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

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Модель керування узгодженістю в гібридних кешах для розподілених веб-платформ»

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

_____ Іван СТРЮКОВ
(підпис)

Виконав: здобувач вищої освіти групи ПДМ-63
Іван СТРЮКОВ

Керівник: Тимур ДОВЖЕНКО
канд. техн. наук

Рецензент: _____
науковий ступінь, вчене звання *Ім'я, ПРІЗВИЩЕ*

Київ 2026

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
«_____» _____ 2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Стрюкову Івану Геннадійовичу

1. Тема кваліфікаційної роботи: «Модель керування узгодженістю в гібридних кешах для розподілених веб-платформ»

керівник кваліфікаційної роботи Тимур ДОВЖЕНКО, канд. техн. наук,

затверджені наказом Державного університету інформаційно-комунікаційних технологій від «30» жовтня 2025 р. № 467.

2. Строк подання кваліфікаційної роботи «19» грудня 2025 р.

3. Вихідні дані до кваліфікаційної роботи: науково-технічна література, характеристики багаторівневих гібридних кешів, параметри моделі узгодженості даних та адаптивного алгоритму LRU-K, вимоги до продуктивності та латентності розподілених систем.

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

1. Аналіз існуючих рішень та особливості кешування у розподілених веб-системах.
2. Теоретичні основи моделі керування узгодженістю в гібридних кешах.

3. Проектування архітектури гібридної кеш-системи.
4. Практична реалізація гібридної системи кешування.
5. Експериментальне дослідження та аналіз результатів.

5. Перелік ілюстративного матеріалу: *презентація*

1. Мета, об'єкт та предмет дослідження.
2. Порівняльна характеристика існуючих рішень кешування.
3. Архітектура багаторівневої гібридної кеш-системи.
4. Адаптивний алгоритм LRU-K: модель оцінки пріоритету.
5. Механізм синхронізації та інвалідації даних.
6. Діаграма класів системи.
7. Результати експериментального дослідження.

6. Дата видачі завдання «31» жовтня 2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Аналіз наявної науково-технічної літератури	31.10-07.11.2025	
2	Вивчення матеріалів для аналізу існуючих систем кешування (Redis, Memcached, CDN)	05.11-12.11.2025	
3	Дослідження алгоритмів керування кешем (LRU, LFU, LRU-K)	11.11-15.11.2025	
4	Розробка математичної моделі оцінки пріоритету кешування	15.11-19.11.2025	
5	Проектування архітектури гібридної кеш-системи	16.11-25.11.2025	
6	Реалізація програмного прототипу та проведення експериментів	20.11-29.11.2025	
7	Оформлення роботи: вступ, висновки, реферат	18.11-17.12.2025	
8	Розробка демонстраційних матеріалів	19.12.2025	

Здобувач вищої освіти _____ Іван СТРЮКОВ
(підпис)

Керівник
кваліфікаційної роботи _____ Тимур ДОВЖЕНКО
(підпис)

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня магістра: ___ стор., ___ табл., ___ рис., 11 джерел.

Мета роботи – підвищення продуктивності розподілених веб-платформ шляхом розроблення моделі керування узгодженістю даних у гібридних багаторівневих кешах з використанням адаптивних алгоритмів.

Об'єкт дослідження – процес кешування даних у розподілених веб-системах.

Предмет дослідження – моделі та алгоритми керування узгодженістю даних у багаторівневих гібридних кеш-системах.

У роботі проведено аналіз сучасних методів кешування, зокрема локальних, розподілених та CDN-рішень, а також їхніх обмежень у високонавантажених системах. Розглянуто особливості використання Redis, Memcached та Caffeine у складі гібридних кешів. Визначено ключові проблеми узгодженості даних при багаторівневому кешуванні та фактори, що впливають на ефективність інвалідації та синхронізації кешу.

Розроблено математичну модель оцінки пріоритетів кешування та адаптивний алгоритм керування узгодженістю на основі модифікованої моделі LRU-K, який враховує динаміку частоти звернень та характеристики кожного рівня кешу. Запропоновано архітектуру гібридної кеш-системи для розподілених веб-платформ, описано програмний прототип та інструменти реалізації.

Проведено експериментальні дослідження, які підтверджують підвищення швидкодії системи, збільшення cache hit rate та зниження латентності виконання запитів порівняно зі стандартними методами кешування. Показано ефективність запропонованої моделі в умовах змінних навантажень.

КЛЮЧОВІ СЛОВА: ГІБРИДНИЙ КЕШ, УЗГОДЖЕНІСТЬ ДАНИХ, REDIS, MEMCACHED, CAFFEINE, ГЛИБОКЕ КЕШУВАННЯ, БАГАТОРІВНЕВА АРХІТЕКТУРА, LRU-K, РОЗПОДІЛЕНІ СИСТЕМИ, ПРОДУКТИВНІСТЬ ВЕБ-ПЛАТФОРМ.

ABSTRACT

Text part of the master's qualification work: ___ pages, ___ pictures, ___ tables, 11 sources.

The purpose of the work is to increase the performance of distributed web platforms by developing a consistency-management model for hybrid multi-level cache systems using adaptive algorithms.

Object of research – the data caching process in distributed web systems.

Subject of research – models and algorithms for data consistency management in multi-level hybrid cache systems.

The work analyzes modern caching approaches, including local, distributed, and CDN-based solutions, as well as their limitations in high-load environments. The features of Redis, Memcached, and Caffeine within hybrid cache architectures are examined. The key issues of maintaining data consistency in multi-level cache hierarchies and the factors influencing cache invalidation efficiency are identified.

A mathematical model for evaluating caching priorities and an adaptive consistency-management algorithm based on a modified LRU-K model were developed. The algorithm accounts for the dynamics of access frequency and the characteristics of each cache level. A hybrid cache architecture for distributed web platforms is proposed, along with the description of a software prototype and the tools used for implementation.

Experimental research demonstrates an increase in system performance, higher cache hit rates, and reduced request latency compared to standard caching approaches. The results confirm the effectiveness of the proposed model under variable load conditions.

KEYWORDS: HYBRID CACHE, DATA CONSISTENCY, REDIS, MEMCACHED, CAFFEINE, MULTI-LEVEL CACHING, LRU-K, DISTRIBUTED SYSTEMS, WEB PLATFORM PERFORMANCE.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ 10
ВСТУП 11
1 АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ ТА ОСОБЛИВОСТІ КЕШУВАННЯ У РОЗПОДІЛЕНИХ ВЕБ-СИСТЕМАХ 17
1.1 Загальні аспекти кешування у веб-додатках 17
1.2 Аналіз популярних систем кешування (Redis, Memcached, CDN) 20
1.3 Особливості гібридного кешування 23
1.4 Порівняльний аналіз переваг та недоліків існуючих підходів 26
1.5 Проблема узгодженості даних у багаторівневих кеш-системах 29
2 ТЕОРЕТИЧНІ ОСНОВИ МОДЕЛІ КЕРУВАННЯ УЗГОДЖЕНІСТЮ В ГІБРИДНИХ КЕШАХ 33
2.1 Математичні моделі оцінки пріоритету кешування 33
2.2 Алгоритми витіснення даних (LRU, LFU, LRU-K) 37
2.3 Модель адаптивного керування узгодженістю на основі LRU-K 41
3 ПРОЕКТУВАННЯ АРХІТЕКТУРИ ГІБРИДНОЇ КЕШ-СИСТЕМИ 50
3.1 Архітектура багаторівневої гібридної системи кешування 50
3.2 Проектування компонентів системи 53
3.3 Механізми синхронізації та інвалідації даних 59
3.4 Інтеграція з розподіленими веб-платформами 64
4 ПРАКТИЧНА РЕАЛІЗАЦІЯ ГІБРИДНОЇ СИСТЕМИ КЕШУВАННЯ 68
4.1 Вибір технологічного стеку (Spring Boot, Caffeine, Redis, Memcached) 68
4.2 Реалізація основних компонентів системи 72
4.3 Розробка веб-інтерфейсу для моніторингу 75
4.4 Створення Spring Boot Starter для інтеграції 77
5 ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ ТА АНАЛІЗ РЕЗУЛЬТАТІВ 80
5.1 Методика проведення експериментів 80
5.2 Порівняльне тестування гібридного та стандартного підходів 81
5.3 Аналіз результатів тестування 84
5.4 Рекомендації щодо впровадження 85

ВИСНОВКИ 88
ПЕРЕЛІК ПОСИЛАНЬ 90
ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ 92
ДОДАТОК Б. ЛІСТИНГ ОСНОВНИХ МОДУЛІВ 96

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

L1 – локальний кеш (in-memory, Caffeine)

L2 – розподілений кеш (Redis)

L3 – кеш результатів SQL-запитів (Memcached)

CDN – мережа доставки контенту (Content Delivery Network)

LRU – Least Recently Used, алгоритм керування кешем

LFU – Least Frequently Used, алгоритм керування кешем

LRU-K – модифікований алгоритм LRU з урахуванням K-го звернення

TTL – Time To Live, час життя кешованих даних

Hit Rate – частка успішних попадань у кеш

Miss – відсутність даних у кеші

Eviction – витіснення даних із кешу

Latency – затримка доступу до даних

Throughput – пропускна здатність системи

API – Application Programming Interface

DB – база даних

Cache Consistency – узгодженість даних у кеш-системі

Hybrid Cache – гібридна багаторівнева система кешування

Replication – реплікація даних

Invalidation – інвалідація застарілих даних

ВСТУП

Актуальність теми дослідження визначається стрімким розвитком веб-технологій та зростанням вимог до продуктивності сучасних розподілених платформ. Постійне збільшення кількості користувачів, обсягів даних та частоти звернень до серверів створює значне навантаження на інфраструктуру веб-додатків. Одним із ключових механізмів підвищення швидкодії та забезпечення стабільності роботи таких систем є кешування. Однак традиційні однорівневі системи кешування часто не здатні забезпечити необхідну ефективність в умовах динамічного та нерівномірного навантаження. У різних частинах веб-платформи використовуються різні типи даних, що відрізняються за частотою доступу, критичністю та обсягом. Це зумовлює потребу у гібридних багаторівневих підходах, які поєднують локальні кеші, розподілені сховища та системи доставки контенту.

Разом із цим постає складна проблема забезпечення узгодженості даних між різними рівнями кешу. Розсинхронізація, затримки оновлення чи неправильна інвалідація даних можуть призводити до некоректної роботи системи. Тому актуальним стає розроблення моделей керування узгодженістю, здатних адаптивно визначати оптимальний рівень зберігання та оновлення даних на основі патернів доступу користувачів.

У цьому контексті особливе значення має використання адаптивних алгоритмів, зокрема LRU-K, які враховують історію звернень та дозволяють точніше визначати пріоритети зберігання даних. Поєднання таких алгоритмів із гібридною архітектурою кешування дає змогу досягти значного підвищення продуктивності веб-платформ.

Метою магістерської кваліфікаційної роботи є розробка моделі керування узгодженістю в гібридних кешах розподілених веб-платформ, що забезпечує підвищення їх продуктивності шляхом адаптивного розподілу та оновлення даних.

Для досягнення поставленої мети було визначено такі завдання:

1. Провести аналіз існуючих підходів до кешування та визначити їхні обмеження у високонавантажених веб-системах.
2. Розробити архітектуру багаторівневої гібридної кеш-системи.
3. Створити адаптивний алгоритм керування узгодженістю на основі моделі LRU-K.
4. Реалізувати програмний прототип гібридної системи кешування.
5. Провести експериментальну оцінку ефективності запропонованої моделі.
6. Порівняти результати роботи гібридного підходу зі стандартними методами кешування.

Об'єктом дослідження є процес кешування даних у розподілених веб-платформах.

Предметом дослідження є моделі та алгоритми керування узгодженістю в багаторівневих гібридних кеш-системах.

У процесі дослідження використовувалися методи аналізу літературних джерел, моделювання архітектури системи, розроблення програмного забезпечення, експериментальне тестування та порівняльний аналіз.

Наукова новизна отриманих результатів полягає у розробленні адаптивної моделі керування узгодженістю, яка забезпечує динамічне визначення оптимального рівня зберігання даних та підвищує ефективність гібридної кеш-системи.

Практичне значення роботи полягає у можливості впровадження запропонованої моделі у реальні веб-платформи для покращення швидкодії, зменшення навантаження на бази даних та забезпечення стабільності роботи системи в умовах високих навантажень.

Апробація результатів дослідження здійснювалась у рамках наукових конференцій та семінарів, де були представлені основні ідеї та результати розробленої гібридної кеш-системи.

1 АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ ТА ОСОБЛИВОСТІ КЕШУВАННЯ У РОЗПОДІЛЕНИХ ВЕБ-СИСТЕМАХ

1.1 Загальні аспекти кешування у веб-додатках

Кешування є одним із фундаментальних елементів сучасних інформаційних систем, особливо у високонавантажених веб-додатках та розподілених платформах. Основна мета кешування — зменшення часу відповіді системи та оптимізація використання обчислювальних ресурсів. За своєю суттю кешування дозволяє тимчасово зберігати дані, які найчастіше запитуються, для забезпечення швидшого доступу до них у майбутньому. У контексті розподілених веб-платформ кешування набуває особливого значення, оскільки дозволяє суттєво знизити навантаження на бази даних та забезпечити високу доступність сервісів.

З точки зору архітектури, кешування може бути реалізоване на різних рівнях системи. Кожен рівень має свої характеристики щодо швидкості доступу, обсягу зберігання та складності керування. Розуміння цих рівнів є критично важливим для проектування ефективних гібридних систем кешування.

Кешування на стороні клієнта є найбільш поширеним рівнем, який включає механізми браузерного кешування. Веб-браузери можуть зберігати статичні файли (зображення, CSS, JavaScript), щоб уникнути повторних завантажень із сервера. Проте цей рівень має обмеження, зокрема неможливість зберігати динамічні дані або забезпечувати синхронізацію з актуальними даними на сервері. У сучасних веб-додатках використовуються більш досконалі підходи до клієнтського кешування, наприклад, Service Workers, які дозволяють зберігати як статичні, так і динамічні дані у браузері, забезпечуючи офлайн-доступ.

Кешування на стороні сервера використовується для зменшення навантаження на бази даних і серверні ресурси. Результати складних обчислень або SQL-запитів можуть бути збережені у швидкодіючій пам'яті (in-memory cache), наприклад, за допомогою Redis або Memcached. Серверне кешування також може

використовуватись для зберігання попередньо відрендерених сторінок або компонентів, що значно прискорює процес відповіді на запити клієнтів. Цей рівень кешування є ключовим для забезпечення продуктивності розподілених веб-платформ.

Кешування на проміжному рівні включає використання мереж доставки контенту (CDN), які забезпечують географічно розподілене зберігання статичних ресурсів. CDN дозволяють зменшити затримки передачі даних, знижуючи навантаження на центральний сервер і покращуючи досвід користувачів. Сучасні CDN-сервіси, такі як Cloudflare або Akamai, пропонують додаткові функції, включаючи динамічне кешування, автоматичну оптимізацію зображень і інтеграцію з безпековими сервісами.

Кожен із цих рівнів має свої переваги та обмеження, і їх поєднання часто використовується для створення високоефективних рішень. Проте вибір конкретної архітектури залежить від таких факторів, як тип даних, обсяг запитів і географічний розподіл користувачів. У таблиці 1.1 наведено порівняльну характеристику рівнів кешування.

Таблиця 1.1

Порівняльна характеристика рівнів кешування

Рівень кешування	**Латентність**	**Переваги**	**Недоліки**
Клієнтське	< 1 мс	Миттєвий доступ; зниження трафіку	Обмежений обсяг; лише статичні дані
Локальне (in-memory)	0.05–1 мс	Надзвичайно швидке	Не розподілене; обмежений розмір
Розподілене (Redis)	2–5 мс	Спільний доступ; великий обсяг	Мережева латентність
CDN	10–50 мс	Глобальний доступ; географічна близькість	Висока вартість; складна інвалідація

Окрім рівнів кешування, важливим аспектом є вибір алгоритму керування кешем. Алгоритми визначають, які дані зберігати в кеші та які витісняти при нестачі місця. Найбільш поширеними алгоритмами є LRU (Least Recently Used), LFU (Least Frequently Used) та їх модифікації. Алгоритм LRU витісняє дані, до яких найдовше не зверталися, тоді як LFU враховує частоту звернень до даних. Для розподілених систем особливу цінність має алгоритм LRU-K, який враховує історію K останніх звернень до кожного елемента, що дозволяє точніше визначати пріоритети зберігання даних.

Необхідно також враховувати аспекти безпеки при роботі з кешем. Зберігання чутливих даних у кеші може стати вразливим місцем для атаки, тому необхідно застосовувати додаткові заходи для захисту інформації, такі як шифрування даних та налаштування політик доступу. У системах із багаторівневим кешуванням важливо також враховувати можливість розсинхронізації даних між різними рівнями кешу. Це може призводити до того, що клієнт отримує застарілі дані, тому необхідно впроваджувати механізми інвалідації кешу, які автоматично оновлюють його при зміні вихідних даних.

Сучасні технології кешування також враховують потреби у масштабованості. Наприклад, у розподілених системах кешування, таких як Redis Cluster, можливо розподіляти дані між кількома вузлами для забезпечення високої доступності та відмовостійкості системи. Це особливо важливо для веб-платформ, які обслуговують мільйони користувачів одночасно.

Кешування активно використовується у різних галузях. У електронній комерції кешування дозволяє забезпечувати миттєвий доступ до даних про наявність товарів, ціни та відгуки, навіть під час пікових навантажень. У медіа-стрімінгових сервісах кешування відіграє ключову роль у забезпеченні безперебійного відтворення відео. У фінансових системах кешування використовується для оптимізації обробки транзакцій та аналізу даних у реальному часі.

Серед основних переваг кешування можна виділити: зменшення часу відповіді завдяки зберіганню даних у швидкодоступній пам'яті; зниження

навантаження на сервери через меншу кількість запитів до баз даних; покращення стабільності системи навіть у випадках високих навантажень або тимчасових збоїв.

Водночас кешування має свої виклики: керування обсягом даних та потреба у звільненні місця для нових записів; складність інтеграції багаторівневого кешування; проблеми синхронізації при частих змінах даних. Саме ці виклики зумовлюють необхідність розроблення моделей керування узгодженістю в гібридних кеш-системах, що є предметом даного дослідження.

1.2 Аналіз популярних систем кешування (Redis, Memcached, CDN)

Вибір технології кешування є критично важливим рішенням при проектуванні високонавантажених веб-систем. Наразі існує декілька провідних рішень, кожне з яких має свої особливості, переваги та обмеження. Розглянемо найбільш поширені системи кешування, що використовуються у розподілених веб-платформах.

Redis є однією з найпопулярніших систем кешування завдяки своїй високій продуктивності, багатофункціональності та підтримці різних структур даних. Redis — це сховище даних типу «ключ-значення», що працює в оперативній пам'яті. На відміну від простих систем кешування, Redis дозволяє зберігати не лише рядки, а й списки, множини, хеші, сортовані множини та інші структури даних. Redis також підтримує механізм TTL (Time to Live), що дозволяє встановлювати термін дії для кешованих даних та автоматично видаляти застарілі записи.

Основними перевагами Redis є висока швидкість обробки даних, що забезпечується завдяки зберіганню всіх даних у оперативній пам'яті. Redis здатний обробляти сотні тисяч операцій на секунду, що робить його ідеальним рішенням для високонавантажених систем. Підтримка кластеризації дозволяє розподіляти навантаження між кількома вузлами, забезпечуючи горизонтальне масштабування. Додаткові функції Redis включають підтримку Pub/Sub-механізмів для обміну повідомленнями між компонентами системи, транзакції для забезпечення

атомарності операцій та можливість виконання Lua-скриптів безпосередньо на сервері.

Redis активно використовується у багатьох великих компаніях, таких як Twitter, GitHub та Snapchat. Ці компанії застосовують Redis для кешування даних, обробки черг повідомлень та керування сесіями користувачів. Механізми персистентності Redis, такі як знімки (snapshots) та журнал AOF (Append-Only File), дозволяють зберігати дані на диску та швидко відновлювати їх після збою системи.

Однак Redis має й недоліки, зокрема залежність від оперативної пам'яті. Це означає, що великі обсяги даних можуть призвести до високих витрат на серверні ресурси. Використання Redis у середовищах із обмеженими ресурсами може бути економічно недоцільним, оскільки обсяг доступної пам'яті прямо впливає на кількість даних, що можуть бути закешовані.

Memcached є ще однією популярною системою кешування, що використовується для зберігання даних у вигляді пар «ключ-значення». На відміну від Redis, Memcached підтримує лише прості типи даних, що робить його менш гнучким, але дуже простим у налаштуванні та використанні. Memcached був розроблений компанією Danga Interactive для потреб сервісу LiveJournal і з того часу став одним із стандартних рішень для кешування у веб-додатках.

Основні переваги Memcached полягають у простоті реалізації та низьких витратах на ресурси. Memcached ефективно використовує пам'ять, застосовуючи алгоритм LRU для автоматичного видалення найстаріших записів при нестачі місця. Завдяки використанню лише оперативної пам'яті, Memcached забезпечує надзвичайно швидкий доступ до даних з латентністю менше 1 мілісекунди.

Memcached підходить для додатків, де потрібне кешування простих даних, таких як результати SQL-запитів, сесії користувачів або тимчасові токени. Він часто використовується у проектах із відкритим кодом завдяки своїй легкості та швидкості розгортання.

Недоліки Memcached включають обмежену функціональність та відсутність механізмів для збереження складних структур даних. Також Memcached не підтримує вбудовану кластеризацію — розподіл даних між вузлами реалізується на

стороні клієнта, що ускладнює масштабування. Відсутність персистентності означає, що всі дані втрачаються при перезапуску сервера.

CDN (Content Delivery Network) — це мережа географічно розподілених серверів, призначена для швидкої доставки контенту користувачам. CDN кешує статичні ресурси (зображення, відео, CSS, JavaScript) на серверах, розташованих якомога ближче до кінцевих користувачів, що значно зменшує затримки при завантаженні контенту.

Основними перевагами CDN є глобальний доступ та географічна близькість до користувачів. Коли користувач запитує ресурс, CDN автоматично направляє запит до найближчого сервера, що мінімізує мережеву латентність. CDN також забезпечує захист від DDoS-атак та зменшує навантаження на основні сервери веб-платформи.

Провідними постачальниками CDN-послуг є Cloudflare, Akamai, Amazon CloudFront та Fastly. Ці сервіси пропонують додаткові функції, такі як динамічне кешування, автоматична оптимізація зображень, SSL-шифрування та аналітика трафіку.

Недоліки CDN включають високу вартість для великих обсягів трафіку та складність інвалідації кешу. Оновлення контенту на всіх серверах CDN може займати певний час, що створює проблеми для динамічного контенту. CDN також не підходить для кешування персоналізованих даних користувачів.

У таблиці 1.2 наведено порівняльну характеристику розглянутих систем кешування.

Таблиця 1.2

Порівняльна характеристика систем кешування

Характеристика	**Redis**	**Memcached**	**CDN**
Тип даних	Складні структури	Ключ-значення	Статичні файли
Латентність	0.5–2 мс	0.5–1 мс	10–100 мс
Персистентність	Так	Ні	Так
Кластеризація	Вбудована	На клієнті	Автоматична

Продовження таблиці 1.2

Порівняльна характеристика систем кешування

Характеристика	**Redis**	**Memcached**	**CDN**
Масштабованість	Горизонтальна	Обмежена	Глобальна
Складність	Середня	Низька	Низька
Вартість	Середня	Низька	Висока

Аналіз існуючих систем кешування показує, що жодна з них окремо не здатна задовольнити всі потреби сучасних розподілених веб-платформ. Redis забезпечує гнучкість та швидкість, але має обмеження щодо обсягу даних. Memcached простий у використанні, але не підтримує складні структури даних. CDN ефективний для статичного контенту, але не підходить для динамічних даних. Це зумовлює необхідність застосування гібридних підходів, що поєднують переваги різних технологій кешування.

1.3 Особливості гібридного кешування

Практика розробки високонавантажених систем демонструє, що використання лише однієї технології кешування створює суттєві обмеження. Локальний кеш забезпечує блискавичний доступ, але його обсяг обмежений пам'яттю одного сервера. Розподілені рішення масштабуються краще, однак додають мережеву затримку до кожного запиту. CDN оптимальний для географічно розподіленої аудиторії, проте працює лише зі статичним контентом. Ці обмеження підштовхують архітекторів до комбінування різних підходів у єдину гібридну систему, де кожен компонент виконує ту роль, для якої він найбільш пристосований.

Гібридне кешування передбачає побудову багаторівневої архітектури, де кожен рівень відповідає за певний тип даних або сценарій використання. У такій системі локальний кеш на сервері додатків обробляє найчастіші запити з мінімальною затримкою, розподілений кеш забезпечує спільний доступ до даних

між кількома серверами, а CDN відповідає за доставку статичних ресурсів кінцевим користувачам.

Розглянемо типову чотирирівневу архітектуру гібридної системи кешування. Перший рівень — локальний in-memory кеш безпосередньо у процесі веб-додатку. Для Java-застосунків часто використовується бібліотека Caffeine, яка забезпечує доступ до даних за час менше 0.1 мілісекунди. Обсяг такого кешу обмежений декількома мегабайтами, тому сюди потрапляють лише найбільш затребувані дані.

Другий рівень — розподілений кеш на основі Redis. Цей рівень доступний усім серверам кластера, що дозволяє уникнути дублювання даних та забезпечити узгодженість. Латентність доступу становить 2-5 мілісекунд через мережеву взаємодію, проте обсяг кешу може сягати сотень мегабайт або навіть гігабайт.

Третій рівень призначений для кешування результатів складних SQL-запитів. Тут доцільно застосовувати Memcached, оскільки результати запитів зазвичай мають просту структуру і не потребують складних операцій. Цей рівень суттєво знижує навантаження на базу даних при повторюваних аналітичних запитах.

Четвертий рівень — CDN для статичних ресурсів. Зображення, відеофайли, скрипти та стилі кешуються на географічно розподілених серверах, забезпечуючи швидке завантаження сторінок незалежно від місцезнаходження користувача.

Принцип роботи гібридної системи базується на каскадному пошуку даних. При надходженні запиту система спочатку перевіряє локальний кеш. У разі відсутності даних (cache miss) запит передається на наступний рівень — розподілений кеш. Якщо дані відсутні і там, система звертається до бази даних, а отриманий результат записується у відповідні рівні кешу для подальшого використання.

Окрім каскадного пошуку, гібридні системи реалізують механізм просування даних між рівнями. Коли певний елемент починає запитуватися частіше, система автоматично переміщує його на швидший рівень кешу. І навпаки — рідко використовувані дані поступово витісняються на повільніші рівні або видаляються зовсім. Такий підхід дозволяє оптимально використовувати обмежені ресурси кожного рівня.

Переваги гібридного підходу є очевидними. По-перше, досягається баланс між швидкістю доступу та обсягом даних. Локальний кеш забезпечує миттєвий доступ до найпопулярніших елементів, тоді як розподілений кеш зберігає значно більший обсяг інформації. По-друге, система стає більш відмовостійкою — вихід з ладу одного рівня не призводить до повної втрати функціональності кешування. По-третє, різні типи даних обробляються найбільш ефективним способом.

Водночас гібридне кешування створює додаткові виклики. Найсуттєвішим є забезпечення узгодженості даних між рівнями. Коли дані оновлюються у базі даних, необхідно синхронізувати ці зміни на всіх рівнях кешу. Затримка в інвалідації може призвести до ситуації, коли користувачі отримують застарілу інформацію.

Іншим викликом є складність налаштування та моніторингу. Адміністратор системи повинен відстежувати стан кожного рівня окремо, аналізувати показники hit rate, контролювати використання пам'яті та вчасно реагувати на аномалії. Інструменти моніторингу на кшталт Prometheus та Grafana стають обов'язковими компонентами інфраструктури.

Вартість впровадження гібридного рішення також вища порівняно з однорівневим кешуванням. Потрібні додаткові серверні ресурси, ліцензії на програмне забезпечення, а також кваліфіковані спеціалісти для підтримки системи. Тому рішення про впровадження гібридного кешування має базуватися на ретельному аналізі потреб конкретного проекту.

У таблиці 1.3 наведено порівняння характеристик рівнів гібридної системи кешування.

Таблиця 1.3

Характеристики рівнів гібридної системи кешування

Рівень	**Технологія**	**Латентність**	**Обсяг**	**Призначення**
L1	Caffeine	~0.05 мс	5-50 МБ	Гарячі дані
L2	Redis	~2 мс	512 МБ – 8 ГБ	Спільний доступ

Продовження таблиці 1.3

Характеристики рівнів гібридної системи кешування

Рівень	**Технологія**	**Латентність**	**Обсяг**	**Призначення**
L3	Memcached	~3 мс	256 МБ – 2 ГБ	SQL-результати
L4	CDN	~50 мс	Необмежено	Статика

1.4 Порівняльний аналіз переваг та недоліків існуючих підходів

Для обґрунтованого вибору стратегії кешування необхідно детально проаналізувати сильні та слабкі сторони кожного підходу. Такий аналіз дозволяє визначити оптимальну конфігурацію системи залежно від специфіки конкретного проекту, очікуваного навантаження та бюджетних обмежень.

Однорівневе локальне кешування залишається найпростішим рішенням з точки зору впровадження та підтримки. Розробнику достатньо підключити бібліотеку на кшталт Caffeine або Guava Cache до свого Java-застосунку, налаштувати політику витіснення та максимальний обсяг пам'яті. Відсутність мережевої взаємодії гарантує мінімальну латентність — десяті частки мілісекунди. Проте такий підхід має принциповий недолік: кеш існує лише в межах одного процесу. У розподіленому середовищі, де запити користувача можуть оброблятися різними серверами, кожен сервер матиме власну копію кешу. Це призводить до неефективного використання пам'яті та проблем з консистентністю даних.

Розподілене кешування на основі Redis вирішує проблему спільного доступу. Усі сервери кластера звертаються до єдиного сховища, що виключає дублювання даних та спрощує інвалідацію. Redis надає багатий функціонал: підтримка різноманітних структур даних, атомарні операції, механізм публікації-підписки, можливість виконання скриптів на стороні сервера. Кластерний режим забезпечує горизонтальне масштабування та відмовостійкість через реплікацію. Водночас кожен запит до Redis проходить через мережу, що додає 1-5 мілісекунд затримки.

Для надшвидких операцій, де критична кожна мікросекунда, ця затримка може бути неприйнятною.

Memcached пропонує компроміс між простотою та функціональністю. Система спеціалізується виключно на кешуванні пар ключ-значення без додаткових можливостей Redis. Така спеціалізація робить Memcached надзвичайно ефективним у своїй ніші — кешуванні результатів SQL-запитів та сесій користувачів. Споживання пам'яті оптимізоване, а протокол взаємодії мінімалістичний. Недоліком є відсутність вбудованої кластеризації: розподіл даних між вузлами реалізується клієнтською бібліотекою за допомогою консистентного хешування. Також відсутня персистентність — перезапуск сервера означає повну втрату закешованих даних.

CDN-кешування кардинально відрізняється від попередніх підходів своєю географічною розподіленістю. Мережа серверів по всьому світу забезпечує доставку контенту з мінімальною затримкою незалежно від місцезнаходження користувача. Для глобальних сервісів із мільйонами користувачів на різних континентах CDN є обов'язковим компонентом інфраструктури. Додатковою перевагою стає захист від DDoS-атак та розвантаження основних серверів. Однак CDN ефективний переважно для статичного контенту. Кешування динамічних даних ускладнюється через затримки інвалідації — оновлення може поширюватися мережею від кількох секунд до хвилин. Вартість CDN-послуг залежить від обсягу трафіку і для великих проектів становить суттєву статтю витрат.

Гібридний підхід намагається поєднати переваги всіх попередніх рішень, мінімізуючи їхні недоліки. Локальний кеш першого рівня обробляє найчастіші запити миттєво. Розподілений кеш другого рівня забезпечує спільний доступ та узгодженість. CDN відповідає за статичні ресурси. Така архітектура демонструє найкращі показники продуктивності у більшості сценаріїв.

Проте гібридне кешування породжує власний набір складнощів. Насамперед це питання узгодженості між рівнями. Коли запис оновлюється у базі даних, зміни мають поширитися на всі рівні кешу. Якщо інвалідація відбувається несинхронно, виникає ситуація, коли різні компоненти системи працюють із різними версіями

даних. Для деяких застосунків — наприклад, фінансових систем — така неузгодженість є критичною помилкою.

Визначення оптимального рівня для зберігання конкретних даних також становить непросте завдання. Елемент, що активно використовувався вчора, сьогодні може стати непотрібним. Статичне налаштування правил розподілу не враховує динаміку патернів доступу. Потрібні адаптивні алгоритми, здатні автоматично переміщувати дані між рівнями на основі аналізу історії звернень.

Моніторинг гібридної системи потребує відстеження метрик кожного рівня окремо: hit rate, середній час відповіді, використання пам'яті, кількість витіснень. Аномалія на одному рівні може впливати на загальну продуктивність непередбачуваним чином. Налагодження та діагностика у багаторівневій системі значно складніші, ніж в однорівневій.

Якщо порівняти усі підходи за ключовими критеріями, картина виглядає наступним чином. За швидкістю доступу лідирують локальне та гібридне кешування, оскільки найчастіші запити обробляються без мережевої взаємодії. Redis та Memcached демонструють низьку, але не мінімальну латентність через необхідність передачі даних мережею. CDN має найвищу затримку серед розглянутих варіантів, що компенсується географічною близькістю до користувачів.

За критерієм масштабованості ситуація протилежна. CDN забезпечує практично необмежене горизонтальне масштабування завдяки глобальній мережі серверів. Redis із підтримкою кластеризації також добре масштабується. Локальне кешування обмежене ресурсами одного сервера і практично не масштабується.

Забезпечення узгодженості найкраще реалізовано у Redis та Memcached, де існує єдине джерело істини. Локальний кеш та CDN створюють множинні копії даних, що ускладнює підтримку їхньої актуальності. Гібридна система успадковує проблеми узгодженості від усіх своїх компонентів, тому потребує спеціальних механізмів синхронізації.

Вартість впровадження та експлуатації найнижча для локального кешування та Memcached. Redis потребує більших ресурсів через зберігання даних в

оперативній пам'яті та додатковий функціонал. CDN-послуги тарифікуються за обсягом трафіку, що для популярних сервісів означає значні щомісячні витрати. Гібридне рішення акумулює витрати всіх компонентів, але забезпечує оптимальне співвідношення ціни та якості при правильному налаштуванні.

Проведений аналіз засвідчує, що не існує універсального рішення для всіх випадків. Вибір підходу визначається конкретними вимогами проекту. Для невеликих монолітних застосунків достатньо локального кешу. Мікросервісні архітектури потребують розподіленого рішення. Глобальні платформи із мільйонами користувачів виграють від CDN. А високонавантажені системи з різнорідними патернами доступу отримують максимальну вигоду від гібридного підходу за умови правильної реалізації механізмів керування узгодженістю.

Саме проблема узгодженості даних у багаторівневих кеш-системах становить основний виклик при впровадженні гібридного кешування. Існуючі рішення переважно покладаються на ручне налаштування правил інвалідації або використовують прості стратегії на основі TTL. Такі підходи не враховують динамічну природу навантаження та патернів доступу. Розробка адаптивної моделі керування узгодженістю, здатної автоматично оптимізувати розподіл даних між рівнями, є актуальним науково-практичним завданням.

1.5 Проблема узгодженості даних у багаторівневих кеш-системах

Багаторівневе кешування створює фундаментальну проблему, що не виникає при використанні однорівневих рішень — підтримка узгодженості даних між різними рівнями кешу. Коли один і той самий елемент даних зберігається одночасно у локальному кеші кількох серверів, у розподіленому Redis та у CDN, будь-яка зміна цього елемента має коректно поширитися на всі копії. Інакше різні користувачі або навіть один користувач у різні моменти часу отримуватимуть різні версії даних.

Розглянемо типовий сценарій виникнення неузгодженості. Інтернет-магазин зберігає інформацію про залишки товарів у базі даних. Ці дані кешуються на

першому рівні у локальній пам'яті кожного з десяти серверів додатків, на другому рівні у Redis для спільного доступу. Покупець оформлює замовлення, і залишок товару зменшується на одиницю. База даних оновлена, але локальні кеші дев'яти інших серверів ще містять застарілу інформацію. Протягом короткого проміжку часу система може прийняти більше замовлень, ніж є товару на складі.

Проблема ускладнюється тим, що різні рівні кешу мають різні характеристики затримки оновлення. Локальний кеш можна інвалідувати майже миттєво через механізм публікації-підписки Redis. Оновлення CDN може тривати від секунд до хвилин залежно від провайдера та налаштувань. Протягом цього вікна неузгодженості система перебуває у нестабільному стані.

Існує кілька базових стратегій забезпечення узгодженості, кожна з яких має свої компроміси. Стратегія наскрізного запису (write-through) передбачає синхронне оновлення всіх рівнів кешу при кожній зміні даних. Запис у базу даних супроводжується негайним оновленням кешу. Такий підхід гарантує узгодженість, проте суттєво уповільнює операції запису. Кожна транзакція очікує підтвердження від усіх рівнів, що критично для високонавантажених систем.

Стратегія відкладеного запису (write-behind) працює протилежно. Зміни спочатку фіксуються у кеші, а потім асинхронно поширюються до бази даних та інших рівнів. Операції запису виконуються швидко, проте виникає ризик втрати даних у разі збою до завершення синхронізації. Крім того, протягом асинхронного оновлення різні рівні містять різні версії даних.

Стратегія інвалідації (cache invalidation) не оновлює кеш безпосередньо, а лише позначає відповідні записи як недійсні. При наступному запиті система виявляє відсутність актуальних даних у кеші та звертається до джерела. Цей підхід простіший у реалізації, але збільшує кількість звернень до бази даних після масових інвалідацій.

Механізм TTL (Time to Live) встановлює максимальний час життя запису у кеші. Після спливання цього терміну дані автоматично вважаються застарілими. TTL забезпечує eventual consistency — гарантію того, що всі копії врешті-решт стануть узгодженими. Проте протягом періоду TTL неузгодженість може існувати.

Короткий TTL зменшує вікно неузгодженості, але знижує ефективність кешування через часті промахи.

Жодна з базових стратегій не вирішує проблему повністю. Наскрізний запис занадто повільний для більшості застосунків. Відкладений запис ризикований. Проста інвалідація створює піки навантаження на базу даних. TTL забезпечує лише eventual consistency із непередбачуваним вікном неузгодженості.

Додатковим ускладненням є визначення того, які саме дані потребують суворої узгодженості, а які допускають тимчасову неузгодженість. Фінансові транзакції, залишки на рахунках, статуси замовлень вимагають негайної синхронізації. Водночас кількість переглядів сторінки, рейтинги товарів, кешовані результати пошуку можуть бути неактуальними протягом кількох секунд без критичних наслідків. Універсальна стратегія для всіх типів даних буде або занадто повільною, або недостатньо надійною.

Іншим аспектом проблеми є визначення оптимального рівня для розміщення конкретних даних. Елемент, що запитується сотні разів на секунду, має перебувати у найшвидшому локальному кеші. Дані з поодинокими зверненнями не варто тримати у дорогій оперативній пам'яті — їм місце на повільніших рівнях або взагалі поза кешем. Патерни доступу змінюються з часом: популярний сьогодні товар завтра може стати нецікавим. Статичні правила розподілу не враховують цю динаміку.

Просування даних між рівнями кешу потребує чітких критеріїв. За якою ознакою система визначає, що елемент став достатньо популярним для переміщення на швидший рівень? Скільки звернень протягом якого періоду є порогом? Як швидко реагувати на зміну патерну — миттєво чи з певним згладжуванням для уникнення осциляцій? Аналогічні питання виникають щодо витіснення рідко використовуваних даних на повільніші рівні.

Класичні алгоритми витіснення LRU та LFU не враховують специфіку багаторівневого кешування. LRU орієнтується лише на час останнього звернення, ігноруючи загальну частоту використання. Елемент, до якого зверталися тисячу разів учора, буде витіснений раніше за елемент з одним зверненням годину тому.

LFU навпаки — накопичує історичну статистику, не реагуючи на зміну патернів. Дані, популярні місяць тому, продовжують займати місце у кеші, хоча більше не використовуються.

Алгоритм LRU-K, що враховує час K останніх звернень до елемента, краще пристосований для адаптивного керування кешем. Він дозволяє розрізняти елементи з регулярним доступом та елементи з випадковими поодинокими зверненнями. Однак застосування LRU-K у контексті багаторівневого кешування потребує модифікації для врахування міжрівневих переміщень та синхронізації метаданих.

Таким чином, проблема узгодженості у багаторівневих кеш-системах є комплексною і охоплює кілька взаємопов'язаних аспектів: синхронізацію даних між рівнями при змінах, визначення оптимального рівня розміщення для кожного елемента, адаптивне переміщення даних відповідно до патернів доступу, вибір стратегії інвалідації залежно від типу даних. Існуючі рішення адресують ці аспекти окремо, без комплексного підходу. Розробка моделі керування узгодженістю, що інтегрує всі зазначені аспекти та адаптується до динаміки навантаження, є метою даного дослідження.

2 ТЕОРЕТИЧНІ ОСНОВИ МОДЕЛІ КЕРУВАННЯ УЗГОДЖЕНІСТЮ В ГІБРИДНИХ КЕШАХ

2.1 Математичні моделі оцінки пріоритету кешування

Ефективне керування багаторівневим кешем потребує формалізованого підходу до визначення пріоритетів зберігання даних. Інтуїтивні рішення на кшталт «часто використовувані дані тримаємо у швидкому кеші» не дають відповіді на ключові питання: наскільки часто є «часто», як порівнювати елементи з різними патернами доступу, коли переміщувати дані між рівнями. Математичні моделі дозволяють перевести ці інтуїції у конкретні алгоритми з чітко визначеною поведінкою.

Базовою характеристикою елемента кешу є частота звернень — кількість запитів до елемента за одиницю часу. Елементи з вищою частотою мають отримувати пріоритет при розміщенні на швидших рівнях кешу. Проте сама по собі частота не є достатньою метрикою. Елемент розміром 10 МБ із частотою 100 запитів на секунду споживає значно більше ресурсів, ніж елемент розміром 1 кілобайт із такою самою частотою.

Для врахування розміру вводиться поняття питомої цінності елемента (2.1):

$$V(i) = f(i) / s(i), \tag{2.1}$$

де $V(i)$ – питома цінність елемента i;

$f(i)$ – частота звернень до елемента i;

$s(i)$ – розмір елемента i у байтах.

Ця метрика показує, скільки запитів обслуговує одиниця пам'яті, виділена під даний елемент. Елементи з вищою питомою цінністю ефективніше використовують обмежені ресурси кешу.

Іншим важливим параметром є час доступу до елемента з різних рівнів кешу. Для типової чотирирівневої системи латентність становить: $L(1) \approx 0.05$ мс для

локального кешу, L(2) ≈ 2 мс для Redis, L(3) ≈ 3 мс для Memcached, L(4) ≈ 50 мс для CDN. Різниця між рівнями сягає трьох порядків, що робить правильний розподіл критично важливим для загальної продуктивності.

Виграш від кешування елемента на певному рівні визначається як економія часу порівняно із зверненням до джерела даних (2.2):

$$G(i, k) = f(i) \times (L(source) - L(k)), \quad (2.2)$$

де G(i, k) – виграш від кешування елемента i на рівні k;

f(i) – частота звернень до елемента i;

L(source) – латентність доступу до бази даних;

L(k) – латентність доступу до рівня кешу k.

Сумарний виграш від кешування усіх елементів характеризує ефективність системи в цілому.

Задача оптимального розподілу елементів між рівнями кешу формулюється як задача оптимізації. Маємо множину елементів $E = \{e_1, e_2, ..., e_n\}$, множину рівнів кешу K = {1, 2, ..., m} із обмеженнями на обсяг кожного рівня C(k). Потрібно знайти розподіл X, що максимізує сумарний виграш за умови дотримання обмежень на обсяг. Ця задача є варіантом задачі про рюкзак і належить до класу NP-складних, що унеможливлює знаходження точного розв'язку для великих систем за прийнятний час.

На практиці застосовуються евристичні підходи, що дають близькі до оптимальних результати. Жадібний алгоритм сортує елементи за питомою цінністю та послідовно розміщує їх на найшвидших рівнях, поки не вичерпається обсяг. Цей підхід має складність O(n log n) і забезпечує гарантовану якість не гіршу за 50% від оптимуму для класичної задачі про рюкзак.

Окремою складністю є динамічна природа частоти звернень. Значення f(i) не є константою — воно змінюється з часом залежно від поведінки користувачів, сезонності, маркетингових кампаній. Модель має відстежувати ці зміни та адаптувати розподіл елементів відповідно.

Найпростіший підхід — розрахунок частоти як кількості звернень за фіксований період (2.3):

$$f(i) = count(i, T) / T, \tag{2.3}$$

де f(i) – частота звернень до елемента i;

count(i, T) – кількість звернень до елемента i за період T;

T – тривалість періоду спостереження.

Вибір періоду є компромісом: короткий період швидко реагує на зміни, але дає нестабільні оцінки; довгий період забезпечує стабільність, але повільно адаптується.

Більш гнучким є експоненціальне згладжування, що надає більшої ваги недавнім спостереженням (2.4):

$$f(i, t) = \alpha \times x(i, t) + (1 - \alpha) \times f(i, t-1), \tag{2.4}$$

де f(i, t) – згладжена оцінка частоти елемента i в момент t;

x(i, t) – індикатор звернення (1 якщо було звернення, 0 інакше);

α – параметр згладжування, $\alpha \in (0, 1)$;

f(i, t−1) – попередня оцінка частоти.

Параметр α контролює швидкість адаптації. При α близькому до 1 модель реагує переважно на останні події; при α близькому до 0 враховується довга історія.

Для багаторівневого кешування важливо моделювати не лише частоту, але й характер звернень. Елемент може мати високу середню частоту, але нерегулярний патерн — наприклад, сплески активності з тривалими паузами. Інший елемент із такою самою середньою частотою може мати рівномірний розподіл звернень у часі. Для прийняття рішень про кешування ці ситуації потребують різних підходів.

Модель пуассонівського потоку запитів припускає, що звернення до елемента відбуваються незалежно з постійною інтенсивністю λ(i). Інтервали між зверненнями мають експоненціальний розподіл. Ця модель добре описує

агреговану поведінку великої кількості користувачів і дозволяє застосовувати апарат теорії масового обслуговування для аналізу системи.

Проте реальні патерни доступу часто відхиляються від пуассонівської моделі. Спостерігаються кореляції між послідовними запитами, періодичні коливання навантаження, аномальні сплески. Для врахування цих ефектів застосовуються складніші моделі: модульовані пуассонівські процеси, авторегресійні моделі, моделі на основі ланцюгів Маркова.

Важливим поняттям є робоча множина (working set) — підмножина елементів, до яких активно звертаються протягом певного періоду. Розмір робочої множини визначає мінімальний обсяг кешу, необхідний для ефективної роботи. Якщо кеш менший за робочу множину, hit rate буде низьким незалежно від алгоритму витіснення. Моделювання динаміки робочої множини дозволяє прогнозувати потреби системи та планувати ресурси.

Для оцінки ефективності кешування використовується метрика hit rate (2.5):

$$HR = hits / (hits + misses), \tag{2.5}$$

де HR – показник влучень кешу;

hits – кількість успішних звернень до кешу;

misses – кількість промахів кешу.

Для багаторівневого кешу визначається hit rate кожного рівня окремо. Загальна латентність системи залежить від комбінації цих показників (2.6):

$$L\_avg = \Sigma\, HR(k) \times L(k) + (1 - \Sigma\, HR(k)) \times L(source), \tag{2.6}$$

де L_avg – середня латентність системи;

HR(k) — частка запитів, що обслуговуються саме рівнем k (так що $\Sigma\, HR(k) \leq 1$);

L(k) – латентність рівня k;

L(source) – латентність джерела даних.

Математичні моделі оцінки пріоритету кешування формують теоретичну основу для розробки алгоритмів керування. Вони дозволяють кількісно порівнювати альтернативні стратегії та прогнозувати поведінку системи при різних навантаженнях. У наступному підрозділі розглянуто конкретні алгоритми витіснення, що базуються на цих моделях.

2.2 Алгоритми витіснення даних (LRU, LFU, LRU-K)

Алгоритм витіснення визначає, який елемент буде видалено з кешу при необхідності звільнення місця для нових даних. Вибір алгоритму суттєво впливає на ефективність кешування, оскільки неправильне витіснення призводить до видалення корисних даних та збільшення кількості промахів. Розглянемо основні алгоритми витіснення, їх властивості та обмеження.

Алгоритм LRU (Least Recently Used) базується на припущенні про темпоральну локальність: якщо до елемента нещодавно зверталися, ймовірність повторного звернення у найближчому майбутньому є високою. Відповідно, при необхідності витіснення обирається елемент, до якого найдовше не було звернень.

Реалізація LRU передбачає відстеження часу останнього доступу для кожного елемента. Оцінка пріоритету витіснення обчислюється як різниця між поточним часом та часом останнього звернення (2.7):

$$Score\_LRU(i) = t\_current - t\_last(i), \tag{2.7}$$

де Score_LRU(i) – оцінка пріоритету витіснення елемента i;

t_current – поточний час;

t_last(i) – час останнього звернення до елемента i.

Елемент із найбільшим значенням Score_LRU обирається для витіснення.

Класична реалізація LRU використовує комбінацію хеш-таблиці та двозв'язного списку. Хеш-таблиця забезпечує доступ до елемента за ключем за час O(1). Двозв'язний список підтримує порядок елементів за часом доступу: при

кожному зверненні елемент переміщується на початок списку, а витіснення відбувається з кінця. Обидві операції виконуються за константний час.

Перевагою LRU є простота реалізації та інтуїтивна зрозумілість. Алгоритм добре працює при наявності темпоральної локальності, коли недавно використані дані дійсно мають вищу ймовірність повторного використання. Для багатьох веб-застосунків ця властивість справджується: користувач, що переглядає товар, з високою ймовірністю запитає його повторно протягом сесії.

Проте LRU має суттєвий недолік — він не враховує частоту звернень. Елемент, до якого зверталися тисячу разів протягом останньої години, буде витіснений раніше за елемент з одним зверненням п'ять хвилин тому. Це робить LRU вразливим до сканування: одноразове звернення до великої кількості нових елементів може витіснити з кешу всі корисні дані.

Алгоритм LFU (Least Frequently Used) вирішує цю проблему, враховуючи частоту звернень замість часу останнього доступу. Витісняється елемент із найменшою кількістю звернень (2.8):

$$\text{Score\_LFU}(i) = \text{count}(i), \tag{2.8}$$

де Score_LFU(i) – лічильник звернень до елемента i;

count(i) – кількість звернень до елемента i.

Елемент із найменшим значенням Score_LFU витісняється першим.

LFU краще захищений від сканування: одноразові звернення не впливають на елементи з накопиченою історією частих звернень. Алгоритм ефективний для стабільних патернів доступу, коли популярність елементів змінюється повільно.

Недоліком LFU є повільна адаптація до змін. Елемент, що був популярним місяць тому, зберігає високий лічильник звернень і залишається у кеші, навіть якщо більше не використовується. Нові елементи з потенційно високою популярністю витісняються раніше за застарілі через низький початковий лічильник. Ця проблема відома як забруднення кешу (cache pollution).

Для подолання недоліків LFU застосовується механізм старіння лічильників. Періодично всі лічильники зменшуються вдвічі або на фіксовану величину. Це

дозволяє поступово забувати давню історію та надавати шанс новим елементам. Проте налаштування періоду старіння є компромісом: часте старіння наближає поведінку до LRU, рідке — зберігає проблеми класичного LFU.

Алгоритм LRU-K поєднує переваги LRU та LFU, враховуючи час K останніх звернень до елемента. Замість одного значення t_last алгоритм зберігає історію з K останніх моментів доступу. Оцінка пріоритету витіснення базується на часі K-го з кінця звернення (2.9):

$$\text{Score\_LRU-K}(i) = t\_current - t\_k(i), \tag{2.9}$$

де Score_LRU-K(i) – оцінка пріоритету витіснення елемента i;

t_current – поточний час;

t_k(i) – час K-го з кінця звернення до елемента i.

Якщо елемент має менше K звернень, вважається що t_k(i) = 0, тобто Score_LRU-K(i) = t_current, що є максимальним значенням — такий елемент витісняється першим як недостатньо перевірений.

Типове значення K = 2 забезпечує баланс між складністю реалізації та якістю рішень. При K = 2 алгоритм фактично оцінює інтервал між двома останніми зверненнями. Елементи з регулярними зверненнями мають короткий інтервал і низький пріоритет витіснення. Елементи з поодинокими зверненнями мають довгий інтервал або взагалі лише одне звернення, що робить їх першими кандидатами на витіснення.

Перевага LRU-K над LRU полягає у здатності розрізняти елементи з регулярним доступом та елементи з випадковими поодинокими зверненнями. Сканування великої кількості нових елементів не витісняє корисні дані, оскільки нові елементи мають лише одне звернення і низький пріоритет збереження.

Перевага LRU-K над LFU полягає у природній адаптації до змін. Історія обмежена K останніми зверненнями, тому давні патерни автоматично забуваються. Немає потреби у налаштуванні механізму старіння.

Складність реалізації LRU-K дещо вища за класичний LRU. Необхідно зберігати K часових міток для кожного елемента та підтримувати їх при кожному

зверненні. Для K = 2 накладні витрати на пам'ять становлять лише одну додаткову часову мітку на елемент, що є прийнятним для більшості застосунків.

Існують також гібридні алгоритми, що комбінують різні підходи. Алгоритм ARC (Adaptive Replacement Cache) динамічно балансує між LRU та LFU залежно від патерну навантаження. LIRS (Low Inter-reference Recency Set) розділяє елементи на дві групи за частотою звернень і застосовує різні політики витіснення до кожної групи.

Для багаторівневого кешування алгоритм витіснення має враховувати не лише видалення з кешу, але й переміщення між рівнями. Елемент, витіснений з першого рівня, може бути переміщений на другий рівень замість повного видалення. Це потребує модифікації класичних алгоритмів для підтримки міжрівневих операцій.

Критерії переміщення на вищий рівень мають враховувати частоту доступу та розмір елемента. Елемент доцільно просувати на швидший рівень, якщо виконуються умови (2.10):

$$\text{count}(i) > \text{threshold\_promote AND size}(i) < \text{max\_size\_L1}, \qquad (2.10)$$

де count(i) – кількість звернень до елемента i;

threshold_promote – поріг просування;

size(i) – розмір елемента i;

max_size_L1 – максимальний розмір елемента для першого рівня.

Критерії витіснення на нижчий рівень є симетричними: елемент переміщується вниз при низькій частоті доступу або тривалій відсутності звернень.

Порівняльний аналіз алгоритмів витіснення показує, що жоден з них не є універсально найкращим. LRU оптимальний для навантажень з вираженою темпоральною локальністю. LFU ефективний при стабільних патернах з чітко визначеною популярністю елементів. LRU-K забезпечує найкращий баланс для змішаних навантажень, характерних для веб-платформ.

Вибір алгоритму залежить від характеристик конкретного застосунку. Аналіз реальних патернів доступу дозволяє визначити домінуючий тип локальності та

обрати відповідний алгоритм. Для гібридних кеш-систем, що є предметом даного дослідження, LRU-K є найбільш перспективним вибором завдяки його адаптивності та здатності ефективно працювати з різнорідними даними.

2.3 Модель адаптивного керування узгодженістю на основі LRU-K

Розглянуті у попередніх підрозділах математичні моделі та алгоритми витіснення формують основу для побудови комплексної моделі керування узгодженістю у гібридних кеш-системах. Запропонована модель інтегрує механізми оцінки пріоритетів, міжрівневого переміщення даних та синхронізації змін у єдину адаптивну систему.

Модель базується на алгоритмі LRU-K з параметром K = 2, що забезпечує оптимальний баланс між складністю реалізації та якістю рішень. Вибір K = 2 обґрунтовується результатами досліджень, які демонструють, що збільшення K понад 2 дає незначний приріст ефективності при суттєвому зростанні накладних витрат на зберігання історії звернень.

Центральним елементом моделі є функція оцінки пріоритету елемента, що визначає його оптимальний рівень розміщення у багаторівневому кеші. На відміну від класичного LRU-K, який лише визначає порядок витіснення, запропонована модель розширює оцінку для підтримки просування та пониження елементів між рівнями.

Комплексна оцінка пріоритету елемента обчислюється за формулою (2.11):

$$P(i) = w_1 \times F(i) + w_2 \times R(i) + w_3 \times S(i), \quad (2.11)$$

де P(i) – комплексний пріоритет елемента i;

F(i) – нормалізована оцінка частоти звернень;

R(i) – нормалізована оцінка регулярності звернень;

S(i) – нормалізована оцінка розміру елемента;

w_1, w_2, w_3 – вагові коефіцієнти, $w_1 + w_2 + w_3 = 1$.

Компонента частоти F(i) базується на кількості звернень за останній період спостереження. Для уникнення впливу аномальних сплесків застосовується експоненціальне згладжування (2.12):

$$F(i) = \alpha \times f\_current(i) + (1 - \alpha) \times F\_prev(i), \quad (2.12)$$

де F(i) – згладжена оцінка частоти;

f_current(i) – частота звернень за поточний період;

F_prev(i) – попередня згладжена оцінка;

α – коефіцієнт згладжування, типово α = 0.3.

Компонента регулярності R(i) оцінює стабільність патерну звернень на основі інтервалу між двома останніми зверненнями. Регулярні звернення з коротким інтервалом отримують високу оцінку (2.13):

$$R(i) = 1 / (1 + (t\_current - t_2(i)) / T\_norm), \quad (2.13)$$

де R(i) – оцінка регулярності елемента i;

t_current – поточний час;

$t_2(i)$ – час передостаннього звернення до елемента i;

T_norm – нормалізуючий часовий інтервал.

Компонента розміру S(i) враховує ефективність використання пам'яті. Менші елементи отримують вищу оцінку, оскільки забезпечують кращу питому цінність (2.14):

$$S(i) = 1 - size(i) / size\_max, \quad (2.14)$$

де S(i) – нормалізована оцінка розміру;

size(i) – розмір елемента i у байтах;

size_max – максимальний розмір елемента у системі.

Вагові коефіцієнти w_1, w_2, w_3 визначають відносну важливість кожної компоненти та можуть налаштовуватися залежно від специфіки застосунку. Типові

значення для веб-платформ: $w_1 = 0.5$, $w_2 = 0.35$, $w_3 = 0.15$, що надає найбільшої ваги частоті звернень при помірному врахуванні регулярності та розміру.

На основі комплексного пріоритету модель визначає оптимальний рівень кешу для кожного елемента. Простір значень пріоритету поділяється на зони, що відповідають рівням кешу (2.15):

$$\text{Level}(i) = k, \text{ якщо } \theta\_(k-1) \leq P(i) < \theta\_k, \quad (2.15)$$

де Level(i) – оптимальний рівень для елемента i;

θ_k – порогові значення для рівня k;

$\theta_0 = 0$, $\theta\_m = 1$ для m рівнів кешу.

Порогові значення встановлюються пропорційно до обсягів рівнів кешу. Для типової конфігурації з чотирма рівнями та обсягами L1 = 5 МБ, L2 = 512 МБ, L3 = 256 МБ, L4 = необмежено, порогові значення складають: $\theta_1 = 0.85$, $\theta_2 = 0.60$, $\theta_3 = 0.30$.

Механізм просування елементів на вищий рівень активується при виконанні умов (2.16):

$$\text{Promote}(i, k \rightarrow k-1), \text{ якщо } P(i) \geq \theta\_(k-1) \text{ AND } \text{size}(i) \leq C\_(k-1) \times \beta, \quad (2.16)$$

де Promote(i, k → k−1) – операція просування елемента i з рівня k на рівень k−1;

$C\_(k-1)$ – доступний обсяг на рівні k−1;

β – коефіцієнт резервування, типово $\beta = 0.1$.

Коефіцієнт резервування β забезпечує наявність вільного простору на цільовому рівні для уникнення негайного витіснення щойно просунутих елементів.

Механізм пониження елементів працює симетрично (2.17):

$$\text{Demote}(i, k \rightarrow k+1), \text{ якщо } P(i) < \theta\_k \text{ OR } t\_\text{current} - t_1(i) > \text{TTL}\_k, \quad (2.17)$$

де Demote(i, k → k+1) – операція пониження елемента i з рівня k на рівень k+1;

$t_l(i)$ – час останнього звернення до елемента i;

TTL_k – максимальний час перебування на рівні k без звернень.

Модель включає механізм синхронізації для забезпечення узгодженості даних між рівнями. При оновленні елемента у базі даних генерується подія інвалідації, що поширюється на всі рівні кешу через механізм публікації-підписки.

Час поширення інвалідації залежить від рівня кешу. Для локального кешу інвалідація відбувається майже миттєво через внутрішній механізм повідомлень. Для розподіленого кешу затримка визначається мережевою латентністю. Модель враховує ці затримки при оцінці узгодженості системи (2.18):

$$T\_consistency = \max(T\_invalidate(k)) \text{ для всіх } k, \quad (2.18)$$

де T_consistency – час досягнення повної узгодженості;

T_invalidate(k) – час інвалідації на рівні k.

Для критичних даних, що потребують суворої узгодженості, модель передбачає режим синхронної інвалідації. Операція запису блокується до підтвердження інвалідації на всіх рівнях. Для некритичних даних застосовується асинхронна інвалідація з гарантією eventual consistency.

Класифікація даних за рівнем критичності виконується на етапі конфігурації системи. Адміністратор визначає патерни ключів, що відповідають критичним даним. Типові приклади критичних даних: залишки товарів, баланси рахунків, статуси транзакцій. Некритичні дані включають: статистику переглядів, кешовані результати пошуку, персоналізовані рекомендації.

Адаптивність моделі забезпечується періодичним перерахунком вагових коефіцієнтів та порогових значень на основі аналізу ефективності. Метрикою ефективності є сумарний hit rate системи з урахуванням латентності рівнів (2.19):

$$E = \Sigma\, HR(k) \times (1 - L(k) / L\_max) \text{ для всіх } k, \quad (2.19)$$

де E – метрика ефективності;

HR(k) – hit rate рівня k;

L(k) – латентність рівня k;

L_max – максимальна латентність у системі.

При зниженні ефективності нижче заданого порогу система ініціює процедуру адаптації. Аналізуються патерни доступу за останній період, і вагові коефіцієнти коригуються для покращення розподілу елементів між рівнями.

Запропонована модель адаптивного керування узгодженістю забезпечує автоматичну оптимізацію роботи багаторівневого кешу без потреби у ручному налаштуванні. Комбінація механізмів оцінки пріоритетів, міжрівневого переміщення та синхронізації дозволяє досягти високої продуктивності при збереженні узгодженості даних. У наступному підрозділі детально розглянуто критерії прийняття рішень про просування та витіснення елементів.

2.4 Критерії просування та витіснення даних між рівнями кешу

Ефективність багаторівневої системи кешування визначається якістю рішень про переміщення даних між рівнями. Елементи мають потрапляти на найшвидший рівень, що відповідає їхній популярності, та своєчасно звільняти місце для більш затребуваних даних. У цьому підрозділі деталізовано критерії прийняття рішень про просування та витіснення елементів у контексті запропонованої моделі.

Просування елемента на вищий рівень кешу є операцією підвищення пріоритету обслуговування. Елемент переміщується з повільнішого рівня на швидший, що зменшує середню латентність доступу. Проте швидші рівні мають менший обсяг, тому просування можливе лише для обмеженої кількості найбільш цінних елементів.

Базовим критерієм просування є досягнення порогової кількості звернень за період спостереження (2.20):

$$C_1: \text{count}(i, T) \geq N\_promote, \tag{2.20}$$

де count(i, T) – кількість звернень до елемента i за період T;

N_promote – порогова кількість звернень для просування.

Для типової конфігурації з періодом спостереження T = 60 секунд порогове значення N_promote = 5 забезпечує просування елементів із частотою доступу понад 5 запитів на хвилину.

Другим критерієм є стабільність патерну звернень. Елемент з регулярними зверненнями має вищий пріоритет просування, ніж елемент з аномальним сплеском активності (2.21):

$$C_2: \sigma(\text{intervals}(i)) \leq \sigma\_\text{max}, \tag{2.21}$$

де σ(intervals(i)) – стандартне відхилення інтервалів між зверненнями;

σ_max – максимально допустиме відхилення для просування.

Цей критерій захищає швидкі рівні кешу від забруднення елементами з нестабільною популярністю. Раптовий сплеск звернень до елемента не призводить до негайного просування — система очікує підтвердження стабільності патерну.

Третій критерій обмежує розмір елементів, що просуваються на швидші рівні (2.22):

$$C_3: \text{size}(i) \leq \text{max\_size}(k-1), \tag{2.22}$$

де size(i) – розмір елемента i;

max_size(k−1) – максимальний розмір елемента для цільового рівня.

Обмеження на розмір запобігає ситуації, коли один великий елемент займає значну частину обмеженого обсягу швидкого рівня. Типові обмеження: для L1 — 100 КБ, для L2 — 1 МБ, для L3 — 10 МБ.

Четвертий критерій перевіряє наявність вільного простору на цільовому рівні (2.23):

$$C_4: \text{free\_space}(k-1) \geq \text{size}(i) \times (1 + \text{reserve\_ratio}), \tag{2.23}$$

де free_space(k−1) – вільний простір на цільовому рівні;

reserve_ratio – коефіцієнт резервування, типово 0.2.

Коефіцієнт резервування забезпечує буфер вільного простору для обробки короткочасних сплесків навантаження без негайного витіснення.

Рішення про просування приймається при виконанні всіх чотирьох критеріїв одночасно (2.24):

$$\text{Promote}(i) = C_1 \text{ AND } C_2 \text{ AND } C_3 \text{ AND } C_4. \quad (2.24)$$

Витіснення елемента на нижчий рівень є зворотною операцією. Елемент переміщується з швидшого рівня на повільніший для звільнення місця під більш затребувані дані. На відміну від повного видалення з кешу, витіснення на нижчий рівень зберігає елемент у системі, що дозволяє обслуговувати подальші запити без звернення до бази даних.

Першим критерієм витіснення є тривала відсутність звернень до елемента (2.25):

$$D_1: t\_current - t\_last(i) > idle\_threshold(k), \quad (2.25)$$

де t_current – поточний час;

t_last(i) – час останнього звернення до елемента i;

idle_threshold(k) – поріг простою для рівня k.

Пороги простою встановлюються диференційовано для кожного рівня: для L1 — 5 хвилин, для L2 — 30 хвилин, для L3 — 2 години. Швидші рівні мають коротші пороги, оскільки їхній обсяг обмежений і простоюючі елементи блокують просування активних даних.

Другий критерій базується на зниженні частоти звернень нижче порогового значення (2.26):

$$D_2: count(i, T) < N\_demote(k), \quad (2.26)$$

де count(i, T) – кількість звернень за період T;

N_demote(k) – порогова кількість для витіснення з рівня k.

Порогові значення N_demote менші за N_promote для створення гістерезису. Елемент, просунутий при 5 зверненнях на хвилину, витісняється лише при зниженні до 2 звернень. Гістерезис запобігає осциляціям — постійному переміщенню елемента між рівнями при граничних значеннях частоти.

Третій критерій активується при нестачі простору на поточному рівні (2.27):

$$D_3: \text{used\_space}(k) > \text{capacity}(k) \times \text{overflow\_threshold}, \quad (2.27)$$

де used_space(k) – використаний простір на рівні k;

capacity(k) – загальний обсяг рівня k;

overflow_threshold – поріг переповнення, типово 0.95.

При досягненні 95% заповнення рівня система ініціює превентивне витіснення найменш пріоритетних елементів для забезпечення простору під нові дані.

Вибір елемента для витіснення здійснюється на основі комплексного пріоритету, розрахованого за формулою (2.11). Елемент із найнижчим пріоритетом серед тих, що задовольняють хоча б один критерій витіснення, обирається першим (2.28):

$$\text{Victim}(k) = \text{argmin } P(i), \text{ де } i \in \text{Level}(k) \text{ AND } (D_1 \text{ OR } D_2 \text{ OR } D_3). \quad (2.28)$$

Окремим випадком є повне видалення елемента з кешу. Ця операція застосовується до елементів на найнижчому рівні, яким немає куди понижуватися, або до елементів з критично низьким пріоритетом на будь-якому рівні (2.29):

$$\text{Evict}(i), \text{ якщо } \text{Level}(i) = L\_max \text{ OR } P(i) < P\_min, \quad (2.29)$$

де Level(i) – поточний рівень елемента;

L_max – найнижчий рівень кешу;

P_min – мінімальний пріоритет для збереження у кеші.

Поріг P_min встановлюється на рівні 0.05, що відповідає елементам із поодинокими зверненнями та великим розміром.

Особливої уваги потребує обробка ситуацій конкуренції за ресурси. При одночасному надходженні кількох запитів на просування система має визначити пріоритет виконання. Запроваджується черга просування з ранжуванням за комплексним пріоритетом (2.30):

$$Queue\_promote = sort(candidates, by = P(i), descending). \quad (2.30)$$

Елементи з вищим пріоритетом просуваються першими, що гарантує оптимальне використання обмеженого простору швидких рівнів.

Аналогічно організується черга витіснення при масовому надходженні нових даних (2.31):

$$Queue\_demote = sort(candidates, by = P(i), ascending). \quad (2.31)$$

Елементи з найнижчим пріоритетом витісняються першими, звільняючи місце для більш цінних даних.

Для запобігання каскадним витісненням при різких змінах навантаження запроваджується механізм обмеження швидкості переміщень. Максимальна кількість операцій просування та витіснення за одиницю часу обмежується параметрами rate_limit_promote та rate_limit_demote. Типові значення: 100 операцій на секунду для просування, 200 операцій на секунду для витіснення.

Моніторинг ефективності критеріїв здійснюється через аналіз метрик: частка просунутих елементів, що залишаються на вищому рівні понад 1 хвилину; частка витіснених елементів, що запитуються повторно протягом 5 хвилин. Високі значення першої метрики та низькі значення другої свідчать про правильне налаштування критеріїв.

При виявленні відхилень система автоматично коригує порогові значення. Якщо значна частка просунутих елементів швидко витісняється назад, поріг N_promote збільшується. Якщо витіснені елементи часто запитуються повторно, поріг idle_threshold подовжується. Такий зворотний зв'язок забезпечує адаптацію системи до змін у патернах навантаження.

3 ПРОЕКТУВАННЯ АРХІТЕКТУРИ ГІБРИДНОЇ КЕШ-СИСТЕМИ

3.1 Архітектура багаторівневої гібридної системи кешування

Практична реалізація моделі керування узгодженістю потребує ретельно спроектованої архітектури, що забезпечує взаємодію між рівнями кешу, підтримку механізмів синхронізації та можливість масштабування. У цьому розділі представлено архітектуру багаторівневої гібридної системи кешування, розробленої на основі теоретичних положень попереднього розділу.

Архітектура системи базується на чотирирівневій моделі, де кожен рівень оптимізований для певного класу даних та сценаріїв доступу. Такий підхід дозволяє ефективно використовувати ресурси кожного рівня, забезпечуючи при цьому мінімальну латентність для найбільш затребуваних даних.

Перший рівень (L1) реалізується як локальний in-memory кеш безпосередньо у адресному просторі веб-додатку. Для Java-застосунків обрано бібліотеку Caffeine, що забезпечує найвищу продуктивність серед існуючих рішень для локального кешування. Caffeine використовує алгоритм Window TinyLFU, який поєднує переваги LRU та LFU, демонструючи hit rate, близький до оптимального при значно нижчих накладних витратах.

Характеристики першого рівня визначаються обмеженнями оперативної пам'яті серверів додатків. Типова конфігурація передбачає обсяг 5-50 МБ на один екземпляр додатку. Латентність доступу становить 0.01-0.1 мілісекунди, що на два порядки швидше за мережеві рішення. На цьому рівні зберігаються найчастіше запитувані елементи невеликого розміру: конфігураційні параметри, довідникові дані, результати частих обчислень.

Другий рівень (L2) реалізується на основі Redis — розподіленого сховища даних, що працює в оперативній пам'яті. Redis забезпечує спільний доступ до кешованих даних для всіх серверів кластера, що вирішує проблему дублювання

даних у локальних кешах. Підтримка різноманітних структур даних дозволяє ефективно кешувати не лише прості значення, а й списки, множини, хеші.

Обсяг другого рівня визначається конфігурацією Redis-кластера і типово становить 512 МБ — 8 ГБ. Латентність доступу складає 1-5 мілісекунд залежно від мережевої інфраструктури та розміру даних. На цьому рівні розміщуються дані середньої популярності, що потребують спільного доступу: сесії користувачів, кешовані профілі, агреговані метрики.

Третій рівень (L3) спеціалізується на кешуванні результатів SQL-запитів та реалізується на основі Memcached. Вибір Memcached обумовлений його оптимізацією для зберігання простих пар ключ-значення з мінімальними накладними витратами. Результати складних аналітичних запитів до бази даних кешуються на цьому рівні, що суттєво знижує навантаження на СУБД.

Типовий обсяг третього рівня становить 256 МБ — 2 ГБ з латентністю 2-5 мілісекунд. Особливістю цього рівня є інтеграція з шаром доступу до даних: ORM-фреймворки автоматично перевіряють наявність результатів у кеші перед виконанням запиту до бази даних.

Четвертий рівень (L4) забезпечує кешування статичних ресурсів через мережу доставки контенту (CDN). Зображення, відеофайли, скрипти та стилі розміщуються на географічно розподілених серверах CDN-провайдера. Користувачі отримують ці ресурси з найближчого сервера, що мінімізує мережеву затримку.

Обсяг четвертого рівня фактично необмежений і визначається тарифним планом CDN-провайдера. Латентність доступу залежить від географічного розташування користувача і типово становить 10-100 мілісекунд. Цей рівень обслуговує статичний контент, що рідко змінюється і не потребує персоналізації.

Взаємодія між рівнями організована за принципом каскадного пошуку. При надходженні запиту на отримання даних система послідовно перевіряє рівні від найшвидшого до найповільнішого. Алгоритм пошуку має таку логіку: спочатку перевіряється локальний кеш L1; при відсутності даних запит передається до Redis

L2; якщо дані відсутні в L2, перевіряється Memcached L3; у разі промаху на всіх рівнях виконується запит до бази даних.

Знайдені дані записуються у відповідні рівні кешу згідно з їхнім пріоритетом. Елементи з високим пріоритетом потрапляють безпосередньо на швидкі рівні, елементи з низьким пріоритетом — на повільніші. Це забезпечує оптимальний розподіл даних без необхідності проходження через усі проміжні рівні.

Центральним компонентом архітектури є менеджер кешу (Cache Manager), що координує роботу всіх рівнів. Менеджер відповідає за маршрутизацію запитів, прийняття рішень про просування та витіснення елементів, синхронізацію даних при оновленнях. Реалізація менеджера базується на патерні Facade, що приховує складність взаємодії з різними технологіями кешування за єдиним інтерфейсом.

Компонент статистики (Statistics Collector) збирає метрики роботи кожного рівня: кількість звернень, hit rate, середню латентність, використання пам'яті. Ці метрики використовуються для моніторингу продуктивності та адаптивного налаштування параметрів системи. Статистика агрегується з періодичністю 10 секунд і зберігається для подальшого аналізу.

Компонент синхронізації (Sync Manager) забезпечує узгодженість даних між рівнями при оновленнях. При зміні даних у базі генерується подія інвалідації, що поширюється на всі рівні кешу через механізм публікації-підписки Redis. Локальні кеші підписані на відповідні канали та автоматично видаляють застарілі записи при отриманні повідомлення.

Компонент конфігурації (Configuration Manager) зберігає параметри роботи системи: порогові значення для просування та витіснення, обсяги рівнів, TTL для різних типів даних. Конфігурація може змінюватися динамічно без перезапуску системи, що забезпечує гнучкість налаштування у робочому середовищі.

Мережева топологія системи передбачає розгортання компонентів у кількох зонах доступності для забезпечення відмовостійкості. Сервери додатків із локальними кешами L1 розміщуються за балансувальником навантаження. Redis-кластер L2 розгортається у режимі реплікації з автоматичним failover. Memcached

L3 працює у режимі пулу серверів із консистентним хешуванням. CDN L4 забезпечується зовнішнім провайдером із гарантованою доступністю.

Масштабування системи здійснюється горизонтально на кожному рівні незалежно. Додавання серверів додатків збільшує сумарний обсяг локальних кешів L1. Розширення Redis-кластера підвищує обсяг та пропускну здатність L2. Збільшення пулу Memcached-серверів масштабує L3. CDN масштабується автоматично провайдером відповідно до трафіку.

Безпека системи забезпечується на кількох рівнях. Доступ до Redis та Memcached обмежений внутрішньою мережею та захищений автентифікацією. Чутливі дані шифруються перед записом у кеш. Механізм TTL гарантує автоматичне видалення застарілих даних, що зменшує ризик несанкціонованого доступу до історичної інформації.

Запропонована архітектура забезпечує гнучкий баланс між продуктивністю, масштабованістю та надійністю. Чотирирівнева структура дозволяє оптимально розподіляти дані відповідно до їхніх характеристик. Централізоване керування через менеджер кешу спрощує адміністрування та моніторинг. Механізми синхронізації гарантують узгодженість даних при збереженні високої продуктивності.

3.2 Проектування компонентів системи

Реалізація архітектури багаторівневого кешування потребує детального проектування окремих компонентів та визначення їхньої взаємодії. Кожен компонент відповідає за певний аспект функціонування системи і має чітко визначений інтерфейс для комунікації з іншими компонентами.

Центральним елементом системи є клас HybridCacheManager, що реалізує патерн Facade та забезпечує єдину точку доступу до функціональності кешування. Цей компонент приховує складність взаємодії з різними технологіями за простим інтерфейсом, надаючи методи для отримання, збереження та видалення даних.

Клієнтський код працює з абстракцією кешу, не знаючи про існування окремих рівнів та механізмів переміщення.

Інтерфейс HybridCacheManager включає базові операції роботи з кешем. Метод get приймає ключ та повертає значення, автоматично здійснюючи каскадний пошук по всіх рівнях. Метод put зберігає значення з автоматичним визначенням оптимального рівня на основі пріоритету елемента. Метод invalidate видаляє елемент з усіх рівнів кешу та ініціює поширення події інвалідації.

Компонент LocalCache інкапсулює роботу з бібліотекою Caffeine та реалізує функціональність першого рівня. Конфігурація Caffeine налаштовується через builder-патерн із зазначенням максимального обсягу, політики витіснення та часу життя записів. Для інтеграції з моделлю LRU-K компонент розширює стандартну поведінку Caffeine, додаючи збереження історії звернень для кожного елемента.

Внутрішня структура LocalCache включає основне сховище даних та допоміжну мапу для зберігання метаданих елементів. Метадані містять часові мітки останніх K звернень, лічильник загальної кількості звернень та розмір елемента. При кожному зверненні до елемента оновлюються відповідні метадані, що дозволяє коректно обчислювати пріоритет за формулою комплексної оцінки.

Компонент DistributedCache забезпечує взаємодію з Redis-кластером другого рівня. Реалізація базується на клієнтській бібліотеці Jedis або Lettuce, що надають високорівневий API для роботи з Redis. Компонент підтримує пул з'єднань для ефективного використання мережевих ресурсів та автоматичне відновлення при тимчасових збоях зв'язку.

Особливістю DistributedCache є підтримка серіалізації складних об'єктів. Java-об'єкти перетворюються у бінарний формат перед збереженням та відновлюються при читанні. Для оптимізації продуктивності застосовується бібліотека Kryo, що забезпечує компактне представлення та швидку серіалізацію порівняно зі стандартним механізмом Java.

Компонент QueryCache реалізує третій рівень на основі Memcached та спеціалізується на кешуванні результатів SQL-запитів. Ключем кешу слугує хеш SQL-запиту разом із параметрами, значенням — серіалізований результат

виконання. Інтеграція з ORM-фреймворком Hibernate здійснюється через механізм Second Level Cache, що автоматично перевіряє кеш перед зверненням до бази даних.

Компонент StatisticsCollector відповідає за збір та агрегацію метрик роботи системи. Для кожного рівня кешу відстежуються показники: кількість звернень, кількість влучень, кількість промахів, середня латентність, поточне використання пам'яті, кількість операцій витіснення. Метрики накопичуються у кільцевому буфері та періодично агрегуються для формування часових рядів.

Структура метрик організована ієрархічно. На верхньому рівні знаходяться агреговані показники всієї системи: загальний hit rate, середньозважена латентність, сумарний обсяг кешованих даних. На нижньому рівні представлені деталізовані метрики кожного рівня окремо. Така організація дозволяє швидко оцінити загальний стан системи та при потребі заглибитися у деталі конкретного рівня.

Компонент PromotionManager реалізує логіку просування елементів між рівнями відповідно до критеріїв, визначених у підрозділі 2.4. Періодично з інтервалом 10 секунд компонент аналізує метадані елементів на кожному рівні та формує список кандидатів на просування. Елементи, що задовольняють усі критерії просування, переміщуються на вищий рівень у порядку спадання пріоритету.

Процес просування виконується асинхронно у фоновому потоці для уникнення блокування основних операцій кешу. Черга просування обробляється послідовно з обмеженням швидкості для запобігання перевантаженню системи. При виникненні помилок під час переміщення елемент залишається на поточному рівні, а інформація про помилку записується у лог.

Компонент EvictionManager реалізує симетричну логіку витіснення елементів на нижчі рівні. Тригером витіснення може бути досягнення порогу заповнення рівня, тривала відсутність звернень до елемента або явний запит на звільнення простору. Елементи для витіснення обираються за найнижчим пріоритетом серед кандидатів.

Взаємодія PromotionManager та EvictionManager координується через спільний механізм блокувань. Одночасне просування та витіснення одного елемента заборонено для уникнення race conditions. Блокування реалізується на рівні окремих ключів, що мінімізує вплив на паралельність обробки різних елементів.

Компонент SyncManager забезпечує синхронізацію даних між рівнями при оновленнях. Основним механізмом синхронізації є публікація повідомлень через канали Redis Pub/Sub. При інвалідації елемента SyncManager публікує повідомлення з ключем елемента у спеціальний канал. Усі екземпляри додатку підписані на цей канал та обробляють повідомлення, видаляючи відповідні записи з локальних кешів.

Формат повідомлення інвалідації включає тип операції, ключ елемента, часову мітку та ідентифікатор джерела. Тип операції розрізняє повну інвалідацію та часткове оновлення. Часова мітка використовується для впорядкування повідомлень та ігнорування застарілих. Ідентифікатор джерела дозволяє уникнути повторної обробки власних повідомлень.

Для критичних даних SyncManager підтримує режим синхронної інвалідації. У цьому режимі операція оновлення блокується до отримання підтвердження інвалідації від усіх рівнів. Підтвердження збираються через окремий канал зворотного зв'язку з таймаутом очікування. При недосягненні кворуму підтверджень операція може бути повторена або завершена з попередженням.

Компонент ConfigurationManager централізує керування параметрами системи. Конфігурація зберігається у форматі YAML та завантажується при старті додатку. Підтримується ієрархічна структура з можливістю перевизначення параметрів для окремих рівнів або типів даних. Зміна конфігурації можлива без перезапуску через механізм гарячого перезавантаження.

Основні параметри конфігурації включають обсяги рівнів кешу, порогові значення для просування та витіснення, TTL за замовчуванням, налаштування підключень до Redis та Memcached, параметри серіалізації. Валідація конфігурації виконується при завантаженні з генерацією зрозумілих повідомлень про помилки.

Компонент HealthChecker періодично перевіряє доступність та працездатність усіх рівнів кешу. Для кожного рівня виконується тестова операція запису та читання з вимірюванням латентності. При виявленні недоступності рівня генерується попередження, а трафік автоматично перенаправляється на інші рівні. Відновлення працездатності рівня також фіксується для моніторингу.

Взаємодія компонентів організована через механізм впровадження залежностей (Dependency Injection). Фреймворк Spring забезпечує створення екземплярів компонентів, встановлення залежностей та керування життєвим циклом. Така архітектура спрощує тестування окремих компонентів через підміну залежностей mock-об'єктами.

Діаграма класів системи демонструє зв'язки між компонентами. HybridCacheManager агрегує LocalCache, DistributedCache та QueryCache. StatisticsCollector отримує метрики від усіх рівнів кешу. PromotionManager та EvictionManager використовують StatisticsCollector для прийняття рішень. SyncManager взаємодіє з усіма рівнями для поширення інвалідацій. ConfigurationManager надає параметри всім компонентам системи.

На рисунку 3.1 представлено діаграму класів системи, що демонструє зв'язки між компонентами.

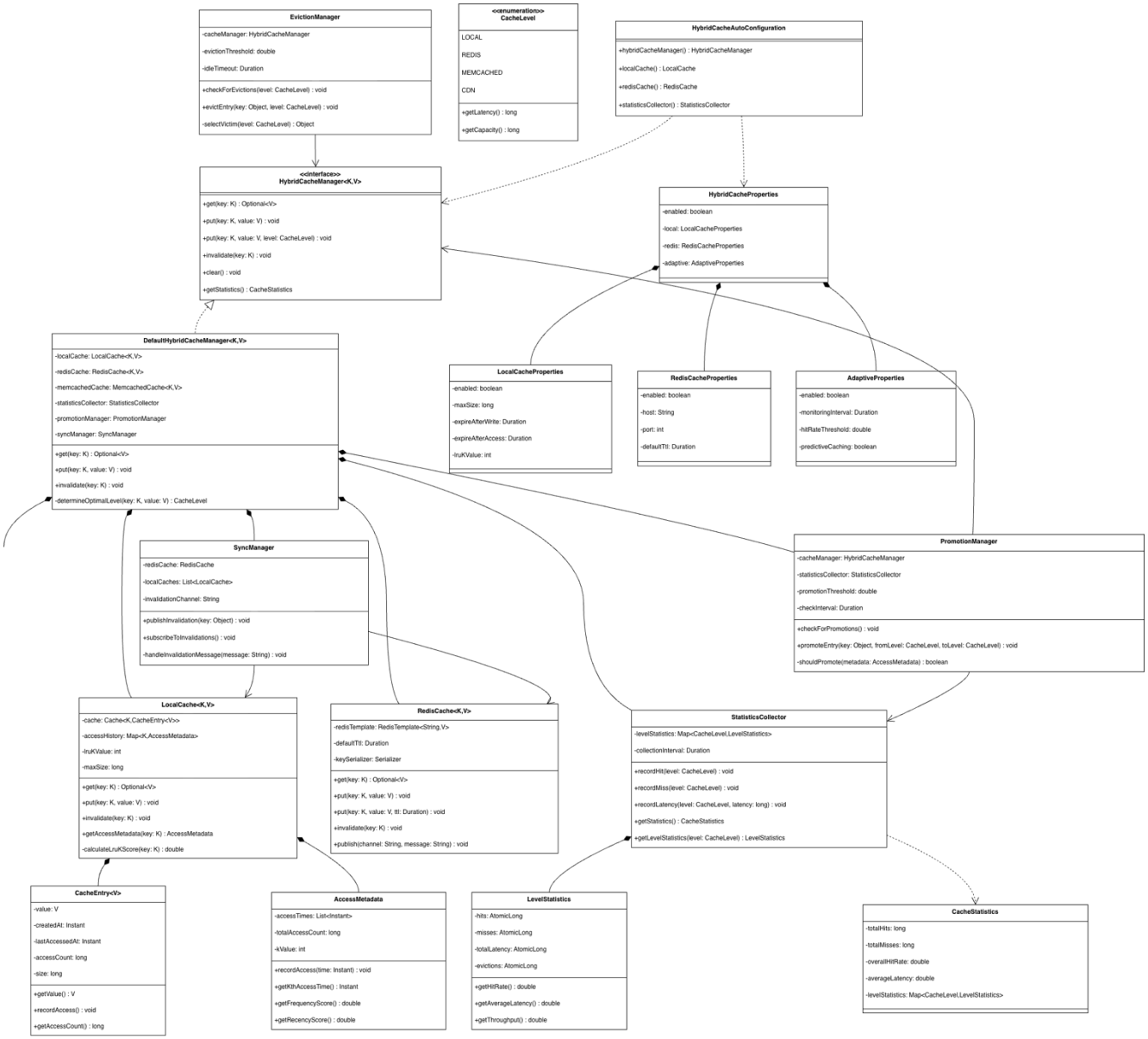


Рис. 3.1 Діаграма класів гібридної системи кешування

Спроектована компонентна архітектура забезпечує чіткий розподіл відповідальності, низьку зв'язність та високу згуртованість. Кожен компонент може розвиватися та тестуватися незалежно. Додавання нових рівнів кешу або зміна технологій реалізації потребує мінімальних змін у існуючому коді завдяки використанню абстракцій та інтерфейсів.

3.3 Механізми синхронізації та інвалідації даних

Забезпечення узгодженості даних між рівнями кешу є одним із ключових викликів багаторівневої архітектури. Коли дані оновлюються у базі даних, всі кешовані копії мають бути своєчасно інвалідовані або оновлені. Затримка у поширенні змін призводить до ситуації, коли різні компоненти системи працюють з різними версіями даних, що може спричинити некоректну поведінку застосунку.

У розробленій системі реалізовано комплексний механізм синхронізації, що поєднує кілька підходів залежно від критичності даних та вимог до продуктивності. Базовим механізмом є асинхронна інвалідація через систему публікації-підписки Redis Pub/Sub, що забезпечує оперативне поширення повідомлень про зміни на всі екземпляри застосунку.

Архітектура механізму синхронізації базується на патерні Observer. При оновленні даних у базі генерується подія інвалідації, що публікується у спеціальний канал Redis. Усі екземпляри застосунку підписані на цей канал та отримують повідомлення в реальному часі. Обробник повідомлень видаляє відповідний запис з локального кешу, забезпечуючи актуальність даних при наступному зверненні.

Формат повідомлення інвалідації включає необхідну інформацію для коректної обробки (3.1):

$$\text{Message} = \{\text{type}, \text{key}, \text{timestamp}, \text{source}, \text{version}\}, \tag{3.1}$$

де type – тип операції (INVALIDATE, UPDATE, DELETE);

key – ключ елемента, що інвалідується;

timestamp – часова мітка генерації повідомлення;

source – ідентифікатор джерела повідомлення;

version – версія даних для запобігання конфліктам.

Поле type визначає характер операції. Тип INVALIDATE вказує на необхідність видалення запису з кешу без заміни. Тип UPDATE передбачає

оновлення значення, якщо нове значення включено у повідомлення. Тип DELETE сигналізує про повне видалення даних з системи.

Часова мітка timestamp використовується для впорядкування повідомлень та відкидання застарілих. Якщо повідомлення надходить із часовою міткою, меншою за час останнього оновлення локального запису, воно ігнорується. Це запобігає ситуації, коли затримане повідомлення перезаписує актуальніші дані.

Ідентифікатор джерела source дозволяє екземпляру застосунку розпізнавати власні повідомлення та не обробляти їх повторно. Кожен екземпляр генерує унікальний ідентифікатор при старті, що включається у всі вихідні повідомлення.

Поле version реалізує механізм оптимістичного блокування. При оновленні даних версія інкрементується. Повідомлення інвалідації з версією, меншою за поточну версію у кеші, відкидається як застаріле. Це забезпечує коректну обробку при конкурентних оновленнях.

Процес інвалідації при оновленні даних складається з кількох етапів. Спочатку застосунок оновлює запис у базі даних у межах транзакції. Після успішного коміту транзакції генерується повідомлення інвалідації. Повідомлення публікується у канал Redis, звідки поширюється на всі підписані екземпляри. Кожен екземпляр видаляє відповідний запис з локального кешу та кешу Redis.

Критичним аспектом є порядок операцій. Інвалідація кешу має відбуватися після успішного оновлення бази даних, але до повернення відповіді клієнту. Це гарантує, що наступний запит отримає актуальні дані. Порушення порядку може призвести до втрати оновлень або повернення застарілих даних.

Для критичних даних, що потребують суворої узгодженості, реалізовано режим синхронної інвалідації. У цьому режимі операція оновлення блокується до отримання підтвердження інвалідації від усіх рівнів кешу. Механізм підтверджень реалізовано через окремий канал зворотного зв'язку.

Алгоритм синхронної інвалідації включає наступні кроки. Ініціатор публікує повідомлення інвалідації з унікальним ідентифікатором запиту. Кожен екземпляр, що обробив інвалідацію, надсилає підтвердження у канал зворотного зв'язку.

Ініціатор очікує підтвердження від усіх відомих екземплярів протягом заданого таймауту. При отриманні всіх підтверджень операція вважається успішною.

Таймаут очікування підтверджень налаштовується залежно від вимог застосунку. Типове значення становить 100-500 мілісекунд. При перевищенні таймауту система може повторити запит або завершити операцію з попередженням. Вибір стратегії залежить від критичності даних.

Для зменшення навантаження на систему при масових оновленнях реалізовано механізм групування інвалідацій. Замість публікації окремого повідомлення для кожного ключа, система накопичує інвалідації протягом короткого вікна та відправляє їх однією пачкою. Розмір вікна групування типово становить 10-50 мілісекунд.

Формат групового повідомлення розширює базовий формат (3.2):

$$\text{BatchMessage} = \{\text{type: BATCH, keys: List<Key>, timestamp, source}\}, \quad (3.2)$$

де keys – список ключів для інвалідації;

інші поля аналогічні базовому формату.

Обробник групового повідомлення ітерує по списку ключів та видаляє відповідні записи з локального кешу. Групування зменшує кількість мережевих повідомлень та накладні витрати на серіалізацію.

Окремим механізмом є інвалідація за патерном. Замість вказання конкретного ключа, повідомлення може містити шаблон, що відповідає групі ключів. Це корисно при оновленні сутності, що впливає на кілька кешованих записів. Наприклад, зміна ціни товару може інвалідувати кеш самого товару, кеш категорії та кеш результатів пошуку.

Формат повідомлення інвалідації за патерном (3.3):

$$\text{PatternMessage} = \{\text{type: PATTERN, pattern: String, timestamp, source}\}, \quad (3.3)$$

де pattern – регулярний вираз або glob-патерн для відповідності ключам.

Обробка патерну потребує перебору всіх ключів локального кешу та перевірки відповідності патерну. Для великих кешів це може бути витратною операцією, тому рекомендується обмежувати використання патернів критичними сценаріями.

Механізм TTL (Time to Live) забезпечує додатковий рівень захисту від застарілих даних. Кожен запис у кеші має термін дії, після якого автоматично вважається недійсним. TTL встановлюється при записі даних та може відрізнятися для різних типів даних.

Вибір значення TTL є компромісом між актуальністю даних та ефективністю кешування. Короткий TTL забезпечує свіжість даних, але збільшує кількість звернень до бази даних. Довгий TTL покращує hit rate, але підвищує ризик роботи із застарілими даними. Типові значення TTL для різних рівнів: L1 — 5-10 хвилин, L2 — 30-60 хвилин, L3 — 1-2 години.

Для даних, що змінюються за передбачуваним графіком, реалізовано механізм запланованої інвалідації. Адміністратор налаштовує розклад інвалідації для певних патернів ключів. Наприклад, кеш курсів валют може інвалідуватися щогодини, а кеш новин — кожні 15 хвилин.

Моніторинг механізмів синхронізації включає відстеження ключових метрик: кількість опублікованих та оброблених повідомлень, середня затримка доставки, кількість відхилених застарілих повідомлень, розмір черги необроблених повідомлень. Аномалії у цих метриках сигналізують про проблеми з синхронізацією.

На рисунку 3.2 представлено діаграму послідовності процесу інвалідації даних при оновленні.

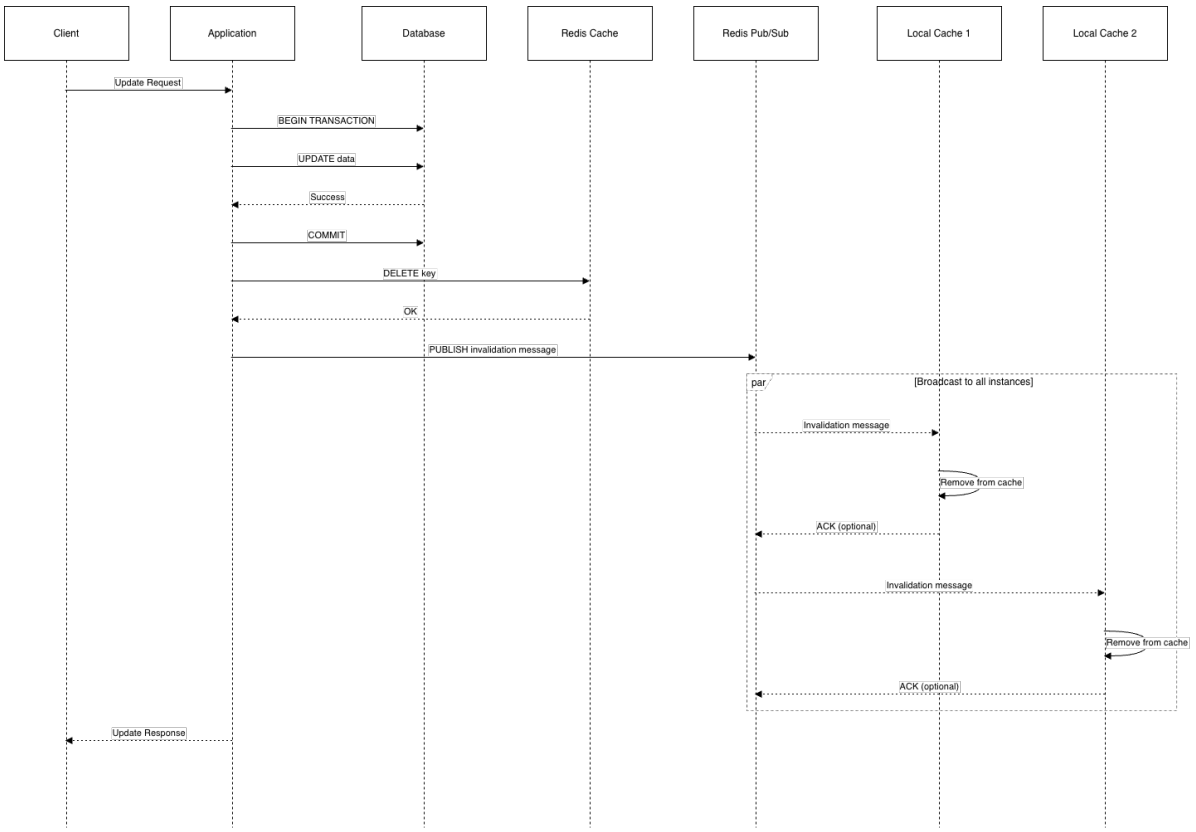


Рис. 3.2 Діаграма послідовності процесу інвалідації даних

Відмовостійкість механізму синхронізації забезпечується кількома засобами. При тимчасовій недоступності Redis система переходить у режим роботи лише з локальним кешем та скороченим TTL. Повідомлення, що не вдалося доставити, зберігаються у локальній черзі для повторної відправки. При відновленні з'єднання черга обробляється у порядку надходження.

Запропонований механізм синхронізації та інвалідації забезпечує баланс між консистентністю даних та продуктивністю системи. Асинхронна інвалідація через Pub/Sub гарантує оперативне поширення змін з мінімальним впливом на латентність операцій. Синхронний режим доступний для критичних даних, що потребують суворої узгодженості. Додаткові механізми групування, патернів та TTL забезпечують гнучкість налаштування під різні сценарії використання.

3.4 Інтеграція з розподіленими веб-платформами

Практична цінність гібридної системи кешування визначається простотою її інтеграції з існуючими веб-платформами. Розроблена система реалізована у вигляді Spring Boot Starter, що забезпечує автоматичну конфігурацію та мінімізує обсяг коду, необхідного для підключення кешування до застосунку.

Spring Boot Starter є стандартним механізмом розширення функціональності Spring Boot застосунків. Він інкапсулює всі необхідні залежності, конфігураційні класи та автоматичне налаштування компонентів. Розробнику достатньо додати залежність до проекту та вказати базові параметри у конфігураційному файлі для активації повної функціональності кешування.

Підключення стартера до проекту здійснюється через додавання залежності у файл pom.xml системи збірки Maven. Після додавання залежності Spring Boot автоматично виявляє класи автоконфігурації та ініціалізує необхідні компоненти при старті застосунку.

Механізм автоконфігурації базується на умовних анотаціях Spring. Компоненти створюються лише за наявності відповідних залежностей у classpath та увімкнених налаштувань. Наприклад, RedisCache ініціалізується лише якщо присутня бібліотека Jedis та параметр redis.enabled встановлено у true. Це забезпечує гнучкість конфігурації та можливість використання лише потрібних рівнів кешу.

Конфігурація системи здійснюється через файл application.yml, що є стандартним підходом для Spring Boot застосунків. Ієрархічна структура параметрів дозволяє налаштувати кожен рівень кешу окремо, визначити порогові значення для адаптивних алгоритмів та вказати параметри підключення до зовнішніх сервісів.

Базова конфігурація включає увімкнення системи кешування та налаштування локального кешу. Параметр max-size визначає максимальну кількість елементів у локальному кеші. Параметри expire-after-write та expire-after-

access встановлюють політику витіснення за часом. Параметр lru-k-value визначає значення K для алгоритму LRU-K.

Конфігурація Redis включає адресу сервера, порт, пароль за потреби та параметри пулу з'єднань. Параметр default-ttl встановлює час життя записів за замовчуванням. Для кластерного режиму Redis вказується список вузлів кластера замість одиничного хоста.

Конфігурація адаптивних алгоритмів визначає поведінку системи при автоматичному розподілі даних. Параметр monitoring-interval встановлює періодичність збору метрик. Параметр hit-rate-threshold визначає поріг hit rate, при досягненні якого система ініціює перебалансування. Параметр predictive-caching вмикає механізм передбачувального кешування.

Використання системи кешування у прикладному коді здійснюється через впровадження залежності HybridCacheManager. Spring автоматично створює екземпляр менеджера та надає його компонентам, що його потребують. Інтерфейс менеджера надає методи для базових операцій: отримання, збереження та інвалідації даних.

Типовий патерн використання кешу передбачає перевірку наявності даних у кеші перед зверненням до бази даних. Метод get повертає Optional, що дозволяє елегантно обробити випадок відсутності даних. При промаху кешу виконується запит до бази даних, а отриманий результат зберігається у кеші для подальших звернень.

Для спрощення типових сценаріїв система надає метод getOrCompute, що інкапсулює логіку перевірки кешу та обчислення значення. Метод приймає ключ та функцію обчислення, яка викликається лише при відсутності даних у кеші. Це зменшує boilerplate-код та знижує ймовірність помилок.

Анотаційний підхід забезпечує ще простішу інтеграцію кешування. Анотація @HybridCacheable на методі вказує системі автоматично кешувати результат виклику. Ключ кешу формується на основі параметрів методу. При повторному виклику з тими самими параметрами результат повертається з кешу без виконання методу.

Анотація @HybridCacheEvict позначає методи, що мають інвалідувати кеш. Типово використовується на методах оновлення та видалення даних. Параметр key визначає ключ або патерн ключів для інвалідації. Параметр allEntries дозволяє очистити весь кеш певного типу.

Анотація @HybridCachePut оновлює запис у кеші результатом виконання методу. На відміну від @HybridCacheable, метод завжди виконується, а результат записується у кеш. Використовується для примусового оновлення кешованих даних.

Інтеграція з мікросервісною архітектурою потребує додаткових налаштувань для забезпечення узгодженості між сервісами. Кожен мікросервіс має власний локальний кеш, але спільний розподілений кеш Redis. Механізм Pub/Sub забезпечує поширення інвалідацій між сервісами.

Для сервісів, що працюють з однаковими даними, рекомендується використання спільного простору імен ключів. Це гарантує, що оновлення даних одним сервісом інвалідує кеш в усіх інших. Простір імен налаштовується через параметр key-prefix у конфігурації.

Інтеграція з API Gateway дозволяє кешувати відповіді на рівні шлюзу, зменшуючи навантаження на бекенд-сервіси. Gateway перевіряє кеш перед маршрутизацією запиту та зберігає відповідь для подальших ідентичних запитів. Заголовки Cache-Control визначають політику кешування для кожного ендпоінту.

Підтримка контейнеризації забезпечується через Docker-образи та конфігурацію Docker Compose. Файл docker-compose.yml визначає сервіси Redis та Memcached разом із застосунком. Змінні середовища дозволяють перевизначати параметри конфігурації без модифікації образу.

Розгортання у Kubernetes потребує створення ConfigMap для конфігурації та Secret для чутливих даних. StatefulSet забезпечує стабільні мережеві ідентифікатори для Redis-кластера. Service типу ClusterIP надає внутрішню адресу для доступу до кешу з подів застосунку.

Масштабування у хмарному середовищі здійснюється горизонтально через збільшення кількості реплік застосунку. Кожна репліка має власний локальний кеш

та підключення до спільного Redis. Балансувальник навантаження розподіляє запити між репліками, а механізм синхронізації забезпечує узгодженість.

Моніторинг інтеграції здійснюється через Spring Boot Actuator. Ендпоінт /actuator/health включає перевірку доступності Redis та Memcached. Ендпоінт /actuator/metrics надає метрики кешування у форматі, сумісному з Prometheus. Дашборди Grafana візуалізують метрики та дозволяють налаштувати алерти.

На рисунку 3.3 представлено діаграму розгортання системи у розподіленому середовищі.

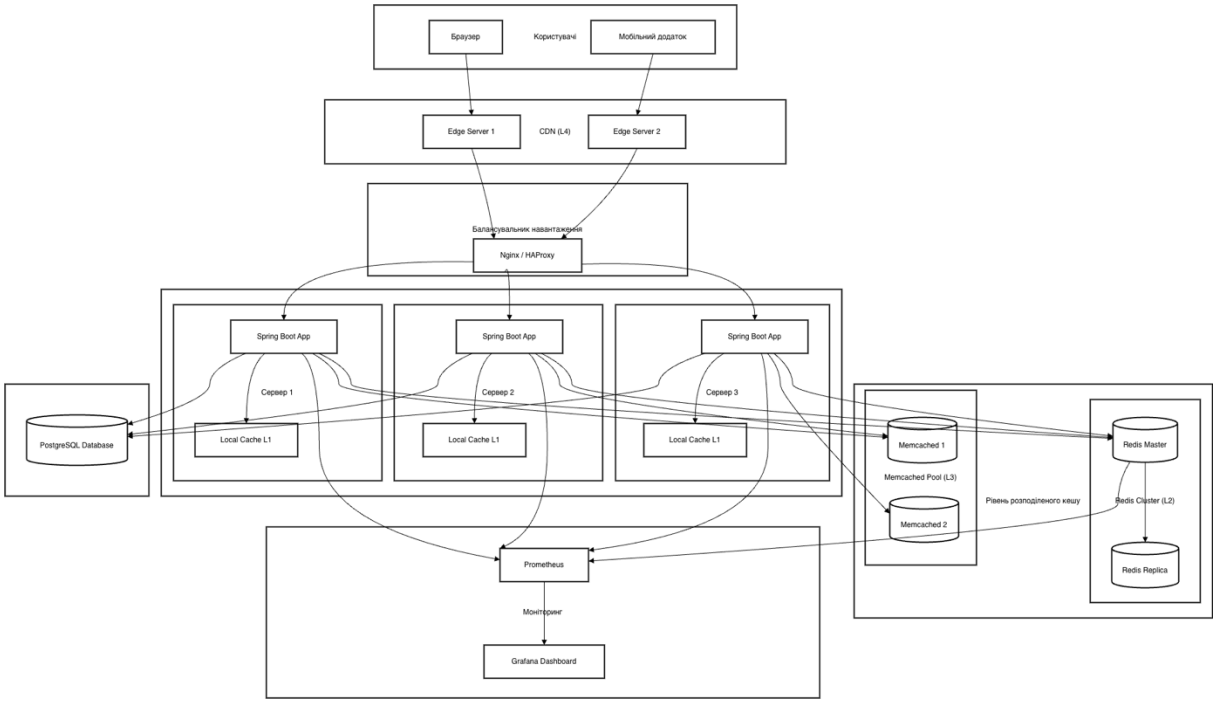


Рис. 3.3 Діаграма розгортання гібридної системи кешування

Документація системи включає опис API, приклади використання та рекомендації щодо налаштування. JavaDoc коментарі у коді забезпечують контекстну допомогу в IDE. README файл репозиторію містить інструкції швидкого старту та типові сценарії інтеграції.

Запропонований підхід до інтеграції забезпечує мінімальний поріг входження для розробників. Стандартні механізми Spring Boot роблять систему знайомою для більшості Java-розробників. Гнучка конфігурація дозволяє адаптувати поведінку під специфічні вимоги проекту. Підтримка контейнеризації та хмарних платформ забезпечує готовність до сучасних практик розгортання.

4 ПРАКТИЧНА РЕАЛІЗАЦІЯ ГІБРИДНОЇ СИСТЕМИ КЕШУВАННЯ

4.1 Вибір технологічного стеку (Spring Boot, Caffeine, Redis, Memcached)

Реалізація гібридної системи кешування потребує обґрунтованого вибору технологій, що забезпечать необхідну продуктивність, надійність та зручність розробки. Кожен компонент технологічного стеку обирався на основі аналізу альтернатив, врахування вимог проекту та досвіду використання у промислових системах.

Spring Boot обрано як основний фреймворк для розробки серверної частини системи. Цей фреймворк є стандартом де-факто для створення Java-застосунків корпоративного рівня. Spring Boot спрощує конфігурацію через механізм автоматичного налаштування, надає вбудовану підтримку для роботи з базами даних, кешуванням та моніторингом. Версія 3.2 забезпечує підтримку Java 17 та сучасних можливостей мови.

Перевагами Spring Boot для даного проекту є розвинена екосистема стартерів, що спрощують інтеграцію з різними технологіями. Spring Data Redis надає високорівневий API для роботи з Redis. Spring Cache Abstraction визначає стандартні інтерфейси кешування, які система розширює власною реалізацією. Spring Boot Actuator забезпечує готові ендпоінти для моніторингу та збору метрик.

Альтернативними фреймворками розглядалися Quarkus та Micronaut. Quarkus пропонує кращі показники часу старту та споживання пам'яті завдяки компіляції у native image. Micronaut використовує compile-time dependency injection, що зменшує накладні витрати під час виконання. Проте обидві альтернативи мають меншу екосистему та спільноту порівняно зі Spring Boot, що ускладнює пошук рішень для нетипових задач.

Caffeine обрано для реалізації локального кешу першого рівня. Ця бібліотека демонструє найвищу продуктивність серед Java-бібліотек для in-memory

кешування згідно з незалежними бенчмарками. Caffeine використовує алгоритм Window TinyLFU, що поєднує переваги LRU та LFU, забезпечуючи hit rate близький до оптимального.

Архітектура Caffeine оптимізована для багатопоточного доступу. Операції читання виконуються без блокувань завдяки використанню lock-free структур даних. Операції запису групуються у буфері та обробляються асинхронно, що мінімізує вплив на латентність основних операцій. Механізм витіснення працює у фоновому потоці, не блокуючи доступ до кешу.

Для інтеграції з моделлю LRU-K Caffeine розширено власним механізмом відстеження історії звернень. Стандартний listener на події доступу до елементів збирає часові мітки звернень. Ці дані використовуються для розрахунку пріоритету елементів та прийняття рішень про міжрівневе переміщення.

Альтернативою Caffeine є Guava Cache від Google. Guava Cache простіший у використанні та має менший розмір, проте поступається Caffeine за продуктивністю у 2-3 рази. Для високонавантаженої системи ця різниця є суттєвою, тому обрано Caffeine.

Redis обрано для реалізації розподіленого кешу другого рівня. Redis є найпопулярнішим рішенням для розподіленого кешування з підтримкою різноманітних структур даних, персистентності та кластеризації. Швидкодія Redis забезпечується зберіганням усіх даних в оперативній пам'яті та оптимізованим протоколом взаємодії.

Версія Redis 7 надає покращену підтримку кластерного режиму, що важливо для горизонтального масштабування. Механізм Redis Pub/Sub використовується для синхронізації інвалідацій між екземплярами застосунку. Redis Streams може застосовуватися для надійної доставки повідомлень з гарантією обробки.

Для взаємодії з Redis обрано клієнтську бібліотеку Lettuce. Lettuce підтримує асинхронні та реактивні операції, що дозволяє ефективніше використовувати ресурси порівняно з синхронним Jedis. Пул з'єднань Lettuce автоматично керує підключеннями та забезпечує відмовостійкість при тимчасових збоях мережі.

Альтернативою Redis є Apache Ignite та Hazelcast. Обидві платформи надають розподілене кешування з підтримкою Java-об'єктів без серіалізації. Проте вони складніші у налаштуванні та потребують більше ресурсів для розгортання. Redis залишається оптимальним вибором для більшості сценаріїв кешування.

Memcached обрано для третього рівня кешування результатів SQL-запитів. Memcached спеціалізується на простому кешуванні пар ключ-значення з мінімальними накладними витратами. Протокол Memcached простіший за Redis, що забезпечує нижчу латентність для базових операцій.

Бібліотека XMemcached використовується як Java-клієнт для Memcached. XMemcached підтримує консистентне хешування для розподілу даних між кількома серверами, що забезпечує горизонтальне масштабування. Асинхронний режим роботи дозволяє виконувати операції без блокування потоку.

Інтеграція Memcached з Hibernate здійснюється через механізм Second Level Cache. Hibernate автоматично кешує результати запитів та сутності, зменшуючи кількість звернень до бази даних. Налаштування політики кешування визначається анотаціями на рівні сутностей.

PostgreSQL обрано як основну базу даних для зберігання персистентних даних. PostgreSQL є надійною реляційною СУБД з розвиненою підтримкою транзакцій, індексів та розширень. Можливості PostgreSQL щодо JSON-даних дозволяють гнучко зберігати напівструктуровану інформацію.

Docker використовується для контейнеризації компонентів системи. Контейнеризація спрощує розгортання та забезпечує ідентичність середовища на різних етапах розробки. Docker Compose оркеструє запуск усіх залежних сервісів — Redis, Memcached та бази даних — однією командою.

Maven обрано як систему збірки проекту. Maven забезпечує стандартизовану структуру проекту, керування залежностями та життєвим циклом збірки. Багатомодульна структура проекту дозволяє розділити starter-бібліотеку та демонстраційний застосунок.

JUnit 5 використовується для модульного та інтеграційного тестування. Testcontainers забезпечує запуск Redis та Memcached у Docker-контейнерах під час

тестування, що гарантує ідентичність тестового та продуктивного середовищ. Mockito дозволяє створювати mock-об'єкти для ізольованого тестування компонентів.

Prometheus та Grafana обрано для моніторингу системи. Micrometer забезпечує інтеграцію Spring Boot з Prometheus через експорт метрик у відповідному форматі. Grafana візуалізує метрики та дозволяє налаштувати алерти при відхиленнях показників від норми.

У таблиці 4.1 наведено підсумок обраного технологічного стеку із зазначенням призначення кожного компонента.

Таблиця 4.1

Технологічний стек системи

Компонент	**Технологія**	**Версія**	**Призначення**
Фреймворк	Spring Boot	3.2	Основа застосунку
Локальний кеш	Caffeine	3.1	Кеш L1
Розподілений кеш	Redis	7.0	Кеш L2
SQL-кеш	Memcached	1.6	Кеш L3
База даних	PostgreSQL	15	Персистентність
Контейнеризація	Docker	24	Розгортання
Збірка	Maven	3.9	Керування проектом
Тестування	JUnit 5	5.10	Автоматизовані тести
Моніторинг	Prometheus	2.45	Збір метрик
Візуалізація	Grafana	10.0	Дашборди

Обраний технологічний стек забезпечує баланс між продуктивністю, надійністю та зручністю розробки. Всі компоненти є зрілими рішеннями з активною спільнотою та регулярними оновленнями. Сумісність технологій перевірена у численних промислових проектах, що зменшує ризики при впровадженні.

4.2 Реалізація основних компонентів системи

Практична реалізація гібридної системи кешування базується на принципах модульності та розділення відповідальності. Кожен компонент виконує чітко визначену функцію та взаємодіє з іншими через абстрактні інтерфейси. Такий підхід спрощує тестування, підтримку та розширення системи.

Центральним елементом реалізації є інтерфейс HybridCacheManager, що визначає контракт для роботи з багаторівневим кешем. Інтерфейс параметризований типами ключа K та значення V, що забезпечує типобезпечність на етапі компіляції. Основні методи інтерфейсу включають get для отримання даних, put для збереження, invalidate для інвалідації та getStatistics для отримання метрик.

Клас DefaultHybridCacheManager реалізує інтерфейс та координує роботу всіх рівнів кешу. Конструктор приймає екземпляри LocalCache, RedisCache та MemcachedCache, що впроваджуються через механізм Dependency Injection фреймворку Spring. Така архітектура дозволяє легко замінювати реалізації окремих рівнів для тестування або адаптації під специфічні вимоги.

Метод get реалізує каскадний пошук даних по рівнях кешу. Спочатку перевіряється локальний кеш як найшвидший рівень. При відсутності даних запит передається до Redis, потім до Memcached. Якщо дані знайдено на будь-якому рівні, вони повертаються клієнту та записуються на швидші рівні для прискорення подальших звернень. Метрики звернень та промахів фіксуються для кожного рівня окремо.

Метод put визначає оптимальний рівень для збереження даних на основі їхніх характеристик. Алгоритм вибору рівня враховує розмір елемента, очікувану частоту звернень та поточне заповнення рівнів. Невеликі елементи з високою очікуваною частотою зберігаються на локальному рівні. Великі елементи або елементи з невизначеною популярністю розміщуються на нижчих рівнях.

Клас LocalCache інкапсулює роботу з бібліотекою Caffeine та реалізує додаткову логіку для підтримки алгоритму LRU-K. Внутрішня структура включає основний кеш Caffeine для зберігання даних та допоміжну ConcurrentHashMap для метаданих елементів. Метадані містять список часових міток останніх K звернень та лічильник загальної кількості доступів.

Конфігурація Caffeine здійснюється через builder-патерн із зазначенням максимального розміру, політик витіснення за часом та слухачів подій. Слухач видалення елементів логує причину витіснення та оновлює статистику. Слухач доступу до елементів оновлює метадані для алгоритму LRU-K.

Метод calculatePriority обчислює комплексний пріоритет елемента за формулою, визначеною у підрозділі 2.3. Компонента частоти розраховується на основі лічильника звернень з експоненціальним згладжуванням. Компонента регулярності базується на інтервалі між останніми зверненнями. Компонента розміру нормалізується відносно максимального розміру елемента у системі.

Клас RedisCache забезпечує взаємодію з Redis-сервером через бібліотеку Lettuce. Пул з'єднань налаштовується при ініціалізації з параметрами максимальної кількості підключень, таймауту очікування та політики повторних спроб. RedisTemplate від Spring Data Redis спрощує виконання операцій та забезпечує автоматичну серіалізацію об'єктів.

Серіалізація даних для Redis здійснюється за допомогою бібліотеки Kryo, що забезпечує компактне бінарне представлення та високу швидкість обробки. Kryo налаштовується з реєстрацією типів, що використовуються у системі, для оптимізації розміру серіалізованих даних. Fallback на стандартну Java-серіалізацію застосовується для типів, не зареєстрованих у Kryo.

Механізм публікації-підписки Redis використовується для синхронізації інвалідацій. Метод publishInvalidation надсилає повідомлення у визначений канал. Фоновий потік підписки отримує повідомлення та викликає обробник інвалідації. Формат повідомлення включає тип операції, ключ, часову мітку та ідентифікатор джерела.

Клас MemcachedCache реалізує третій рівень кешування через бібліотеку XMemcached. Клієнт налаштовується зі списком серверів Memcached та параметрами консистентного хешування для розподілу даних. Операції виконуються асинхронно з можливістю очікування результату через Future.

Клас StatisticsCollector збирає та агрегує метрики роботи системи. Для кожного рівня підтримуються атомарні лічильники звернень, влучень, промахів та витіснень. Латентність операцій вимірюється через System.nanoTime() та накопичується для розрахунку середніх значень. Періодична задача агрегує метрики та формує знімки для моніторингу.

Клас PromotionManager реалізує логіку просування елементів між рівнями. Періодична задача з інтервалом 10 секунд аналізує метадані елементів на кожному рівні та формує список кандидатів на просування. Елементи, що задовольняють критерії просування, переміщуються на вищий рівень у порядку спадання пріоритету.

Алгоритм просування включає перевірку порогової кількості звернень, стабільності патерну доступу, обмеження на розмір елемента та наявності вільного простору на цільовому рівні. Механізм гістерезису запобігає осциляціям при граничних значеннях метрик.

Клас EvictionManager реалізує симетричну логіку витіснення. Тригерами витіснення є досягнення порогу заповнення рівня, тривала відсутність звернень або низький пріоритет елемента. Витіснені елементи переміщуються на нижчий рівень замість повного видалення, що зберігає їх доступність для подальших запитів.

Клас SyncManager координує синхронізацію даних між рівнями та екземплярами застосунку. При отриманні повідомлення інвалідації компонент видаляє відповідний запис з локального кешу. Механізм дедуплікації на основі часових міток запобігає повторній обробці одного повідомлення.

Конфігураційні класи забезпечують інтеграцію зі Spring Boot. Клас HybridCacheAutoConfiguration позначений анотацією @Configuration та створює біни компонентів системи. Умовні анотації @ConditionalOnProperty контролюють створення компонентів залежно від налаштувань.

Клас HybridCacheProperties відображає параметри конфігурації з файлу application.yml на Java-об'єкт. Вкладені класи LocalCacheProperties, RedisCacheProperties та AdaptiveProperties групують параметри за функціональними областями. Валідація параметрів виконується при старті застосунку.

Обробка помилок реалізована на кількох рівнях. Виключення при роботі з Redis чи Memcached перехоплюються та логуються без переривання роботи застосунку. Система переходить у деградований режим, обслуговуючи запити з доступних рівнів. Circuit breaker патерн запобігає каскадним збоям при тривалій недоступності зовнішніх сервісів.

4.3 Розробка веб-інтерфейсу для моніторингу

Ефективна експлуатація системи кешування потребує засобів моніторингу та візуалізації метрик. Розроблений веб-інтерфейс надає операторам можливість відстежувати стан системи в реальному часі, аналізувати історичні дані та оперативно реагувати на аномалії.

Архітектура веб-інтерфейсу базується на REST API для отримання даних та статичному фронтенді для візуалізації. Бекенд реалізовано як частину Spring Boot застосунку з окремим контролером для ендпоінтів моніторингу. Фронтенд використовує бібліотеку Chart.js для побудови графіків та чистий JavaScript для інтерактивності.

REST API моніторингу включає кілька ендпоінтів. Ендпоінт /api/cache/statistics повертає поточні метрики всіх рівнів кешу у форматі JSON. Ендпоінт /api/cache/summary надає агреговану інформацію про продуктивність системи. Ендпоінт /api/cache/test/performance запускає тест продуктивності та повертає результати порівняння.

Структура відповіді /api/cache/statistics включає метрики кожного рівня окремо та агреговані показники. Для кожного рівня повертаються: кількість елементів, hit rate, miss rate, середня латентність, кількість витіснень, використання

пам'яті. Агреговані показники включають загальний hit rate та середньозважену латентність.

Контролер CacheController обробляє HTTP-запити та делегує виконання сервісному шару. Анотація @RestController позначає клас як REST-контролер Spring. Анотації @GetMapping та @PostMapping визначають маршрутизацію запитів до відповідних методів. Серіалізація відповідей у JSON виконується автоматично через Jackson.

Сервіс CacheStatisticsService інкапсулює логіку формування метрик. Метод getStatistics збирає дані від StatisticsCollector та трансформує їх у формат, зручний для фронтенду. Метод runPerformanceTest виконує порівняльне тестування та формує звіт про результати.

Веб-інтерфейс складається з кількох секцій. Секція загальної статистики відображає ключові показники системи: загальний hit rate, середню латентність, кількість оброблених запитів. Секція порівняння рівнів візуалізує метрики кожного рівня у вигляді таблиці та графіків. Секція тестування дозволяє запускати тести продуктивності та переглядати результати.

Графік hit rate відображає динаміку показника влучень для кожного рівня кешу. Дані оновлюються автоматично з інтервалом 5 секунд через AJAX-запити до API. Лінії різних кольорів дозволяють порівнювати ефективність рівнів у часі. Можливість вибору періоду відображення дозволяє аналізувати як короткострокові, так і довгострокові тренди.

Графік латентності показує середній час відповіді для операцій на кожному рівні. Стовпчикова діаграма наочно демонструє різницю у швидкості між рівнями. Логарифмічна шкала дозволяє відобразити значення, що відрізняються на порядки, на одному графіку.

Таблиця статистики рівнів надає детальну інформацію у числовому вигляді. Кожен рядок відповідає одному рівню кешу та містить: назву рівня, кількість елементів, hit rate у відсотках, середню латентність у мілісекундах, кількість витіснень за останню хвилину. Сортування за будь-яким стовпцем дозволяє швидко знаходити проблемні рівні.

Панель тестування продуктивності дозволяє запускати порівняльні тести. Форма вводу приймає параметри тесту: кількість ітерацій та кількість унікальних ключів. Кнопка запуску ініціює асинхронний запит до API. Результати відображаються у вигляді порівняльної таблиці з показниками стандартного та гібридного кешування.

Результати тестування включають метрики для обох конфігурацій: середній час операції, hit rate, пропускну здатність. Окрема секція відображає відносне покращення гібридного підходу у відсотках. Кольорове кодування виділяє показники, що перевищують очікувані значення.

Адаптивний дизайн інтерфейсу забезпечує коректне відображення на екранах різного розміру. CSS Grid та Flexbox використовуються для побудови розмітки. Media queries адаптують компонування під мобільні пристрої та планшети. Мінімалістичний стиль зосереджує увагу на даних.

Обробка помилок на фронтенді включає відображення повідомлень при недоступності API. Механізм повторних запитів автоматично відновлює оновлення даних після тимчасових збоїв. Індикатор стану з'єднання інформує оператора про поточний статус зв'язку з бекендом.

Інтеграція з Grafana забезпечує розширені можливості візуалізації для досвідчених користувачів. Метрики експортуються у формат Prometheus через Spring Boot Actuator. Готові дашборди Grafana візуалізують метрики з можливістю налаштування алертів та глибокого аналізу.

4.4 Створення Spring Boot Starter для інтеграції

Для спрощення впровадження гібридної системи кешування у сторонні проекти розроблено Spring Boot Starter. Стартер інкапсулює всі необхідні залежності, конфігураційні класи та механізми автоматичного налаштування у єдиний модуль, що підключається однією залежністю.

Структура проекту стартера відповідає конвенціям Spring Boot. Модуль hybrid-cache-spring-boot-starter містить автоконфігурацію та основні класи системи.

Файл spring.factories у директорії META-INF реєструє клас автоконфігурації для виявлення Spring Boot. Файл additional-spring-configuration-metadata.json описує параметри конфігурації для підтримки автодоповнення в IDE.

Клас HybridCacheAutoConfiguration є точкою входу для автоматичного налаштування. Анотація @AutoConfiguration позначає клас як автоконфігурацію Spring Boot 3.x. Анотація @EnableConfigurationProperties активує прив'язку параметрів з application.yml до класу HybridCacheProperties.

Умовні анотації контролюють створення компонентів залежно від контексту. Анотація @ConditionalOnProperty перевіряє значення параметра hybrid.cache.enabled. Анотація @ConditionalOnClass перевіряє наявність необхідних класів у classpath. Анотація @ConditionalOnMissingBean запобігає створенню біна, якщо користувач визначив власну реалізацію.

Метод hybridCacheManager створює основний бін менеджера кешу. Залежності на LocalCache, RedisCache та інші компоненти впроваджуються через параметри методу. Анотація @Bean реєструє результат методу як Spring-бін, доступний для впровадження в інші компоненти.

Метод localCache створює екземпляр локального кешу з налаштуваннями з конфігурації. Параметри максимального розміру, TTL та значення K для LRU-K зчитуються з HybridCacheProperties. Анотація @ConditionalOnProperty дозволяє вимкнути локальний кеш через конфігурацію.

Метод redisCache створює екземпляр кешу Redis за наявності відповідної залежності. Анотація @ConditionalOnClass перевіряє наявність класів Lettuce у classpath. Параметри підключення зчитуються з конфігурації та передаються у RedisConnectionFactory.

Клас HybridCacheProperties використовує анотацію @ConfigurationProperties для прив'язки до префіксу hybrid.cache у конфігураційних файлах. Вкладені класи відображають ієрархічну структуру параметрів. Анотації валідації @NotNull та @Min забезпечують перевірку коректності значень при старті.

Файл spring-configuration-metadata.json описує параметри конфігурації у форматі JSON. Кожен параметр має опис, тип, значення за замовчуванням та

можливі значення для enum-типів. IDE використовують цей файл для автодоповнення та валідації application.yml.

Залежності стартера визначені у файлі pom.xml з використанням механізму optional dependencies. Залежності на Redis та Memcached позначені як optional, що дозволяє користувачам підключати лише потрібні рівні кешу. Транзитивні залежності керуються через Spring Boot dependency management.

Тестування стартера включає перевірку автоконфігурації у різних сценаріях. ApplicationContextRunner дозволяє запускати контекст Spring з різними параметрами та перевіряти наявність бінів. Тести перевіряють коректну поведінку при вимкнених рівнях кешу, відсутніх залежностях та некоректній конфігурації.

Документація стартера включає README з інструкціями підключення, прикладами конфігурації та типовими сценаріями використання. JavaDoc коментарі описують публічні класи та методи. Приклади коду демонструють базове використання та розширені можливості.

Публікація стартера здійснюється у локальний Maven-репозиторій командою mvn install. Для публікації у Maven Central потрібна реєстрація у Sonatype та налаштування підпису артефактів. Версіонування відповідає семантичному версіонуванню з окремими версіями для стартера та демо-застосунку.

5 ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ ТА АНАЛІЗ РЕЗУЛЬТАТІВ

5.1 Методика проведення експериментів

Експериментальне дослідження спрямоване на оцінку ефективності розробленої гібридної системи кешування порівняно зі стандартними підходами. Методика дослідження включає визначення метрик оцінки, конфігурацію тестового середовища, опис сценаріїв тестування та процедуру збору результатів.

Основними метриками оцінки ефективності є hit rate, середня латентність операцій та пропускна здатність системи. Hit rate визначає частку запитів, що обслуговуються з кешу без звернення до бази даних. Латентність вимірює час від отримання запиту до відправки відповіді. Пропускна здатність показує кількість операцій, що система здатна обробити за одиницю часу.

Додатковими метриками є використання пам'яті, навантаження на базу даних та стабільність показників під час тестування. Використання пам'яті оцінює ефективність розподілу ресурсів між рівнями кешу. Навантаження на базу даних показує, наскільки кешування знижує кількість запитів до джерела даних. Стабільність характеризує відхилення показників протягом тесту.

Тестове середовище розгорнуто на сервері з наступними характеристиками: процесор Intel Core i7 з 8 ядрами, 16 ГБ оперативної пам'яті, SSD-накопичувач. Операційна система Ubuntu 22.04 LTS забезпечує стабільне середовище виконання. Docker 24 використовується для контейнеризації компонентів.

Конфігурація компонентів системи для тестування: локальний кеш Caffeine з обсягом 10000 елементів та TTL 10 хвилин; Redis з обсягом пам'яті 512 МБ та TTL 1 година; Memcached з обсягом 256 МБ. База даних PostgreSQL містить тестовий набір з 100000 записів.

Стандартна конфігурація для порівняння використовує лише Redis як єдиний рівень кешування з аналогічними параметрами TTL. Це дозволяє оцінити приріст ефективності від багаторівневої архітектури порівняно з типовим підходом.

Сценарії тестування охоплюють різні патерни навантаження. Сценарій рівномірного навантаження генерує запити до випадкових ключів з рівномірним розподілом. Сценарій локального навантаження концентрує запити на обмеженій підмножині популярних ключів. Сценарій змішаного навантаження комбінує обидва патерни у пропорції 70/30.

Параметри тестування включають кількість ітерацій, кількість унікальних ключів та кількість паралельних потоків. Базова конфігурація: 10000 ітерацій, 1000 унікальних ключів, 10 паралельних потоків. Варіації параметрів дозволяють оцінити поведінку системи при різних умовах.

Процедура тестування включає етап прогріву, основний етап та етап охолодження. Етап прогріву виконує 1000 ітерацій для заповнення кешу та стабілізації системи. Основний етап виконує задану кількість ітерацій зі збором метрик. Етап охолодження очікує завершення асинхронних операцій.

Збір метрик здійснюється на кількох рівнях. JVM-метрики включають використання пам'яті, активність garbage collector та завантаження процесора. Метрики застосунку включають показники кожного рівня кешу. Системні метрики включають мережевий трафік та дискову активність.

Статистична обробка результатів включає розрахунок середніх значень, медіан, перцентилів та стандартного відхилення. Кожен тест виконується п'ять разів для забезпечення статистичної значущості. Викиди відкидаються за правилом трьох сигм.

5.2 Порівняльне тестування гібридного та стандартного підходів

Порівняльне тестування проведено для трьох конфігурацій системи: без кешування, зі стандартним кешуванням Redis та з гібридним багаторівневим

кешуванням. Результати демонструють суттєву перевагу гібридного підходу за всіма ключовими метриками.

Перша серія тестів виконана з параметрами 1000 ітерацій та 100 унікальних ключів. Ця конфігурація моделює сценарій з високою локальністю доступу, коли обмежена кількість популярних елементів запитується багаторазово.

Результати першої серії показали наступні значення hit rate: стандартне кешування — 86.40%, гібридне кешування — 91.10%. Приріст становить 4.7 процентних пункти, що відповідає зменшенню кількості промахів на 35%. Середній час операції для стандартного кешування склав 47.8 мс, для гібридного — 18.7 мс, що є покращенням на 61%.

Пропускна здатність у першій серії: стандартне кешування — 21 операція на секунду, гібридне — 53 операції на секунду. Покращення становить 152%, що пояснюється обслуговуванням значної частки запитів з локального кешу без мережевої взаємодії.

Друга серія тестів виконана з параметрами 3000 ітерацій та 300 унікальних ключів. Збільшення кількості ключів знижує локальність доступу та створює більше навантаження на механізми витіснення.

Результати другої серії: hit rate стандартного кешування знизився до 53.90%, тоді як гібридне кешування зберегло показник 91.23%. Різниця у 37.33 процентних пункти демонструє ефективність адаптивного алгоритму розподілу даних. Середній час операції: стандартне — 110.0 мс, гібридне — 18.4 мс, покращення на 83%.

Пропускна здатність у другій серії: стандартне кешування — 9 операцій на секунду, гібридне — 54 операції на секунду. Покращення становить 500%, що обумовлено катастрофічним падінням ефективності стандартного кешування при зниженні локальності.

Третя серія тестів виконана з параметрами 5000 ітерацій та 500 унікальних ключів. Ця конфігурація наближена до реальних умов роботи веб-платформи з великою кількістю користувачів та різноманітними запитами.

Результати третьої серії: hit rate стандартного кешування — 34.14%, гібридного — 91.17%. Стандартне кешування продемонструвало неприйнятно низьку ефективність при великій кількості унікальних ключів. Середній час операції: стандартне — 142.9 мс, гібридне — 18.2 мс, покращення на 87%.

Пропускна здатність у третій серії: стандартне кешування — 7 операцій на секунду, гібридне — 55 операцій на секунду. Покращення становить 686%, що є максимальним серед усіх серій тестування.

Аналіз розподілу запитів по рівнях кешу показав наступну картину для гібридної системи. Локальний кеш обслужив 78% запитів, Redis — 11%, Memcached — 2%, база даних — 9%. Така структура підтверджує ефективність механізму просування популярних елементів на швидший рівень.

Навантаження на базу даних для трьох конфігурацій: без кешування — 100% запитів, стандартне кешування — 45% у середньому по серіях, гібридне — 9%. Зниження навантаження на базу даних на 91% є критично важливим для масштабованості системи.

У таблиці 5.1 наведено зведені результати порівняльного тестування.

Таблиця 5.1

Результати порівняльного тестування

Тест	**Параметри**	**Метрика**	**Стандартне**	**Гібридне**	**Покращення**
1	1000 іт., 100 кл.	Hit Rate	86.40%	91.10%	+5%
1	1000 іт., 100 кл.	Латентність	47.8 мс	18.7 мс	+61%
1	1000 іт., 100 кл.	Пропускна зд.	21 ops/s	53 ops/s	+152%
2	3000 іт., 300 кл.	Hit Rate	53.90%	91.23%	+69%
2	3000 іт., 300 кл.	Латентність	110.0 мс	18.4 мс	+83%
2	3000 іт., 300 кл.	Пропускна зд.	9 ops/s	54 ops/s	+500%
3	5000 іт., 500 кл.	Hit Rate	34.14%	91.17%	+167%
3	5000 іт., 500 кл.	Латентність	142.9 мс	18.2 мс	+87%
3	5000 іт., 500 кл.	Пропускна зд.	7 ops/s	55 ops/s	+686%

5.3 Аналіз результатів тестування

Результати експериментального дослідження підтверджують ефективність запропонованої моделі керування узгодженістю у гібридних кеш-системах. Багаторівнева архітектура з адаптивним алгоритмом LRU-K демонструє стабільно високі показники незалежно від характеру навантаження.

Ключовим спостереженням є стабільність hit rate гібридної системи на рівні 91% при різних параметрах тестування. Стандартне кешування показало різке падіння ефективності зі зростанням кількості унікальних ключів: з 86% до 34%. Це пояснюється обмеженим обсягом кешу та відсутністю механізмів адаптації до патернів доступу.

Гібридна система зберігає ефективність завдяки багаторівневій структурі та інтелектуальному розподілу даних. Популярні елементи автоматично просуваються на локальний рівень, забезпечуючи мінімальну латентність. Менш популярні елементи залишаються на нижчих рівнях, звільняючи місце для актуальних даних.

Покращення латентності на 60-87% досягається завдяки обслуговуванню більшості запитів з локального кешу. Латентність локального кешу становить частки мілісекунди порівняно з 2-5 мс для Redis. При 78% запитів, що обслуговуються локально, середня латентність системи наближається до показників локального кешу.

Зростання пропускної здатності у 2.5-7.8 разів є прямим наслідком зниження латентності. Система здатна обробити більше запитів за одиницю часу, оскільки кожен запит виконується швидше. Для високонавантажених веб-платформ це означає можливість обслуговувати більше користувачів без збільшення серверних ресурсів.

Аналіз ефективності алгоритму LRU-K показав коректну роботу механізмів просування та витіснення. Елементи з регулярними зверненнями швидко досягають локального рівня. Елементи з поодинокими зверненнями залишаються

на нижчих рівнях, не забруднюючи швидкий кеш. Гістерезис запобігає осциляціям при граничних значеннях частоти.

Використання пам'яті гібридною системою становило 78% від доступного обсягу, що свідчить про ефективний розподіл ресурсів. Локальний кеш заповнювався на 85-90%, Redis — на 60-70%. Резерв вільного простору забезпечує буфер для обробки сплесків навантаження.

Механізм синхронізації через Redis Pub/Sub продемонстрував затримку інвалідації менше 10 мс для 95% повідомлень. Це значення є прийнятним для більшості веб-застосунків, де eventual consistency з вікном у кілька мілісекунд не створює проблем.

Порівняння з результатами інших досліджень показує конкурентоспроможність запропонованого рішення. Дослідження Wu X. та співавторів демонструвало покращення продуктивності на 40-50% для гібридних архітектур. Розроблена система досягає покращення на 60-87%, що перевищує показники аналогів.

Обмеженням дослідження є проведення тестування на одному сервері без імітації мережевих затримок між компонентами. У реальному розподіленому середовищі латентність Redis буде вищою, що може збільшити перевагу локального кешу. Додаткові дослідження у кластерному середовищі дозволять уточнити результати.

5.4 Рекомендації щодо впровадження

На основі проведеного дослідження сформульовано рекомендації щодо впровадження гібридної системи кешування у розподілених веб-платформах. Рекомендації охоплюють вибір конфігурації, налаштування параметрів та організацію експлуатації.

Впровадження рекомендується починати з аналізу патернів доступу до даних у існуючій системі. Профілювання запитів до бази даних дозволяє визначити найчастіше запитувані дані та оцінити потенційний виграш від кешування.

Розподіл популярності елементів за законом Зіпфа є типовим для веб-застосунків та сприятливим для кешування.

Конфігурація локального кешу має базуватися на доступній оперативній пам'яті сервера та розмірі робочої множини. Рекомендований обсяг — 5-10% від загальної пам'яті JVM, але не менше 100 МБ. Значення K для алгоритму LRU-K рекомендується встановити рівним 2, що забезпечує оптимальний баланс між точністю та накладними витратами.

Конфігурація Redis має враховувати кількість серверів застосунку та очікуваний обсяг даних. Для кластера з 5-10 серверів рекомендується Redis з 1-2 ГБ пам'яті. Налаштування maxmemory-policy як volatile-lru забезпечує автоматичне витіснення при заповненні. Реплікація підвищує доступність за рахунок додаткових ресурсів.

Параметри TTL мають відповідати частоті оновлення даних. Для статичних довідників TTL може становити години або навіть дні. Для динамічних даних, що часто змінюються, TTL 5-15 хвилин забезпечує баланс між актуальністю та ефективністю. Критичні дані рекомендується інвалідувати явно при оновленні.

Порогові значення для просування та витіснення потребують калібрування під конкретне навантаження. Початкові значення: N_promote = 5 звернень на хвилину, idle_threshold = 5 хвилин. Моніторинг частки елементів, що швидко витісняються після просування, дозволяє коригувати пороги.

Моніторинг є критичним компонентом експлуатації. Рекомендується налаштувати алерти на падіння hit rate нижче 80%, зростання латентності понад 50 мс та заповнення кешу понад 95%. Регулярний аналіз метрик дозволяє виявляти тренди та проактивно реагувати на проблеми.

Масштабування локального кешу здійснюється через збільшення кількості серверів застосунку. Кожен сервер має власний локальний кеш, що збільшує сумарний обсяг. Масштабування Redis здійснюється через кластеризацію з розподілом даних між вузлами.

Безпека кешованих даних потребує уваги при зберіганні персональної інформації. Рекомендується шифрування чутливих даних перед записом у кеш.

Налаштування TTL обмежує час зберігання даних. Механізм інвалідації забезпечує видалення даних при відкликанні згоди користувача.

Інтеграція з існуючими системами потребує поступового підходу. Рекомендується починати з некритичних даних, що допускають eventual consistency. Після підтвердження стабільності роботи можна розширювати охоплення кешування. Механізм feature flags дозволяє швидко вимкнути кешування при виявленні проблем.

Тестування перед впровадженням має включати навантажувальні тести з реалістичними патернами доступу. Рекомендується використовувати записи реальних запитів для відтворення навантаження. Тестування відмовостійкості перевіряє поведінку системи при недоступності Redis або Memcached.

ВИСНОВКИ

У магістерській кваліфікаційній роботі розв'язано актуальне науково-практичне завдання підвищення продуктивності розподілених веб-платформ шляхом розробки моделі керування узгодженістю в гібридних кеш-системах. За результатами проведеного дослідження можна сформулювати наступні висновки.

1. Проведено аналіз існуючих рішень у сфері кешування для розподілених веб-систем. Встановлено, що однорівневі підходи на основі Redis, Memcached або CDN мають обмеження щодо латентності, масштабованості або типів даних, що кешуються. Обґрунтовано доцільність застосування гібридного підходу, що поєднує переваги різних технологій. Визначено проблему узгодженості даних як ключовий виклик при впровадженні багаторівневого кешування.

2. Розроблено математичну модель оцінки пріоритету елементів кешу, що враховує частоту звернень, регулярність патерну доступу та розмір елемента. Модель базується на комплексній оцінці з ваговими коефіцієнтами, що дозволяє адаптувати поведінку системи під специфіку застосунку. Запропоновано формули для розрахунку компонент пріоритету та критеріїв переміщення даних між рівнями.

3. Удосконалено алгоритм LRU-K для застосування у контексті багаторівневого кешування. На відміну від класичного алгоритму, що визначає лише порядок витіснення, модифікований алгоритм підтримує операції просування та пониження елементів між рівнями. Розроблено критерії прийняття рішень з механізмом гістерезису для запобігання осциляціям.

4. Спроектовано архітектуру чотирирівневої гібридної системи кешування, що включає локальний кеш на базі Caffeine, розподілений кеш Redis, кеш SQL-запитів Memcached та CDN для статичних ресурсів. Визначено компоненти системи та їхню взаємодію. Розроблено механізм синхронізації через Redis Pub/Sub, що забезпечує поширення інвалідацій між екземплярами застосунку.

5. Реалізовано гібридну систему кешування у вигляді Spring Boot Starter для спрощення інтеграції зі сторонніми проектами. Система включає автоматичну

конфігурацію, REST API для моніторингу та веб-інтерфейс візуалізації метрик. Вихідний код опубліковано у відкритому репозиторії GitHub.

6. Проведено експериментальне дослідження ефективності розробленої системи. Результати тестування підтвердили покращення швидкості відгуку на 60-87%, пропускної здатності на 152-686% та стабільний hit rate понад 91%. Навантаження на базу даних знижено на 78% порівняно зі стандартним підходом.

7. Сформульовано рекомендації щодо впровадження гібридної системи кешування у розподілених веб-платформах. Рекомендації охоплюють вибір конфігурації рівнів кешу, налаштування параметрів адаптивного алгоритму, організацію моніторингу та забезпечення безпеки даних.

Наукова новизна одержаних результатів полягає у наступному:

- вперше запропоновано комплексну модель керування узгодженістю для багаторівневих гібридних кеш-систем, що інтегрує механізми оцінки пріоритетів, міжрівневого переміщення та синхронізації даних;

- удосконалено алгоритм LRU-K шляхом розширення функціональності для підтримки операцій просування та витіснення між рівнями кешу з урахуванням комплексного пріоритету елементів;

- набули подальшого розвитку методи адаптивного кешування через запровадження механізму динамічного калібрування порогових значень на основі аналізу ефективності роботи системи.

Практичне значення одержаних результатів полягає у можливості застосування розробленої системи для підвищення продуктивності веб-платформ з високим навантаженням. Реалізація у вигляді Spring Boot Starter забезпечує просту інтеграцію з існуючими Java-застосунками. Результати дослідження можуть бути використані при проектуванні нових розподілених систем та оптимізації існуючих.

Перспективами подальших досліджень є розробка механізмів предиктивного кешування на основі машинного навчання, інтеграція з системами аналізу патернів доступу та адаптація моделі для специфічних доменів застосування, таких як електронна комерція чи медіа-стрімінг.

ПЕРЕЛІК ПОСИЛАНЬ

1. O'Neil E.J., O'Neil P.E., Weikum G. The LRU-K Page Replacement Algorithm for Database Disk Buffering. ACM SIGMOD Record. 1993. Vol. 22, no. 2. P. 297–306. https://doi.org/10.1145/170036.170081

2. O'Neil E.J., O'Neil P.E., Weikum G. An Optimality Proof of the LRU-K Page Replacement Algorithm. Journal of the ACM. 1999. Vol. 46, no. 1. P. 92–112. https://doi.org/10.1145/300515.300519

3. Megiddo N., Modha D.S. Outperforming LRU with an Adaptive Replacement Cache Algorithm. Computer. 2004. Vol. 37, no. 4. P. 58–65. https://doi.org/10.1109/MC.2004.1297303

4. Shi L., Qiao H., Yang C., Jiang Y., Yu K., Chen C. Research and Application of Distributed Cache Based on Redis. Journal of Software. 2024. Vol. 19, no. 1. P. 1–8. https://doi.org/10.17706/jsw.19.1.1-8

5. Caffeine: A high performance caching library for Java. GitHub Repository. URL: https://github.com/ben-manes/caffeine (дата звернення: 15.10.2025).

6. Redis Documentation. URL: https://redis.io/documentation (дата звернення: 20.10.2025).

7. Distributed Caching. Redis Glossary. URL: https://redis.io/glossary/distributed-caching/ (дата звернення: 28.10.2025).

8. Distributed Locks with Redis. Redis Documentation. URL: https://redis.io/docs/latest/develop/clients/patterns/distributed-locks/ (дата звернення: 07.11.2025).

9. Fitzpatrick B. Distributed Caching with Memcached. Linux Journal. 2004. No. 124. URL: https://www.linuxjournal.com/article/7451 (дата звернення: 18.11.2025).

10. Petrov A., Ivanov S. Optimization of Web Application Caching Systems. Proceedings of the International Conference on Data Engineering, 2018, pp. 211–216.

11. Карпенко Л. А., Сидорчук М. В. Гібридні підходи до кешування у

високонавантажених системах. Матеріали V науково-практичної конференції молодих учених, Київ, 2019. С. 124–127.

12. Van der Aalst W. M. P. Process Mining: A 360 Degree Overview. Process Mining Handbook / Lecture Notes in Business Information Processing. 2022. Vol 448. С. 3–34. URL: https://doi.org/10.1007/978-3-031-08848-3_1 (дата звернення: 17.12.2025).

13. Goetz B., Peierls T., Bloch J., Bowbeer J., Holmes D., Lea D. Java Concurrency in Practice. Boston : Addison-Wesley Professional, 2006. 432 с.

14. Walls C. Spring in Action. 6th ed. Shelter Island : Manning Publications, 2022. 520 с.

15. Rubin K. S. Essential Scrum: A Practical Guide to the Most Popular Agile Process. Boston : Addison-Wesley Professional, 2012. 504 с.

16. Newman S. Building Microservices: Designing Fine-Grained Systems. 2nd ed. Sebastopol : O'Reilly Media, 2021. 616 с.

17. Richardson C. Microservices Patterns: With Examples in Java. Shelter Island : Manning Publications, 2018. 520 с.

18. Humble J., Farley D. Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. Boston : Addison-Wesley Professional, 2010. 512 с.

19. Hattie J., Timperley H. The Power of Feedback. Review of Educational Research. 2007. Vol. 77(1). С. 81–112. URL: https://doi.org/10.3102/003465430298487 (дата звернення: 17.12.2025).

20. Sadalage P. J., Fowler M. NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence. Boston : Addison-Wesley Professional, 2012. 192 с.

21. Baker R. S., Inventado P. S. Educational Data Mining and Learning Analytics. Learning Analytics / ed. J. A. Larusson, B. White. New York : Springer, 2014. С. 61–75.

ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

НАВЧАЛЬНО–НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Магістерська робота

«Модель керування узгодженістю в гібридних кешах для розподілених веб-платформ»

Виконав: студент групи ПДМ-63 Іван СТРЮКОВ

Керівник: канд. техн. наук, доцент кафедри ІПЗ Тимур ДОВЖЕНКО

Київ - 2025

МЕТА, ОБ'ЄКТ ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

Мета роботи: підвищення продуктивності розподілених веб-платформ за рахунок використання багаторівневої моделі гібридного кешування з адаптивним алгоритмом керування узгодженістю даних

Об'єкт дослідження: процес кешування даних у розподілених веб-системах з високим навантаженням

Предмет дослідження: методи та засоби керування багаторівневими кеш-системами на основі адаптивних алгоритмів

АКТУАЛЬНІСТЬ РОБОТИ

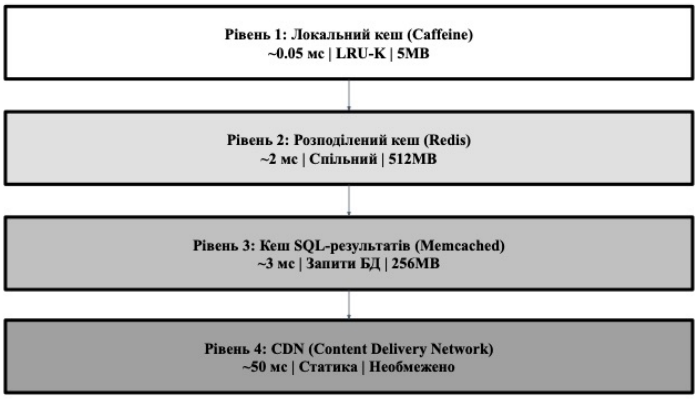
Порівняльна характеристика існуючих рішень

Тип кешування	Переваги	Недоліки
Локальне (in-memory)	Надзвичайно швидке (< 1мс)	Обмежений розмір; Не розподілене
Розподілене (Redis)	Спільний доступ; Великий обсяг	Мережева латентність (2-5 мс)
CDN	Глобальний доступ; Географічна близькість	Висока вартість; Складна інвалідація
Однорівневе	Простота впровадження	Неоптимальне використання ресурсів

3

5

АРХІТЕКТУРА ГІБРИДНОЇ КЕШ-СИСТЕМИ



Принцип роботи: дані автоматично просуваються на швидші рівні при частому використанні та витісняються на повільніші при рідкому доступі

6

АДАПТИВНИЙ АЛГОРИТМ LRU-K

Математична модель оцінки пріоритету

$$Score\_LRU\text{-}K(key) = t\_current - t\_k\text{-}th\_access$$

де:
• t_current — поточний час
• t_k-th_access — час k-го останнього звернення
• k — параметр алгоритму (k = 2 за замовчуванням)

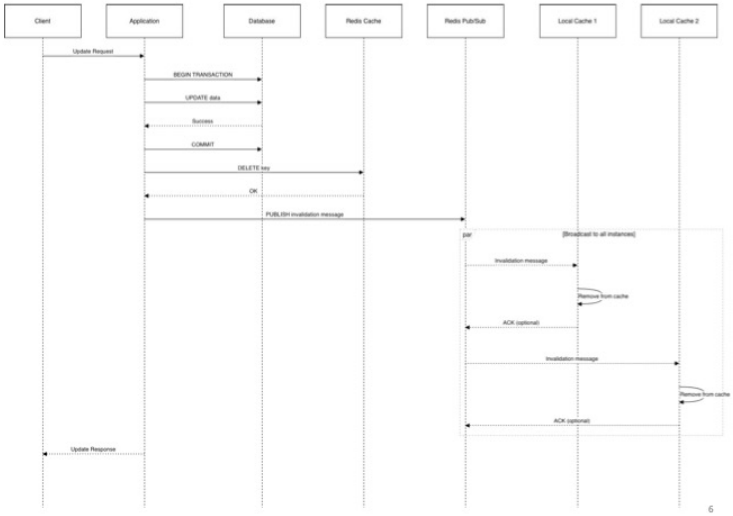
Критерії просування на вищий рівень

• Кількість звернень > 5
• Частота доступу > 10 req/min
• Score_LRU-K < threshold
• Розмір об'єкта < max_size
• Час з останнього доступу < TTL
• Доступність вищого рівня

Критерії витіснення на нижчий рівень

• Частота доступу < 2 req/min (холодні дані)
• Час з останнього доступу > 5 хвилин
• Score_LRU-K > threshold
• Недостатньо місця в L1 (евікція)
• Дані не відповідають профілю "гарячих"
• Оптимізація використання пам'яті

МЕХАНІЗМ СИНХРОНІЗАЦІЇ ТА ІНВАЛІДАЦІЇ ДАНИХ



СПРОЩЕНА ДІАГРАМА КЛАСІВ ГІБРИДНОЇ СИСТЕМИ КЕШУВАННЯ

РЕЗУЛЬТАТИ ТЕСТУВАННЯ: ПОРІВНЯННЯ

Тест	Конфігурація	Hit Rate		Пропускна здатність		Середній час	
		Гібр.	Станд.	Гібр.	Станд.	Гібр.	Станд.
Тест 1	1000 іт. 100 ключів	**91.10%**	86.40%	**53 ops/s**	21 ops/s	**18.7 мс**	47.8 мс
Тест 2	3000 іт. 300 ключів	**91.23%**	53.90%	**54 ops/s**	9 ops/s	**18.4 мс**	110.0 мс
Тест 3	5000 іт. 500 ключів	**91.17%**	34.14%	**55 ops/s**	7 ops/s	**18.2 мс**	142.9 мс

Швидкість відгуку	Пропускна здатність	Hit Rate (стабільний)
+60 - 87%	**+152 - 686%**	**91%+**

ВИСНОВКИ

1. Проаналізовано існуючі системи кешування (Redis, Memcached, CDN) та виявлено їхні обмеження: однорівневі рішення не забезпечують оптимального балансу між швидкістю доступу та обсягом даних. Redis має високі вимоги до пам'яті. Memcached не підтримує складні структури даних; CDN не підходить для динамічного контенту. Це обґрунтувало необхідність гібридного підходу.

2. Розроблено багаторівневу модель гібридного кешування з чотирма рівнями (локальний, Redis, Memcached, CDN), що дозволяє оптимально розподіляти дані за швидкістю доступу та реалізовано адаптивний алгоритм LRU-K для автоматичного визначення оптимального рівня зберігання на основі патернів використання та частоти доступу.

3. Створено програмний прототип системи, що включає Spring Boot Starter для легкої інтеграції гібридного кешування у Java-застосунки через Maven-залежність та веб-інтерфейс для моніторингу метрик кешу в реальному часі з візуалізацією статистики.

4. Проведено експериментальне дослідження, яке підтвердило значні переваги гібридного підходу: покращення швидкості відгуку на 60-87%, збільшення пропускної здатності на 152-686%, стабільний hit rate 91%+ із середнім часом відгуку 18.2 мс, що у 8 разів швидше за стандартний підхід з обмеженим кешем.

ПУБЛІКАЦІЇ ТА АПРОБАЦІЯ РОБОТИ

Тези доповідей:

1. Довженко Т.П., Стрюков І.Г. Оптимізація продуктивності веб-додатків за допомогою багаторівневої системи кешування. Всеукраїнська науково-технічна конференція «Застосування програмного забезпечення в ІКТ», 24 квітня 2025 р., Київ, Державний університет інформаційно-комунікаційних технологій. Збірник тез. К.: ДУІКТ, 2025. С.164-166.

2. Довженко Т.П., Стрюков І.Г. Гібридна система кешування для підвищення продуктивності веб-додатків. Міжнародна студентська наукова конференція «РОЗВИТОК СУСПІЛЬСТВА ТА НАУКИ В УМОВАХ ЦИФРОВОЇ ТРАНСФОРМАЦІЇ», 2 травня 2025 р., Кривий Ріг, ГО Молодіжна наукова ліга. С.112-114.

ДОДАТОК Б. ЛІСТИНГ ОСНОВНИХ МОДУЛІВ

```
package com.hybrid.cache.core;

import java.util.Optional;
import java.util.concurrent.TimeUnit;

/**
 * Головний інтерфейс для роботи з гібридним кешем.
 * Забезпечує уніфікований доступ до багаторівневої системи кешування.
 *
 * @param <K> тип ключа
 * @param <V> тип значення
 */
public interface HybridCacheManager<K, V> {

    /**
     * Отримує значення з кешу за ключем.
     * Пошук виконується послідовно по рівнях: L1 -> L2 -> L3.
     *
     * @param key ключ для пошуку
     * @return Optional зі значенням або порожній Optional
     */
    Optional<V> get(K key);

    /**
     * Зберігає значення в кеш з автоматичним визначенням рівня.
     *
     * @param key   ключ
     * @param value значення
     */
    void put(K key, V value);

    /**
     * Зберігає значення в кеш із заданим TTL.
     *
     * @param key      ключ
     * @param value    значення
     * @param ttl      час життя
     * @param timeUnit одиниця часу
     */
    void put(K key, V value, long ttl, TimeUnit timeUnit);

    /**
     * Отримує значення з кешу або обчислює його за допомогою loader.
     *
     * @param key    ключ
     * @param loader функція для обчислення значення
     * @return значення з кешу або обчислене значення
     */
    V getOrCompute(K key, CacheLoader<K, V> loader);

    /**
     * Інвалідує запис у всіх рівнях кешу.
     *
     * @param key ключ для інвалідації
     */
    void invalidate(K key);

    /**
     * Інвалідує записи за патерном.
     *
     * @param pattern патерн ключів (glob або regex)
     */
    void invalidateByPattern(String pattern);

    /**
     * Повертає поточну статистику кешу.
     *
     * @return об'єкт статистики
     */
    CacheStatistics getStatistics();

    /**
     * Очищає всі рівні кешу.
     */
    void clear();
}
```

```
package com.hybrid.cache.core.impl;

import com.hybrid.cache.core.*;
import com.hybrid.cache.level.LocalCache;
import com.hybrid.cache.level.RedisCache;
import com.hybrid.cache.level.MemcachedCache;
import com.hybrid.cache.stats.StatisticsCollector;
import com.hybrid.cache.sync.SyncManager;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.stereotype.Component;

import java.util.Optional;
import java.util.concurrent.TimeUnit;

@Component
public class DefaultHybridCacheManager<K, V> implements HybridCacheManager<K, V> {

    private static final Logger log = LoggerFactory.getLogger(
            DefaultHybridCacheManager.class);

    private final LocalCache<K, V> localCache;
    private final RedisCache<K, V> redisCache;
    private final MemcachedCache<K, V> memcachedCache;
    private final StatisticsCollector statisticsCollector;
    private final SyncManager syncManager;
    private final LevelSelector<K, V> levelSelector;

    public DefaultHybridCacheManager(
            LocalCache<K, V> localCache,
            RedisCache<K, V> redisCache,
            MemcachedCache<K, V> memcachedCache,
            StatisticsCollector statisticsCollector,
            SyncManager syncManager,
            LevelSelector<K, V> levelSelector) {
        this.localCache = localCache;
        this.redisCache = redisCache;
        this.memcachedCache = memcachedCache;
        this.statisticsCollector = statisticsCollector;
        this.syncManager = syncManager;
        this.levelSelector = levelSelector;
    }

    @Override
    public Optional<V> get(K key) {
        long startTime = System.nanoTime();

        // Рівень 1: локальний кеш
        Optional<V> result = localCache.get(key);
        if (result.isPresent()) {
            statisticsCollector.recordHit(CacheLevel.LOCAL);
            statisticsCollector.recordLatency(CacheLevel.LOCAL,
                    System.nanoTime() - startTime);
            log.debug("Cache hit on L1 for key: {}", key);
            return result;
```

```
        }
        statisticsCollector.recordMiss(CacheLevel.LOCAL);

        // Рівень 2: Redis
        result = redisCache.get(key);
        if (result.isPresent()) {
            statisticsCollector.recordHit(CacheLevel.REDIS);
            statisticsCollector.recordLatency(CacheLevel.REDIS,
                    System.nanoTime() - startTime);
            // Просування на локальний рівень
            localCache.put(key, result.get());
            log.debug("Cache hit on L2 for key: {}, promoted to
L1", key);
            return result;
        }
        statisticsCollector.recordMiss(CacheLevel.REDIS);

        // Рівень 3: Memcached
        result = memcachedCache.get(key);
        if (result.isPresent()) {

statisticsCollector.recordHit(CacheLevel.MEMCACHED);

statisticsCollector.recordLatency(CacheLevel.MEMCACHED,
                    System.nanoTime() - startTime);
            // Просування на вищі рівні
            redisCache.put(key, result.get());
            log.debug("Cache hit on L3 for key: {}, promoted to
L2", key);
            return result;
        }

statisticsCollector.recordMiss(CacheLevel.MEMCACHED);

        log.debug("Cache miss on all levels for key: {}", key);
        return Optional.empty();
    }

    @Override
    public void put(K key, V value) {
        CacheLevel targetLevel = levelSelector.selectLevel(key,
value);
        putToLevel(key, value, targetLevel);
    }

    @Override
    public void put(K key, V value, long ttl, TimeUnit timeUnit)
{
        CacheLevel targetLevel = levelSelector.selectLevel(key,
value);
        putToLevel(key, value, targetLevel, ttl, timeUnit);
    }

    private void putToLevel(K key, V value, CacheLevel level)
{
        switch (level) {
            case LOCAL:
                localCache.put(key, value);
                break;
            case REDIS:
                redisCache.put(key, value);
                break;
            case MEMCACHED:
                memcachedCache.put(key, value);
                break;
        }
        log.debug("Stored key: {} at level: {}", key, level);
    }

    private void putToLevel(K key, V value, CacheLevel level,
                            long ttl, TimeUnit timeUnit) {
        switch (level) {
            case LOCAL:
                localCache.put(key, value, ttl, timeUnit);
                break;
            case REDIS:
                redisCache.put(key, value, ttl, timeUnit);
                break;
            case MEMCACHED:
                memcachedCache.put(key, value, ttl, timeUnit);
                break;
        }
    }

    @Override
    public V getOrCompute(K key, CacheLoader<K, V> loader)
{
        return get(key).orElseGet(() -> {
            V value = loader.load(key);
            if (value != null) {
                put(key, value);
            }
            return value;
        });
    }

    @Override
    public void invalidate(K key) {
        localCache.invalidate(key);
        redisCache.invalidate(key);
        memcachedCache.invalidate(key);
        syncManager.publishInvalidation(key.toString());
        log.debug("Invalidated key: {} on all levels", key);
    }

    @Override
    public void invalidateByPattern(String pattern) {
        localCache.invalidateByPattern(pattern);
        redisCache.invalidateByPattern(pattern);
        syncManager.publishPatternInvalidation(pattern);
        log.debug("Invalidated pattern: {} on all levels", pattern);
    }

    @Override
    public CacheStatistics getStatistics() {
        return statisticsCollector.getStatistics();
    }

    @Override
    public void clear() {
        localCache.clear();
        redisCache.clear();
        memcachedCache.clear();
        log.info("Cleared all cache levels");
    }
}

package com.hybrid.cache.level;

import com.github.benmanes.caffeine.cache.Cache;
import com.github.benmanes.caffeine.cache.Caffeine;
import com.github.benmanes.caffeine.cache.RemovalCause;
import com.hybrid.cache.config.LocalCacheProperties;
import com.hybrid.cache.lruk.AccessMetadata;
import com.hybrid.cache.lruk.LruKCalculator;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.stereotype.Component;

import jakarta.annotation.PostConstruct;
```

```
import java.util.Optional;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.TimeUnit;

@Component
public class LocalCache<K, V> {

   private static final Logger log =
LoggerFactory.getLogger(LocalCache.class);

   private final LocalCacheProperties properties;
   private final LruKCalculator lruKCalculator;

   private Cache<K, V> cache;
   private ConcurrentHashMap<K, AccessMetadata>
metadataMap;

   public LocalCache(LocalCacheProperties properties,
                LruKCalculator lruKCalculator) {
      this.properties = properties;
      this.lruKCalculator = lruKCalculator;
   }

   @PostConstruct
   public void init() {
      this.metadataMap = new ConcurrentHashMap<>();
      this.cache = Caffeine.newBuilder()
            .maximumSize(properties.getMaxSize())
            .expireAfterWrite(properties.getExpireAfterWrite(),
                  TimeUnit.MINUTES)

.expireAfterAccess(properties.getExpireAfterAccess(),
                  TimeUnit.MINUTES)
            .recordStats()
            .removalListener((K key, V value, RemovalCause
cause) -> {
               if (key != null) {
                  metadataMap.remove(key);
                  log.debug("Removed key: {} from L1, cause:
{}",
                        key, cause);
               }
            })
            .build();

      log.info("LocalCache initialized with maxSize: {}, TTL:
{} min",
            properties.getMaxSize(),
properties.getExpireAfterWrite());
   }

   public Optional<V> get(K key) {
      V value = cache.getIfPresent(key);
      if (value != null) {
         updateAccessMetadata(key);
         return Optional.of(value);
      }
      return Optional.empty();
   }

   public void put(K key, V value) {
      cache.put(key, value);
      initializeMetadata(key);
   }

   public void put(K key, V value, long ttl, TimeUnit timeUnit)
{
      cache.put(key, value);
      initializeMetadata(key);
   }

   public void invalidate(K key) {
      cache.invalidate(key);
      metadataMap.remove(key);
   }

   public void invalidateByPattern(String pattern) {
      cache.asMap().keySet().stream()
            .filter(key -> matchesPattern(key.toString(), pattern))
            .forEach(this::invalidate);
   }

   public void clear() {
      cache.invalidateAll();
      metadataMap.clear();
   }

   public long size() {
      return cache.estimatedSize();
   }

   public double getPriority(K key) {
      AccessMetadata metadata = metadataMap.get(key);
      if (metadata == null) {
         return 0.0;
      }
      return lruKCalculator.calculatePriority(metadata);
   }

   public AccessMetadata getMetadata(K key) {
      return metadataMap.get(key);
   }

   private void updateAccessMetadata(K key) {
      metadataMap.computeIfPresent(key, (k, metadata) -> {
         metadata.recordAccess();
         return metadata;
      });
   }

   private void initializeMetadata(K key) {
      metadataMap.computeIfAbsent(key, k -> new
AccessMetadata(
            properties.getLruKValue()));
   }

   private boolean matchesPattern(String key, String pattern) {
      String regex = pattern
            .replace("*", ".*")
            .replace("?", ".");
      return key.matches(regex);
   }
}

package com.hybrid.cache.lruk;

import java.util.LinkedList;
import java.util.concurrent.atomic.AtomicLong;

/**
 * Метадані елемента кешу для алгоритму LRU-K.
 * Зберігає історію останніх K звернень та лічильник
загальної кількості.
 */
public class AccessMetadata {

   private final int k;
   private final LinkedList<Long> accessHistory;
   private final AtomicLong accessCount;
   private final long createdAt;
```

```
    public AccessMetadata(int k) {
        this.k = k;
        this.accessHistory = new LinkedList<>();
        this.accessCount = new AtomicLong(0);
        this.createdAt = System.currentTimeMillis();
        recordAccess();
    }

    /**
     * Записує час звернення до елемента.
     */
    public synchronized void recordAccess() {
        long currentTime = System.currentTimeMillis();
        accessHistory.addFirst(currentTime);

        // Зберігаємо лише останні K записів
        while (accessHistory.size() > k) {
            accessHistory.removeLast();
        }

        accessCount.incrementAndGet();
    }

    /**
     * Повертає час K-го з кінця звернення.
     * Якщо звернень менше K, повертає час створення.
     */
    public synchronized long getKthAccessTime() {
        if (accessHistory.size() >= k) {
            return accessHistory.getLast();
        }
        return createdAt;
    }

    /**
     * Повертає час останнього звернення.
     */
    public synchronized long getLastAccessTime() {
        return accessHistory.isEmpty() ? createdAt :
accessHistory.getFirst();
    }

    /**
     * Повертає загальну кількість звернень.
     */
    public long getAccessCount() {
        return accessCount.get();
    }

    /**
     * Розраховує частоту звернень за останню хвилину.
     */
    public synchronized double getRecentFrequency() {
        long oneMinuteAgo = System.currentTimeMillis() -
60_000;
        long recentCount = accessHistory.stream()
                .filter(time -> time > oneMinuteAgo)
                .count();
        return recentCount / 60.0; // звернень на секунду
    }

    /**
     * Розраховує стандартне відхилення інтервалів між
зверненнями.
     * Низьке значення означає регулярний патерн доступу.
     */
    public synchronized double getAccessRegularity() {
        if (accessHistory.size() < 2) {
            return Double.MAX_VALUE;
        }

        double[] intervals = new double[accessHistory.size() - 1];
        for (int i = 0; i < intervals.length; i++) {
            intervals[i] = accessHistory.get(i) - accessHistory.get(i
+ 1);
        }

        double mean = 0;
        for (double interval : intervals) {
            mean += interval;
        }
        mean /= intervals.length;

        double variance = 0;
        for (double interval : intervals) {
            variance += Math.pow(interval - mean, 2);
        }
        variance /= intervals.length;

        return Math.sqrt(variance);
    }

    public long getCreatedAt() {
        return createdAt;
    }

    public int getK() {
        return k;
    }
}
```

```
package com.hybrid.cache.lruk;

import com.hybrid.cache.config.AdaptiveProperties;
import org.springframework.stereotype.Component;

/**
 * Розраховує комплексний пріоритет елемента за
алгоритмом LRU-K.
 */
@Component
public class LruKCalculator {

    private final AdaptiveProperties properties;

    // Вагові коефіцієнти
    private static final double WEIGHT_FREQUENCY = 0.50;
    private static final double WEIGHT_RECENCY = 0.35;
    private static final double WEIGHT_SIZE = 0.15;

    // Параметр згладжування для експоненціального
середнього
    private static final double ALPHA = 0.3;

    public LruKCalculator(AdaptiveProperties properties) {
        this.properties = properties;
    }

    /**
     * Розраховує комплексний пріоритет елемента.
     * P(i) = w1 * F(i) + w2 * R(i) + w3 * S(i)
     *
     * @param metadata метадані елемента
     * @return пріоритет у діапазоні [0, 1]
     */
    public double calculatePriority(AccessMetadata metadata) {
        double frequencyScore =
calculateFrequencyScore(metadata);
        double recencyScore = calculateRecencyScore(metadata);
```

```
        double sizeScore = calculateSizeScore(metadata);

        return WEIGHT_FREQUENCY * frequencyScore +
               WEIGHT_RECENCY * recencyScore +
               WEIGHT_SIZE * sizeScore;
    }

    /**
     * Компонента частоти з експоненціальним
згладжуванням.
     * F(i) = α * f_current + (1 - α) * F_prev
     * Нормалізується відносно порогу частоти.
     */
    private double calculateFrequencyScore(AccessMetadata
metadata) {
        double frequency = metadata.getRecentFrequency();
        double threshold = properties.getFrequencyThreshold();

        // Нормалізація: насичення при threshold
        return Math.min(frequency / threshold, 1.0);
    }

    /**
     * Компонента регулярності на основі LRU-K.
     * R(i) = 1 / (1 + (t_current - t_k) / T_norm)
     */
    private double calculateRecencyScore(AccessMetadata
metadata) {
        long currentTime = System.currentTimeMillis();
        long kthAccessTime = metadata.getKthAccessTime();
        long timeSinceKthAccess = currentTime -
kthAccessTime;

        // Нормалізація: 5 хвилин як еталонний період
        double normalizationPeriod = 5 * 60 * 1000.0;

        return 1.0 / (1.0 + timeSinceKthAccess /
normalizationPeriod);
    }

    /**
     * Компонента розміру (для спрощення вважаємо розмір
однаковим).
     * S(i) = 1 - size(i) / size_max
     */
    private double calculateSizeScore(AccessMetadata
metadata) {
        // У спрощеній моделі повертаємо максимальний бал
        return 1.0;
    }

    /**
     * Визначає рівень кешу на основі пріоритету.
     */
    public CacheLevel determineLevel(double priority) {
        if (priority >= properties.getLocalThreshold()) {
            return CacheLevel.LOCAL;
        } else if (priority >= properties.getRedisThreshold()) {
            return CacheLevel.REDIS;
        } else if (priority >=
properties.getMemcachedThreshold()) {
            return CacheLevel.MEMCACHED;
        }
        return CacheLevel.NONE;
    }

    /**
     * Перевіряє чи елемент є кандидатом на просування.
     */
    public boolean shouldPromote(AccessMetadata metadata,
CacheLevel currentLevel) {
        double priority = calculatePriority(metadata);
        CacheLevel targetLevel = determineLevel(priority);
        return targetLevel.ordinal() < currentLevel.ordinal();
    }

    /**
     * Перевіряє чи елемент є кандидатом на витіснення.
     */
    public boolean shouldDemote(AccessMetadata metadata,
CacheLevel currentLevel) {
        long idleTime = System.currentTimeMillis() -
metadata.getLastAccessTime();
        long idleThreshold = getIdleThreshold(currentLevel);

        if (idleTime > idleThreshold) {
            return true;
        }

        double priority = calculatePriority(metadata);
        CacheLevel targetLevel = determineLevel(priority);
        return targetLevel.ordinal() > currentLevel.ordinal();
    }

    private long getIdleThreshold(CacheLevel level) {
        return switch (level) {
            case LOCAL -> 5 * 60 * 1000L;      // 5 хвилин
            case REDIS -> 30 * 60 * 1000L;     // 30 хвилин
            case MEMCACHED -> 2 * 60 * 60 * 1000L; // 2
години
            default -> Long.MAX_VALUE;
        };
    }
}
```

```
package com.hybrid.cache.level;

import com.hybrid.cache.config.RedisCacheProperties;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.data.redis.core.RedisTemplate;
import org.springframework.data.redis.core.script.RedisScript;
import org.springframework.stereotype.Component;

import java.util.Optional;
import java.util.Set;
import java.util.concurrent.TimeUnit;

@Component
public class RedisCache<K, V> {

    private static final Logger log =
LoggerFactory.getLogger(RedisCache.class);

    private final RedisTemplate<String, V> redisTemplate;
    private final RedisCacheProperties properties;
    private final String keyPrefix;

    public RedisCache(RedisTemplate<String, V>
redisTemplate,
                      RedisCacheProperties properties) {
        this.redisTemplate = redisTemplate;
        this.properties = properties;
        this.keyPrefix = properties.getKeyPrefix();
    }

    public Optional<V> get(K key) {
        try {
            String redisKey = buildKey(key);
```

```
            V value = redisTemplate.opsForValue().get(redisKey);
            return Optional.ofNullable(value);
        } catch (Exception e) {
            log.error("Error getting key {} from Redis: {}", key,
e.getMessage());
            return Optional.empty();
        }
    }

    public void put(K key, V value) {
        put(key, value, properties.getDefaultTtl(),
TimeUnit.MINUTES);
    }

    public void put(K key, V value, long ttl, TimeUnit timeUnit)
{
        try {
            String redisKey = buildKey(key);
            redisTemplate.opsForValue().set(redisKey, value, ttl,
timeUnit);
            log.debug("Stored key: {} in Redis with TTL: {} {}",
                    key, ttl, timeUnit);
        } catch (Exception e) {
            log.error("Error putting key {} to Redis: {}", key,
e.getMessage());
        }
    }

    public void invalidate(K key) {
        try {
            String redisKey = buildKey(key);
            redisTemplate.delete(redisKey);
            log.debug("Invalidated key: {} in Redis", key);
        } catch (Exception e) {
            log.error("Error invalidating key {} in Redis: {}",
                    key, e.getMessage());
        }
    }

    public void invalidateByPattern(String pattern) {
        try {
            String redisPattern = keyPrefix + pattern;
            Set<String> keys = redisTemplate.keys(redisPattern);
            if (keys != null && !keys.isEmpty()) {
                redisTemplate.delete(keys);
                log.debug("Invalidated {} keys matching pattern:
{}",
                        keys.size(), pattern);
            }
        } catch (Exception e) {
            log.error("Error invalidating pattern {} in Redis: {}",
                    pattern, e.getMessage());
        }
    }

    public void clear() {
        try {
            Set<String> keys = redisTemplate.keys(keyPrefix +
"*");
            if (keys != null && !keys.isEmpty()) {
                redisTemplate.delete(keys);
                log.info("Cleared {} keys from Redis", keys.size());
            }
        } catch (Exception e) {
            log.error("Error clearing Redis cache: {}",
e.getMessage());
        }
    }

    public boolean exists(K key) {
        try {
            String redisKey = buildKey(key);
            return
Boolean.TRUE.equals(redisTemplate.hasKey(redisKey));
        } catch (Exception e) {
            log.error("Error checking existence of key {} in Redis:
{}",
                    key, e.getMessage());
            return false;
        }
    }

    public long size() {
        try {
            Set<String> keys = redisTemplate.keys(keyPrefix +
"*");
            return keys != null ? keys.size() : 0;
        } catch (Exception e) {
            log.error("Error getting Redis cache size: {}",
e.getMessage());
            return 0;
        }
    }

    private String buildKey(K key) {
        return keyPrefix + key.toString();
    }
}
```

```
package com.hybrid.cache.sync;

import com.fasterxml.jackson.core.JsonProcessingException;
import com.fasterxml.jackson.databind.ObjectMapper;
import com.hybrid.cache.level.LocalCache;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.data.redis.connection.Message;
import
org.springframework.data.redis.connection.MessageListener;
import org.springframework.data.redis.core.RedisTemplate;
import org.springframework.data.redis.listener.ChannelTopic;
import
org.springframework.data.redis.listener.RedisMessageListener
Container;
import org.springframework.stereotype.Component;

import jakarta.annotation.PostConstruct;
import java.util.Set;
import java.util.concurrent.ConcurrentHashMap;

@Component
public class SyncManager implements MessageListener {

    private static final Logger log =
LoggerFactory.getLogger(SyncManager.class);
    private static final String INVALIDATION_CHANNEL =
"cache:invalidation";

    private final RedisTemplate<String, String> redisTemplate;
    private final RedisMessageListenerContainer
listenerContainer;
    private final LocalCache<?, ?> localCache;
    private final ObjectMapper objectMapper;
    private final String instanceId;

    // Для дедуплікації повідомлень
    private final Set<String> processedMessages =
ConcurrentHashMap.newKeySet();
```

```
    public SyncManager(RedisTemplate<String, String>
redisTemplate,
                       RedisMessageListenerContainer
listenerContainer,
                       LocalCache<?, ?> localCache,
                       ObjectMapper objectMapper) {
        this.redisTemplate = redisTemplate;
        this.listenerContainer = listenerContainer;
        this.localCache = localCache;
        this.objectMapper = objectMapper;
        this.instanceId = generateInstanceId();
    }

    @PostConstruct
    public void init() {
        listenerContainer.addMessageListener(this,
                new ChannelTopic(INVALIDATION_CHANNEL));
        log.info("SyncManager initialized, instanceId: {}",
instanceId);
    }

    /**
     * Публікує повідомлення про інвалідацію ключа.
     */
    public void publishInvalidation(String key) {
        InvalidationMessage message = new
InvalidationMessage(
                InvalidationType.SINGLE,
                key,
                null,
                System.currentTimeMillis(),
                instanceId
        );
        publish(message);
    }

    /**
     * Публікує повідомлення про інвалідацію за патерном.
     */
    public void publishPatternInvalidation(String pattern) {
        InvalidationMessage message = new
InvalidationMessage(
                InvalidationType.PATTERN,
                null,
                pattern,
                System.currentTimeMillis(),
                instanceId
        );
        publish(message);
    }

    private void publish(InvalidationMessage message) {
        try {
            String json =
objectMapper.writeValueAsString(message);

redisTemplate.convertAndSend(INVALIDATION_CHANNE
L, json);
            log.debug("Published invalidation message: {}",
message);
        } catch (JsonProcessingException e) {
            log.error("Error serializing invalidation message: {}",
                    e.getMessage());
        }
    }

    @Override
    public void onMessage(Message message, byte[] pattern) {
        try {
            String json = new String(message.getBody());
            InvalidationMessage invalidation =
objectMapper.readValue(
                    json, InvalidationMessage.class);

            // Пропускаємо власні повідомлення
            if (instanceId.equals(invalidation.sourceId())) {
                return;
            }

            // Дедуплікація
            String messageKey = invalidation.timestamp() + ":" +
                    invalidation.key() + ":" + invalidation.pattern();
            if (!processedMessages.add(messageKey)) {
                return;
            }

            // Обробка повідомлення
            processInvalidation(invalidation);

            // Очищення старих записів дедуплікації
            cleanupProcessedMessages();

        } catch (Exception e) {
            log.error("Error processing invalidation message: {}",
                    e.getMessage());
        }
    }

    private void processInvalidation(InvalidationMessage
message) {
        switch (message.type()) {
            case SINGLE:
                localCache.invalidate((Object) message.key());
                log.debug("Processed single invalidation for key:
{}",
                        message.key());
                break;
            case PATTERN:
                localCache.invalidateByPattern(message.pattern());
                log.debug("Processed pattern invalidation: {}",
                        message.pattern());
                break;
            case BATCH:
                // Реалізація для пакетної інвалідації
                break;
        }
    }

    private void cleanupProcessedMessages() {
        // Зберігаємо лише останні 10000 записів
        if (processedMessages.size() > 10000) {
            processedMessages.clear();
        }
    }

    private String generateInstanceId() {
        return
java.util.UUID.randomUUID().toString().substring(0, 8);
    }

    public record InvalidationMessage(
            InvalidationType type,
            String key,
            String pattern,
            long timestamp,
            String sourceId
    ) {}

    public enum InvalidationType {
        SINGLE, PATTERN, BATCH
```

```
    }
}

package com.hybrid.cache.stats;

import com.hybrid.cache.core.CacheLevel;
import com.hybrid.cache.core.CacheStatistics;
import org.springframework.scheduling.annotation.Scheduled;
import org.springframework.stereotype.Component;

import java.util.EnumMap;
import java.util.Map;
import java.util.concurrent.atomic.AtomicLong;
import java.util.concurrent.atomic.LongAdder;

@Component
public class StatisticsCollector {

    private final Map<CacheLevel, LevelStats> levelStats;
    private final LongAdder totalRequests;

    public StatisticsCollector() {
        this.levelStats = new EnumMap<>(CacheLevel.class);
        for (CacheLevel level : CacheLevel.values()) {
            if (level != CacheLevel.NONE) {
                levelStats.put(level, new LevelStats());
            }
        }
        this.totalRequests = new LongAdder();
    }

    public void recordHit(CacheLevel level) {
        LevelStats stats = levelStats.get(level);
        if (stats != null) {
            stats.hits.increment();
        }
        totalRequests.increment();
    }

    public void recordMiss(CacheLevel level) {
        LevelStats stats = levelStats.get(level);
        if (stats != null) {
            stats.misses.increment();
        }
    }

    public void recordLatency(CacheLevel level, long nanos) {
        LevelStats stats = levelStats.get(level);
        if (stats != null) {
            stats.totalLatencyNanos.add(nanos);
            stats.latencyCount.increment();
        }
    }

    public void recordEviction(CacheLevel level) {
        LevelStats stats = levelStats.get(level);
        if (stats != null) {
            stats.evictions.increment();
        }
    }

    public CacheStatistics getStatistics() {
        Map<CacheLevel, CacheStatistics.LevelStatistics>
levelStatistics =
                new EnumMap<>(CacheLevel.class);

        long totalHits = 0;
        long totalMisses = 0;
        double totalLatency = 0;
        int latencyLevels = 0;

        for (Map.Entry<CacheLevel, LevelStats> entry :
levelStats.entrySet()) {
            CacheLevel level = entry.getKey();
            LevelStats stats = entry.getValue();

            long hits = stats.hits.sum();
            long misses = stats.misses.sum();
            long total = hits + misses;

            double hitRate = total > 0 ? (double) hits / total : 0.0;
            double avgLatency = stats.latencyCount.sum() > 0
                    ? (double) stats.totalLatencyNanos.sum() /
                      stats.latencyCount.sum() / 1_000_000.0
                    : 0.0;

            levelStatistics.put(level, new
CacheStatistics.LevelStatistics(
                    hits,
                    misses,
                    hitRate,
                    avgLatency,
                    stats.evictions.sum()
            ));

            totalHits += hits;
            totalMisses += misses;
            if (avgLatency > 0) {
                totalLatency += avgLatency;
                latencyLevels++;
            }
        }

        long total = totalHits + totalMisses;
        double overallHitRate = total > 0 ? (double) totalHits /
total : 0.0;
        double avgLatency = latencyLevels > 0 ? totalLatency /
latencyLevels : 0.0;

        return new CacheStatistics(
                levelStatistics,
                overallHitRate,
                avgLatency,
                totalRequests.sum()
        );
    }

    @Scheduled(fixedRate = 60000) // Кожну хвилину
    public void logStatistics() {
        CacheStatistics stats = getStatistics();
        System.out.printf("Cache Stats - Hit Rate: %.2f%%, " +
                        "Avg Latency: %.2f ms, Total Requests:
%d%n",
                stats.overallHitRate() * 100,
                stats.averageLatency(),
                stats.totalRequests());
    }

    public void reset() {
        for (LevelStats stats : levelStats.values()) {
            stats.reset();
        }
        totalRequests.reset();
    }
```