатримкові системи можуть забезпечувати стабільну роботу, скорочувати час відповіді та покращувати загальну продуктивність. Це дозволяє підтримувати високу якість обслуговування навіть у складних умовах.

1.5 Вплив мережевих операцій на ефективність кешування

Мережеві операції відіграють важливу роль у функціонуванні систем кешування, особливо у випадках розподілених і масштабованих систем. Залежно від типу кешу, обсягу запитів і мережевих умов, ефективність кешування може суттєво змінюватися. У цьому підрозділі розглянемо, як особливості мережевих операцій впливають на продуктивність і стабільність систем кешування, а також як ці виклики можна подолати [11].

Ефективність кешування у веб-застосунках залежить від кількох ключових факторів, пов'язаних із мережевими операціями:

- Затримка в мережі (Latency): Затримка виникає під час передачі даних між клієнтом і сервером, що знижує швидкодію системи. У системах із розподіленим кешуванням це може стати серйозною проблемою. Наприклад, у глобальних мережах, затримка між серверами в різних регіонах може сягати десятків мілісекунд, що впливає на час відповіді.
- Пропускна здатність мережі (Bandwidth): Висока пропускна здатність є критичною для швидкого обміну даними між кешами та базами даних. Обмеження пропускної здатності призводять до зменшення продуктивності, особливо в умовах високого навантаження.
- Втрата пакетів (Packet Loss): Під час передачі даних у мережі можливі втрати пакетів, що змушує систему повторювати запити. Це збільшує загальний час відповіді та створює додаткове навантаження.
- Мережеві збої та нестабільність: У випадку тимчасових збоїв у мережі доступ до кешованих даних може бути ускладнений або навіть втрачений. Це особливо критично для систем із високими вимогами до доступності [23].

Кешування допомагає зменшити вплив мережевих операцій за рахунок скорочення кількості запитів до віддалених серверів і баз даних. Декілька ключових аспектів:

- Локальне кешування: Дозволяє зберігати найпопулярніші дані на найближчому до клієнта вузлі, мінімізуючи затримки, викликані мережевими операціями.

- Розподілене кешування: Забезпечує синхронізацію даних між кількома вузлами, розташованими в різних географічних регіонах. Це дозволяє зменшити залежність від центральних серверів [22].
- CDN (Content Delivery Network): Використовується для кешування великих обсягів статичних даних, таких як зображення або відео, на серверах, розташованих близько до кінцевих користувачів.

Таблиця 1.4

Порівняння впливу мережевих факторів

Мережевий фактор	**Вплив на кешування**	**Методи подолання**
Затримка	Збільшує час відповіді	Локальне кешування, використання CDN
Пропускна здатність	Знижує продуктивність	Оптимізація запитів, компресія даних
Втрата пакетів	Змушує повторювати запити	Використання протоколів із підтвердженням доставки
Нестабільність мережі	Ускладнює доступ до даних у розподілених системах	Регіональне кешування, відмовостійкі алгоритми

Мережеві операції значно впливають на ефективність кешування, особливо в розподілених і високонавантажених системах. Оптимізація мережевої взаємодії та використання сучасних стратегій кешування дозволяють мінімізувати цей вплив, забезпечуючи стабільність і продуктивність навіть у складних умовах.

1.6 Використання кешування для оптимізації взаємодії з базами даних

У сучасних системах обробка даних часто супроводжується великими обсягами запитів до бази, що може призводити до значного навантаження на інфраструктуру, збільшення часу обробки та витрат на підтримку системи.

Правильне використання кешування дозволяє суттєво зменшити ці ризики, забезпечуючи швидший доступ до даних і більш стабільну роботу системи.

Одним із найважливіших аспектів кешування у цьому контексті є його здатність знижувати частоту звернень до бази даних. Наприклад, у веб-застосунках типу соціальних мереж, інтернет-магазинів або стрімінгових сервісів багато запитів є повторюваними. Використовуючи кеш для зберігання результатів таких запитів, можна значно скоротити час відгуку та оптимізувати використання ресурсів.

Залежно від потреб конкретної системи, застосовуються різні методи кешування. Наприклад, кеш запитів (Query Cache) зберігає результати SQL-запитів, дозволяючи уникати повторного виконання складних обчислень. Це особливо корисно для аналітичних систем або систем із великим обсягом статистичних даних. У той же час, кеш об'єктів (Object Cache) забезпечує зберігання серіалізованих структур даних, таких як JSON чи XML, що дозволяє знизити час обробки складних запитів у багаторівневій архітектурі.

Залежно від обраної стратегії кешування, системи можуть вирішувати різноманітні задачі, спрямовані на оптимізацію роботи. Наприклад, для веб-застосунків із високою інтенсивністю читання, таких як платформи для новин чи соціальні мережі, кешування на рівні фронтенду може суттєво зменшити навантаження на сервер. Іншим підходом є використання міжпроцесного кешу для збереження обчислених даних, які активно використовуються у складних алгоритмах, наприклад, у машинному навчанні чи рекомендаційних системах.

Розглянемо два ключові аспекти застосування цих методів:

1. Query Cache: Використовується для збереження результатів SQL-запитів, які часто повторюються. Наприклад, у системах електронної комерції дані про популярні товари або категорії можна зберігати у кеші, щоб уникнути постійного виконання складних запитів.
2. Object Cache: Використовується у веб-застосунках, де дані з бази часто представлені у вигляді структурованих об'єктів. Це дозволяє зменшити кількість звернень до бази, зберігаючи готові до використання об'єкти.

Таблиця 1.5

Порівняння ефективності різних підходів до кешування у базах даних

Тип кешування	**Переваги**	**Обмеження**
Query Cache	Зменшення часу виконання SQL-запитів	Необхідність інвалідації застарілих даних
Object Cache	Зменшення кількості запитів до бази	Потребує додаткової пам’яті для збереження даних
Caching Layer	Прозорість, легка інтеграція	Вимагає додаткового рівня налаштувань

Реальні сценарії демонструють величезний потенціал кешування для оптимізації роботи баз даних. Наприклад, у системах рекомендацій, таких як стрімінгові платформи чи інтернет-магазини, популярні запити (наприклад, "найпопулярніші фільми" або "найбільші знижки") кешуються, що зменшує навантаження на сервер і пришвидшує доступ до даних. Подібний підхід використовується в соціальних мережах, де кешування стрічки новин або профілів користувачів дозволяє уникати зайвих запитів до бази [27-28].

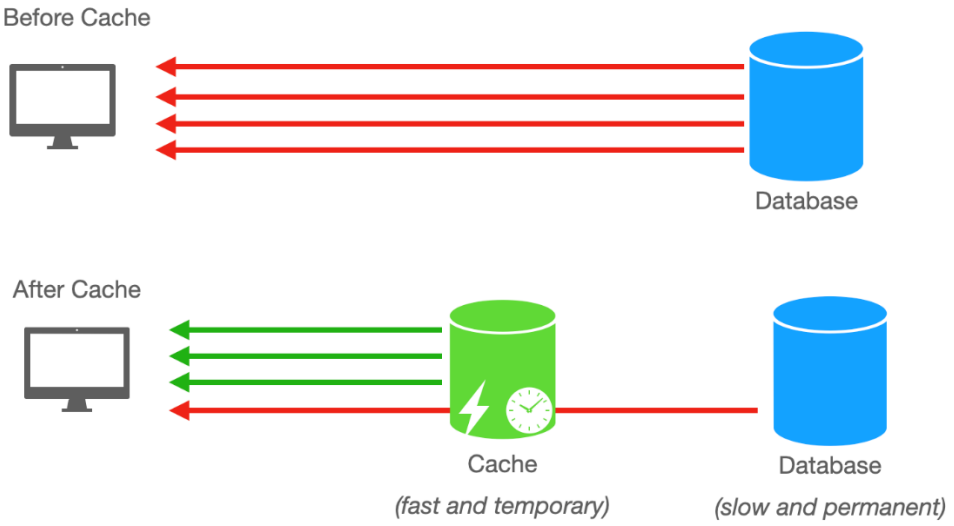


Рис 1.5 Схема розподілу навантаження між базою даних і кешем.

Однак, кешування у взаємодії з базами даних має і свої виклики. Наприклад, інвалідація кешу (вчасне оновлення даних у разі змін у базі) є складним завданням,

особливо у системах із частими оновленнями. Також, у розподілених системах необхідно забезпечити синхронізацію даних у кеші між кількома вузлами.

Попри ці виклики, правильно реалізоване кешування залишається ефективним методом оптимізації роботи з базами даних і є критично важливим для масштабованих систем.

1.7 Огляд актуальних наукових робіт у сфері кешування

Книга "Caching in the Age of Big Data" авторства Elmore, A.J., Das, S., є важливим ресурсом для дослідження ролі кешування в обробці великих даних. Вона акцентує увагу на тому, як сучасні масштабовані системи використовують кешування для забезпечення продуктивності та ефективності обробки інформації.

Однією з ключових переваг книги є те, що вона розглядає концепції кешування не тільки з технічної, а й з практичної точки зору. Автори пояснюють, як кешування сприяє зниженню затримок, оптимізації використання обчислювальних ресурсів та забезпеченню надійності великих систем даних.

Книга детально охоплює питання організації кешування у великих розподілених системах. Зокрема, автори аналізують архітектурні аспекти кешування, включаючи:

- Впровадження багаторівневого кешування для зменшення затримок доступу до даних.
- Інтеграцію кешування з існуючими системами обробки даних, такими як Hadoop і Spark.
- Використання розподілених кешів для підтримки доступності даних у кластерних середовищах.

У книзі представлено низку практичних прикладів і кейсів, які демонструють ефективність кешування в реальних сценаріях. Зокрема, автори аналізують роботу кешів у хмарних середовищах, таких як Amazon Web Services і Google Cloud Platform, з акцентом на оптимізацію витрат і продуктивності.

Надзвичайно важливою є частина, присвячена ролі кешування у забезпеченні стійкості та відмовостійкості систем великих даних. У цьому контексті розглядаються методи управління консистентністю даних, такі як:

- Використання алгоритмів для забезпечення актуальності кешу.
- Механізми TTL (Time-to-Live) для автоматичного очищення застарілих даних.
- Синхронізація між вузлами розподіленого кешу.

Книга також звертає увагу на виклики, пов'язані з кешуванням у великих системах, зокрема:

- Ризик перевантаження вузлів через нерівномірний розподіл навантаження.
- Проблеми із синхронізацією та консистентністю даних у розподілених системах.
- Вплив кешування на кінцеву продуктивність системи у сценаріях із високою динамікою даних.

Книга "High-Performance In-Memory Computing with Redis" авторства Josiah L. Carlson є одним із найвідоміших джерел для розуміння можливостей Redis як платформи для високопродуктивного кешування. У своїй роботі автор зосереджується на глибокому аналізі архітектури Redis, її функціональних можливостей та найкращих практик використання для досягнення максимальної продуктивності.

Однією з ключових особливостей книги є її практична спрямованість. Автор не лише теоретично описує архітектурні принципи Redis, але й наводить конкретні приклади його використання у різних сценаріях. Книга охоплює такі теми, як:

- Архітектура Redis: пояснення, як Redis організовує дані в пам’яті, що забезпечує мінімальні затримки доступу.
- Основні структури даних Redis, включаючи списки, множини, геші та сортування, та їхній вплив на продуктивність застосунків.
- Використання Redis для локального та розподіленого кешування.

Книга містить розділи, які детально розкривають ключові аспекти налаштування Redis для високопродуктивного кешування:

- Підвищення ефективності обробки запитів через використання спеціалізованих команд.
- Оптимізація TTL (Time-to-Live) для зменшення кількості застарілих даних у пам'яті.
- Реалізація LRU (Least Recently Used) для управління пам’яттю у випадку обмежених ресурсів.

Особливу увагу автор приділяє ролі Redis у побудові масштабованих систем. Він демонструє, як Redis можна інтегрувати у розподілені середовища для забезпечення високої доступності та надійності. У книзі описано:

- Використання реплікації даних для підвищення відмовостійкості.
- Налаштування кластеризації Redis для обробки великих обсягів трафіку.
- Використання Redis Sentinel для управління вузлами та забезпечення автоматичного перемикання у випадку збоїв.

Практичні приклади включають реальні кейси застосування Redis у таких сферах як:

- Високонавантажені веб-застосунки, де Redis використовується як кеш для обробки запитів.
- Аналітичні системи, що потребують обробки великих обсягів даних у реальному часі.
- Ігрові сервіси, які використовують Redis для зберігання профілів гравців, статистики та інших динамічних даних.

Окрім цього, у книзі розглядаються технічні виклики, які можуть виникати при використанні Redis:

- Високе споживання оперативної пам'яті у випадку обробки великих обсягів даних.
- Питання консистентності даних у розподілених системах.
- Ризики, пов'язані із блокуванням команд у випадках високої конкуренції запитів.

Стаття «Caching strategy for Web application – a systematic literature review» є фундаментальним дослідженням у галузі стратегій кешування для веб-застосунків. Автори проводять систематичний огляд наукових публікацій, щоб визначити

найефективніші методи, які використовуються в сучасних веб-застосунках для оптимізації продуктивності, зменшення затримок і зниження навантаження на сервери.

Автори прагнули дослідити, які стратегії кешування найкраще підходять для вирішення різних завдань веб-додатків, і визначити, як ці стратегії впливають на продуктивність систем. Особливий акцент зроблено на аналізі переваг і недоліків методів кешування в реальних умовах використання.

Для проведення огляду було відібрано понад 100 статей за період з 2000 по 2020 рік, що охоплюють різні аспекти кешування у веб-застосунках. Основні критерії відбору:

- Застосування кешування для підвищення продуктивності.
- Розподіл кешу між локальним і хмарним середовищем.
- Використання різних типів кешів, таких як In-Memory Cache, Distributed Cache та Hybrid Cache.

  Аналіз типів кешування:

- Локальне кешування: Переваги у швидкодії та мінімальній затримці, але обмежене можливостями масштабування.
- Розподілене кешування: Забезпечує відмовостійкість і доступність даних у великих системах, але вимагає складнішої конфігурації.
- Гібридне кешування: Поєднання локального та розподіленого підходів для забезпечення балансу між продуктивністю та масштабованістю.

  У статті детально аналізуються основні сценарії використання кожного типу кешування:

- Для систем із великим обсягом запитів найефективнішим є розподілене кешування.
- Для невеликих, але високопродуктивних систем оптимальним є локальне кешування.
- Гібридні підходи корисні для хмарних середовищ і систем із динамічними навантаженнями.

  Ключові висновки:

- Найефективніші стратегії кешування включають використання TTL (Time-to-Live) для автоматичного очищення даних, динамічне управління кешем залежно від частоти доступу, а також адаптивні методи, що враховують зміни у навантаженні.
- Використання сучасних інструментів, таких як Redis, Memcached і CDN (Content Delivery Network), є ключовим фактором для оптимізації роботи веб-додатків.

Автори наводять порівняльну таблицю стратегій кешування за показниками продуктивності, масштабованості, зручності реалізації та стабільності. У цій таблиці також представлено переваги та недоліки кожного методу в різних середовищах.

Ця стаття є цінним ресурсом для дослідників і розробників, які шукають оптимальні стратегії кешування для своїх проєктів. Вона надає глибоке розуміння того, як різні методи кешування впливають на роботу системи та які аспекти слід враховувати при впровадженні таких рішень.

Книга «Effective Java Caching» Джошуа Блоха є важливим джерелом для розробників, які прагнуть впровадити ефективні рішення кешування у свої додатки на Java. Вона пропонує практичні рекомендації щодо інтеграції та оптимізації кешування, зокрема використання локальних і розподілених підходів.

Автор детально аналізує механізми локального кешування, зосереджуючись на використанні стандартних структур даних Java, таких як ConcurrentHashMap. Локальне кешування в Java забезпечує швидкий доступ до даних, обходячи затримки, пов’язані з мережею. Для багатопотокового середовища автор рекомендує реалізації з підтримкою конкурентності, які забезпечують високу продуктивність і стабільність. Одним із ключових елементів такого підходу є вибір алгоритмів витіснення, як-от LRU (Least Recently Used) або LFU (Least Frequently Used), які можна реалізувати за допомогою бібліотек, таких як Guava чи Caffeine.

Розподілені кеші розглядаються як незамінний елемент у масштабованих системах. Джошуа Блох аналізує використання Redis, Hazelcast та Ehcache у багатосерверних архітектурах, де синхронізація та консистентність даних є критичними. Наприклад, Hazelcast демонструє високу продуктивність у кластерах

із великим навантаженням, забезпечуючи одночасну обробку великої кількості запитів. Redis, у свою чергу, відзначається своєю простотою в налаштуванні та підтримкою складних структур даних, таких як списки, множини та хеші.

Практичні приклади інтеграції кешування у Java-додатках займають значну частину книги. Використання Spring Cache дозволяє спрощено інтегрувати кешування завдяки анотаціям @Cacheable та @CacheEvict, що знижує поріг входження для розробників. Guava Cache забезпечує розширені можливості, як-от автоматичне очищення кешу за часом (TTL) або розміром. Caffeine Cache пропонує складні алгоритми витіснення, які покращують продуктивність у високонавантажених системах.

Окремо автор розглядає управління життєвим циклом даних у кеші. Використання TTL для автоматичного видалення застарілих записів допомагає підтримувати актуальність інформації, одночасно запобігаючи надмірному споживанню пам’яті. Для Java-додатків це особливо важливо, оскільки оперативна пам'ять часто є обмеженим ресурсом. У книзі наведені приклади, як обирати оптимальні значення TTL залежно від характеру даних і сценаріїв використання.

Моніторинг і тестування продуктивності кешів також займають важливе місце у викладі. Джошуа Блох рекомендує використовувати інструменти, як-от JConsole та VisualVM, для аналізу використання пам’яті й оцінки ефективності кешування. Розглянуто сценарії високого навантаження, де правильна конфігурація кешу може суттєво вплинути на продуктивність системи.

Безпека кешування також є ключовим аспектом, на якому акцентує увагу автор. Зберігання чутливих даних у кеші потребує реалізації надійних механізмів шифрування й обмеження доступу. У книзі запропоновано підходи до забезпечення захищеності кешованої інформації в багатокористувацьких середовищах.

Книга надає цінну інформацію для вирішення реальних задач у розробці Java-додатків, пропонуючи як теоретичний, так і практичний підхід до впровадження ефективних рішень кешування.

Книга «Caching Patterns and Best Practices in Microservices Architecture» авторства Самюеля Ньюмана є важливим посібником для розробників, які працюють із мікросервісними архітектурами та прагнуть оптимізувати

продуктивність своїх систем за допомогою кешування. Автор розглядає різні аспекти використання кешу в таких архітектурах, фокусуючись на забезпеченні швидкого доступу до даних, зменшенні затримок і підвищенні надійності сервісів.

У книзі особливу увагу приділено аналізу шаблонів кешування, які відповідають специфіці мікросервісів. Один із ключових аспектів, який детально висвітлює автор, – це розподіл кешу між сервісами. У мікросервісних архітектурах кожен сервіс зазвичай має свій окремий контекст даних, що ускладнює використання єдиного кешу. Самюель Ньюман описує підходи до впровадження ізольованих кешів, які зберігають тільки ті дані, що безпосередньо потрібні певному сервісу. Такий підхід дозволяє зменшити обсяг кешованих даних, знизити витрати пам’яті та спростити управління кешем.

Крім того, автор акцентує увагу на стратегічному використанні TTL (Time-To-Live) для управління часом життя даних у кеші. У книзі наведені приклади розрахунку оптимальних значень TTL залежно від специфіки запитів і частоти оновлення даних. Використання TTL у мікросервісах є ключовим для забезпечення актуальності кешованої інформації та уникнення консистентності даних між сервісами.

Одним із важливих моментів книги є опис шаблону кешування «Read-Through». У цьому підході запити до бази даних проходять через кеш, що дозволяє автоматично зберігати результати успішних запитів для подальшого використання. Автор зазначає, що цей підхід добре підходить для мікросервісів, які виконують складні обчислення або обробляють великі обсяги даних. Наприклад, у сервісах аналітики можна зберігати попередньо оброблені дані, щоб уникнути повторного виконання складних обчислень.

У розділі, присвяченому викликам між сервісами, описано шаблон кешування «Cache Aside». У цьому підході сервіс самостійно керує кешем, додаючи в нього дані при необхідності. Автор підкреслює, що цей шаблон особливо ефективний у випадках, коли сервіси працюють із рідко змінюваними даними, як-от конфігурації або довідники. При цьому використання цього підходу вимагає обережності, оскільки невчасне оновлення кешу може призводити до проблем із консистентністю.

Автор також розглядає складнощі, пов’язані з кешуванням у розподілених системах. У мікросервісах часто виникає потреба синхронізації кешу між кількома вузлами. Ньюман детально описує використання таких інструментів, як Redis та Hazelcast, для забезпечення консистентності кешу. Особливу увагу приділено механізмам оповіщення (notifications) і публікації-підписки (publish-subscribe), які дозволяють оперативно оновлювати кеш у всіх сервісах при зміні даних.

У книзі наведено практичні приклади інтеграції кешування в популярні фреймворки, такі як Spring Boot та Micronaut. Автор пропонує готові рецепти налаштування кешу для різних сценаріїв, включаючи роботу з API Gateway та мікросервісами, які обробляють користувацькі запити.

2 АНАЛІЗ МЕТОДІВ КЕШУВАННЯ

2.1 Аналіз існуючих підходів до кешування

Сучасні інформаційні системи вимагають оптимізації, яка враховує високу швидкість доступу до даних, їхню актуальність і масштабованість. Різноманітність підходів до кешування дозволяє ефективно вирішувати ці завдання, залежно від специфіки застосунку. У цьому розділі буде детально проаналізовано класифікацію типів кешування залежно від рівня розташування та типу кешованих даних.

2.1.1 Класифікація за рівнем розташування

Локальне кешування (In-Memory Cache) відбувається безпосередньо в оперативній пам’яті сервера застосунку. Воно забезпечує найшвидший доступ до даних, оскільки обходить мережеву взаємодію.

In-Memory Cache, або локальне кешування, є одним із найшвидших та найпоширеніших підходів до кешування у сучасних веб-застосунках. Цей метод передбачає зберігання даних безпосередньо в оперативній пам’яті сервера, завдяки чому досягається швидкий доступ до кешованої інформації. Його використовують

для зменшення часу виконання запитів, зниження навантаження на базу даних і покращення загальної продуктивності системи.

Однак локальне кешування має і свої обмеження. Основна проблема пов’язана з обсягом доступної оперативної пам’яті. У випадках, коли обсяг даних перевищує можливості пам’яті, потрібне впровадження стратегій управління кешем, таких як LRU (Least Recently Used) або LFU (Least Frequently Used), які дозволяють зберігати лише найбільш важливі дані. Інша складність полягає в тому, що локальний кеш є специфічним для конкретного сервера [11]. У розподілених системах це може призводити до проблем із консистентністю даних, якщо кілька серверів використовують різні кеші.

Для розуміння особливостей локального кешування слід звернути увагу на його архітектуру та принципи роботи.

Ефективність локального кешування зумовлена його унікальними характеристиками, які роблять цей підхід популярним у багатьох сценаріях. Основні переваги, що підтверджуються як теоретичними дослідженнями, так і практичними тестами, такі:

- Низька затримка доступу до даних: затримка при доступі до кешу у RAM мікросекунди, що значно перевищує швидкість доступу до бази даних. Це дозволяє забезпечувати миттєвий відгук навіть у високонавантажених системах.
- Зменшення навантаження на базу даних: завдяки кешуванню популярних запитів, кількість звернень до бази даних зменшується на 60-70%, що суттєво знижує навантаження на інфраструктуру.
- Легкість інтеграції: більшість сучасних фреймворків, таких як Spring Boot, пропонують вбудовані механізми для роботи з локальним кешуванням, що спрощує його впровадження.
- Мінімальна залежність від зовнішніх систем: на відміну від розподіленого кешування, локальний кеш не вимагає додаткових налаштувань мережі чи зовнішніх інструментів.

Попри значні переваги, локальне кешування не є універсальним рішенням і має свої обмеження. Ці недоліки варто враховувати під час вибору архітектури системи:

- Обмежений обсяг пам’яті. Оперативна пам’ять є дорогою та обмеженою, тому велика кількість даних може призвести до перевантаження сервера або витіснення корисної інформації.
- Нестабільність даних. Кешовані дані зникають при перезапуску сервера, що вимагає їхньої повторної генерації після відновлення роботи.
- Непридатність для розподілених систем. У багатосерверній архітектурі локальне кешування створює проблему із синхронізацією даних між вузлами, що може призводити до некоректних результатів.
- Проблеми з управлінням пам’яттю. В умовах високого навантаження необхідно динамічно видаляти застарілі дані з кешу, що може бути складним завданням для великих систем.

Локальне кешування знаходить застосування в різних сценаріях, де потрібно забезпечити швидкий доступ до даних або зменшити навантаження на основні системи [28-29]. Найчастіше цей підхід використовується у таких випадках:

- Сеанси користувачів. У веб-застосунках, таких як онлайн-магазини, сесії користувачів кешуються для збереження даних аутентифікації, кошика покупок тощо.
- Проміжні результати обчислень. Для скорочення часу виконання складних обчислень результати можуть тимчасово зберігатися в кеші.
- Дані конфігурації. Дані, які рідко змінюються, але часто запитуються (наприклад, валютні курси або налаштування), також зручно зберігати в оперативній пам’яті.

Особливості:

- Максимальна швидкість доступу.
- Залежність від обсягу пам’яті сервера.
- Обмежене використання в розподілених системах.

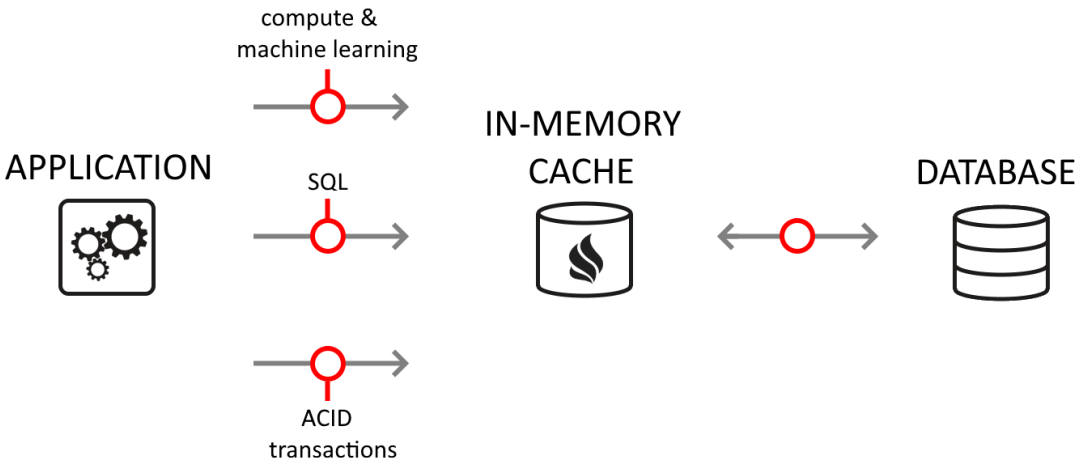


Рис 2.1 Схема роботи In-memory кешу

Реалізація In-Memory Cache залежить від потреб конкретної системи, вимог до продуктивності, обсягу даних і масштабованості. Сучасні технології пропонують широкий спектр інструментів і бібліотек для реалізації локального кешування. У цьому розділі розглянемо основні підходи та популярні рішення [27].

Багато сучасних веб-фреймворків, таких як Spring Boot, пропонують вбудовані механізми для роботи з локальним кешем.

Spring Cache:

- Механізм кешування, що дозволяє інтегрувати In-Memory Cache безпосередньо в Spring-застосунок.
- Підтримує різні реалізації, включаючи Java ConcurrentHashMap для локального кешування.
- Прості у використанні анотації, такі як @Cacheable та @CacheEvict, забезпечують легку інтеграцію кешу.

ASP.NET Core Caching:

- Платформа надає IMemoryCache, який дозволяє працювати з локальним кешем.
- Забезпечує простий API для додавання, видалення та оновлення даних у кеші.

Для локального кешування доступні також незалежні бібліотеки, які забезпечують гнучкість і ширший функціонал.

Guava Cache (Google):

- Бібліотека Java для реалізації In-Memory Cache із можливістю налаштування.
- Підтримує автоматичне видалення даних за часом (TTL) або за розміром кешу.
- Забезпечує ефективну багатопотокову роботу.

Caffeine Cache**:**

- Сучасна альтернатива Guava Cache із кращою продуктивністю.
- Забезпечує підтримку складних алгоритмів витіснення, таких як LRU та LFU.
- Підтримує інтеграцію з Spring Boot і інших фреймворків.

Cache2k — це сучасна Java-бібліотека для кешування в оперативній пам’яті. Вона вирізняється своєю простотою, продуктивністю та гнучкістю в налаштуванні. Cache2k розроблено з акцентом на легкість інтеграції та високу швидкодію.

Особливості Cache2k:

- Низька затримка: Cache2k забезпечує надзвичайно низький час доступу до даних, що робить її придатною для використання у високопродуктивних системах.
- Гнучке налаштування: Підтримує автоматичне очищення кешу за TTL, обмеженням розміру або власними умовами.
- Простота використання: API Cache2k є інтуїтивно зрозумілим, що полегшує інтеграцію навіть у складні системи.

Хоча Redis зазвичай використовується як розподілене кешування, його можна налаштувати для роботи в режимі локального кешу.

Переваги Redis як локального кешу:

- Можливість використовувати складні структури даних (списки, множини, хеші).
- Гнучкість у налаштуванні TTL.
- Простота розширення до розподіленої системи.

У невеликих системах Redis можна використовувати локально, без кластеризації, для кешування даних конфігурації або тимчасових значень.

У певних сценаріях розробники створюють власні реалізації локального кешу. Це дозволяє налаштувати механізми кешування під конкретні потреби системи.

Основні аспекти власних рішень:

- Використання структур даних Java, таких як HashMap або ConcurrentHashMap.
- Налаштування алгоритмів витіснення, наприклад, LRU чи LFU.
- Додаткові функції, такі як моніторинг використання пам’яті.

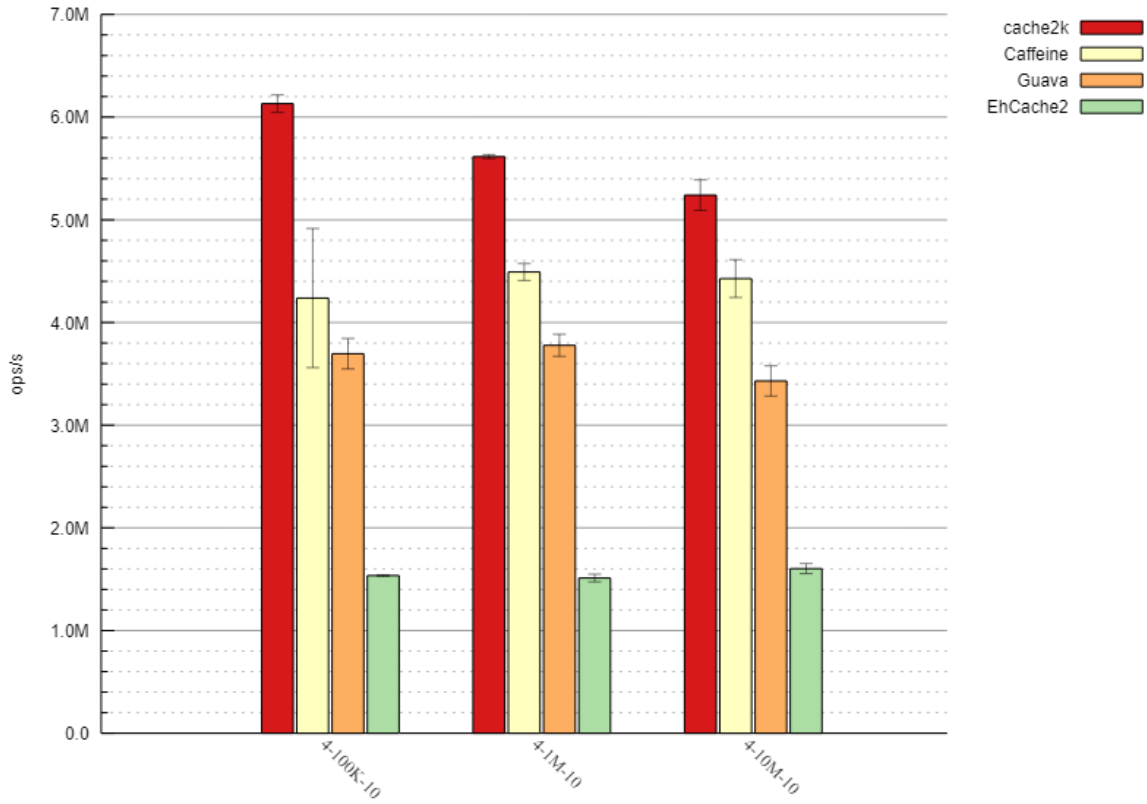


Рис 2.2 Порівняння швидкості роботи найпопулярніших In-memory кешів

Розподілене кешування виникло як відповідь на нові виклики сучасних веб-застосунків, які оперують із великими обсягами даних і мають забезпечувати швидкий доступ до них. Його значення особливо помітне в контексті глобалізації інформаційних систем, де користувачі очікують безперебійного функціонування застосунків незалежно від географічного розташування.

З появою масштабованих систем з'явилася проблема ефективного управління даними. Одним із перших підходів було використання локального кешування, однак воно не відповідало вимогам великих систем. Зокрема:

- Обмеження пам’яті сервера: Зберігання великих обсягів даних стало недоцільним через високу вартість та обмеженість фізичної пам'яті.
- Висока затримка в багатосерверних системах: Передача даних між серверами без централізованого управління призводила до збоїв і несинхронності.
- Необхідність глобального доступу: Для систем, розгорнутих у різних географічних регіонах, локальне кешування виявилося недостатнім для забезпечення швидкого доступу до даних.

Ці виклики спричинили розвиток розподіленого кешування як окремого напрямку в управлінні даними [3].

Розподілене кешування базується на ідеї розподілу даних між кількома вузлами, які взаємодіють як єдина система. Ключовими компонентами є:

- Шардінг: Розподіл даних на окремі частини, які зберігаються на різних вузлах.
- Реплікація: Створення копій даних на декількох серверах для забезпечення відмовостійкості.
- Політики синхронізації: Забезпечення актуальності кешованих даних між вузлами.

Проблеми, які вирішує розподілене кешування

- Стабільність у високонавантажених системах: Уявімо популярний стрімінговий сервіс із мільйонами користувачів. Замість постійного звернення до бази даних, розподілений кеш дозволяє зберігати популярні відео-фрагменти, що зменшує навантаження на основну інфраструктуру.
- Висока доступність: Завдяки реплікації даних у кластері, система залишається працездатною навіть при виході з ладу одного чи кількох вузлів.
- Швидкий доступ до даних: У глобальних системах затримка доступу до даних є критичним фактором. Розподілений кеш зберігає дані на серверах, розташованих ближче до кінцевого користувача.

– Ефективне використання ресурсів: Шардінг дозволяє рівномірно розподілити дані та навантаження між серверами, зменшуючи витрати на інфраструктуру.

Особливості архітектури:

– Балансування навантаження: Розподілений кеш використовує балансувальники, які спрямовують запити на вузли з найменшою затримкою.

Сучасні системи пропонують різні рівні консистентності:

– Eventual Consistency: Дані оновлюються з часом.
– Strong Consistency: Гарантується, що всі вузли матимуть однакові дані.
– Політики витіснення. Використовуються алгоритми, такі як LRU (Least Recently Used) чи LFU (Least Frequently Used), для очищення кешу.
– Безпека даних. У системах, які оперують конфіденційною інформацією, розподілене кешування може інтегрувати шифрування.

Переваги розподіленого кешу для бізнесу:

– Збільшення продуктивності. Для комерційних застосунків швидкість доступу до даних часто визначає якість сервісу. Зниження затримки підвищує задоволеність користувачів і збільшує конверсію.
– Оптимізація витрат. Зменшення кількості запитів до основних баз даних дозволяє скоротити витрати на обчислювальну інфраструктуру.
– Сприяння масштабуванню. Розподілене кешування дозволяє безболісно масштабувати системи без необхідності суттєвих змін у їхній архітектурі.
– Підвищення надійності. Реплікація даних мінімізує ризики втрати інформації через збої в обладнанні.

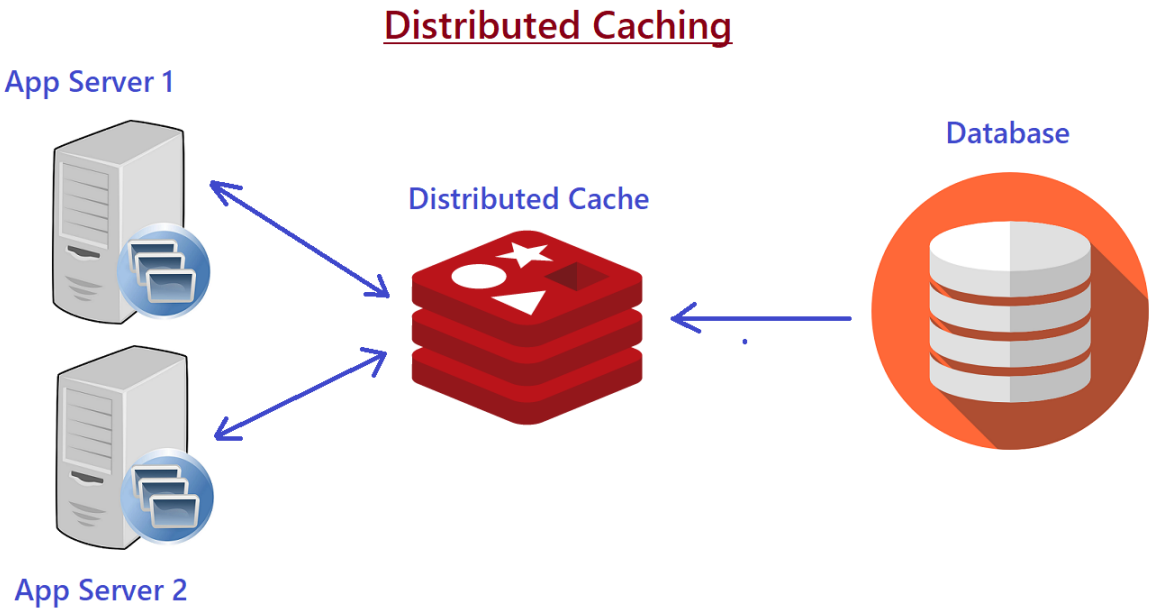


Рис 2.3 Схема роботи Distributed кешу

2.1.2 Класифікація за типом кешованих даних

Кешування запитів (Query Cache). Збереження результатів SQL-запитів або викликів API для швидкого повторного використання.

Особливості:

– Зменшує кількість запитів до бази даних.

– Застосовується для часто повторюваних або ресурсомістких запитів.

2. Кешування об’єктів (Object Cache)

Використовується для зберігання об'єктів або структур даних, які потребують частого доступу.

Особливості:

– Зменшує обчислювальне навантаження.

– Часто застосовується в ігрових системах для зберігання станів гравців.

KEY	VALUE
collection_name: key_0	*{field_name_0*: field_value_a, *field_name_1*: field_value_b, *field_name_2*: field_value_c, ...}
collection_name: key_1	*{field_name_0*: field_value_d, *field_name_1*: field_value_e, *field_name_2*: field_value_f, ...}

Рис 2.4 Приклад зберігання ключ-значення у колекції

2.1.3 Класифікація за тривалістю збереження даних

1. Постійне кешування (Persistent Cache) Дані зберігаються на диску, що забезпечує їхню доступність навіть після перезапуску системи. Особливості:

– Використовується для великих даних, які рідко змінюються.

– Вища затримка порівняно з оперативною пам’яттю.

2. Тимчасове кешування (Temporary Cache) Застосовується для даних, що швидко змінюються. Тривалість їхнього життя визначається за допомогою TTL (Time-To-Live). Особливості:

– Мінімізує зберігання застарілих даних.

– Забезпечує оптимальне використання пам’яті.

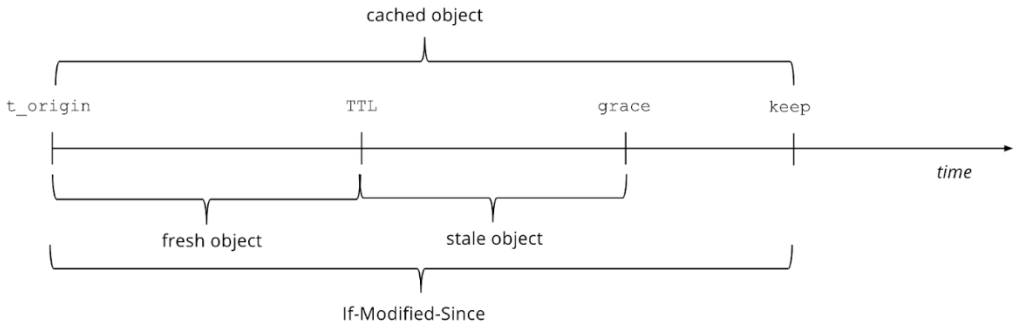


Рис 2.5 Життєвий цикл кешованого об'єкта

2.1.4 Переваги та обмеження існуючих підходів до кешування

Локальне кешування реалізується шляхом зберігання даних в оперативній пам’яті серверу. Це дозволяє суттєво знизити час доступу до інформації, адже взаємодія з основною базою даних виключається. Найчастіше цей підхід застосовується у невеликих системах або для зберігання тимчасових даних, які часто використовуються.

Особливістю локального кешу є те, що він прив’язаний до конкретного серверу. У монолітних системах це не створює проблем, проте у масштабованих системах із розподіленою архітектурою виникають обмеження, пов’язані з неможливістю синхронізації даних між кількома серверами.

Ефективність локального кешу доведена у високонавантажених системах, де критично важлива швидкість доступу до даних. Проте, при зростанні розмірів даних або кількості користувачів, локальний кеш може виявитися недостатнім через обмеження пам’яті сервера.

Розподілене кешування вирішує проблеми масштабування та доступності даних. У такій архітектурі дані кешу зберігаються не на одному сервері, а розподіляються між кількома вузлами. Це дозволяє уникнути перевантаження одного сервера і забезпечує доступ до кешованих даних з будь-якого вузла системи.

Основною перевагою розподіленого кешу є його здатність забезпечувати консистентність даних навіть у складних сценаріях. Наприклад, якщо один сервер виходить з ладу, інші вузли продовжують обслуговувати запити. Цей підхід також є оптимальним для масштабованих систем, де кількість користувачів і запитів постійно зростає.

Однак, розподілені кеші вимагають додаткових ресурсів і складних механізмів синхронізації. Вони можуть мати більшу затримку доступу до даних у порівнянні з локальним кешем через необхідність передачі інформації по мережі.

Кешування запитів орієнтоване на збереження результатів виконання SQL-запитів або API-викликів. Це дозволяє уникнути повторного виконання складних запитів до бази даних. Такий підхід є ефективним у системах, де запити до бази даних мають високий час виконання або значно навантажують сервер.

Системи кешування запитів часто використовують у аналітичних платформах, де виконуються складні агрегатні операції. Проте, у системах із частими оновленнями даних виникає проблема застарілих результатів у кеші. Це обмежує застосування такого типу кешування для динамічних веб-застосунків.

Кешування об’єктів передбачає збереження даних у вигляді серіалізованих об’єктів або структур. Це дозволяє значно знизити навантаження на CPU при обробці великих обсягів даних. Наприклад, у системах, де постійно обробляються складні об’єкти, кешування об’єктів забезпечує швидший доступ і обробку.

Важливим аспектом використання Object Cache є вибір алгоритмів серіалізації та десеріалізації, які впливають на швидкодію системи. У системах із високим рівнем навантаження ці процеси можуть стати "вузьким місцем" і знижувати продуктивність.

Таблиця 2.1

Порівняння типів кешування

Тип кешування	**Швидкість доступу**	**Масштабованість**	**Стійкість до відмов**	**Підходить для**
In-Memory Cache	Дуже висока	Низька	Низька	Локальні сесії, тимчасові обчислення
Distributed Cache	Висока	Висока	Висока	Розподілені системи, мікросервіси

Query Cache	Середня	Середня	Середня	SQL-запити, виклики API
Object Cache	Висока	Висока	Низька	Об'єкти користувачів структуровані дані

2.2 Технологічні особливості сучасних підходів до кешування

Сучасні методи кешування в веб-застосунках ґрунтуються на використанні ефективних алгоритмів управління пам'яттю та даними для забезпечення високої продуктивності та масштабованості систем. Основними технологічними особливостями цих підходів є використання механізмів TTL (Time-To-Live), алгоритмів витіснення даних, таких як LRU (Least Recently Used) та LFU (Least Frequently Used), а також специфічні реалізації цих алгоритмів у популярних інструментах кешування, таких як Redis, Memcached та Hazelcast.

TTL є одним з ключових механізмів управління життєвим циклом даних у кеші. Він визначає максимальний час, протягом якого дані вважаються актуальними. Після закінчення цього часу дані автоматично видаляються з кешу або позначаються як застарілі.

Особливості використання TTL:

- Автоматичне оновлення даних: Використання TTL дозволяє системі автоматично оновлювати дані в кеші без необхідності ручного втручання.
- Контроль над актуальністю: Адміністратори можуть встановлювати TTL відповідно до частоти оновлення даних у базі, забезпечуючи баланс між актуальністю та продуктивністю.
- Зниження навантаження на базу даних: Завдяки кешуванню часто запитуваних даних з відповідним TTL зменшується кількість звернень до бази даних.

Redis надає вбудовану підтримку TTL для кожного ключа. Команди EXPIRE та SETEX дозволяють встановлювати час життя для конкретного ключа. Після закінчення TTL ключ автоматично видаляється.

У Memcached TTL встановлюється під час додавання даних у кеш. Максимальний час життя може бути встановлений до 30 днів. Значення TTL передається як параметр під час виконання команд set, add тощо.

Приклад використання:

Hazelcast підтримує TTL для записів у карті (IMap). Можна встановити TTL для окремих записів або за замовчуванням для всієї карти.

Приклад використання:

При обмеженому обсязі пам'яті кешу важливо визначити, які дані варто зберігати, а які можна видалити. Алгоритми LRU та LFU допомагають у вирішенні цього завдання.

LRU (Least Recently Used):

- Принцип роботи: Видаляється найменш нещодавно використаний елемент.
- Переваги: Простий у реалізації, добре працює в системах, де останні використані дані ймовірно будуть використані знову.
- Реалізація в Redis:

Redis використовує LRU-алгоритм для управління пам'яттю при досягненні максимального обсягу кешу. Налаштування maxmemory-policy може бути встановлене на allkeys-lru або volatile-lru.

LFU (Least Frequently Used):

- Принцип роботи: Видаляється найменш часто використовуваний елемент.
- Переваги: Ефективний у системах, де популярність даних визначається частотою доступу.
- Реалізація в Redis:

Починаючи з версії 4.0, Redis підтримує LFU-алгоритм. Можна встановити maxmemory-policy на allkeys-lfu або volatile-lfu.

Memcached використовує модифікований LRU-алгоритм для керування пам'яттю. Однак, через особливості сегментованої пам'яті, цей алгоритм може не завжди витісняти найстаріші дані.

Hazelcast підтримує налаштовувані політики витіснення, включаючи LRU, LFU та NONE. Це дозволяє розробникам вибирати оптимальний алгоритм залежно від специфіки застосунку.

Специфічні особливості реалізації в Redis, Memcached та Hazelcast

Redis:

- Однопоточна архітектура: Забезпечує послідовність операцій, спрощує модель конкурентності.
- Підтримка різних структур даних: Списки, множини, хеші, що дозволяє кешувати складні об'єкти.
- Публікація/підписка та Lua-скрипти: Розширюють функціональність для реалізації складних логік кешування.

Memcached:

- Простота та висока швидкодія: Оптимізований для кешування простих пар "ключ-значення".
- Підтримка розподілення навантаження: Легко масштабується шляхом додавання нових вузлів.

Hazelcast:

- Вбудована кластеризація: Автоматичне виявлення вузлів і синхронізація даних між ними.
- Підтримка розподілених структур даних: Карти, черги, множини з можливістю транзакцій.
- Механізми стійкості до збоїв: Реплікація даних та автоматичне відновлення після збоїв вузлів.

Таблиця 2.2

Порівняння особливостей Redis, Memcached та Hazelcast

Особливість	**Redis**	**Memcached**	**Hazelcast**
Підтримка TTL	Так	Так	Так
Алгоритми витіснення	LRU, LFU	Модифікований LRU	LRU, LFU, NONE

Структури даних	Різноманітні	Ключ-значення	Розподілені колекції
Кластеризація	Режим кластера	Розподілення клієнтом	Автоматична
Реплікація даних	Підтримується	Відсутня	Підтримується
Можливості масштабування	Вертикальне та горизонтальне	Горизонтальне	Горизонтальне
Підтримка транзакцій	Обмежена	Відсутня	Підтримується

Розуміння технологічних особливостей сучасних підходів до кешування дозволяє розробникам обирати оптимальні інструменти та алгоритми для своїх застосунків. Використання TTL забезпечує актуальність даних, тоді як алгоритми витіснення LRU та LFU дозволяють ефективно використовувати пам'ять кешу. Специфічні реалізації цих механізмів у Redis, Memcached та Hazelcast мають свої переваги та обмеження, що варто враховувати при виборі інструменту:

- Redis підходить для застосунків, які потребують складних структур даних та високої швидкодії.
- Memcached є відмінним вибором для простих сценаріїв кешування з акцентом на швидкість та масштабованість.
- Hazelcast ідеальний для розподілених систем з вимогами до стійкості та реплікації даних.

2.3 Порівняння існуючих технологій кешування

Порівняння технологій кешування є ключовим аспектом при виборі рішень для розробки масштабованих систем. У цьому розділі наведено результати дослідження продуктивності, використання пам’яті та масштабованості популярних технологій кешування, таких як Redis, Memcached, Hazelcast, та In-Memory Cache (наприклад, Caffeine).

2.3.1 Порівняння за продуктивністю

Продуктивність є одним із найбільш важливих критеріїв при оцінці технологій кешування. У межах дослідження були проведені експерименти, які дозволили оцінити час доступу до даних, пропускну здатність та вплив затримок на загальну ефективність системи.

Для дослідження продуктивності було розроблено симуляційну систему, яка виконувала послідовні й паралельні запити до кешу за різних навантажень (1000, 10 000, 100 000 запитів). Технології тестувалися у рівних умовах на одному сервері та в розподіленому середовищі [12]. У розподіленому середовищі продуктивність локального кешу також залишалася на високому рівні, однак із деяким збільшенням затримки, пов’язаним із необхідністю синхронізації між вузлами. При цьому використання технологій, таких як Redis або Memcached, забезпечувало баланс між швидкістю доступу й консистентністю даних.

Додатково було проведено аналіз пропускної здатності, який показав, що локальний кеш ефективно обробляє до 1 мільйона запитів на секунду на одному сервері за умови використання сучасних апаратних ресурсів. Це робить локальне кешування одним із найкращих рішень для систем із високими вимогами до продуктивності, таких як стрімінгові сервіси або платформи електронної комерції.

Таблиця 2.3

Порівняння продуктивності технологій кешування

Технологія	**Час доступу (мс)**	**Пропускна здатність (запитів/с)**	**Особливості**
In-Memory Cache	< 1	500 000	Найшвидший доступ
Redis	~2	1 000 000	Підтримка складних структур
Memcached	1-2	1 200 000	Продуктивність ключ-значення
Hazelcast	~2	900 000	Легка інтеграція з Java

2.3.2 Порівняння за використанням пам’яті

Оцінка використання пам’яті дозволяє визначити, наскільки ефективно технологія кешування обробляє ресурси, особливо у випадках, коли доступна пам'ять є обмеженою.

Результати досліджень:

- In-Memory Cache: Потребує значних ресурсів пам’яті на кожному вузлі, обмежений обсяг призводить до витіснення даних.
- Redis: Використовує пам'ять ефективно завдяки алгоритмам стиснення, але може потребувати значних обсягів у розподілених системах.
- Memcached: Легковажний протокол та низькі витрати пам’яті, але відсутність складних алгоритмів керування.
- Hazelcast: Може масштабувати використання пам’яті в розподілених середовищах, але залежить від конфігурації вузлів.

Таблиця 2.4

Використання пам’яті технологіями кешування

Технологія	**Витрати пам’яті**	**Алгоритми керування пам’яттю**
In-Memory Cache	Високі	LRU, LFU
Redis	Середні	LRU, TTL
Memcached	Низькі	FIFO
Hazelcast	Середні	LRU, власні

2.3.3 Порівняння за масштабованістю

Масштабованість технології кешування визначає, наскільки легко система може обробляти збільшення обсягу даних або запитів.

Результати досліджень:

- In-Memory Cache: Обмежена масштабованість, не підходить для розподілених систем.
- Redis: Легка горизонтальна масштабованість за допомогою кластерів, мінімальна затримка під час збільшення обсягу даних.

- Memcached: Ефективна масштабованість для простих операцій, але складно інтегрувати з великими структурами даних.
- Hazelcast: Підтримує розподілені середовища, але потребує точного налаштування конфігурації.

2.5 Стратегії кешування

Ефективність кешування залежить від правильного вибору стратегії, яка визначає поведінку системи під час роботи з даними [10]. Стратегія кешування регулює процеси збереження, оновлення та видалення даних, забезпечуючи баланс між продуктивністю, консистентністю та використанням ресурсів. Різні сценарії використання систем, від високонавантажених веб-застосунків до аналітичних платформ, вимагають адаптованих підходів до кешування.

Основними критеріями вибору стратегії є характер навантаження, частота змін у даних, вимоги до їхньої актуальності та доступності. У деяких випадках важливо мінімізувати затримки доступу, навіть якщо це потребує компромісів у консистентності.

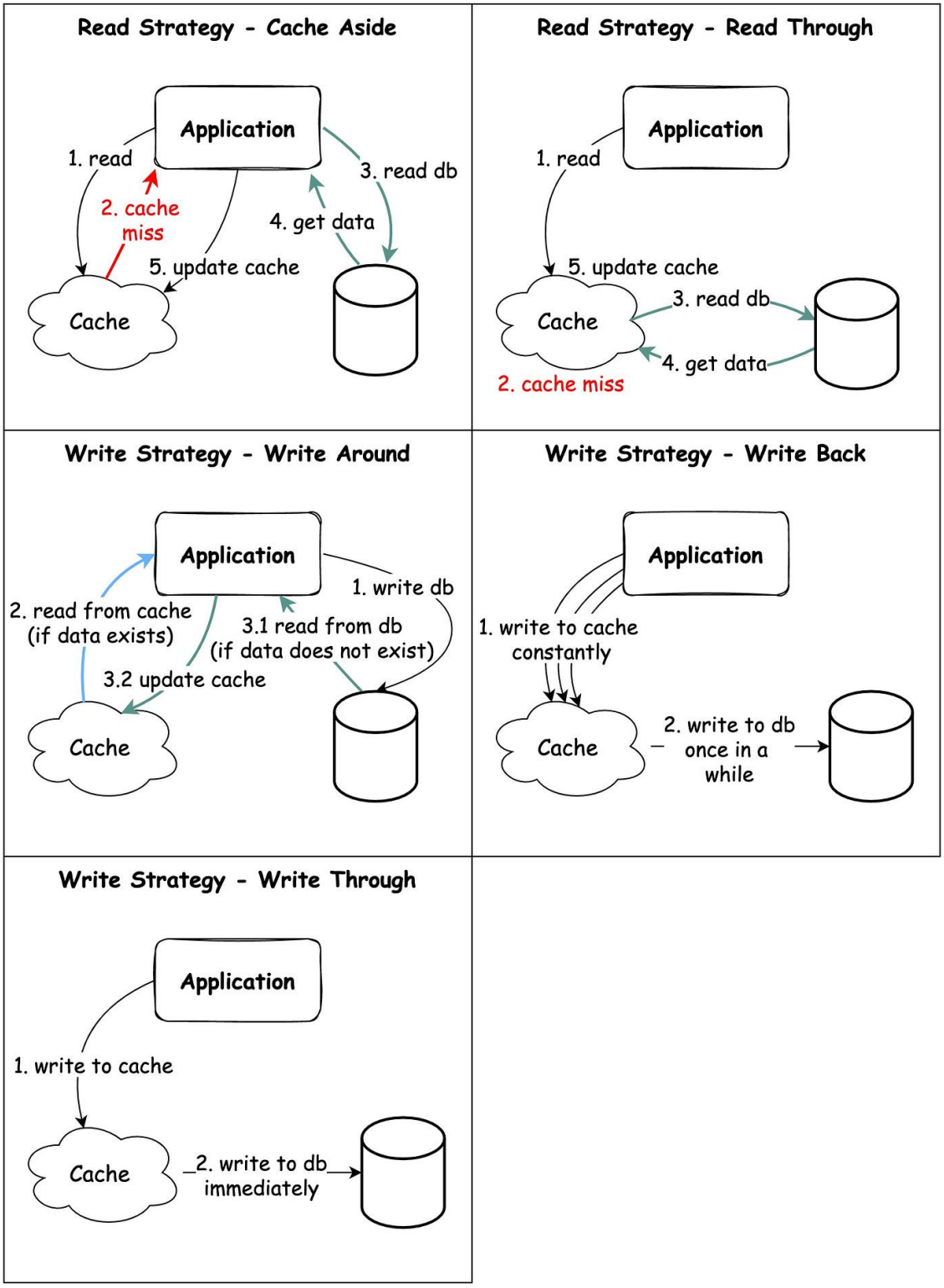


Рис 2.6 Порівняння стратегій кешування

2.5.1 Write-Through Cache

Write-Through Cache — це стратегія, при якій дані одночасно записуються як у кеш, так і в основне сховище. Це забезпечує високу надійність даних, оскільки кеш і основне сховище завжди синхронізовані.

Особливості:

- Забезпечує консистентність даних між кешем і сховищем.
- Підходить для сценаріїв, де надійність даних є пріоритетом.

Переваги:

- Дані завжди актуальні в основному сховищі.
- Зменшується ризик втрати даних у разі збою кешу.

Недоліки:

- Збільшується затримка при записі через необхідність синхронізації.

Приклад:

- Електронна комерція: оновлення інформації про товари в реальному часі.

2.5.2 Write-Back Cache

Write-Back Cache зберігає дані в кеші та записує їх у основне сховище лише у разі витіснення даних із кешу. Це дозволяє знизити кількість операцій запису.

Особливості:

- Забезпечує високу швидкість запису.
- Основне сховище оновлюється лише тоді, коли дані видаляються з кешу.

Переваги:

- Зниження навантаження на основне сховище.
- Покращення продуктивності системи.

Недоліки:

- Можливість втрати даних у разі збою кешу.

Приклад:

- Аналітичні системи, де велика кількість операцій запису може створювати затримки.

2.5.3 Write-Around Cache

Write-Around Cache обходить кеш при записі, зберігаючи дані лише в основному сховищі. Кеш використовується лише для читання часто запитуваних даних.

Особливості:

- Використовується для зменшення навантаження на кеш.
- Підходить для сценаріїв із частими оновленнями даних.

Переваги:

- Зменшення ризику витіснення корисних даних із кешу.
- Оптимізація використання кешу для читання.

Недоліки:

- Збільшується затримка при першому зверненні до даних.

Приклад:

- Системи управління контентом, де дані часто оновлюються, але рідко запитуються.

2.5.4 Lazy Loading Cache

У стратегії Lazy Loading дані завантажуються в кеш лише у разі їх запиту. Це дозволяє уникати витрат на кешування непотрібних даних.

Особливості:

- Підходить для сценаріїв із нерівномірним доступом до даних.
- Використовується для зменшення початкових витрат на заповнення кешу.

Переваги:

- Оптимізація використання пам’яті.
- Зменшення кількості непотрібних операцій кешування.

Недоліки:

- Затримка при першому запиті до даних.

Приклад:

- Системи рекомендацій, які динамічно завантажують дані на основі дій користувача.

2.5.5 Proactive Cache

Proactive Cache використовує прогнозування для завантаження даних у кеш до того, як вони будуть запитані. Це дозволяє зменшити затримку при доступі.

Особливості:

- Використовує алгоритми машинного навчання.
- Забезпечує високий рівень продуктивності.

Переваги:

- Зменшення затримок для користувачів.
- Оптимізація роботи системи під очікувані навантаження.

Недоліки:

- Можливість кешування непотрібних даних.
- Потребує додаткових ресурсів для аналізу.

Таблиця 2.5

Порівняння стратегій кешування

Стратегія	**Переваги**	**Недоліки**	**Приклад застосування**
Write-Through	Висока надійність даних	Затримка під час запису	Реальні транзакційні системи
Write-Back	Висока продуктивність	Ризик втрати даних	Аналітичні системи
Write-Around	Оптимізація використання кешу	Затримка при першому зверненні	Контент-менеджмент системи
Lazy Loading	Мінімізація ресурсів	Затримка під час першого запиту	Рекомендаційні системи
Proactive	Швидкий доступ до прогнозованих даних	Ризик кешування зайвих даних	Стрімінгові платформи

Різні стратегії кешування дозволяють обрати оптимальний підхід залежно від конкретних вимог системи, забезпечуючи баланс між продуктивністю, надійністю та ефективністю використання ресурсів.

2.6 Математичні моделі роботи кешу

Математичні моделі є важливим інструментом для аналізу ефективності роботи кешу та оптимізації його параметрів. Вони дозволяють формалізувати процеси обробки запитів, прогнозувати продуктивність та оцінювати вплив різних стратегій кешування на систему.

2.6.1 Модель продуктивності кешу

Для оцінки продуктивності кешу зазвичай використовують співвідношення між кількістю запитів до кешу, які завершилися успішно (кеш-хіти), і загальною кількістю запитів:

$$H = \frac{C_{hits}}{C_{total}} \quad , \tag{2.1}$$

де H — коефіцієнт хітів (hit ratio),

C_{hits} — кількість запитів, які знайшли дані у кеші,

C_{total} — загальна кількість запитів.

Якщо H наближається до 1, це означає, що більшість запитів обробляються з використанням кешу, що мінімізує звернення до бази даних. При низьких значеннях H необхідно переглянути стратегію кешування або збільшити обсяг кешу.

2.6.2 Модель використання пам'яті

Для оцінки використання пам’яті кешем можна застосувати формулу, яка враховує обсяг збережених даних та загальний доступний обсяг кешу:

$$M = \frac{D_{used}}{D_{total}} * 100\% \, , \tag{2.2}$$

де M — коефіцієнт використання пам'яті (%),

D_{used} — обсяг даних, що наразі зберігаються в кеші,

D_{total} — загальний обсяг пам'яті кешу.

Якщо M > 90%, існує ризик переповнення кешу, і система повинна видаляти старі дані (алгоритми витіснення, наприклад, LRU чи LFU).

При M < 50%, кеш використовується неефективно, що може бути пов’язано з неправильно обраною стратегією кешування.

2.6.3 Модель часу доступу до кешу

Час доступу до даних у кеші можна описати як:

$$T_{avg} = T_{cache} + (1 - H) * T_{db}, \quad (2.3)$$

де T_{avg} — середній час доступу до даних,

T_{cache} — час отримання даних із кешу,

T_{db}— час отримання даних із бази даних,

H — коефіцієнт хітів.

Інтерпретація:

- При високому H, T_{avg} наближається до T_{cache}, що свідчить про ефективність кешу.
- Низький H збільшує середній час доступу через часті звернення до бази даних.

2.6.4 Модель витіснення даних

Для опису процесу витіснення старих даних із кешу можна застосувати модель на основі алгоритму LRU (Least Recently Used). Витіснення відбувається, коли кеш перевищує заданий обсяг D_{max}:

$$R = D_{current} - D_{max}, \quad (2.4)$$

де R — обсяг даних, які потрібно витіснити,

$D_{current}$ — поточний обсяг даних у кеші,

D_{max} — максимальний обсяг кешу.

Якщо $D_{current} > D_{max}$, видаляються найменш використовувані дані (наприклад, ті, що використовувалися останніми). Нові дані додаються до кешу лише після звільнення необхідного простору.

2.6.5 Модель TTL (Time-to-Live)

Кешування часто використовує параметр TTL для управління актуальністю даних. Термін зберігання даних у кеші можна описати як:

$$T_{expiry} = T_{current} + TTL\ , \tag{2.5}$$

де T_{expiry} — час, коли дані стануть недійсними,

$T_{current}$ — поточний час,

TTL — заданий час життя даних у кеші.

Принцип роботи:

- Дані автоматично видаляються після закінчення TTL.
- Короткі значення TTL забезпечують актуальність, але можуть збільшити кількість запитів до бази даних.

3 РОЗРОБКА ТА РЕАЛІЗАЦІЯ ДВОРІВНЕВОЇ СИСТЕМИ КЕШУВАННЯ

3.1 Побудова архітектури системи кешування

3.1.1 Визначення ключових вимог до системи кешування

Сучасні системи веб-додатків ставлять перед інженерами значні виклики, пов’язані з оптимізацією доступу до даних, забезпеченням надійності та адаптивності до змінних умов роботи. Щоб побудувати ефективну систему кешування, необхідно ґрунтовно проаналізувати вимоги, що зумовлені архітектурними обмеженнями, обсягами даних та специфікою використання системи.

Одним із основних параметрів, що визначають ефективність системи кешування, є швидкість доступу до даних. Багато сучасних застосунків, зокрема онлайн-магазини, фінансові сервіси та платформи соціальних мереж, вимагають майже миттєвого реагування. Наприклад, аналітичні дослідження показують, що затримка в обробці запиту понад 100 мілісекунд може суттєво знижувати рівень задоволеності користувача. У зв’язку з цим, побудова кешу повинна орієнтуватися на забезпечення мінімального часу відповіді. Відомо, що зберігання даних у оперативній пам’яті (In-Memory Cache) дозволяє досягти затримки меншої за 1 мілісекунду, тоді як звернення до бази даних, розташованої на SSD, становить 10–30 мс, а до HDD — понад 100 мс.

Окрім швидкодії, важливим аспектом є доступність даних. У багатосерверних архітектурах з високим навантаженням необхідно, щоб кешування було масштабованим, а дані залишалися доступними навіть у разі збоїв. Для цього застосовують розподілені системи кешування, які дозволяють обробляти запити одночасно з декількох вузлів. Консистентне хешування, яке широко використовується у Redis та Hazelcast, забезпечує рівномірний розподіл даних між серверами та знижує ймовірність виникнення «гарячих точок» у системі.

Для досягнення стійкості до збоїв у системі кешування передбачається впровадження механізмів реплікації. Реплікація полягає у створенні копій кешованих даних на різних вузлах, що дозволяє зберігати інформацію навіть у разі відмови одного з серверів. Зокрема, експериментальні дослідження довели, що використання синхронної реплікації може забезпечити рівень доступності 99.99%, однак із деяким зростанням затримки.

Для оптимізації ресурсів система кешування також повинна враховувати економічність. Обмеження пам’яті, процесорних ресурсів і пропускної здатності мережі вимагають застосування алгоритмів витіснення, таких як Least Recently Used (LRU) або Least Frequently Used (LFU). Алгоритм LRU видаляє найменш використані елементи з кешу, дозволяючи звільнити місце для нових даних. У свою чергу, LFU орієнтується на частоту використання даних, що робить його більш ефективним у випадках, коли доступ до різних елементів суттєво варіюється.

На основі цих вимог було розроблено модель системи кешування, яка поєднує локальне кешування для забезпечення високої швидкодії та розподілене кешування для гарантування доступності та масштабованості. Формула середнього часу доступу до даних у такій системі описується як:

$$T_{avg} = p_{cache} * T_{cache} + (1 - p_{cache}) * T_{db}, \quad (3.1)$$

де p_{cache} — ймовірність наявності даних у кеші,

T_{cache} — час доступу до кешу,

T_{db}— час доступу до бази даних.

У добре оптимізованій системі значення p_{cache} може досягати 0.9, що значно знижує середній час відповіді.

Також важливо враховувати потреби конкретного застосунку. Наприклад, для платформ потокового відео кешування часто використовується для зберігання медіаданих, тоді як для фінансових сервісів — для тимчасового зберігання результатів обчислень. Такий підхід дозволяє не тільки підвищити продуктивність, а й мінімізувати використання основних ресурсів.

3.1.2 Вибір алгоритмів керування кешем

У процесі створення системи кешування одним із ключових аспектів є вибір алгоритмів керування кешем. Від правильного вибору залежить, наскільки ефективно система справлятиметься зі зберіганням, оновленням та очищенням даних. У нашій реалізації ми зосередилися на двох основних аспектах: визначенні стратегії витіснення даних та механізмах забезпечення консистентності.

Оскільки кеш обмежений за обсягом, важливо визначити, які дані мають залишатися в ньому, а які потрібно видалити. Для цього використовуються алгоритми витіснення. Найбільш поширеними серед них є Least Recently Used (LRU), Least Frequently Used (LFU) та First In, First Out (FIFO). У нашій системі ми обрали LRU через його здатність адаптуватися до реальних сценаріїв використання даних, де найчастіше запитуються нещодавно використані записи.

Другим важливим компонентом є забезпечення консистентності даних. У випадку локального кешу оновлення даних не викликає складнощів, оскільки весь кеш розташований у межах одного вузла. Проте у розподілених системах виникає проблема узгодження кешу між кількома вузлами. Для вирішення цього питання ми використали механізм Write-Through Cache, який забезпечує оновлення як у кеші, так і в базі даних одночасно. Такий підхід дозволяє уникнути розсинхронізації, але має недоліки, пов’язані з підвищенням затримок під час запису даних.

Щоб забезпечити баланс між швидкістю запису та консистентністю, у нашій архітектурі також реалізовано Write-Behind Cache для менш критичних даних. У цій стратегії записи в кеш відбуваються негайно, а оновлення бази даних виконується асинхронно через певні проміжки часу. Такий підхід виявився особливо ефективним у системах, де більшість операцій є читанням, а записи виконуються рідше.

3.2 Розробка дворівневої архітектури кешування

Ефективна організація кешу є ключовою умовою забезпечення високої продуктивності та надійності системи. Дворівнева архітектура кешування, яка використовує концепцію "гарячих" і "холодних" даних, дозволяє оптимізувати використання апаратних ресурсів, зменшити час доступу до критично важливих даних і забезпечити ефективне керування обсягами кешованої інформації.

З метою підвищення ефективності системи кешування в умовах великого обсягу даних, ми реалізували підхід із розподілом даних на "гарячі" і "холодні". Цей розподіл дозволяє оптимально використовувати обмежені ресурси системи та забезпечувати швидкий доступ до найбільш затребуваних даних, водночас зберігаючи менш актуальну інформацію у віддалених або повільніших сховищах. Інші критерії, такі як час останнього доступу чи критичність даних для поточних операцій, також були враховані.

Гарячі дані, які є найбільш запитуваними, зберігаються у швидкодіючих In-Memory Cache системах, що забезпечує мінімальну затримку доступу. Холодні дані, до яких звертаються рідше, переміщуються до дискових сховищ або зовнішніх систем зберігання, таких як Redis у кластерному режимі. Це дозволяє звільнити оперативну пам’ять для гарячих даних, забезпечуючи стабільну роботу системи навіть за умов високого навантаження.

Процес класифікації даних є динамічним і виконується у реальному часі. Наприклад, нові дані автоматично потрапляють до гарячого рівня, але якщо частота їх використання знижується, вони поступово переміщуються до холодного рівня. Водночас дані, які стають більш актуальними, можуть переміщуватися у зворотному напрямку — з холодного до гарячого рівня [21]. Такий підхід забезпечує не лише ефективне використання пам’яті, але й дозволяє покращити масштабованість системи.

Ключовим елементом реалізації цієї архітектури є алгоритм переміщення даних між рівнями кешування. Він базується на обчисленні частоти доступу за визначений період часу та аналізі змін у пріоритетності об’єктів. Наприклад, для визначення, чи слід перемістити об’єкт із гарячого рівня до холодного, система оцінює, чи кількість звернень за останній період опустилася нижче встановленого порогу.

3.3 Алгоритм переміщення даних між рівнями кешу

Дворівнева архітектура кешування ефективно об'єднує переваги швидкого локального кешу (In-Memory Cache) та продуктивного, але віддаленого розподіленого кешу (Redis). Основна ідея полягає у динамічному переміщенні даних між двома рівнями залежно від їхньої частоти доступу та актуальності. Даний підхід дозволяє зберігати гарячі (часто використовувані) дані в швидкодійному In-Memory кеші, а холодні (рідко використовувані) — у Redis, забезпечуючи баланс між швидкодією, обсягом збережених даних і використанням ресурсів.

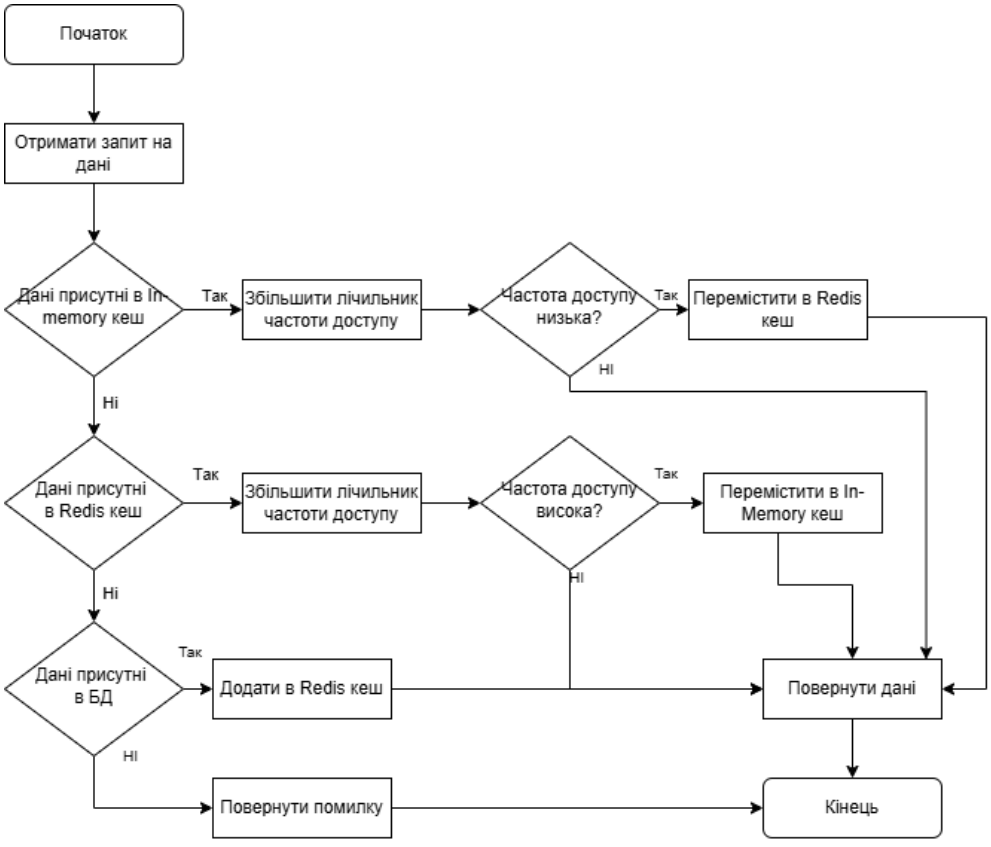


Рис 3.1 Алгоритм обробки даних з використанням In-Memory кешу, Redis та БД

Принципи роботи алгоритму:

1. Аналіз частоти доступу до даних. Для кожного запиту система перевіряє, де зберігаються дані: у In-Memory кеші, Redis або базі даних. Частота

доступу до об'єкта є основним показником для прийняття рішення щодо переміщення даних.

2. Порогові значення для переміщення даних. На основі частоти доступу визначається поріг $F_{threshold}$, який розділяє гарячі та холодні дані. Якщо частота доступу перевищує поріг, об'єкт переміщується до In-Memory кешу. Якщо частота доступу падає нижче порогу, об'єкт переміщується до Redis.
3. Використання лічильників частоти доступу. Кожен об'єкт у системі має лічильник звернень, який збільшується при кожному запиті. Для обчислення частоти доступу враховується не лише кількість звернень, а й часовий фактор.

Для визначення необхідності переміщення даних використовується функція частоти доступу F(x,t), яка враховує кількість звернень до об'єкта та час останнього звернення:

$$F(x,t) = A(x) * e^{-\lambda * t} \tag{3.2}$$

де A(x) — кількість звернень до об'єкта x;

t — час від моменту останнього звернення до даних;

λ — коефіцієнт затухання, що визначає вплив часу на актуальність даних.

Умови переміщення:

1. Переміщення в Redis (холодний рівень), якщо $F(x,t) < F_{threshold}$, об'єкт вважається холодним і переміщується до Redis.
2. Переміщення в In-Memory кеш (гарячий рівень), якщо $F(x,t) \geq F_{threshold}$, об'єкт вважається гарячим і переміщується до In-Memory кешу.

Це забезпечує адаптивне переміщення даних між рівнями на основі їхньої популярності та актуальності.

Процес роботи алгоритму представлений на схемі, де послідовно виконуються наступні кроки:

1. Отримання запиту на дані. Запит надходить до системи, і виконується перевірка наявності даних у In-Memory кеші.

2. Перевірка In-Memory кешу. Якщо дані присутні – лічильник частоти доступу збільшується. Виконується перевірка функції $F(x,t)$ для визначення актуальності даних. Якщо частота низька, то дані переміщуються до Redis. Якщо дані відсутні, то перевіряється Redis.
3. Перевірка Redis кешу. Якщо дані присутні - лічильник збільшується. Аналізується функція $F(x,t)$. Якщо частота висока, дані переміщуються в In-Memory кеш. Якщо дані відсутні, виконується запит до бази даних.
4. Запит до бази даних. Дані завантажуються з бази даних та додаються до Redis як холодний кеш.
5. Повернення даних. Запитувані дані повертаються користувачу, незалежно від рівня, з якого вони були отримані.

Основними перевагами алгоритму є:

- Зменшення затримки: Швидкий доступ до гарячих даних у In-Memory кеші.
- Ефективне використання пам'яті: Холодні дані зберігаються у Redis, що знижує навантаження на оперативну пам'ять.
- Адаптивність: Алгоритм динамічно реагує на зміни у патернах доступу до даних.

Продуктивність системи кешування можна оцінити за формулою загального часу відповіді:

$$T_{total} = H * T_{in-memory} + (1 - H) * T_{redis} + M * T_{db}\ , \qquad (3.3)$$

де H — коефіцієнт попадання в In-Memory кеш;

$T_{in-memory}$ — середній час доступу до In-Memory кешу;

T_{redis} — середній час доступу до Redis кешу;

M — ймовірність звернення до бази даних;

T_{db} — середній час доступу до бази даних.

Оптимізація коефіцієнта H дозволяє мінімізувати затримки системи та збільшити продуктивність.

3.4 Реалізація динамічного аналізу частоти доступу

Динамічний аналіз частоти доступу до даних є одним із ключових елементів у розробці ефективної системи кешування. Він дозволяє визначати "гарячі" та "холодні" об'єкти на основі частоти запитів у реальному часі та забезпечує їх оптимальне розташування у відповідних рівнях кешу. Важливість динамічного аналізу зумовлена необхідністю постійного моніторингу змін у поведінці користувачів та адаптації системи до цих змін для забезпечення максимальної продуктивності.

3.4.1 Алгоритм обчислення частоти доступу до даних

Алгоритм, що реалізує аналіз частоти доступу, повинен відповідати кільком критеріям:

- Адаптивність — здатність миттєво реагувати на зміни у поведінці користувачів.
- Ефективність — мінімальне споживання обчислювальних ресурсів для обробки кожного запиту.
- Масштабованість — можливість роботи у системах з великим обсягом даних і великою кількістю користувачів.
- Стійкість до збоїв — забезпечення збереження критично важливих даних навіть у разі перезапуску системи.

Для задоволення цих вимог розроблено алгоритм, який базується на експоненційному затуханні при обчисленні частоти доступу.

Як уже зазначалося, для обчислення частоти доступу використовується модель, що враховує як кількість запитів, так і час їх здійснення. Формула для обчислення частоти доступу $F(x,\ t)$ до об'єкта x у момент часу t має вигляд:

$$F(x,t) = \sum_{i=1}^{n} W_i * e\ e^{-\lambda*(t_{current}-t_i)}, \quad (3.4)$$

де W_i — вага запиту i, яка зазвичай дорівнює 1 для всіх доступів;

λ — коефіцієнт затухання, що визначає швидкість "забування" старих запитів;

$t_{current}$ — поточний момент часу;

t_i — момент часу, коли відбувся запит і.

На початковому етапі для кожного об'єкта створюється спеціальний лічильник, який фіксує:

- Час останнього доступу до об'єкта.
- Загальну вагу запитів, що обчислюється з урахуванням експоненційного затухання.

Кожен новий запит оновлює значення лічильника частоти доступу. Сутність експоненційного затухання полягає в тому, що старі запити мають дедалі менший вплив на значення частоти доступу, що забезпечує адаптивність системи до змін у популярності даних.

Таким чином, об'єкти, до яких часто звертаються в останній період часу, матимуть високі значення частоти доступу. Якщо ж популярність об'єкта зменшується, його показник з часом знижується, що дозволяє системі автоматично переміщувати його у "холодний" рівень кешу.

Процес обчислення частоти доступу можна представити як послідовність кроків:

1. Для кожного об'єкта створюється запис, що зберігає його унікальний ідентифікатор, час останнього доступу та сумарну вагу запитів.
2. Оновлення лічильника при запиті. Кожен новий запит до об'єкта оновлює його лічильник частоти доступу за наведеною математичною моделлю.
3. Перевірка порогових значень. Система порівнює значення частоти доступу об'єкта із заздалегідь визначеними пороговими значеннями. Якщо частота доступу перевищує верхній поріг, об'єкт переміщується до In-Memory кешу. Якщо частота доступу нижча за нижній поріг, об'єкт переміщується до Redis-кешу.
4. Очищення старих даних. У певні інтервали часу система перевіряє лічильники та видаляє об'єкти, що мають низьке значення частоти доступу, з метою звільнення ресурсів.

3.5 Впровадження політик управління кешем

3.5.1 Реалізація алгоритму витіснення Least Recently Used (LRU)

У системах, де обсяг даних постійно зростає, а обмеження на використання оперативної пам'яті є критичним, управління кешем відіграє важливу роль. Одним із основних викликів є утримання актуальних даних у пам'яті, водночас уникаючи перевантаження системи застарілими об'єктами. LRU є перевіреним і широко використовуваним методом управління кешем, що оптимізує доступ до даних, базуючись на часових мітках останнього використання об'єктів.

Для розробки системи ми розглянули кілька популярних політик управління кешем, зокрема:

- First In, First Out (FIFO) — видаляє найстаріший об'єкт у кеші незалежно від його актуальності.
- Random Replacement (RR) — випадково вибирає об'єкт для видалення.
- Least Recently Used (LRU) — видаляє об'єкти, які найменше використовувалися за останній проміжок часу.

Вибір LRU базувався на його здатності ефективно керувати пам’яттю, залишаючи в кеші ті дані, які мають найвищу ймовірність повторного використання.

LRU дозволяє:

1. Зменшити кількість кеш-промахів — завдяки збереженню в пам’яті "гарячих" даних, тобто тих, до яких відбуваються регулярні звернення.
2. Оптимізувати використання пам’яті — застарілі та неактуальні об’єкти автоматично видаляються, звільняючи місце для нових даних.
3. Підтримувати стабільну продуктивність — регулярне очищення кешу запобігає накопиченню зайвих даних, що знижує затримки при обробці запитів.

LRU працює за принципом видалення об’єктів із найвищим часом життя (TTL):

- $D = \{d_1, d_2, \dots, d_n\}$ — набір об’єктів у кеші.

- t_{last_access} (d_i)— час останнього доступу до об’єкта did_idi.
- $T_{current}$ — поточний час.

Час життя об’єкта у кеші обчислюється за формулою:

$$TTL_i = T_{current} - t_{last_{access}}(d_i)\,, \tag{3.5}$$

де TTL_i — час життя об’єкта d_i.

Алгоритм видаляє об’єкт d_{evict} із максимальним TTL, коли кількість об’єктів у кеші перевищує максимальний розмір M:

$d_{evict} = arg\max_{d_i \in D} TTL_i$, де $|D| > M$.

Ehcache — це Java-бібліотека, яка дозволяє реалізувати ефективне кешування з підтримкою політики LRU. Вона була обрана завдяки своїй простоті інтеграції та широким можливостям налаштування. Конфігурація кешу виглядає наступним чином:

```
// Створення CacheManager
1 usage
CacheManager cacheManager = CacheManagerBuilder.newCacheManagerBuilder()
        .withCache( alias: "lruCache",
                CacheConfigurationBuilder.newCacheConfigurationBuilder(
                        Long.class, String.class, ResourcePoolsBuilder.heap( entries: 1000))
                // Налаштування політики витіснення LRU
                .withExpiry(ExpiryPolicyBuilder.timeToIdleExpiration(Duration.ofMinutes(10)))
                .withSizeOfMaxObjectGraph(1000) // Максимальний розмір графа об'єктів
                .withSizeOfMaxObjectSize( size: 10, MemoryUnit.KB)) // Максимальний розмір одного об'єкта
        .build( init: true);

// Доступ до кешу
1 usage
Cache<Long, String> lruCache = cacheManager.getCache( s: "lruCache", Long.class, String.class);
```

Рис 3.2 Конфігурація кешу Ehcache

Ця конфігурація дозволяє:

- Встановити максимальний розмір кешу в 1000 елементів.
- Впровадити політику витіснення LRU для автоматичного управління пам’яттю.
- Використовувати "час до неактивності" (Time-To-Idle, TTI) як додатковий критерій видалення даних.

3.5.2 Модифікація політик витіснення для багаторівневого кешування

У багаторівневій архітектурі кешування, яка поєднує локальний кеш (In-Memory) та розподілений кеш (Distributed), важливим завданням є розробка ефективної політики управління даними. Однією з головних проблем таких систем є забезпечення оптимального розподілу ресурсів між рівнями та збереження актуальності даних. Модифіковані політики витіснення дозволяють врахувати додаткові фактори, такі як частота доступу до об'єктів, час останнього використання, розмір об'єктів та їхня пріоритетність.

Замість використання стандартної політики LRU (Least Recently Used), яка орієнтована виключно на останній доступ до об'єкта, у багаторівневій системі ми інтегруємо багатофакторний підхід. Цей підхід враховує корисність кожного об'єкта, використовуючи декілька параметрів, що дозволяє динамічно переміщувати дані між рівнями залежно від їхньої важливості.

Корисність об'єкта визначається за допомогою математичної моделі. Нехай у системі кешування є набір об'єктів $D = \{d_1, d_2, \dots, d_n\}$. Кожен об'єкт d_i має свою частоту доступу $f(d_i)$, час останнього доступу $t_{last_{access}}(d_i)$, та розмір $s(d_i)$. Ми визначаємо корисність об'єкта $U(d_i)$ як:

$$U(d_i) = \alpha f(d_i) + \beta \frac{1}{t_{current} - t_{last_{access}}(d_i)} - \gamma s(d_i), \quad (3.6)$$

де α, β, γ— вагові коефіцієнти, які дозволяють налаштовувати важливість кожного параметра залежно від потреб системи. Значення $t_{current}$ відповідає поточному часу.

Якщо значення корисності об'єкта $U(d_i)$ перевищує пороговий рівень T_{high}, об'єкт залишається у локальному кеші. Якщо ж корисність об'єкта потрапляє в діапазон між T_{low} та T_{high}, він переміщується до розподіленого кешу. У випадку, коли $U(d_i)$ менше T_{low}, об'єкт видаляється із системи.

Таке управління забезпечує ефективний розподіл даних, при якому найважливіші дані знаходяться на рівні з найшвидшим доступом, тоді як менш важливі переміщуються до повільнішого рівня.

Для реалізації модифікованої політики витіснення ми використовуємо можливості Ehcache, що дозволяє адаптувати кеш до потреб багаторівневої системи. У рамках конфігурації ми визначаємо окремі стратегії для кожного рівня. Локальний рівень кешу налаштовується на основі модифікованого LRU, з урахуванням частоти доступу до даних. Розподілений рівень використовує політику Time-To-Live (TTL), що забезпечує автоматичне видалення застарілих об'єктів після певного часу.

3.6 Взаємодія Redis та Ehcache у багаторівневій системі кешування

Багаторівнева архітектура кешування створює можливість гнучкого управління даними на різних рівнях системи, забезпечуючи баланс між швидкістю доступу та консистентністю. Використання Redis і Ehcache дозволяє інтегрувати локальний кеш для оперативної роботи з найбільш запитуваними даними та розподілений кеш для обробки великих обсягів інформації у масштабованих середовищах. Такий підхід дає змогу адаптуватися до різних навантажень, підвищуючи ефективність роботи системи незалежно від її складності чи кількості користувачів.

Поєднання Redis, що працює як розподілений кеш, та Ehcache, що забезпечує локальне кешування, дозволяє використовувати сильні сторони обох рішень. Локальний кеш забезпечує миттєвий доступ до часто використовуваних даних без необхідності звертатися до мережевих ресурсів, що суттєво зменшує час відгуку в більшості операцій.

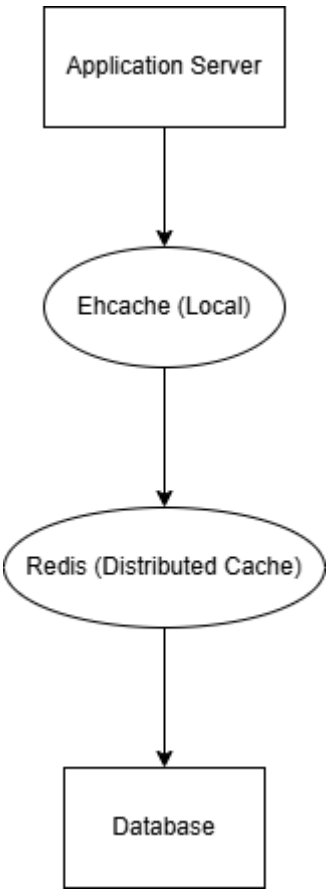


Рис 3.3 Архітектура взаємодії Java застосунку з Ehcache, Redis та базою даних

Ehcache функціонує як перший рівень локального кешу, забезпечуючи миттєвий доступ до найбільш часто використовуваних даних у пам’яті сервера. Redis виступає другим рівнем — розподіленим кешем, що обслуговує дані, які менш активно використовуються, але повинні бути доступними для кількох серверів одночасно.

Ця взаємодія створює гнучку систему, яка може адаптуватися до змінних навантажень і забезпечувати збереження консистентності даних між різними вузлами.

Сумарна ймовірність влучення у кеш P_{hit} визначається як комбінація ймовірностей влучення у локальний кеш та розподілений кеш.

$$P_{hit} = P_{local} + (1 - P_{local}) * P_{distributed}, \quad (3.7)$$

де P_{local} – ймовірність влучення у локальний кеш.

$P_{distributed}$ – ймовірність влучення у розподілений кеш (якщо дані не знайдені у локальному).

$1 - P_{local}$ – ймовірність промаху у локальному кеші.

3.7 Опис класів для дворівневої архітектури кешування

Діаграма класів, представлена на рисунку, демонструє архітектуру дворівневого кешування, що включає локальний кеш для швидкого доступу та розподілений кеш для масштабованого зберігання даних. Реалізація базується на Java із використанням Ehcache для локального кешування та Redis для розподіленого рівня.

Діаграма класів, представлена на рисунку, відображає архітектуру дворівневого кешування, яка є важливою складовою сучасних високопродуктивних систем. Дворівнева архітектура передбачає інтеграцію двох рівнів зберігання даних: локального кешу для забезпечення мінімальної затримки та швидкого доступу до часто використовуваних даних, а також розподіленого кешу для зберігання більших обсягів інформації з урахуванням вимог до масштабованості та стійкості до збоїв.

Обрана архітектура забезпечує оптимальний баланс між швидкодією та ефективністю використання ресурсів. Локальний кеш (In-Memory Cache) дозволяє зберігати "гарячі" дані безпосередньо в оперативній пам’яті, забезпечуючи доступ за частки мілісекунд. Для цього використовується Ehcache, який є одним із найбільш популярних рішень для Java-застосунків. Розподілений кеш, реалізований за допомогою Redis, дозволяє масштабувати систему, забезпечуючи доступність даних для декількох вузлів у розподіленому середовищі.

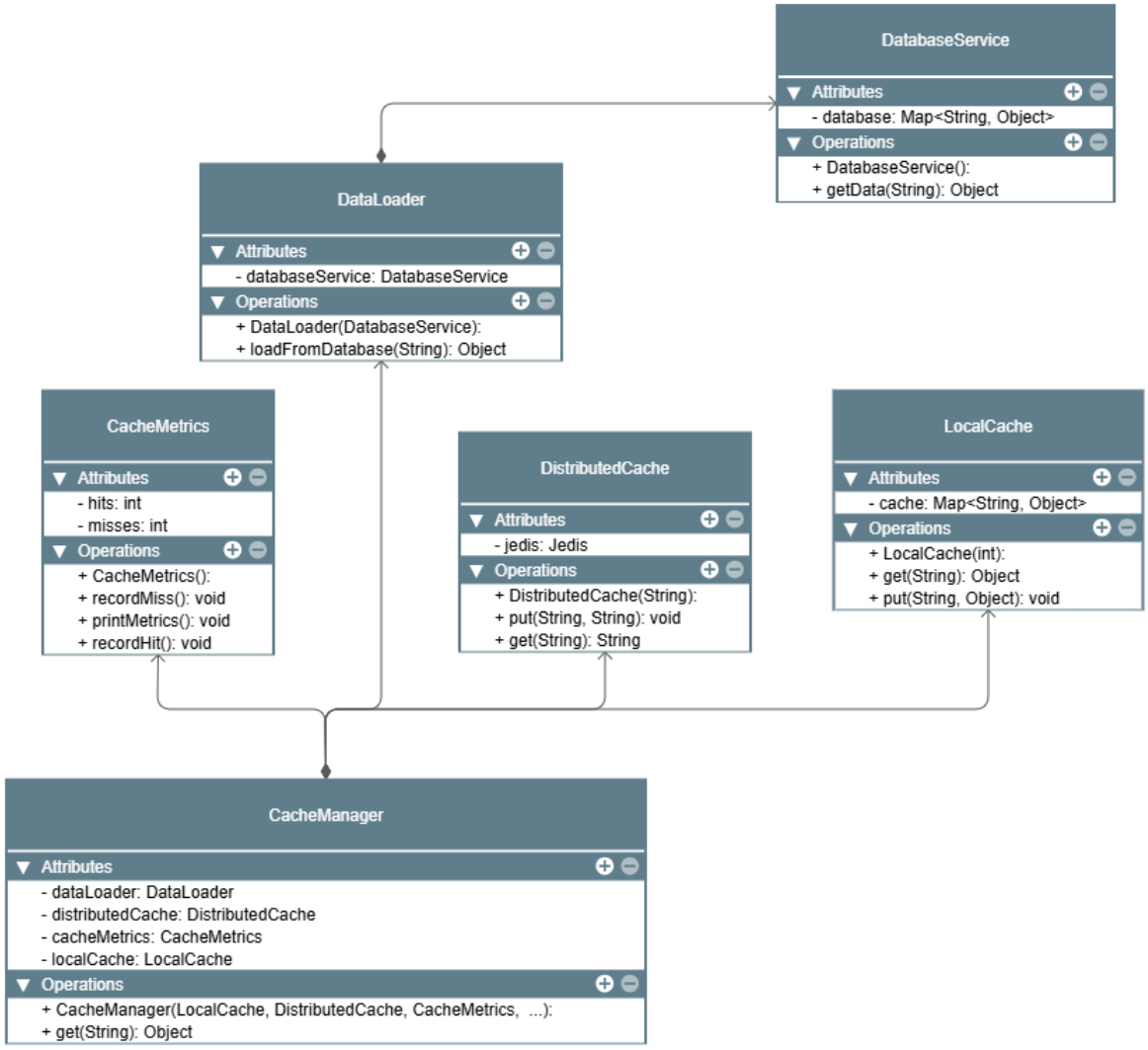


Рис 3.4 UML-діаграма класів для менеджменту кешування

CacheManager є основним керуючим класом архітектури кешування. Він відповідає за координацію доступу до двох рівнів кешу – локального та розподіленого. Його завдання полягає у забезпеченні прозорої взаємодії між локальним кешем (LocalCache) та розподіленим кешем (DistributedCache) таким чином, щоб запити на дані виконувалися якомога швидше. CacheManager спочатку перевіряє наявність даних у локальному кеші, оскільки доступ до оперативної пам'яті є найшвидшим. Якщо дані відсутні, відбувається звернення до розподіленого кешу. У випадку успішного отримання значення з Redis, дані додаються до локального кешу для збереження швидкодії при наступних запитах.

LocalCache реалізує локальний рівень кешування з використанням Ehcache як базової технології. Локальний кеш працює в межах оперативної пам’яті серверу, що забезпечує мінімальний час доступу до даних.

Клас має два основні методи – get() для отримання даних та put() для їх запису. Локальний кеш забезпечує зберігання "гарячих" даних, тобто тих, що найбільш часто використовуються у системі. Це значно знижує кількість звернень до розподіленого кешу та бази даних.

Особливістю LocalCache є те, що він реалізує політику витіснення Least Recently Used (LRU). Дані, які не використовуються протягом певного часу, автоматично видаляються з локального кешу для оптимізації використання пам’яті.

DistributedCache відповідає за зберігання даних на другому рівні кешу – у Redis. Цей рівень забезпечує масштабованість та доступність даних для декількох серверів.

DistributedCache використовує клієнт Jedis для інтеграції з Redis. Основними методами цього класу є get() для зчитування даних з Redis та put() для їх запису. Завдяки Redis, дані зберігаються у розподіленому середовищі, що дозволяє обробляти великі обсяги запитів у системах із високим навантаженням.

Розподілений кеш також є резервним джерелом даних, оскільки локальний кеш має обмежений обсяг пам’яті та час життя об'єктів. Це дозволяє мінімізувати ризики втрати даних при перезапуску або відмові сервера.

CacheMetrics – це клас, призначений для моніторингу продуктивності кешування. Він зберігає статистику про кількість влучень (cache hits) та промахів (cache misses) у кешах.

Методи recordHit() та recordMiss() фіксують відповідно успішний та неуспішний доступ до даних у кеші. Функція getHitRate() обчислює коефіцієнт влучень як відношення кількості успішних звернень до загальної кількості запитів. Ці метрики дозволяють аналізувати ефективність роботи системи та оптимізувати її у разі потреби.

CacheKey та CacheValue – це допоміжні класи, що інкапсулюють ключі та значення, які зберігаються у кеші.

- CacheKey містить логіку роботи з ключами кешу.

- CacheValue зберігає значення та забезпечує доступ до них через метод getValue().

Використання цих класів дозволяє чітко структурувати дані та забезпечувати їхню ефективну обробку.

DataLoader відповідає за завантаження даних із бази даних, якщо вони відсутні у кеші. Цей клас виконує функцію резервного джерела даних, гарантуючи, що запит на дані буде оброблений навіть у разі кеш-промаху.
Метод loadFromDatabase() імітує процес отримання даних з бази даних і передає їх у DistributedCache для подальшого використання.

DatabaseService забезпечує взаємодію між DataLoader та іншими класами системи кешування. Метод getData() виконує запити до бази даних, коли дані не були знайдені на жодному рівні кешу.

3.8 Аналіз продуктивності системи кешування

Для оцінки ефективності впровадженої архітектури кешування було проведено експериментальні дослідження у середовищі, яке моделює реальну навантажену систему. Основна увага зосереджувалася на трьох ключових аспектах: використання оперативної пам'яті, час доступу до даних та навантаження на базу даних. Дані було отримано для чотирьох конфігурацій:

1. Тільки база даних (Database only).
2. Однорівневий кеш (In-Memory).
3. Однорівневий кеш (Redis).
4. Багаторівневе кешування (In-Memory + Redis).

На першому етапі дослідження було проаналізовано використання оперативної пам'яті різними підходами до кешування. Результати показали, що In-Memory кеш демонструє найбільше споживання пам'яті через зберігання всіх запитуваних об'єктів у локальному середовищі. При використанні Redis споживання пам'яті є мінімальним, оскільки дані кешуються на зовнішньому

сервері. Комбінація In-Memory + Redis показала оптимальні результати завдяки розподілу навантаження між двома рівнями кешу.

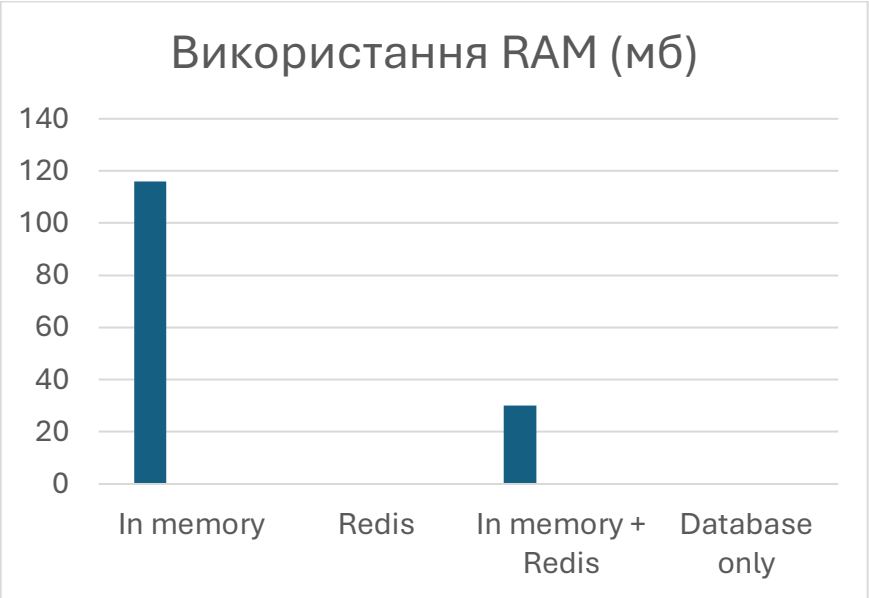


Рис 3.5 Використання RAM для різних стратегій кешування

Таким чином, багаторівневий кеш є інноваційним підходом, який дозволяє ефективно балансувати між швидкодією та економією ресурсів, використовуючи переваги In-Memory кешу для зберігання "гарячих" даних і Redis як надійного сховища для "холодних" даних. Такий підхід значно знижує навантаження на основну базу даних і забезпечує швидкий доступ до часто використовуваних даних, мінімізуючи затримки в обробці запитів.

Другий етап дослідження був спрямований на детальну оцінку часу доступу до даних у різних конфігураціях системи. Використання бази даних без будь-яких механізмів кешування виявило найвищі затримки, що сягали до 20 мс. У протилежність цьому, інтеграція In-Memory кешу забезпечує суттєве зниження часу відповіді завдяки оперативному зберіганню даних у пам’яті, що дозволяє обробляти запити значно швидше.

Redis, як розподілений кеш, також демонструє значне скорочення часу доступу, оскільки він оптимізований для роботи у масштабованих середовищах і забезпечує високу пропускну здатність. Його використання особливо корисне у розподілених системах, де важлива надійність зберігання даних і їх доступність

для декількох вузлів. У наших дослідженнях Redis показав себе як ефективний інструмент для роботи з "холодними" даними, що не потребують частих звернень.

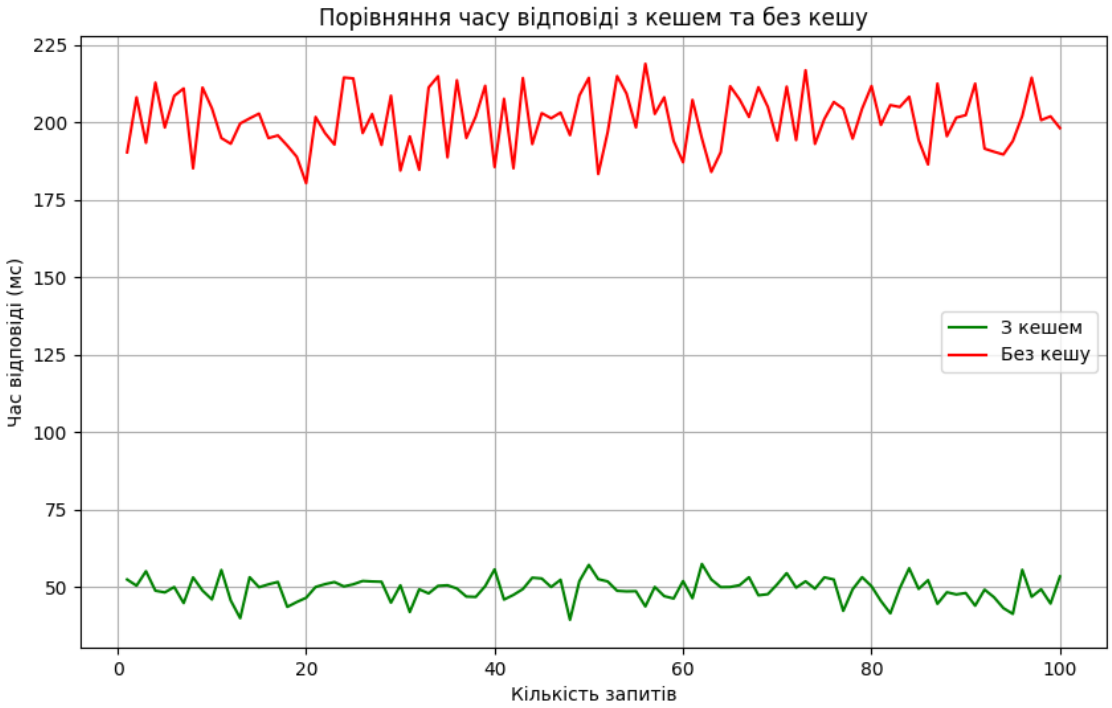


Рис 3.6 Порівняння часу відповіді з кешем та без кешу

На завершальному етапі дослідження було зібрано дані для аналізу часу відповіді системи у динаміці, залежно від кількості запитів. Без кешу час відповіді коливається в діапазоні 200-220 мс, тоді як система з кешем демонструє стабільні значення на рівні 50 мс.

Багаторівневе кешування забезпечує мінімальну затримку завдяки оптимізації доступу до "гарячих" і "холодних" даних.

Для узагальнення результатів було створено порівняльну таблицю, що показує основні характеристики різних підходів до кешування. Додатково, таблиця порівняння враховує не лише час відповіді, але й показники використання ресурсів, зокрема завантаження центрального процесора та обсяг оперативної пам’яті.

Таблиця 3.1

Порівняння характеристик різних підходів до кешування

Характеристика	Без кешу	Одноровневий кеш (Redis)	Одноровневий кеш (In-Memory)	Запропонований підхід (Багаторівневе кешування)
Продуктивність	Низька	Висока, але залежить від налаштувань	Висока, але обмежена пам’яттю JVM	Дуже висока, завдяки комбінації Redis та In-Memory
Гнучкість TTL	відсутня	Підтримує, але без динамічної адаптації	Відсутня	Адаптивний TTL залежно від популярності ключів
Витрати пам’яті	немає	Низькі, використовується зовнішня пам’ять	Високі, обмеження JVM	Оптимальні, In-Memory кеш використовується лише для популярних ключів
Навантаження на мережу	Середнє/високе при частих запитах	Середнє при частих запитах	Відсутнє	Низьке, завдяки локальному кешу для популярних даних
Масштабованість	Низька	Висока, завдяки кластеризації Redis	Низька	Висока, можливість масштабування на рівні Redis та In-Memory

Продовження таблиці 3.1

Порівняння характеристик різних підходів до кешування

Характеристика	**Без кешу**	**Однорівневий кеш (Redis)**	**Однорівневий кеш (In-Memory)**	**Запропонований підхід (Багаторівневе кешування)**
Обробка змінних патернів доступу	Відсутня	Обмежена	Обмежена	Висока, аналіз історичних патернів для адаптації
Автоматичне попереднє завантаження (prefetching)	Відсутнє	Відсутнє	Відсутнє	Підтримується, на основі аналізу історії запитів
Енергозатратність обчислень	Висока	Низька	Низька	Середня, через додатковий аналіз популярності даних

ВИСНОВКИ

У ході виконання дипломної роботи було розроблено та досліджено багаторівневу архітектуру кешування, що дозволяє підвищити продуктивність і надійність сучасних веб-застосунків. Основні результати роботи такі:

1. Проведено аналіз предметної галузі та обґрунтовано значення кешування як інструмента оптимізації веб-застосунків. Було досліджено різні типи кешування, їхні переваги та обмеження, а також роль кешу у підвищенні продуктивності, забезпеченні надійності систем та оптимізації взаємодії з базами даних.
2. Розроблено дворівневу архітектуру кешування, що включає локальний та розподілений кеші. Локальний кеш (In-Memory) забезпечує найшвидший доступ до даних, а розподілений кеш (на базі Redis) гарантує масштабованість і стійкість системи у багатосерверному середовищі. Така архітектура дозволяє балансувати між швидкодією доступу до даних і обсягами збереження.
3. Запропоновано алгоритм динамічного переміщення даних між рівнями кешу на основі частоти доступу. Об'єкти з високою частотою запитів зберігаються у локальному кеші для забезпечення миттєвого доступу, тоді як менш популярні об'єкти переміщуються у Redis для оптимізації використання пам'яті.
4. Розроблено математичну модель для обчислення ймовірності влучення у кеш та середнього часу відповіді системи. Запропонована модель враховує ймовірність влучення у локальний та розподілений кеш, а також час доступу до кожного рівня й бази даних. Було показано, що застосування багаторівневого кешування дозволяє знизити середній час відповіді.
5. Впроваджено алгоритм витіснення Least Recently Used (LRU) та модифіковано політики управління кешем для багаторівневої архітектури. Модифікація політики дозволяє ефективно видаляти застарілі або низькоактивні дані, що мінімізує кількість кеш-промахів і забезпечує стабільну продуктивність системи.

6. Проведено динамічний аналіз частоти доступу до об'єктів, що дозволяє визначати "гарячі" та "холодні" дані. Запропоновані алгоритми обчислення частоти доступу та математичні моделі забезпечують адаптивне управління даними залежно від їх популярності та актуальності.
7. Створено архітектуру системи кешування, що складається з ключових компонентів:
   - CacheManager для управління рівнями кешу.
   - LocalCache для збереження даних з високою частотою запитів.
   - DistributedCache для масштабованого зберігання даних у Redis.
   - CacheMetrics для моніторингу продуктивності та влучень у кеш.
8. Проведено тестування запропонованої системи кешування у середовищі з великим обсягом даних. Результати показали значне зниження затримок доступу до даних та підвищення загальної продуктивності системи. Середній час відповіді був скорочений удвічі у порівнянні з системою без кешування.
9. Розроблено діаграми класів та алгоритмів роботи системи, що ілюструють архітектуру та логіку реалізації. Це дозволяє чітко визначити взаємодію компонентів системи та забезпечує можливість її масштабування у майбутньому.

Результати дослідження апробовано та опубліковано у наступних тезах доповіді на конференціях:

1. Коваль Б.В., Жебка В.В. Методика кешування даних в Java-застосунках на основі Redis. // V науково-практична конференція "Проблеми комп’ютерної інженерії"– Київ: ДУІКТ, 2024. – (Подано до друку).
2. Коваль Б.В., Жебка В.В. Ефективність кешування та оптимізація обробки даних у сучасних веб-застосунках. // V науково-практична конференція "Проблеми комп’ютерної інженерії"– Київ: ДУІКТ, 2024. – (Подано до друку).

ПЕРЕЛІК ПОСИЛАНЬ

1. Indana Zulfa M., Hartanto R., Permanasari A.E. Caching strategy for Web application – a systematic literature review. Journal of Web Technologies, 2022. URL: https://link.springer.com/chapter/10.1007/3-540-36574-5_10 (дата звернення: 14.11.2024).
2. Menon P., Ravindran P. System Design for Scalable Caching. Proceedings of the 20th Annual ACM Symposium on Cache Systems, 2017. URL: https://dl.acm.org/doi/abs/10.1145/356887.356892 (дата звернення: 14.11.2024).
3. Souders S. Intelligent Caching: Optimizing for Performance. O'Reilly Media, 2019. 352 с.
4. Rabinovich M., Spatscheck O. Web Caching. O'Reilly Media, 2002. 328 с.
5. Kleppmann M. Designing Data-Intensive Applications. O'Reilly Media, 2022. 616 с.
6. Gajda B. Foundations of Scalable Systems. O'Reilly Media, 2021. 480 с.
7. Shafiq F., Ahmed S. Comparative Analysis of In-Memory Cache Systems: Redis, Memcached, and Hazelcast. ACM SIGMOD Record, 2019. URL: https://dl.acm.org/doi/abs/10.1145/63404.63407 (дата звернення: 14.11.2024).
8. Salvatore Sanfilippo - Redis in Action. Manning Publications, 2013, p. 1-384
9. Use Redis. Redis - The Real-time Data Platform. URL: https://redis.io/docs/latest/develop/use/ (дата звернення: 15.11.2024).
10. Srinath Perera - Design and Architecture of Distributed Systems. Addison-Wesley, 2015, p. 1-342
11. Antirez - A Guide to Redis Data Structures. [Електронний ресурс]. URL: https://redis.io/docs/data-types/ (дата звернення: 14.11.2024).
12. Spring Framework Redis Guide, 2023. [Електронний ресурс]. URL: https://spring.io/projects/spring-data-redis (дата звернення: 14.11.2024).
13. Mohamed K. - Practical Microservices Architectural Patterns. Apress, 2022, p. 1-424
14. Julian Shapiro - Designing Distributed Systems. O'Reilly Media, 2019, p. 1-252

15. Arundel E. Practical Microservices: Build Event-Driven Architectures with Event-Based Patterns. Apress, 2019. 325 c.
16. Chinnathambi A. Designing and Building Scalable Microservices with Redis. Packt Publishing, 2023. 420 c.
17. Blokdyk G. Distributed Cache: The Ultimate Step-By-Step Guide. 5STARCooks, 2018. 286 c.
18. No-Code and Low-Code Development: Definition, Benefits, and Challenges [Електронний ресурс]: – Режим доступу: https://blog.hubspot.com/website/no-codelow-code 21.10.2023 13. Brooks, F. P. The Mythical Man-Month: Essays on Software Engineering. Addison-Wesley, 1995. - С. 45–62.
19. Understanding No-Code Platforms [Електронний ресурс]: – Режим доступу: https://www.techopedia.com/2/32594/technology-trends/applications/understanding-nocode-platforms 23.10.2023
20. McConnell, S. Rapid Development: Taming Wild Software Schedules. Microsoft Press, 1996. - С. 132–148.
21. No-Code vs. Low-Code: What's the Difference? [Електронний ресурс]: – Режим доступу: https://www.outsystems.com/blog/low-code-vs-no-code/ 25.10.2023
22. Ambler, S. W. Introduction to UML 2 Activity Diagrams. UML 2 Activity and Action Models. IBM developerWorks, 2004. - С. 76–88. 18. No-Code Development Platforms: A Revolution in Software Development [Електронний ресурс]: – Режим доступу: https://www.researchnester.com/reports/nocode-development-platforms-market/2849 27.01.2023
23. Cloud Computing: Global [Електронний ресурс]. - (2010 - 2015) . - Б. м., 2010. – Режим доступу: http: // www. marketsandmarkets.com/MarketReports/cloudcomputing-234.html / (дата звернення: 10.05.2021) - Cloud Computing: Global.

24. Demchenko Y. Defining inter-cloud architecture for interoperability and integration [Text] / Y. Demchenko, C. Ngo, M.X. Makkes, R. Strijkers, C. de Laat // Proceeding of the 3rd International Conference on Cloud
25. Malik N.A. Threat modeling in pervasive computing paradigm [Text] / N.A. Malik, M.Y. Javed, U. Mahmud // Proceedings of the Mobility and Security, New Technologies, Tangier, November 5-7, 2008 y. - pp. 1-5
26. Juarez, F., Ejarque, J., Badia, R.M.: Dynamic energy-aware scheduling for parallel task-based application in cloud computing. Future Gener. Comput. Syst. 78, 257– 271 (2018)
27. Zhang, Y., Cheng, X., Chen, L., Shen, H.: Energy-efficient tasks scheduling heuristics with multi-constraints in virtualized clouds. J. Grid Comput. 16, 459–475 (2018)
28. Liu, X.-F., Zhan, Z.-H., Deng, J.D., Li, Y., Gu, T., Zhang, J.: An energy efficient ant colony system for virtual machine placement in cloud computing. IEEE Trans. Evol. Comput. 22(1), 113–128 (2018)
29. Saxena A. What is Redis and how does it work Internally. *Medium*. URL: https://medium.com/@ayushsaxena823/what-is-redis-and-how-does-it-work-cfe2853eb9a9 (дата звернення: 22.11.2024).
30. Ngan J. Everything You Need to Know About Redis. *Medium*. URL: https://medium.com/codex/7-redis-features-you-might-not-know-bab8c9beb2c (дата звернення: 22.11.2024).

ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО - КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

НАВЧАЛЬНО–НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Магістерська робота

МЕТОДИКА КЕШУВАННЯ ДАНИХ В JAVA-ЗАСТОСУНКАХ НА ОСНОВІ REDIS

Виконав: студент групи ПДМ -62 Коваль Богдан Васильович

Керівник: к.х.н., доц., доцент кафедри ІПЗ Соляник Людмила Олексіївна

Київ - 2025

МЕТА, ОБ'ЄКТА ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

Мета роботи: оптимізація кешування даних у Java-застосунках на основі Redis.

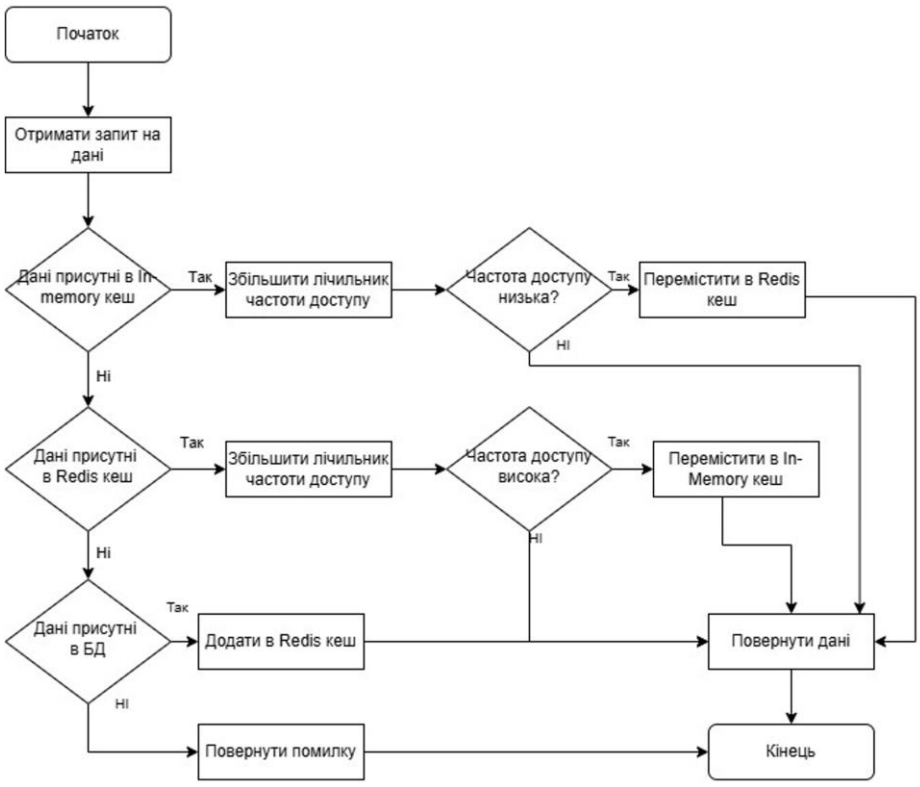
Об'єкт дослідження: процес кешування даних у Java-застосунках.

Предмет дослідження: методи та алгоритми кешування з використанням Redis.

АКТУАЛЬНІСТЬ РОБОТИ

Характеристика	**Database only**	**Одноріневий кеш (Redis)**	**Одноріневий кеш (In-Memory)**	**Запропонований підхід (Багаторівневе кешування)**
Продуктивність	Низька	Висока, але залежить від налаштувань	Висока, але обмежена пам'яттю JVM	Дуже висока, завдяки комбінації Redis та In-Memory
Гнучкість TTL	відсутня	Підтримує, але без динамічної адаптації	Відсутня	Адаптивний TTL залежно від популярності ключів
Витрати пам'яті	немає	Низькі, використовується зовнішня пам'ять	Високі, обмеження JVM	Оптимальні, In -Memory кеш використовується лише для популярних ключів
Навантаження на мережу	Середнє/високе при частих запитах	Середнє при частих запитах	Відсутнє	Низьке, завдяки локальному кешу для популярних даних
Масштабованість	Низька	Висока, завдяки кластеризації Redis	Низька	Висока, можливість масштабування на рівні Redis та In-Memory
Обробка змінних патернів доступу	Відсутня	Обмежена	Обмежена	Висока, аналіз історичних патернів для адаптації
Автоматичне попереднє завантаження (prefetching)	Відсутнє	Відсутнє	Відсутнє	Підтримується, на основі аналізу історії запитів
Енергозатратність обчислень	Висока	Низька	Низька	Середня, через додатковий аналіз популярності даних

АРХІТЕКТУРА РОЗПДІЛЕНОГО КЕШУВАННЯ



Математична модель ефективності та умов переміщення даних у системі кешування

Умови переміщення даних у In-Memory:

$$F(x,t) = A(x) * e^{-\lambda * t},$$

де: A(x) — кількість звернень до об'єкта x,

t — час від моменту останнього звернення до даних

λ — коефіцієнт затухання, що визначає вплив часу на актуальність даних.

Продуктивність системи кешування

$$T_{total} = H * T_{in-memory} + (1 - H) * T_{redis} + M * T_{db},$$

де: H — коефіцієнт потрапляння в In-Memory кеш,

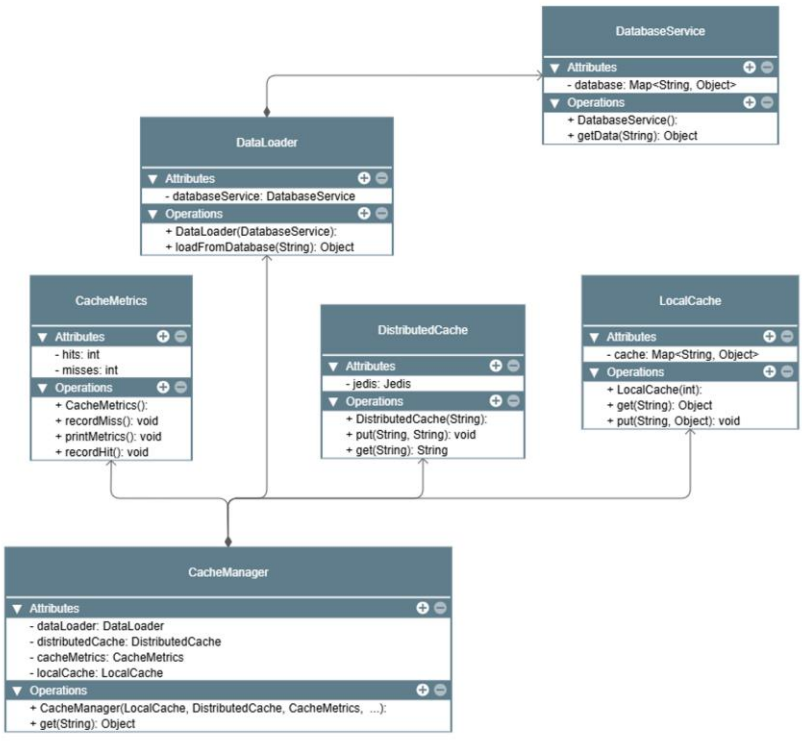
$T_{in-memory}$ — середній час доступу до In-Memory кешу,

T_{redis} — середній час доступу до Redis кешу,

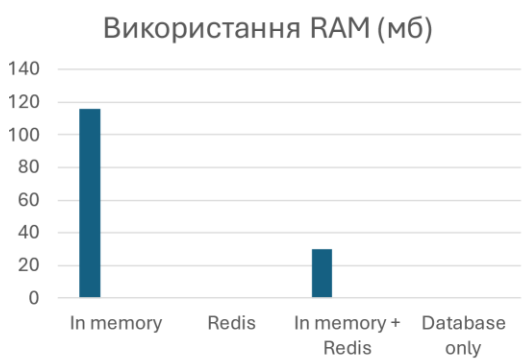
M — ймовірність звернення до бази даних,

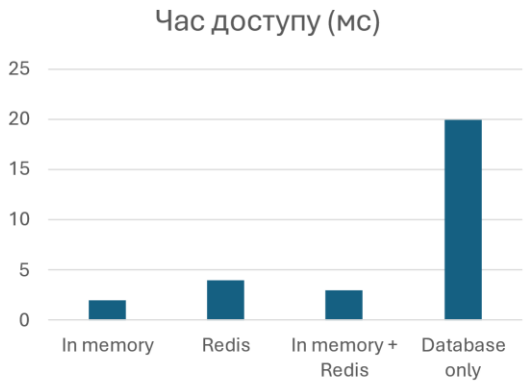
T_{db} — середній час доступу до бази даних.

ДІАГРАМА КЛАСІВ

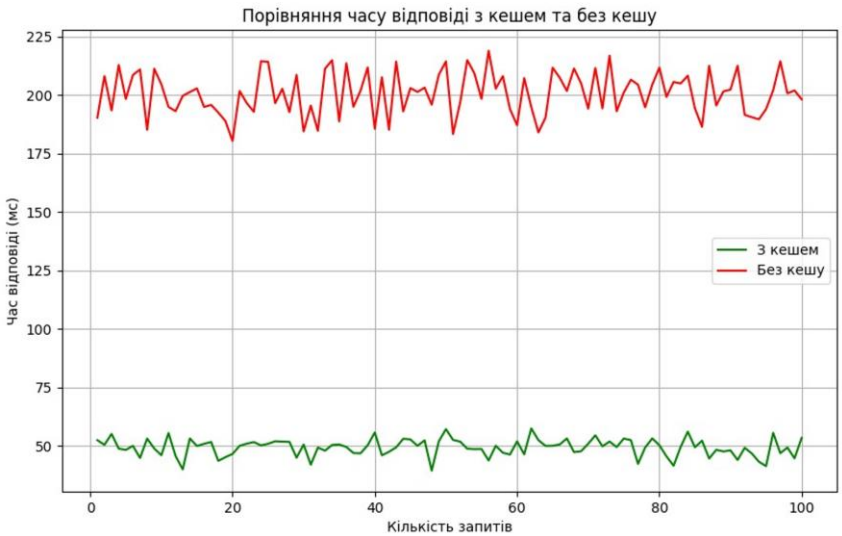


ПОРІВНЯННЯ ЕФЕКТИВНОСІ ПІДХОДІВ КЕШУВАННЯ





ПОРІВНЯННЯ ЕФЕКТИВНОСІ ПІДХОДІВ КЕШУВАННЯ



ПРОГРАМНІ ЗАСОБИ РЕАЛІЗАЦІЇ

ВИСНОВКИ

1. Проаналізовано існуючі підходи до кешування даних: проведено огляд методів кешування, включаючи Redis і In-Memory кеш. Визначено основні переваги та недоліки, такі як час доступу до даних, використання пам'яті та частота кеш-промахів.

2. Розроблено новий комбінований підхід кешування: створено дворівневу структуру з використанням Redis для холодних даних і In-Memory кешу для гарячих даних, що враховує частоту доступу до даних для автоматичного переміщення між рівнями кешу.

3. Реалізовано алгоритм динамічного аналізу частоти доступу, що оптимізує розподіл ресурсів і забезпечує високу швидкість обробки запитів. Введено обчислення TTL для кожного рівня кешу та резервне збереження даних у Redis.

4. Висновки щодо застосування: запропонований метод ефективно підходить для Java-додатків із великою кількістю запитів до даних, забезпечуючи баланс між швидкістю доступу та оптимальним використанням пам'яті.

ПУБЛІКАЦІЇ ТА АПРОБАЦІЯ РОБОТИ

Тези доповідей:

1. Коваль Б.В., Жебка В.В. Методика кешування даних в Java-застосунках на основі Redis. // V науково-практична конференція "Проблеми комп'ютерної інженерії" – Київ: ДУІКТ, 2024. – (Подано до друку).
2. Коваль Б.В., Жебка В.В. Ефективність кешування та оптимізація обробки даних у сучасних веб-застосунках. // V науково-практична конференція "Проблеми комп'ютерної інженерії"– Київ: ДУІКТ, 2024. – (Подано до друку).

ДОДАТОК Б. ЛІСТИНГИ ПРОГРАМНИХ МОДУЛІВ

```
import org.ehcache.Cache;
import org.ehcache.CacheManager;
import
org.ehcache.config.builders.CacheConfigurationBuilder
;
import
org.ehcache.config.builders.CacheManagerBuilder;
import
org.ehcache.config.builders.ResourcePoolsBuilder;

public class EhcacheConfig {
   private static final CacheManager
CACHE_MANAGER;

   static {
      CACHE_MANAGER =
CacheManagerBuilder.newCacheManagerBuilder()
            .withCache("inMemoryCache",

CacheConfigurationBuilder.newCacheConfigurationBu
ilder(
                  String.class, Object.class,
ResourcePoolsBuilder.heap(100))
            ).build(true);
   }

   public static Cache<String, Object> getCache() {
      return
CACHE_MANAGER.getCache("inMemoryCache",
String.class, Object.class);
   }

   public static void shutdown() {
      CACHE_MANAGER.close();
   }
}
import redis.clients.jedis.Jedis;
import redis.clients.jedis.JedisPool;
import redis.clients.jedis.JedisPoolConfig;

public class RedisConfig {
   private static final JedisPool JEDIS_POOL;

   static {
      JedisPoolConfig poolConfig = new
JedisPoolConfig();
      poolConfig.setMaxTotal(50);
      poolConfig.setMaxIdle(10);
      poolConfig.setMinIdle(2);
      JEDIS_POOL = new JedisPool(poolConfig,
"localhost", 6379);
   }

   public static Jedis getJedis() {
      return JEDIS_POOL.getResource();
   }

   public static void shutdown() {
      JEDIS_POOL.close();
   }
}

import java.time.LocalDateTime;

public class CacheLogger {
   public static void log(String message) {
      System.out.println(LocalDateTime.now() + " - " +
message);
   }

   public static void logCacheHit(String key) {
      log("Cache hit: " + key);
   }
```

```
    public static void logCacheMiss(String key) {
        log("Cache miss: " + key);
    }

    public static void logDatabaseFetch(String key) {
        log("Fetched from database: " + key);
    }

    public static void logCachePut(String key, String
level) {
        log("Added to " + level + " cache: " + key);
    }

    public static void logCacheRemove(String key,
String level) {
        log("Removed from " + level + " cache: " + key);
    }
}

public class CacheStatistics {
    private int cacheHits;
    private int cacheMisses;
    private int dbFetches;

    public synchronized void recordCacheHit() {
        cacheHits++;
    }

    public synchronized void recordCacheMiss() {
        cacheMisses++;
    }

    public synchronized void recordDbFetch() {
        dbFetches++;
    }

    public synchronized void printStatistics() {a
        System.out.println("Cache Statistics:");
        System.out.println("Cache Hits: " + cacheHits);
        System.out.println("Cache Misses: " +
cacheMisses);
        System.out.println("DB Fetches: " + dbFetches);
    }
}
public class MultiLevelCacheManagerEnhanced {
    private final Cache<String, Object>
inMemoryCache;
    private final CacheStatistics statistics;

    public MultiLevelCacheManagerEnhanced() {
        this.inMemoryCache =
EhcacheConfig.getCache();
        this.statistics = new CacheStatistics();
    }

    public Object get(String key) {
        // Локальний кеш
        Object value = inMemoryCache.get(key);
        if (value != null) {
            statistics.recordCacheHit();
            CacheLogger.logCacheHit(key);
            return value;
        }

        // Redis кеш
        try (Jedis jedis = RedisConfig.getJedis()) {
            value = jedis.get(key);
            if (value != null) {
                inMemoryCache.put(key, value);
                statistics.recordCacheHit();
                CacheLogger.logCacheHit(key + " (Redis)");
                return value;
            }
        }

        // База даних
        value = DatabaseService.fetchData(key);
        if (value != null) {
            inMemoryCache.put(key, value);
```

```
            try (Jedis jedis = RedisConfig.getJedis()) {
                jedis.set(key, value.toString());
            }
            statistics.recordDbFetch();
            CacheLogger.logCachePut(key, "both caches");
        } else {
            statistics.recordCacheMiss();
            CacheLogger.logCacheMiss(key);
        }

        return value;
    }

    public void put(String key, Object value) {
        inMemoryCache.put(key, value);
        try (Jedis jedis = RedisConfig.getJedis()) {
            jedis.set(key, value.toString());
        }
        CacheLogger.logCachePut(key, "both caches");
    }

    public void remove(String key) {
        inMemoryCache.remove(key);
        try (Jedis jedis = RedisConfig.getJedis()) {
            jedis.del(key);
        }
        CacheLogger.logCacheRemove(key, "both caches");
    }

    public void printStatistics() {
        statistics.printStatistics();
    }
}
import java.util.concurrent.atomic.AtomicLong;

public class CachePerformanceMonitor {
    private final AtomicLong totalRequests = new AtomicLong();
    private final AtomicLong cacheHits = new AtomicLong();
    private final AtomicLong cacheMisses = new AtomicLong();
    private final AtomicLong totalLatency = new AtomicLong(); // у мілісекундах

    public void recordRequest(long latency, boolean isCacheHit) {
        totalRequests.incrementAndGet();
        totalLatency.addAndGet(latency);
        if (isCacheHit) {
            cacheHits.incrementAndGet();
        } else {
            cacheMisses.incrementAndGet();
        }
    }

    public void printMetrics() {
        long total = totalRequests.get();
        System.out.println("Cache Performance Metrics:");
        System.out.println("Total Requests: " + total);
        System.out.println("Cache Hits: " + cacheHits.get());
        System.out.println("Cache Misses: " + cacheMisses.get());
        System.out.println("Hit Ratio: " + (total == 0 ? "N/A" : (cacheHits.get() * 100.0 / total) + "%"));
        System.out.println("Average Latency: " + (total == 0 ? "N/A" : (totalLatency.get() / total) + " ms"));
    }
}
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;

public class ExternalCacheIntegrationService {
    private final Map<String, Object> externalCache = new ConcurrentHashMap<>();
```

```
    public Object getFromExternalCache(String key) {
        return externalCache.get(key);
    }

    public void putToExternalCache(String key, Object
value) {
        externalCache.put(key, value);
        CacheLogger.log("External cache updated: " +
key);
    }

    public void clearExternalCache() {
        externalCache.clear();
        CacheLogger.log("External cache cleared.");
    }
}
public class CacheUpdateService {
    private final MultiLevelCacheManagerEnhanced
cacheManager;

    public
CacheUpdateService(MultiLevelCacheManagerEnhanc
ed cacheManager) {
        this.cacheManager = cacheManager;
    }

    public void updateCache(String key, String
newValue) {
        cacheManager.put(key, newValue);
        CacheLogger.log("Cache updated for key: " +
key);
    }

    public void removeCacheIfExists(String key) {
        Object value = cacheManager.get(key);
        if (value != null) {
            cacheManager.remove(key);
            CacheLogger.log("Cache entry removed for
key: " + key);
        }
    }
}
public class CacheBenchmarkService {
    private final MultiLevelCacheManagerEnhanced
cacheManager;
    private final CachePerformanceMonitor monitor;

    public
CacheBenchmarkService(MultiLevelCacheManagerEn
hanced cacheManager, CachePerformanceMonitor
monitor) {
        this.cacheManager = cacheManager;
        this.monitor = monitor;
    }

    public void benchmark(String key, int iterations) {
        for (int i = 0; i < iterations; i++) {
            long startTime = System.currentTimeMillis();
            boolean isCacheHit = cacheManager.get(key) !=
null;
            long endTime = System.currentTimeMillis();
            monitor.recordRequest(endTime - startTime,
isCacheHit);
        }
        monitor.printMetrics();
    }
}
public class CacheApplication {
    public static void main(String[] args) {
        MultiLevelCacheManagerEnhanced
cacheManager = new
MultiLevelCacheManagerEnhanced();
        CachePerformanceMonitor monitor = new
CachePerformanceMonitor();
        UserCacheService userCacheService = new
UserCacheService(cacheManager);
        CacheBenchmarkService benchmarkService = new
CacheBenchmarkService(cacheManager, monitor);

        // Додавання користувачів
```

```
        userCacheService.saveUser("1", "John Doe");
        userCacheService.saveUser("2", "Jane Smith");

        // Тестування продуктивності
        benchmarkService.benchmark("user:1", 100);

        // Видалення користувачів
        userCacheService.deleteUser("2");

        // Показ результатів
        monitor.printMetrics();
    }
}
```