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

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Гібридний метод кешування на основі ключових ідентифікаторів та метаданих»

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

_____ Артур ГАВОР
(підпис)

Виконав: здобувач вищої освіти групи ПДМ-62
Артур ГАВОР

Керівник: Богдан ХУДІК
доктор філософії (PhD)

Рецензент: Вікторія ЖЕБКА
зав. каф. ТЦР д. т. н. проф.

Київ 2025

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Інженерії програмного забезпечення

Ступінь вищої освіти Магістр

Спеціальність 121 Інженерія програмного забезпечення

Освітньо-професійна програма «Інженерія програмного забезпечення»

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

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Гавору Артуру Станіславовичу

1. Тема кваліфікаційної роботи: «Гібридний метод кешування на основі ключових ідентифікаторів та метаданих»

керівник кваліфікаційної роботи Богдан ХУДІК, доктор філософії (PhD),

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

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

3. Вихідні дані до кваліфікаційної роботи: науково-технічна література, аналіз існуючих алгоритмів кешування, математичні моделі та методи аналізу, критерії оцінки ефективності.

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

1. Огляд алгоритмів та методів кешування даних.
2.Розробка гібридного методу та моделі адаптивного кешування.
3. Експериментальне дослідження та тестування гібридного методу.

5. Перелік ілюстративного матеріалу: *презентація*

1. Мета, об'єкта та предмет дослідження.
2. Актуальність роботи.
3. Алгоритм гібридної адаптації.
4. Алгоритм узгодження даних реплікацій.
5. Алгоритм перевірки цілісності даних.
6. Архітектура гібридного кешування.
7. Результати тестування за законом амдала.
8. Результати тестування та порівняння у шаблонних патернах даних.
9. Результати тестування та порівняння у випадкових патернах даних .

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

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Постановка завдання та визначення мети дослідження	07.09-12.09.2024	
2	Аналіз наявної науково-технічної літератури	12.09.-19.09.2024	
3	Розробка концепції гібридного методу адаптивного кешування	20.09.-30.09.2024	
4	Проектування архітектури системи	01.10 - 09.10.2024	
5	Реалізація гібридного методу	10.10 - 24.10.2024	
6	Експериментальне дослідження та тестування	25.11 - 19.11.2024	
7	Порівняння з існуючими підходами	20.11 - 23.11.2024	
8	Розробка демонстраційних матеріалів	24.11 - 25.11.2024	
9	Попередній захист роботи	26.11.2024	
10	Оформлення роботи, вступ, висновки, реферат	17.12.2024	

Здобувач вищої освіти _____ Артур ГАВОР
(підпис)

Керівник
кваліфікаційної роботи _____ Богдан ХУДІК
(підпис)

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня магістра: 95 стор., 10 табл., 39 рис., 43 джерел.

Мета роботи – оптимізація кешування даних за рахунок використання гібридного методу на основі алгоритмів LRU та MRU з застосуванням механізму архівування, ключових ідентифікаторів та метаданих.

Об'єкт дослідження – процес кешування даних.

Предмет дослідження – методи та алгоритми кешування даних.

Аналіз методів та алгоритмів, зокрема LRU, MRU, LFU, ARC та інших, вивчено їхні принципи роботи, переваги та недоліки, а також на основі інших методів які використовують поєднання алгоритмів, було встановлено можливості їх комбінування для підвищення продуктивності системи.

Розроблений метод який базується на адаптації патернів, що дозволяє динамічно перемикається між алгоритмами LRU та MRU, що залежить від характеру метаданих, частота запитів яка вираховується через дисперсію, і також прийняття рішень щодо архівування даних.

За результатом експерименту гібридний метод забезпечує вищий коефіцієнт кеш-потрапляння та низьку затримку, найкраще він спрацював с різноманітних патернів доступу до даних. Розглянуто використання захист від несанкціонованого доступу та забезпечення цілісності даних та введено механізми для забезпечення узгодженості даних при паралельному доступі та архівуванні.

КЛЮЧОВІ СЛОВА: кешування даних, алгоритми кешування, гібридний метод, LRU, MRU, адаптивне кешування, метадані, архівування даних, продуктивність системи, безпека даних, шифрування, перевірка цілісності.

ABSTRACT

Text part of the master's qualification work: 95 pages, 39 pictures, 10 tables, 43 sources.

Purpose - to optimize data caching by using a hybrid method based on LRU and MRU algorithms with the use of an archiving mechanism, key identifiers, and metadata.

The object of research is the process of data caching.

The subject of research is data caching methods and algorithms.

The analysis of methods and algorithms, in particular LRU, MRU, LFU, ARC and others, studied their principles of operation, advantages and disadvantages, as well as on the basis of other methods that use a combination of algorithms, it was established the possibility of combining them to improve system performance.

A method based on pattern adaptation has been developed that allows dynamic switching between LRU and MRU algorithms, depending on the nature of the metadata, the frequency of requests calculated through variance, and decision-making on data archiving.

According to the experimental results, the hybrid method provides a higher cache hit ratio and low latency, and it worked best with a variety of data access patterns. The use of tamper-proofing and data integrity is considered, and mechanisms are introduced to ensure data consistency during parallel access and archiving.

Translated with DeepL.com (free version)

KEYWORDS: data caching, caching algorithms, hybrid method, LRU, MRU, adaptive caching, metadata, data archiving, system performance, data security, encryption, integrity check.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

ОЗП - Оперативно запам'ятовуючий пристрій

SSD - Solid State Drive

HTML - Hypertext Markup Language

LRU - Least Recently Used

LFU - Least Frequently Used

FIFO - First In, First Out

ARC - Adaptive Replacement Cache

MRU - Most Recently Used

AES - Advanced Encryption Standard

MD5 - Message Digest 5

NVMe - Non-Volatile Memory Host Controller Interface Specification

CDN - Content delivery network

TPS - Transactions Per Second

TPM - Transactions Per Minute

ВСТУП

В наш час, обсяги даних постійно зростають, що призводить до значного навантаження на інформаційні системи, зберігання даних та багаторазове повторення складних операцій створюють суттєві проблеми, знижуючи продуктивність та ефективність роботи системи.

Однією с ключових проблем, яке вирішує кешування, збереження часто використовуваних даних у тимчасовому сховищі, що дозволяє значно зменшити час очікування при зверненні до них. Завдяки цьому кешування дає можливість :

- зменшити затримки при взаємодії з системою, особливо в архітектурі клієнт-сервер;
- підвищити продуктивність системи без необхідності вдосконалення апаратного забезпечення сервера;
- знизити навантаження на систему зберігання даних;
- економити ресурси (пропускну здатність мережі).

Існують такі проблеми, з яким часто зустрічаються при кешовані, дані можуть не оновлюватися, що призводить до помилок на різних реплікаціях даних. Розміри інформації, може дублюватися що призводить до того, що дані які мали би зберігатися в одному екземплярі, зберігаються в декількох, що в свою чергу дає затрати на дисковий простір тощо.

Вирішенню вищезазначених проблем, призначенне відповідне програмне забезпечення, таке як Memcached [1;2], Redis [1;3], Rocksdb [4], Couchbase [5], Hazelcast [6], Apache Ignite [7], Ehcache [8], Coherence [9], CouchDB [10].

Методи кешування на основі ключів та метаданих ґрунтуватимуться на ідентифікації та керування даними в кеші

Об`єкт дослідження – процес кешування даних.

Предмет дослідження – методи та алгоритми кешування даних.

Мета роботи – оптимізація кешування даних за рахунок використання гібридного методу на основі алгоритмів LRU та MRU з застосуванням механізму архівування, ключових ідентифікаторів та метаданих.

Для досягнення поставленої мети в роботі передбачено вирішення наступних завдань:

- провести аналіз наявних алгоритмів, методів та підходів до кешування даних.
- розробити алгоритми для ефективного керування кешем, які включатимуть ключові ідентифікаторів та метадані.
- розробити програмну реалізацію методу гібридного кешування.
- проаналізувати результати тестування з іншими механізмами, методами, алгоритмами, визначити переваги та недоліки розробленого методу.

Методи дослідження включають математичну статистику, математичне моделювання, вставка, пошук, фільтрація та сортування на основі існуючих підходів до кешування даних, експериментальне тестування алгоритмів кешування.

Практична значущість результатів полягає в оптимізації управління кешем у високонавантажених системах, що забезпечує підвищення їхньої продуктивності та ефективності показників кеш-потрапляння та затримок.

Робота пройшла апробацію на науково-практичних конференціях:

1. Всеукраїнська науково-технічна конференція «Застосування програмного забезпечення в ІКТ»

2. ІІ Всеукраїнській науково-технічній конференції «Технологічні горизонти: дослідження та застосування інформаційних технологій для технологічного прогресу України і світу»

За результатами апробації опубліковано тези доповідей [11] та [12].

Робота складається зі вступу, трьох розділів, висновків, списку використаних джерел та додатків. У першому розділі проводиться аналіз літературних джерел та існуючих підходів. Другий розділ присвячений обґрунтування методів, що використовувалися для досягнення поставлених завдань. Третій розділ представляє практичні аспекти впровадження методів, аналіз отриманих результатів та їх обговорення.

1 ОГЛЯД АЛГОРИТМІВ ТА МЕТОДІВ КЕШУВАННЯ ДАНИХ

1.1 Типи та способи зберігання кешованих даних

Для збереження кешованих даних пропонується, різни підходи, та навіть комбінування у гібридні типи. Основні типи систем кешування: в пам'яті; на диску та гібридні системи.

Кешування в ОЗП (Оперативно запам'ятовуючий пристрій). Забезпечує швидкий доступ до даних, оскільки доступ до неї значно швидший, ніж до диска. Відомі програмні рішення для кешування, такі як Redis і Memcached, як раз і використовує цей підхід збереження даних[2; 3].

Використання постійної пам'яті (та/або жорсткі диски або SSD) для довготривалого зберігання в просторі системи. Це дозволяє зберігати більші обсяги даних у порівнянні з оперативною пам'яттю. Серед популярних рішень для даного типу, можна відзначити системи RocksDB і Ehcache, які забезпечують ефективне зберігання у довготривалій пам'яті [4; 5].

Поєднання постійної пам'яті та оперативної в гібридну систему дає переваги всіх інших типів, щоб досягти балансу оптимальної продуктивності та надійності, що характерно для ОЗП, та надійністю та масштабованістю постійної пам’яті. Гібридні системи, як показує практика динамічно підлаштовуються під характер даних та умов навантажень.

Провівши аналіз вище згаданих способів, підсумовуючи результати розглянути можна порівняльну таблицю 1.1, яка описує їхні переваги та недоліки. Наведена інформація допоможе сформувати уявлення про характеристики різних типів та способів кешування, що полегшить вибір оптимального рішення.

Таблиця 1.1

Порівняння характеристик типів та способів кешування даних

Назва	ОЗП	Постійна пам'ять	Поєднання постійної пам'яті та оперативної
Переваги	швидкий доступ до даних, низькі затримки	великий обсяг простору, дані зберігаються після знеструмлення	баланс швидкості та обсягу, що забезпечує оптимальну продуктивність
Недоліки	обмежений обсяг простору, втрата даних при знеструмленні	нижча швидкість доступу до даних, залежність від надійності диску	необхідно керування двома рівнями даних

Розглянувши даний розділ, перейдемо до обговорення різних типів систем кешування та підходів до їх реалізації

1.2 Типи систем кешування та підходи до кешування

Поширені типи систем керування кешем пропонують локальне та розподілене кешування. Також існує такий вид кешування, який працює на рівні програмного забезпечення, це може бути як кешування сесії, тимчасові дані програми, та інші ресурси.

Кешування на стороні клієнта (локальне кешування) передбачає зберігання даних безпосередньо на пристрої користувача, що дозволяє тривало зберігати дані для використання в автоматичному режимі [13]. На рисунку 1.1 зображено схему збереження та використання кешування даних. В даний спосіб можна включати кешування HTML-сторінок, API-запитів, зображень та інших ресурсів. Даний спосіб зберігання кешу дозволяє швидко доступатися до даних без необхідності

кожного разу звертатися до сервера, що підвищує продуктивність та ефективність роботи програм чи систем [14].

Рис. 1.1. Схема кешування на стороні клієнта

Час доступу до локального кешу менший, ніж час доступу до віддалених джерел даних. Це пов'язано з тим, що отримання даних з локальної пам’яті не потребує додаткових мережевих операцій, а визначається переважно лише швидкістю читання з пристрою.

$$T_{total} = T_{read}$$

де T_{total} – загальний час доступу до даних; T_{read} – час, необхідний для зчитування даних з локального кешу.

Розподілене кешування, передбачає зберігання даних у пам’яті на різних вузлах (комп’ютерів), об’єднаних у мережу. Використовуються в розподілених мікро сервісних системах, де доступ до даних може бути повільним через велику відстань або обмежену пропускну здатність [3]. Даний тип кешування працює за схемою, що дозволяє об'єднувати системи в один вузол, на рисунку 1.2 продемонстровано схему взаємодії, клієнт-серверну архітектуру між кешем та базою даних. Зазвичай розташовується окремо від системи, де будуть використовуватися тимчасові дані. Це робиться для того, щоб запобігти впливу кешу на продуктивність основної системи [1].

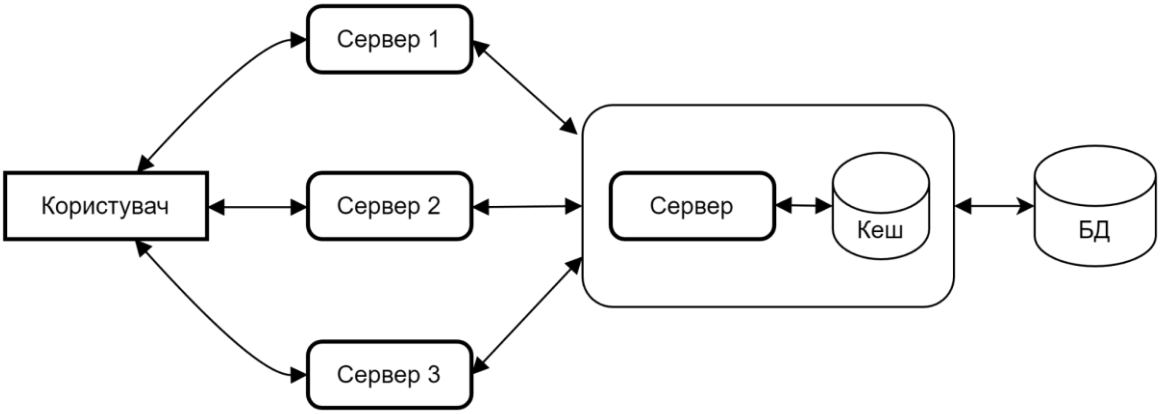


Рис. 1.2. Схема розподільного кешування

Математична модель розподіленого кешування, яка описує взаємодію між клієнтами, кеш-серверами та базою даних [15]. Система з N вузлами, де кожен вузол має локальний кеш можна розрахувати за формулу (1.1) загальне представлення для розрахунку часу доступу до даних у системах з розподіленим кешуванням [13].

$$T_{total} = \sum_{i=1}^{N} p(i) \cdot T_{cache}(i) + \big(1 - p(i)\big) \cdot T_{db}\,, \tag{1.1}$$

де N – кількість вузлів (комп'ютерів) у системі, на яких розподіллено кеш; i – індекс, який представляє кожен вузол у системі з N вузлів; $p(i)$ – ймовірність того, що дані вже кешовані на вузлі i; T_{cache} – час доступу до даних, які вже знаходяться в кеші вузла i; T_{db} – час доступу до основної бази даних, коли дані не знаходяться в локальному кеші.

Ефективність кешування E_{cache} можна оцінити у розподілених системах за формулою (1.2) [15;16].

$$E_{cache} = \frac{K_{\text{у}}}{K_{\text{З}}}\,, \tag{1.2}$$

де $K_{\text{у}}$ – кількість успішних звернень до кешу; $K_{\text{З}}$ – загальна кількість звернень.

Кешування на рівні програмного забезпечення, здебільш реалізується, як структура даних, що зберігає часто використовувані ресурси або результати

обчислень [17].Ця структура даних може бути розташована в пам'яті, на диску або в мережі.

Якщо розглядати, що кеш є структурою даних, тоді час доступу до кешу можна визначити за формулою (1.3) [17].

$$T_{cache} = T_{hit} \cdot P_{hit} + T_{mis} \cdot (1 - P_{hit}), \quad (1.3)$$

де T_{hit} – час доступу до кешу, якщо дані вже знаходяться в кеші; P_{hit} – ймовірність того, що запитані дані вже знаходяться в кеш; T_{mis} – час, необхідний для обчислення та збереження результату в кеші, якщо дані за кешовані.

Основні можливості способів зберігання та кешування даних , які було розглянуто вище, є різноманітними та популярними в сучасній індустрії розробки інформаційних систем [16;17]. Їх застосовують в умовах, коли потрібно обробляти однакову кількість даних або забезпечувати швидкий доступ до часто використовуваних ресурсів. З порівняльними можливостями можна ознайомитися в таблиці 1.2.

Сучасні інформаційні системи використовують різні методи зберігання та кешування даних. Їх застосовують у двох основних сценаріях обробка великих обсягів даних, яке дозволяє зменшити навантаження на джерело даних, розподіливши його на декілька серверів або локальних сховищ та другий сценарій забезпечення швидкого доступу до використовуваних даних в легкодоступному місці, наприклад, у пам'яті сервера або на локальному пристрої користувача[16;17;18].

Не дивлячись на різні методи та їх можливості, краще визначитися з типом даних, вони бувають такі як статичні, сеансові або динамічні дані, що с різним типом кешування дасть можливість, різну ефективність [17]. Рівень продуктивності, інформаційної системи, впливає на вибір методу типу даних та інші метрик [16;18].

Таблиця 1.2

Порівняння можливостей способів кешування даних

Назва	Локальне кешування	Розподілене кешування	Кешування на рівні програмного забезпечення
Зберігає дані на пристрої користувача.	так	ні	можливо
Розподіляє дані на декількох серверах	ні	так	ні
Вбудоване в програмне забезпечення	ні	ні	так
Зберігає статичні ресурси	так	так	так
Зберігає сеансові дані	так	так	так
Зберігає динамічні дані	так	так	так
Зберігає дані об'єктів	так	так	так

Також розглянемо переваги та недоліки характеристик кешування даних в таблиці 1.3 наочно описано переваги та недоліки кожного методу.

Один з факторів який впливає на кешування, обсяг даних, які потрібно кешувати. Якщо потрібно кешувати великий кластер даних, знадобитися використовувати розподілене кешування або кешування на рівні програмного забезпечення. Але все залежить від реалізації, та в необхідності використання того чи іншого методу.

Таблиця 1.3

Порівняння за типами характеристики кешування даних

Метод	**Переваги**	**Недоліки**
Локальне кешування	швидкий доступ до незалежних від мережі даних	обмежена місткість, відсутність синхронізації
Розподілене кешування	масштабованість, висока доступність, гнучкість	складність налаштування, мережеві затримки
Кешування на рівні програмного забезпечення	гнучкість, контроль	витрати на розробку, залежність від середовища

Кожен із розглянутих типів та способів зберігання кешованих даних, має свої унікальні переваги та недоліки, що робить їх корисними для різних сценаріїв їх використання дивлячись від поставленої задачі. Вибір методу залежить від вимог до продуктивності, масштабованості та потреб. Локальне кешування підходить – для швидкого доступу до даних, які знаходяться на клієнтській стороні, розподілене кешування – для масштабованих та високо доступних систем на рівні програмного забезпечення, забезпечує гнучкість і контроль у контексті конкретного додатка.

Оглянувши типи та способи зберігання кешу, потрібно приступити до самого головного, розглянувши алгоритми, які оптимізують кеш, щоб постійно не зберігати копію даних.

1.3 Алгоритми кешування

Алгоритми кешування даних є основним методома оптимізаціїї та балансування даних, вони влаштовані так, що коли до даних звертаються, елементи копіюють в сховище та потім керують цими копіями, щоб максимально підвищити

продуктивність системи. Наданий час є різноманітні алгоритми, кожен з яких використовує свої підходи, методики тощо, для того щоб визначити який елемент видаляти та який залишити.

Розглянемо популярні алгоритми кешування: Least Recently Used (LRU), Least Frequently Used (LFU), First In, First Out (FIFO), Adaptive Replacement Cache (ARC) та Most Recently Used (MRU).

1.3.1 Алгоритм LRU

Одним із найпоширеніших алгоритмів кешування є LRU (Least Recently Used). Його основна ідея полягає в тому, щоб видаляти з кешу той елемент, який найменше використовувався. Забезпечує актуальність збережених даних і зменшує ймовірність повторного звернення до видалених елементів [16;18].

Алгоритму LRU можна представити як абстрактну колекцію даних у формальному визначенні (1.4), а також множина часових міток (1.5) [18].

$$C = (k_1, v_1), (k_2, v_2), \dots, (k_n, v_n) n \leq \mathrm{c}\,, \tag{1.4}$$

$$t = (k_i, t_i) | i = 1, \dots, n\,, \tag{1.5}$$

де C – колекція кешованих даних; k – ключ; v – значення; c – виділений об'єм під зберігання кешу; n – кількість об'єктів у кеші; i – індекс, ітерації по об'єктах у кеші; t – дискретний часовий крок, на якому відбуваються операції з кешем.

Зчитування даних з колекції можна описати, як функцію доступу до об'єкта в кеші (1.6) та запис даних у колекцію (1.7) [16].

$$get(k) = v_i, (k_i, v_i) \in C\,, \tag{1.6}$$

$$f(x) = \begin{cases} C \cup \{(k,v),\ (v,k) \in C \\ t \cup \left(k, t_{t_{current}}\right), (k_i, v_i) \in C \wedge |C| < c \\ C - (k_{LRU}, v_{LRU}), k \in C \wedge |C| = c \\ t - (k_{LRU}, v_{LRU}) \cup \left(k, t_{t_{current}}\right), k \in C \wedge |C| = c \end{cases}, \quad (1.7)$$

де $get(k)$ – функція доступу до об'єкта з ключем k; $put(k,v)$ – функція вставки об'єкта за ключем k та значенням v; $current$ – поточна часова мітка; k_{LRU} – ключ найменш недавно використаного елемента; v_{LRU} – недавно використаний елемент; t_{LRU} – час останнього доступу до найменш використаного елемента.

Алгоритм LRU гарантує що елементи, які використовувалися найменше, ідентифікуються та видаляються, коли це необхідно. Ознайомитися з блок-схемою алгоритму можна на рисунку 1.3 [16;18].

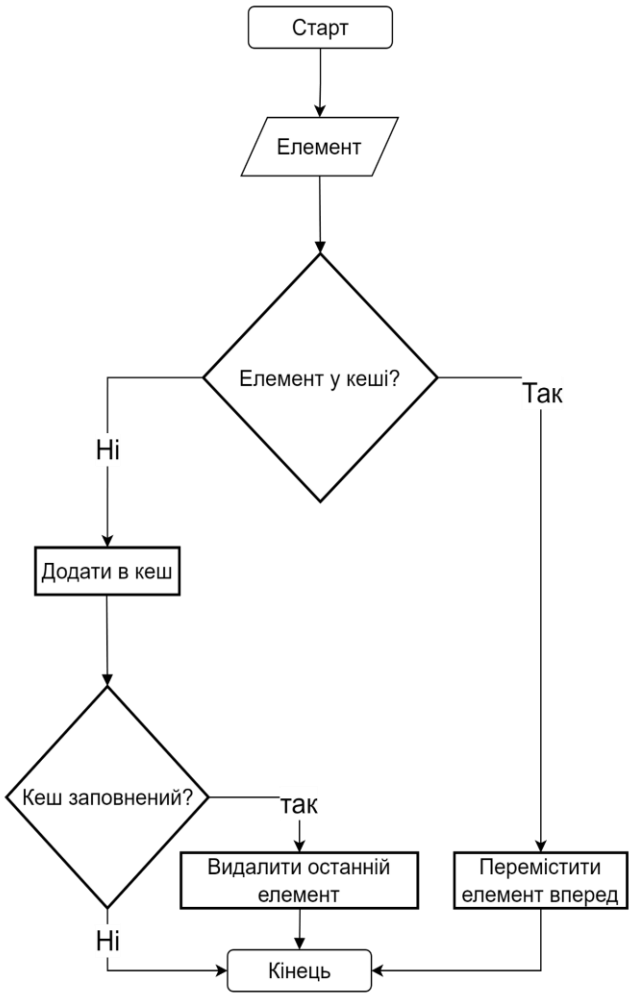


Рис. 1.3. Блок-схема алгоритму LRU

1.3.2. Алгоритм LFU

Алгоритм LFU (Least Frequently Used) керування кешем, який базується на частоті доступу до даних. Основною метою LFU є збереження в кеші елементів, до яких звертаються найчастіше, і видалення тих даних, до яких звертаються найменше.

На рисунку 1.4 зображено блок-схему алгоритму LFU, яка ілюструє кроки та процеси, що виконуються під час роботи цього алгоритму. Блок-схема візуалізує, як відбувається оновлення частот використання елементів у кеші та як здійснюється видалення найрідше використовуваних елементів для забезпечення ефективного управління пам'яттю [18].

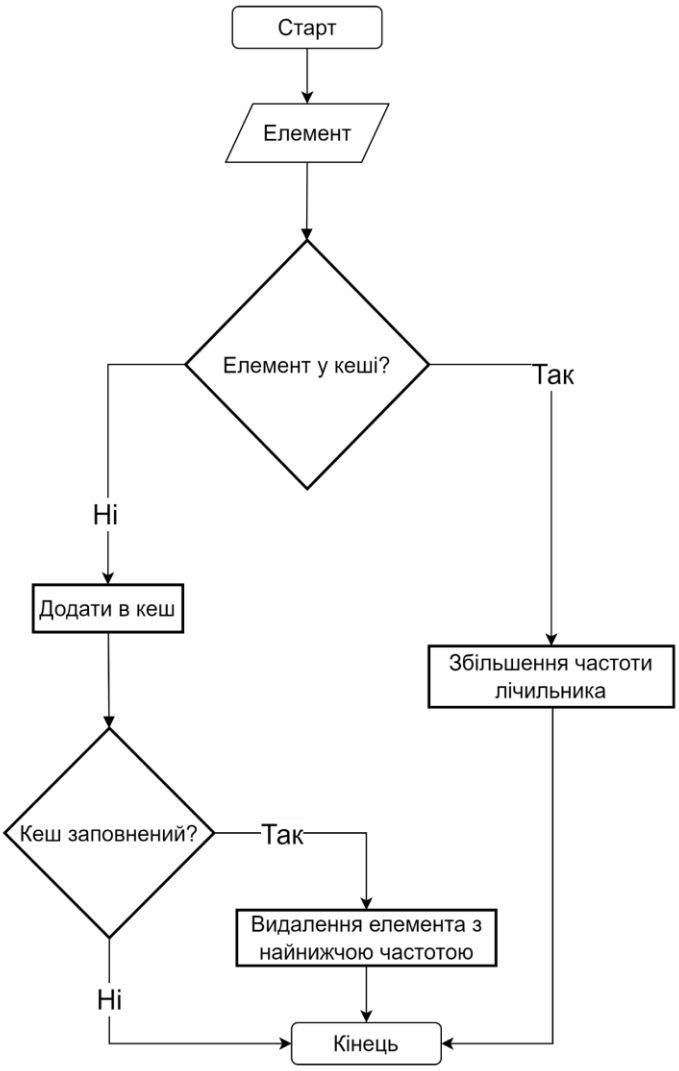


Рис. 1.4. Блок-схема алгоритму LFU

В алгоритмі LFU відстежується не тільки ключем та значення, але й частоту доступу до кожного елемента. Колекцію кешованих даних можна представити, як набір пар ключ-значення та частоти доступу до даних (1.8).

$$C(t) = (k_1, v_1, f_1), (k_2, v_2, f_2), \dots, (k_n, v_n, f_n) n \leq \text{c}, \quad (1.8)$$

де f – частота доступу;

Після здійснення доступу до елемента частота його доступу збільшиться на 1 – також початкова частота доступу до нового елемента. Як приклад, здійснимо доступ до елемента k_1 то колекція даних буде змінена (1.9).

$$C = (k_1, v_1, f_1 + 1), (k_2, v_2, f_2), \quad (1.9)$$

Вставка нового елемента у колекцію даних за формулою (1.10).

$$cache(t+1) = cache(t) \cup (k_t, v_t, 1), \quad (1.10)$$

де $cache(t)$ — набір елементів у кеші на момент часу t; k_t – ключ нового елемента; v_t – значення нового елемента.

Видалення елемента с кешу за формулою (1.11).

$$cache(t+1) = (cache(t) \setminus y) \cup (k_t, v_t, 1), \quad (1.11)$$

де y – елемент із найнижчою частотою доступу.

Оновлення частоти доступу до даних відбувається, якщо елемент з ключем k_t знаходиться в кеші (1.12) [16].

$$f(k_t) = k_t + 1, \quad (1.12)$$

де $f(k_t)$ – частота доступу до елемента k_t, яка збільшується на 1.

Лічильник частоти доступу до елемента в кеші можна представити за формулою (1.13) індикаторна функція, що дорівнює 1, якщо $k_i = k$, і 0 в іншому випадку [16].

$$f(k) = \sum_{i=1}^{n} 1(k_i = k)\,, \tag{1.13}$$

де $f(k)$ – лічильник частоти.

Алгоритм LFU забезпечує ясність та компактність математичного представлення.

1.3.3. Алгоритм FIFO

Даний алгоритм частково не являться алгоритмом, бо використовує поєднання різних підходів в один метод, але оскільки ці підходи описуються як управління запасами, обробка даних та в обчислювальній техніці. То цей метод і стали використовувати, як алгоритм кешування.

FIFO використовують у сферах управління запасами та оцінки запасів. Він передбачає, що перші елементи, додані до запасів, будуть першими використаними [18;19]. Нагадує природний рух товарів, які швидко псуються. На рисунку 1.4 показана блок-схема, яка описує дії алгоритму, наприклад можна це пояснити так: додається новий елемент до кешу, але він вже заповнений, то перший на вихід буде останнім елементом, даному випадку його просто видаляють. Це доволі зручно коли необхідно підтримувати постійну подачу елементів.

Колекція кешованих даних $cache(t)$ на момент часу t визначається як множина пар, де кожна пара містить ключ:

$$cache(t) = (k_1, v_1), (k_1, v_1), \dots, (k_n, v_n) n \leq \text{c}\,,$$

Внесення нового елемента:

$$cache(t+1) = cache(t) \cup (k_t, v_t)\,, \tag{1.14}$$

Видалення елемента с кешу:

1. Знаходимо найстаріший елемент у кеші:

$$o = (k_t, v_t)\,, \tag{1.15}$$

2. Видаляємо найстаріший елемент з кешу:

$$cache(t+1) = cache(t)\{o\}\,, \tag{1.16}$$

3. Додаємо новий елемент з ключем k_t і значенням v_t:

$$cache(t+1) = (cache(t)\{o\} \cup \{(k_t, v_t)\}), \quad (1.17)$$

де o – найстаріший елемент у кеші, який видаляється.

Цей підхід дозволяє уникнути повторень і забезпечує чітке математичне представлення колекції кешованих даних в алгоритмі FIFO.

Рис. 1.5. Блок-схема алгоритму FIFO

1.3.4. Алгоритм ARC

Алгоритм ARC (Adaptive Replacement Cache), був розроблений Німродом Мегіддо та Дейвідом С. Модом у 2003 році, який дозволяє ефективніше використовувати пам’ять за допомогою адаптивного визначення розміру елементів на основі частоти їх доступу [20;21].

Метод керування даного алгоритма полягає у створення двухзон(пар) елементів які будуть зберігати інформацію одночасно враховуючи параметр, як недавні запити, так і частоту доступу до даних [20;21]. Завдяки цьому відбувається адаптація, адаптивного керування, що дозволяє оптимізувати використання обмеженого обсягу пам'яті [20]:

- T1 – нещодавно використані об’єкти;
- B1 – видалені об'єкти, які були в списку T1;
- T2 – часто використовувані об'єкти;
- B2 – видалені об'єкти, які були в списку T2.

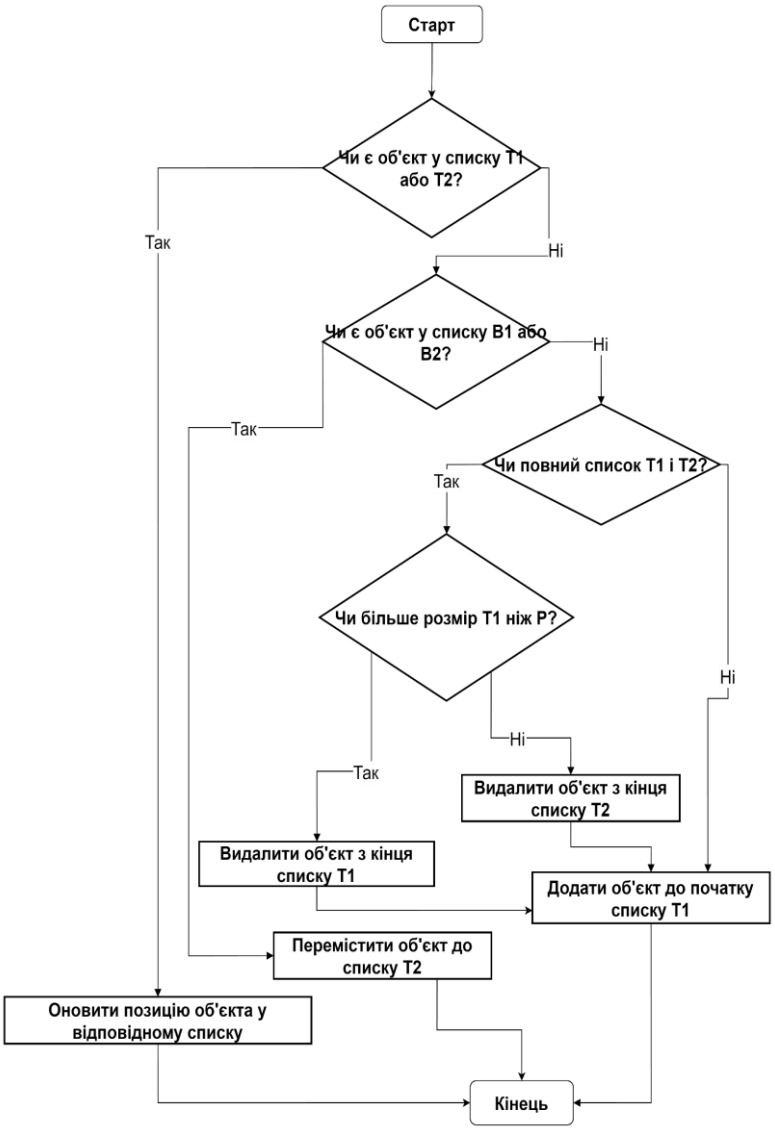


Рис. 1.6. Блок-схема алгоритму ARC

ARC динамічно регулює розмір цих списків на основі поведінки доступу до даних, що дозволяє ефективно адаптуватися до змін у патернах доступу. Коли об'єкт запитується, ARC вирішує, в який зі списків помістити об'єкт, та оновлює списки відповідно. На зображенні 1.6. вказано блок схема алгоритм дій ARC.

Оновлення списків T1 та T2. Зсув елемента до кінця T2 (1.18) та T1 (1.19), при умові, що : $x_t \in T1(t) \cup T2(t)$[21].

$$T2(t+1) = T2(t) \cup x_t\,, \tag{1.18}$$

$$T1(t+1) = T1(t)\; x_t\,, \tag{1.19}$$

де x_t – найстаріший елемент у кеші, який видаляється; t – часовий крок; $T1$(t) – список елементів, які були нещодавно додані у кеш; $T2$(t) – список елементів, які довго знаходяться у кеші;

Адаптивне оновлення даних. Треба збільшити значення p, щоб збалансувати частоту використання між B1 та B2 за формулою (1.20), якщо елемент x_t знаходиться в B1, та коли елемент x_t знаходитися в B2 треба зменшити значення p (1.21) [21].

$$p = min\left(p + max\left(\frac{|B2(t)|}{|B1(t)|}, 1\right), C_{max}\right), \tag{1.20}$$

$$p = max\left(p + min\left(\frac{|B1(t)|}{|B2(t)|}, 1\right), 0\right), \tag{1.21}$$

де p – параметр адаптивного розподілу між T1 та T2; C_{max} – найстаріший елемент у кеші, який видаляється; $B1$(t) – список елементів, що були видалені з $T1$, але ще не видалені з кешу; $B2$(t) – список елементів, що були видалені з $T2$, але ще не видалені з кешу.

Якщо елемент не знаходиться у жодному зі списків, перевіряємо чи кеш заповнений. Сумарний розмір списків одного рівня, який більше за максимальною можливу кількість елементів у кеші за формулою (1.22) тоді сумарний розмір всіх списків менше або дорівнює максимальній сумі кешу (1.23) та також сума дорівнює двом максимальним сумам кешу (1.24) то видаляємо елемент зі списку, що були видалені с іншого рівня списку (1.25) інакше видаляємо елемент зі списку, що нещодавно доданий (1.26) [20;21].

$$|T1(t)| + |B1(t)| < C_{max}\ , \tag{1.22}$$

$$|T1(t)| + |T2(t)| + |B1(t)| + |B2(t)| \geq C_{max}\ , \tag{1.23}$$

$$|T1(t)| + |T2(t)| + |B1(t)| + |B2(t)|\ = 2C_{max}\ , \tag{1.24}$$

$$B2(t+1) = B2(t)\ x_{old}\ , \tag{1.25}$$

$$T1(t+1) = T1(t) \cup x_t \tag{1.26}$$

де x_{old} – найстаріший елемент.

1.3.5. Алгоритм MRU

Алгоритм MRU (Most Recently Used), політика керування кеш-пам'яттю, яка є прямою протилежністю до політики найменшого використання LRU. Якщо в кеш-пам'яті немає вільного місця, то в таких випадках, щоб звільнити місце, MRU видалить об'єкт, до якого звернулися останнім разом [22]. Кожного разу, коли здійснюється доступ до об'єкта, він оновлюється, щоб зберегти позицію, яка вказує на те, що він є найбільш доступним об'єктом. У вигляді блок-схеми можна розглянути даний алгоритм за рисунком 1.7. Алгоритм ефективний у певних типах навантаження, де повторний доступ до найчастіше використаних даних є малоймовірним.

Доступ до елемента відбувається, якщо значення його є в кеші:

$$v_t = v_i\ , \tag{1.27}$$

де t – Дискретні часові кроки; v_t – ключ елемента, до якого здійснюється доступ у момент часу t; v_i - значення i - го елемента.

Додавання нового елемента відбувається коли кеш не заповнений:

$$cache(t+1) = cache(t) \cup v_t, v_i\ , \tag{1.28}$$

$$|cache(t)| <\ C_{max}\ , \tag{1.29}$$

де $cache(t)$ – набір пар у кеші на момент часу t.

Видаляємо найбільший недавно використаний елемент (останній доданий елемент) за умови, що кеш заповнений:

$$cache(t+1) = cache(t)\, x_{new}\,, \tag{1.30}$$

$$|cache(t)| = C_{max}\,, \tag{1.31}$$

де x_{new} – дискретні часові кроки.

Рис. 1.7. Блок-схема алгоритму MRU

Після поглибленого вивчення основних алгоритмів кешування, наведених вище, варто звернути вашу увагу на те, що їх ефективність залежить від ситуації. Однак для більш точного визначення та покращення продуктивності кешування необхідно застосовувати математичні моделі та методи аналізу. Лише за допомогою цих моделей буде здійснено загальний огляд реалізованості різних

алгоритмів і точного налаштування системних параметрів для досягнення максимально можливого рівня продуктивності.

Давайте розглянемо математичні моделі взаємодії користувача та кешу, які дозволять краще зрозуміти поведінку системи щодо кешованих даних.

Після розгляду основних особливостей алгоритмів кешування LRU, LFU, FIFO, ARC та MRU, доцільно систематизувати їхні характеристики. У таблиці 1.4 наведено порівняння ключових характеристик цих алгоритмів.

Таблиця 1.4

Порівняння характеристики алгоритмів

Алгоритм кешування	**Переваги**	**Недоліки**	**Складність алгоритму**
LRU	простота реалізації, ефективність	може не враховувати частоту використання	O(1) при використанні хеш-таблиці та двозв'язного списку
LFU	ефективний для часто використовуваних даних	може не враховувати не давність використання	O(log_2 n) при використанні збалансованого дерева або O(1) з додатковими структурами даних
FIFO	простота реалізації, передбачуваність	може видаляти часто використовувані елементи	O(1)
ARC	висока ефективність у різному сценарію	складність реалізації, високе споживання пам'яті	O(1) для вставки і видалення, O(1) для доступу з хеш-таблицею
MRU	ефективний для деяких специфічних сценаріїв	не підходить для більшості загальних випадків	O(1)

1.4 Математичні моделі та методи аналізу продуктивності кешування

Моделювання поведінки користувачів і оцінка ймовірностей доступу до різних даних є ключовими для розуміння ефективності кешування. Ймовірність P-доступу використовується для визначення ймовірності того, що певний елемент буде запитано за формулою (1.32).

$$P(i) = \frac{n_i}{n_{total}}, \tag{1.32}$$

де $P(i)$ – ймовірність того, що конкретний елемент i буде запитано користувачем або системою; n_i – кількість звернень до елемента i; n_{total} – загальна кількість доступів.

Модель черги (queueing theory) використовуються для оцінки W середнього часу очікування і обробки запитів за формулою (1.33) в системах кешування.

$$W = \frac{\lambda}{\mu(\mu-\lambda)}, \tag{1.33}$$

де λ – інтенсивність запитів; μ – інтенсивність обслуговування.

Аналіз пропускної здатності (Throughput analysis). Оцінка пропускної здатності системи для визначення максимального числа запитів, які система може обробляти за одиницю часу. Пропускну здатність T можна оцінити за формулою (1.34).

$$T = \frac{N}{T_{total}}, \tag{1.34}$$

де N – кількість оброблених запитів; T_{total} – загальний час обробки.

Моделювання ефективності кешу. Ефективність кешу часто вимірюється за допомогою коефіцієнта кешування (cache hit ratio), який визначає частку запитів за формулою (1.35), що обслуговуються кешем.

$$H = \frac{n}{n_r}, \tag{1.35}$$

де H – коефіцієнт кешування; n – кількість звернень до кешу; n_r – загальна кількість запитів.

Вибір оптимального розміру кешу можна здійснити шляхом аналізу функції вартості та ефективності. Наприклад, якщо відомо, що ефективність кешування збільшується пропорційно до логарифма розміру кешу, можна використовувати модель (1.36).

$$E(s) = a \cdot log_2(s) + b\,, \tag{1.36}$$

де E – функція ефективності; s – розмір кешу; a і b – константи.

Визначення часу відповіді системи (1.37). Для оцінки середнього часу відповіді системи можна використовувати моделі черг.

$$R = W + \frac{1}{\mu}, \tag{1.37}$$

де R – середній час відповіді.

Математичні моделі і методи дозволяють точно оцінювати продуктивність систем кешування, визначати оптимальні параметри і прогнозувати навантаження, що значно підвищує ефективність інформаційних систем.

1.5 Висновки до розділу

Було розглянуто типи систем керувань, методи, та алгоритми управлінням кешем, ці механізми відіграють важливу роль в продуктивності зберігання та обробки даних. Вибір відповідного механізму або набір механізмів залежить від конкретних цілей та вимог до ефективності системи.

Локальне кешування мінімізує затримки при доступі до даних у клієнтських додатках, розподілене забезпечує масштабованість у великих системах, а кешування на рівні програмного забезпечення надає контроль над обчисленнями й

доступом до ресурсів. Кожен із цих підходів має свої переваги й недоліки, що залежать від специфіки використання.

Особливу увагу буде приділено використанню метаданих, таких як частота доступу, дисперсія патернів, пріоритети даних та розмір об’єктів, які дозволяють адаптувати вибір алгоритму кешування залежно від характеру даних. Запропоновано концепцію адаптивного вибору між LRU та MRU, яка базується на аналізі метаданих у реальному часі.

Таким чином, у розділі 1 закладено теоретичну основу для подальшого дослідження та розробки гібридного методу кешування, що спрямований на підвищення продуктивності систем за рахунок адаптивного управління пам’яттю.

В наступному розділ розглянемо та змоделюємо поєднання двох алгоритмів шляхом адаптації даних методу та додамо деякі технології які допоможуть моєму гібридному методу конкурувати з іншими алгоритмами, методами, тощо та збільшити пропускну здатність кешування.

2 АНАЛІЗ АДАПТИВНОГО КЕШУВАННЯ ТА ВВЕДЕННЯ МОДЕЛІ ГІБРИДНОГО МЕТОДУ

У розділі розглядається метод оптимізації кешування даних, що базується на адаптації алгоритмів, таких як LRU і MRU поєднанням механізмом архівування. Метою даного підходу є покращення продуктивності системи завдяки поєднанню двох різносторонніх алгоритмів кешування, що можуть в теорії покращити продуктивність, як це було продемонстровано в досліджені [23]. Для цього потрібно лише розробити, механізми, та моделі адаптації [23; 24].

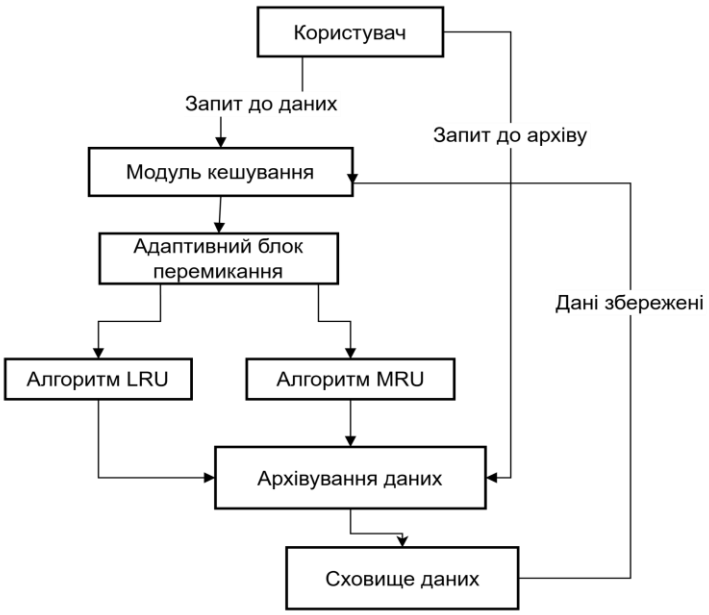


Рис. 2.1. Архітектура гібридного кешування

Розглянемо архітектуру запропонованого методу гібридного кешування на рисунку 2.1, де демонструється запит від користувача який надходить безпосередньо до модуля кешування. Якщо дані відсутні в кеші, запит перенаправляється до архіву для зберігання або отримання збережених даних. Ключовим елементом архітектури є адаптивний блок перемикання, який, на основі аналізу запитів, вирішує, який алгоритм кешування – LRU чи MRU – використовувати в даний момент. Під час завантаження даних у кеш за потреби

задіюється механізм архівування, а сховище даних виступає, як резервний ресурс для збереження інформації [16;17].

2.1 Комбінування алгоритмів кешування

Основою гібридного методу кешування є два алгоритма, LRU, що видаляє не часто використовувані об'єкти, а MRU видаляє дані, які були використані останніми [18; 22]. Цей підхід дозволить балансувати між збереженням часто запитуваних ресурсів і раціональним використанням пам'яті.

Для поєднання даних алгоритмів, потрібно розуміти, що одночасно вони не можуть працювати, і потрібно модель адаптації, яка відбувається на основі поточних патернів доступу. Потрібно аналізувати основні метрики адаптації, такі як, частота доступу – кожен об'єкт у кеші оцінюється на основі частоти звернення до нього за певний період часу, аналіз патернів, який оцінює доступ до об'єктів – регулярні (стабільні) або нерегулярні (сплескові). І на основі цих параметрів, перемикатися між алгоритмами. Якщо частота доступу висока і стабільна, використовувати MRU, оскільки найновіші дані, ймовірно, більше не будуть потрібні. Якщо ж доступ нерегулярний, використовувати LRU, щоб зберігати в кеші дані, які застосовуються частіше [17; 22]. Отже виходячи з цього можна представити модель адаптації (2.1).

$$S(t) = \begin{cases} LRU, f_{lru}(t) > f_{mru}(t) \\ MRU, f_{lru}(t) \leq f_{mru}(t) \end{cases} \quad (2.1)$$

де S – частота доступу в алгоритмі; $f_{lry}(t)$ – частота доступу до даних у кеші LRU за час t; $f_{mry}(t)$ – частота доступу до даних у кеші MRU за час t.

Для кращого розуміння стабільності патернів можна вираховувати дисперсію частоти доступу. Дисперсія, вимірює варіацію частоти запитів до даних, дозволяє оцінити стабільність та змінність патернів доступу. Чим вища дисперсія, тим менша стабільність алгоритму, що дозволяє визначити, наскільки оптимально

використовувати один із алгоритмів (2.2) та (2.3). Для цього необхідно визначити середнє значення частоти доступу (2.4) де кожне значення $f(i)$ зважене на відповідний час Δt_i та суму всіх часових інтервалів (2.5) протягом яких проводилися вимірювання частоти доступу.

$$\sigma^2_{LRU} = \frac{1}{T}\sum_{i=1}^{n}(f_{lry}(t_i) - \bar{f})^2 \Delta t_i\,, \tag{2.2}$$

$$\sigma^2_{MRU} = \frac{1}{T}\sum_{i=1}^{n}(f_{mry}(t_i) - \bar{f})^2 \Delta t_i\,, \tag{2.3}$$

$$\bar{f} = \frac{\sum_{i=1}^{n}(f(i) \times \Delta t_i)}{\sum_{i=1}^{n} \Delta t_i}, \tag{2.4}$$

$$T = \sum_{i=1}^{n} \Delta t_i\,, \tag{2.5}$$

де σ^2_{LRU} – дисперсія частоти доступу для алгоритму LRU; σ^2_{MRU} – дисперсія частоти доступу для алгоритму MRU; T – загальний час; $\bar{f}$ – середнє значення частоти доступу для кожного алгоритму за період часу T; $f_{lry}(t_i)$ та $f_{mry}(t_i)$ – частота доступа до певного ключа у кеші, на момент часу t_i; n - кількість окремих моментів часу t_i , на яких здійснюється вимірювання показників.

Виходячи з того, що простий підрахунок частоти доступу до даних не завжди відображає реальну потребу у видалення даних, замість нього використаємо дисперсію для прийняття рішення. Висока дисперсія свідчить про те, що доступ до даних є нестабільним, і це може вказувати на те, що дані, які використовувалися нещодавно, більше не потрібні. В такому випадку, адаптивна модель кешування обирає алгоритм, який максимально відповідає поточному паттерну (2.6) запитів [23;24]:

$$S(t) = \begin{cases} LRU & \sigma^2_{LRU} < \sigma^2_{MRU} \\ MRU & \sigma^2_{MRU} \le \sigma^2_{LRU} \end{cases} \tag{2.6}$$

Щоб динамічно пристосовуватись до змін у патернах доступу, використовують метод експоненціального згладжування (2.7) для обчислення середньої частоти доступу:

$$F_i(t) = a \times X_i(t) + (1 - a) \times F_i(t-1)\,, \tag{2.7}$$

де $F_i(t)$ – згладжена частота доступу до елемента і в момент часу t; $X_i(t)$ – фактична кількість звернень до елемента i за останній період часу; a – коефіцієнт згладжування.

На рисунку 2.2 нижче представлена блок-схема процесу кешування з адаптивним перемиканням між алгоритмами LRU та MRU. Коли надходить новий запит, система перевіряє наявність даних у кеші. Якщо дані вже кешовані, вони повертаються користувачеві. У випадку відсутності даних у кеші, система аналізує патерни доступу до даних, розраховуючи дисперсію частоти доступу для кожного алгоритму – LRU та MRU.

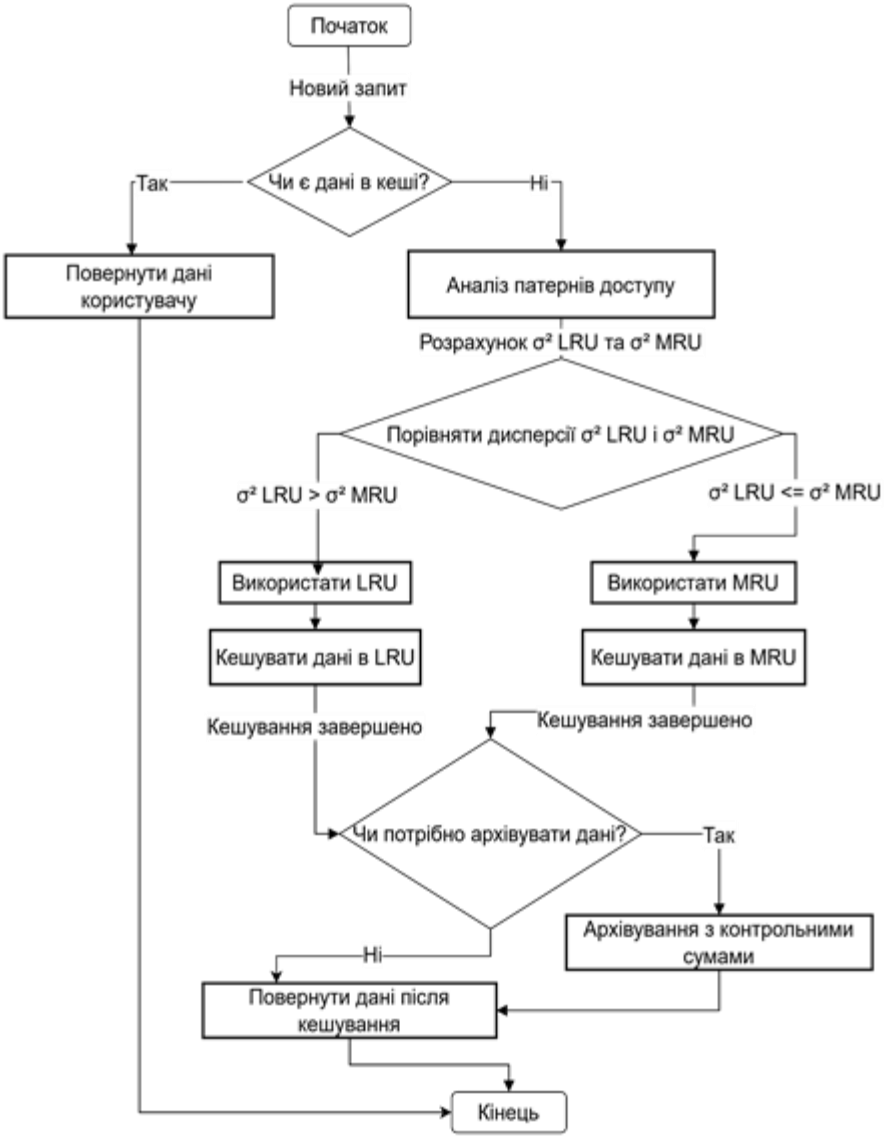


Рис. 2.2. Блок-схема адаптивного кешування

Процес адаптивного кешування. На основі результатів розрахунків система порівнює дисперсії: якщо дисперсія для LRU вища за MRU, обирається LRU, інакше – MRU. Після кешування система перевіряє, чи потрібно архівувати дані. Якщо так, то відбувається архівування з додаванням контрольних сум для забезпечення цілісності даних.

2.2 Архівування даних та метадані

В архітектурі запропонованого гібридного методу кешування, архівування даних із використанням контрольних сум відіграє роль забезпечення довготривалого зберігання та контролю цілісності даних, що не активно використовуються.

У процесі роботи з кешем деякі дані, залежно від частоти звернень та тривалості їх зберігання, можуть бути переміщені в архів. Це допоможе зменшити навантаження на кеш та оптимізувати доступ до найчастіше використовуваних даних. Нижче наведено детальний опис кроків процесу архівування з використанням контрольних сум для забезпечення цілісності даних.

Перевірка необхідності архівування:

- процес кешування алгоритм визначає, чи потрібно архівувати дані. Це може залежати від низки критеріїв, зокрема від того, як часто дані використовуються (2.8) та як довго вони зберігаються у кеші:

$$f(x) < f_{threshold}\,, \tag{2.8}$$

- якщо дані рідко записуються або перебувають у кеші довше встановленого терміну (2.9), їх вважають кандидатами на архівування:

$$t_{cache}(x) > t_{max}\,, \tag{2.9}$$

де $f(x)$ – частота звернень до даних x у певний період часу T; $f_{threshold}$ – мінімальна кількість доступу до даних за певний проміжок часу; $t_{cache}(x)$ – час перебування об'єкта x у кеші; t_{max} – максимальний допустимий час зберігання даних у кеші.

За формулою 2.6 встановлюється, що якщо частота доступу до даних x менша за порогове значення $f_{threshold}$, ці дані вважаються рідко використовуваними та можуть бути за архівовані. Для більшої точності, частоту доступу $f(x)$ можна визначити (2.10) як:

$$f(x) = \frac{n(x)}{T}, \tag{2.10}$$

де $n(x)$— кількість звернень до даних x за період T.

Якщо дані x перебувають у кеші довше за t_{max}, вони вважаються застарілими та можуть бути заархівовані, виходячи с цього час перебування в кеші можна визначити (2.11) так:

$$t_{cache}(x) = t_{current} - t_{insertion}(x), \tag{2.11}$$

де $t_{current}$ – поточний час; $t_{insertion}(x)$ – час, коли дані x були додані в кеш.

Як зазначено раніше, система обирає між LRU та MRU на основі дисперсії частоти доступу (2.12):

$$(f(x) < f_{threshold} \vee t_{cache}(x) > t_{max}) \wedge \text{p(x)} = 0, \tag{2.12}$$

де p(x) — функція пріоритету даних.

Функція $p(x)$ визначена як бінарна функція, що приймає значення 0 або 1:

- 1, якщо дані *x* мають високий пріоритет (не архівуються);
- 0, якщо дані *x* можуть бути за архівовані.

Можна використовувати ймовірність доступу *p(x)* (2.13) для прийняття рішення про архівування:

$$p(x) = \frac{n(x)}{N}, \tag{2.13}$$

де N — загальна кількість звернень до всіх даних за період T.

Якщо $p(x)$ менше за певний поріг $p_{threshold}$ (2.14), дані можна архівувати:

$$p(x) < p_{threshold}. \tag{2.14}$$

Перед архівуванням для кожного блоку даних обчислюється контрольна сума (хеш-значення), яка представляє "підпис" даних на момент архівування. Контрольна сума може обчислюватися за допомогою криптографічних хеш-

функцій, таких як SHA-256, MD5 або інших відповідних алгоритмів [25]. Для кожного блоку даних x перед архівуванням обчислюємо контрольну суму (2.15) $h(x)$:

$$h(x) = hash(x)\,, \tag{2.15}$$

де hash(x) – криптографічна хеш-функція.

Для обчислення контрольних сум використовується криптографічна хеш-функція, яка забезпечує унікальність та стійкість до колізій. У контексті систем кешування та архівування важливо обрати хеш-функцію, мною було обрано SHA-256 [25]. Порівняння популярних хеш-функцій представлено в таблиці 2.1.

Таблиця 2.1

Порівняння популярних хеш-функцій

Хеш-функція	Швидкість обчислення	Стійкість до колізій
MD5	висока	низька
SHA-1	середня	середня
SHA-256	низька	висока
CRC32	висока	низька

Для обчислення контрольних сум використовується криптографічна хеш-функція, яка забезпечує унікальність та стійкість до колізій. Ймовірність того, що дві різні вхідні дані $x \neq y$ мають однакову хеш-суму $h(x) = h(y)$ при використанні хеш-функції з довжиною виходу n біт. Для розрахунку $P_{collision}$ колізії потрібно використати парадокс дня народження (2.16):

$$P_{collision} \approx 1 - e - \frac{k(k-1)}{2\times 2^n}\,, \tag{2.16}$$

де k – кількість різних хешів (або вхідних даних); n – довжина хешу в бітах (для SHA-256, n=256).

Для великих значень k та n, якщо $k \ll 2^n$, ймовірність колізії можна наближено виразити (2.17) як:

$$P_{collision} \approx \frac{k(k-1)}{2 \times 2^n}. \quad (2.17)$$

Для обчислення необхідності кількості хешів потрібно використати константи множника (2.18) C для $P_{collision}$ ймовірності колізії у відсотках, відповідно до таблиці 2.2:

$$k = C \times 2^{n/2} \quad (2.18)$$

Таблиця 2.2

Ймовірностей колізії

Ймовірність колізії $P_{collision}$	Множник C
10%	0.4590
20%	0.6525
30%	0.8456
40%	1.0233
50%	1.1774
70%	1.5517
80%	1.79412
90%	2.14597
95%	2.44775
99%	3.03485

Враховуючи всі вище перелічені дані, середній час доступу з урахуванням архівування можна представити як (2.19) [26]:

$$T_{avg} = P_{cache} \times T_{cache} + P_{archive} \times T_{archive}\ , \quad (2.19)$$

де P_{cache} – ймовірність того, що дані знаходяться в кеші; $P_{archive}$ – ймовірність того, що дані в архіві $1 - P_{cache}$; T_{cache} – середній час доступу до даних у кеші; $T_{archive}$ – середній час доступу до даних в архіві.

Для поєднання рішення архівування даних з алгоритмами кешування LRU та MRU потрібно формулювати залежності порогових значень від алгоритму (2.20), їх можна ввести в залежності від порогових значень (2.21) та обраного алгоритму кешування [23;24] :

$$f_{threshold}^{alg} = \begin{cases} f_{threshold}^{LRU} & LRU \\ f_{threshold}^{LRU} & MRU \end{cases} \tag{2.20}$$

$$t_{max}^{alg} = \begin{cases} t_{max}^{LRU} & LRU \\ t_{max}^{MRU} & MRU \end{cases} \tag{2.21}$$

де $f_{threshold}^{alg}$ та t_{max}^{alg} – порогові значення, залежні від обраного алгоритму кешування (LRU або MRU); $f_{threshold}^{LRU}$ та $f_{threshold}^{MRU}$ – порогова частота доступу для алгоритмів LRU та відповідно MRU; t_{max}^{LRU} та t_{max}^{MRU} – Максимальний час перебування в кеші алгоритмів LRU та MRU.

Тепер потрібно оновити критерій архівування з урахуванням алгоритму (2.22).

$$A(x)\begin{cases} 1, \left(f(x) < f_{threshold}^{alg}\right) \vee \left(t_{cache}(x) > t_{max}^{alg}\right) \wedge p(x) = 0 \\ 0, \text{інакше} \end{cases} \tag{2.22}$$

де $A(x)$ – індикаторна функція, яка визначає, чи повинні дані x бути за архівовані.

Отже, після початкового аналізу, на рисунку 2.3 представлена блок-схема процесу прийняття рішення архівування даних, алгоритм визначення, чи слід архівувати конкретні дані, з урахуванням параметрів доступу.

Рішення про архівування даних приймається на основі таких параметрів, як частота доступу до даних, час їх перебування в кеші та пріоритет даних (2.23).

$$D(x) = \phi(f(x),\ t_{cache}, p(x), alg)\ , \qquad (2.23)$$

де ϕ – логічна функція, яка повертає 1, якщо дані слід архівувати, якщо ні, то 0; alg – алгоритм кешування.

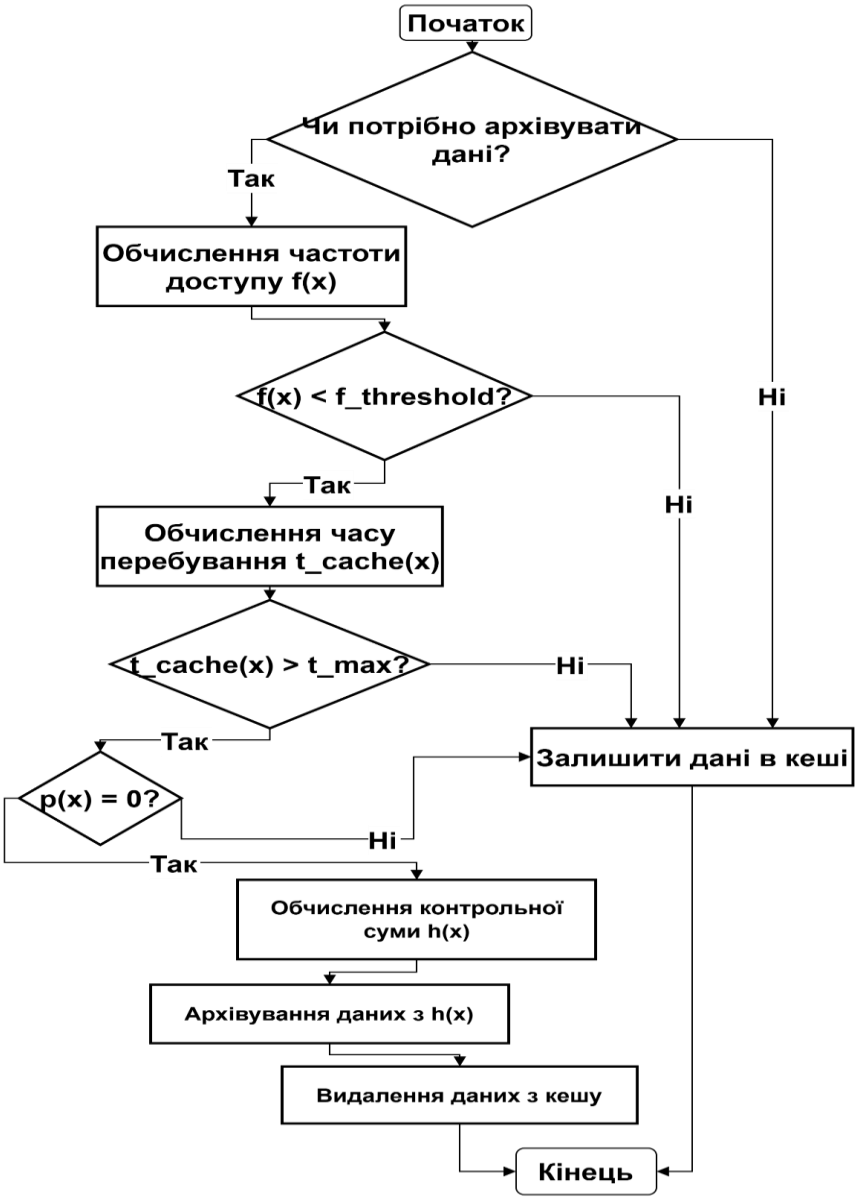


Рис. 2.3. Блок-схема прийняття рішення архівування даних

Один із ключових блоків у блок-схемі, процес прийняття рішення архівування даних – блок "Чи потрібно архівувати дані?" проілюстровано на рисунку 2.4.

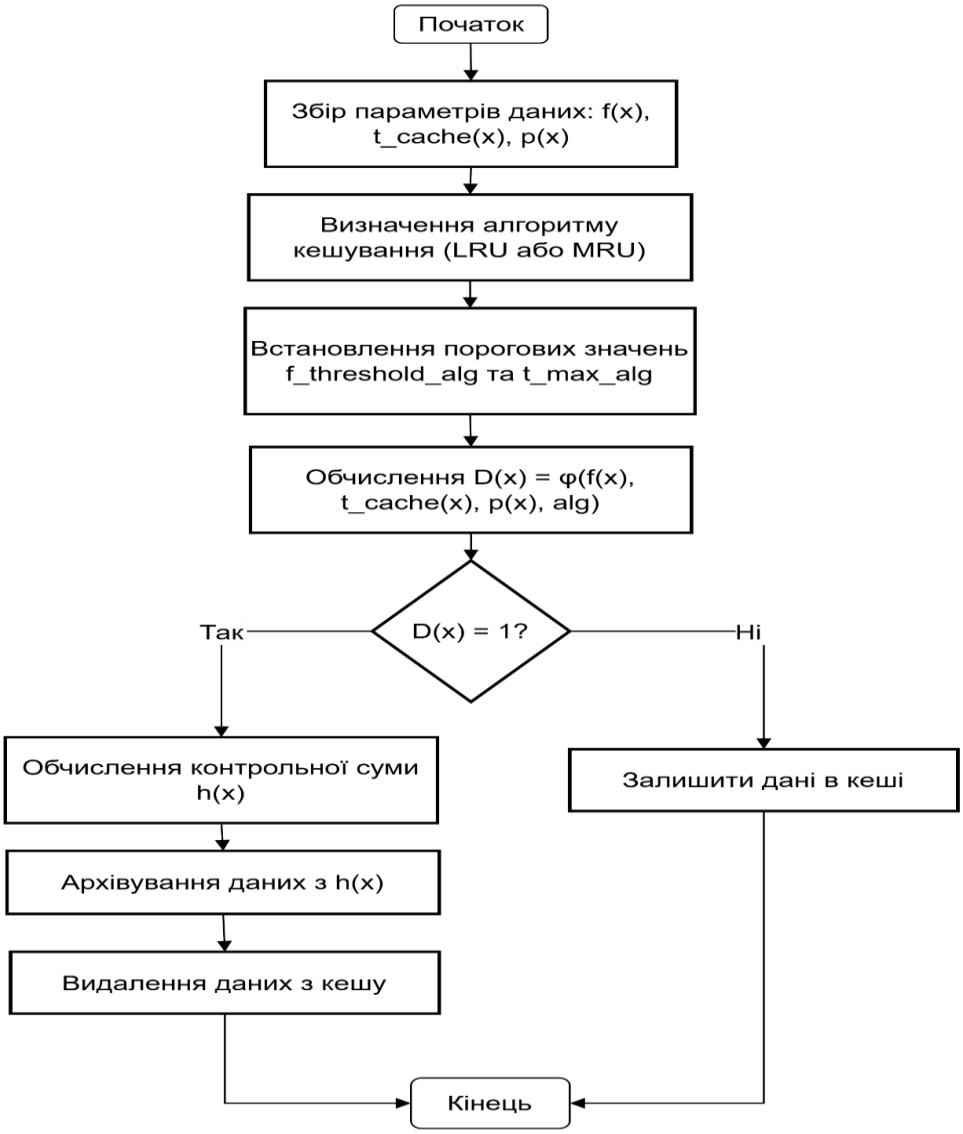


Рис. 2.4. Блок схема взаємодії компонентів

2.3 Вставка, пошук, фільтрація та сортування даних

При роботі з великими обсягами даних та високою частотою запитів важливо забезпечити ефективність операцій кешування. Аналіз великих систем кешування, таких як кеш-кластери в масштабах Twitter, показує важливість оптимізації цих операцій [27].

Розглянемо інтерпретацію операцій ставки, пошуку, фільтрації та сортування в контексті запропонованого адаптивного блоку, гібридного методу кешування.

Вставка даних у кешування, процес який дозволяє додавання нових даних до кешу LRU та MRU, враховується поточний стан адаптації та можливість архівування. Нові дані надходять в кеш в результаті зовнішнього запиту або внутрішньої операції. Метод визначає поточний алгоритм кешування LRU або MRU на основі аналізу патернів доступу.

Механізм розміщення даних у кеші реалізується наступним чином:

1. LRU: додаються до кінця списку кешу.
2. MRU: розміщуються на початку списку кешу, бо ймовірно будуть використані знову.

Якщо кеш заповнений, застосовуються алгоритм LRU або MRU для оптимізації простору, і дані низького пріоритету архівуються . Формально, цей процес можна представити наступною формулою (2.24)

$$\text{Cache} = \text{Cache} \cup \{x\} , \tag{2.24}$$

де Cache – структуру даних або частина пам’яті; x – елемент даних, який зберігається в кеші.

Час вставки $t_{insertion}$ записується, як поточний момент часу, коли дані додаються до кешу (2.25). Це дозволяє відстежувати, коли саме об'єкт був завантажений.

Пріоритет *p(x)* визначається пріоритетом даних (за замовчуванням $p(x)=0$), що означає стандартний рівень важливості. Однак, при необхідності, пріоритет може бути змінений для окремих об'єктів, що дозволяє більш гнучко керувати розміщенням даних у кеші відповідно до їхнього значення та частоти використання.

$$t_{insertion}(x) = t_{current} , \tag{2.25}$$

де $t_{insertion}(x)$ – час вставки даних x в кеш; x – поточний час, коли операція вставки виконується.

Коли кеш досягає свого максимального розміру, виконується видалення даних згідно з обраним алгоритмом кешування та критеріями архівування. На рисунку 2.5 продемонстровано основні етапи вставки нових даних до кешу.

Рис. 2.5. Вставка даних

Пошук даних відбувається з кешу або архіву. Користувач або внутрішній компонент системи надсилає запит на отримання даних, потім ці перевіряють їх наявність, якщо дані знайдено, вони повертаються користувачу, і при необхідності, оновлюються час останнього доступу. Коли дані було не знайдено в кеші, система звертається до архіву, дані завантажуються з архіву, перевіряється їх цілісність за допомогою контрольної суми h(x) і додаються до кешу згідно з процедурою вставки. На рисинку 2.6 відображено перевірку наявності запитаних даних у кеші. У випадку відсутності знайдених даних відбувається звернення до архіву з подальшим завантаженням до кешу та оновленням атрибутів доступу.

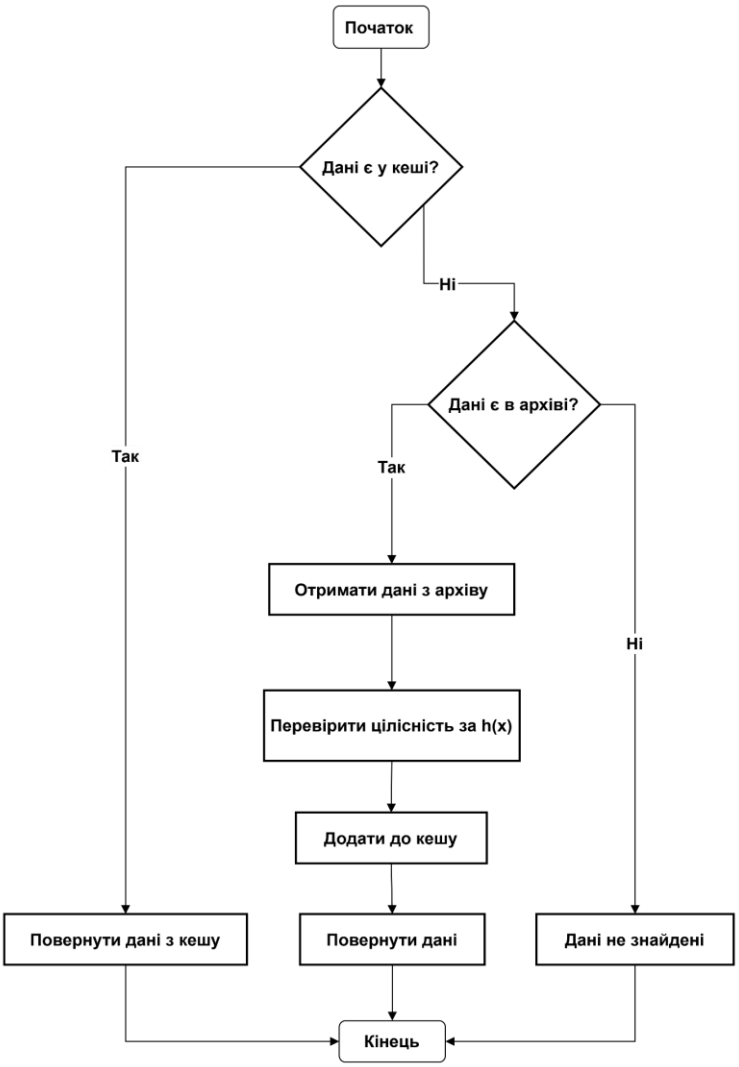


Рис. 2.6. Пошук даних

Блок схема, що на рисунку 2.7 демонструє процес відбору даних у кеші на основі заданих критеріїв. У разі невідповідності в кеші, фільтрація переноситься на архів із збереженням результатів для майбутнього використання.

Фільтрація, вибір підмножини даних, що відповідають заданим критеріям. В кешування фільтрація необхідна для обробки запитів, що стосуються певних властивостей. У процесі фільтрації визначається критеріями фільтрації, який повинен відповідати дані.

Фільтрування в кеші проходить по даних у кеші та відбирає ті, що відповідають критеріям. Для цього використовується структури даних хеш-таблиць.

Застосування фільтрації в архіві. Якщо необхідні дані не знайдено в кеші, фільтрація виконується в архіві. Для цього використовують індекси та ключі за яким зберігаються дані.

Рис. 2.7. Фільтрація даних

Процес фільтрації, запропонований має декілька етапів:

- перенесення часто запитуваних результатів до кешу для прискорення подальших запитів;
- аналіз історії запитів та оптимізація структур даних для підвищення ефективності фільтрації.

Дані в кеші відсортовані при вставці за ключем. При додаванні нових елементів в кеш зберігається у відсортований порядок. Реалізовано за допомогою структур даних, що підтримують впорядкованість збалансовані дерева.

Щоб процес сортування оптимізувати, застосовується такі методи оптимізації:

● інкрементне сортування: сортування в режимі реального часу при зміні даних.

● паралельне сортування: багатопоточності для прискорення процесу сортування великих обсягів даних.

Для сортування використовуються алгоритми, адаптовані до обраних структур даних, пірамідальне сортування. На рисунку 2.8 зображено сортування даних при оновленні або отриманні нового запиту. Інкрементальне та паралельне сортування використовуються для підвищення продуктивності та відповідності кешу в структурі даних.

Рис. 2.8. Сортування даних

2.4. Оптимізація використання пам'яті

У багатокористувацьких системах із високими вимогами до продуктивності, необхідно ефективно керувати пам’яттю, як наведено в аналізу, мультимедійні

файли, які часто завантажують та передають в соціальні мережі Twitter (зараз X), можуть вимагати використання специфічних стратегій обробки [27].

Для пріоритетного розміщення даних можна застосувати алгоритм експоненціального згладжування для обчислення середньої частоти доступу, що дозволить враховувати як поточні, так і історичні дані. Частота доступу може бути використана для динамічного виділення пам'яті [28].

Потрібно на етапі отримання даних звертати на їх специфіку (критичність або терміновість), що може впливати на стратегії кешування. Для цього доцільно використовувати метадані, які визначають важливість даних та для управління обсягом кешу можна використовувати динамічне виділення пам'яті, де обсяг кешу адаптується в реальному часі залежно від навантаження. У реальних системах це реалізуються через використання методів, подібних до тих, що застосовуються в динамічному управлінні ресурсами хмарних середовищ [33]. У періоди високого навантаження кеш може тимчасово зменшуватися за рахунок архіву, а в періоди низького навантаження – збільшуватися.

Збільшення обсягу кешу зазвичай призводить до покращення продуктивності, оскільки зменшується кількість звернень до архіву [23]. Хоч це вимагає більших апаратних ресурсів, але зменшення об'єму кешу має наступні наслідки:

- збільшення часу доступу до даних через частіше звертання до архіву;
- підвищення ризику втрати даних, які могли б бути запитані повторно, через більш часте видалення об'єктів [24].

Можна дотримуватись компромісу між обсягом кешованих даних та архіву, використовуючи для цього “гарячий кеш”, який зберігатиме часто запитувані дані, а інші для менш критичних в архіві [25]. Для оптимального розподілу ресурсів можна запропонувати такі підходи:

1. Адаптивне управління пам'яттю: автоматично перерозподіляти пам'яті між кешем та архівом.

2. Пріоритетне кешування: дані з високою частотою доступу або критичні для бізнесу отримують більше пам'яті.

3. Контроль обмежень: наприклад, максимальний обсяг пам'яті, який може використовувати кеш, встановлюється політикою системи або самого методу.

Модель Лагранжа (2.26) дозволяє мінімізувати цільову функцію, яка визначає ефективність використання пам'яті, враховуючи обмеження ресурсів:

$$L = \sum_{i=1}^{N} w_i \times f(x_i) + \lambda\left(M - \sum_{i=1}^{N} x_i\right), \quad (2.26)$$

де w_i – вага (частота доступу) кожного елемента i; $f(x_i)$ – функція оцінки важливості даних; x_i – загальний доступний обсяг пам'яті; M – загальний обсяг доступної пам'яті; λ – множник Лагранжа, що враховує обмеження на пам'ять.

Враховуючи фактор, що адаптація алгоритму здійснюється шляхом аналізу дисперсії та мінімізації витрат, модель оптимізації можна представити наступним чином (2.27):

$$L = \sum_{i=1}^{N}\left(C_{LRU} \times P_{LRU}(x_i) + C_{MRU} \times P_{MRU}(x_i)\right) + \lambda\left(M - \sum_{i=1}^{N} x_i\right), \quad (2.27)$$

де C_{LRU},C_{MRU} – частота використання LRU або MRU відповідно (час доступу, кількість промахів); $P_{LRU}(x_i)$, $P_{MRU}(x_i)$ – ймовірність вибору LRU або MRU залежно від дисперсії частоти доступу.

$$C_{LRU}(x_i) = w_1 \times t_{cache}(x_i) + w_2 \times f(x_i), \quad (2.28)$$

$$C_{MRU}(x_i) = w_3 \times \left(1 - D(x_i)\right) + w_4 \times f(x_i), \quad (2.29)$$

де $D(x_i)$ – дисперсія частоти доступу.

Для обґрунтування вибору потрібно врахувати фактор дисперсії частоти доступу (2.30), це поєднання дозволить оптимізувати об'єм кешу для кожного алгоритму, та знизити кількість кеш-промахів:

$$L = \sum_{i=1}^{N}\left(C_{LRU} \times P_{LRU}(x_i) \times D_{LRU}(x_i) + C_{MRU} \times D_{MRU}(x_i) \times P_{MRU}(x_i)\right) + \lambda(M - \sum_{i=1}^{N} x_i), \quad (2.30)$$

де $D_{LRU}(x_i)$, $D_{MRU}(x_i)$ – дисперсія частоти доступу для алгоритмів LRU та MRU для об'єкта x_i.

2.5 Безпека та надійність кешованих даних

Окрім самого фактора, кешування потрібно звернути увагу, на безпеку даних, які будуть зберігатися на сервері чи на стороні клієнта. Використовування гібридної моделі підвищить складність системи, що потребує додаткового аналізу теоретичних загроз та захист даних від несанкціонованого доступу[17;27;28].

2.5.1 Безпеки даних у контексті кешування та архівування

Потрібно проаналізувати кешувані в констексті безпеку, та які є ризики. Для виявлення потенційних потрібно промоделювати їх, використовуючи модель STRIDE [30]:

1. Tampering (Підробка) – зміни або пошкодження даних.
2. Information Disclosure (Розкриття інформації) – небезпека витоку.
3. Denial of Service (Відмова в обслуговуванні) – перевантаження системи кешування.

2.5.2 Методи захисту даних від несанкціонованого доступу

Одним з методів захисту можна використати аутентифікацію та авторизацію. Аутентифікація застосовується для впевненості, що доступ отримав саме той користувач, а авторизація визначає, які саме дії та ресурси доступні кожному користувачу після успішної аутентифікації. Модель контролю доступу (2.31).

$$Access(User, Resource) = \begin{cases} alow\ , \ Per(User, Resource) = True \\ deny, \text{ інакше} \end{cases}, \qquad (2.31)$$

де $User$ – ідентифікатор користувача, $Resource$– ресурс, Per – функція, що визначає права доступу; $alow$ – дозволити; $deny$ – заперечити.

Ще одним методом захисту інформації від несанкціонованого доступу є симетричні алгоритми шифрування, AES-256 – різновид симетричного

шифрування, що перетворює дані у зашифрований вигляд за допомогою ключа доступу [25;30;31].

Щоб забезпечити цілісність даних після шифрування, потрібно оцінювати контрольну суму кешованих даних. Контрольна сума даних обчислюється за формулою (2.10), яку ми раніше розглядали у контексті контрольної суми для збереження даних. На рисунку 2.9 представлено сам алгоритм дій для перевірки цілісності даних.

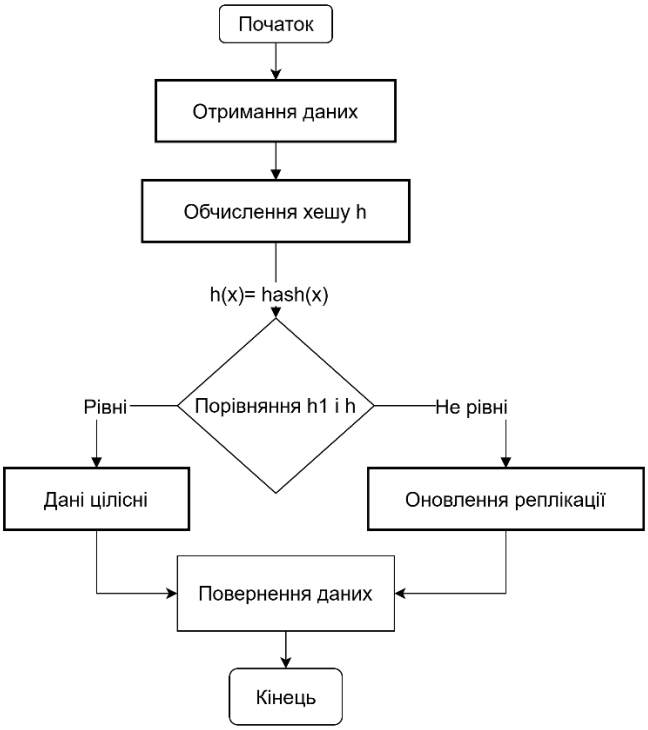


Рис. 2.9. Алгоритм перевірки цілісності даних

2.5.3 Вплив архівування на цілісність та доступність даних

Архівування даних впроваджується як окремий шар зберігання, що взаємодіє з кешем. На рисунку 2.10 представлено схему взаємодії компонентів.

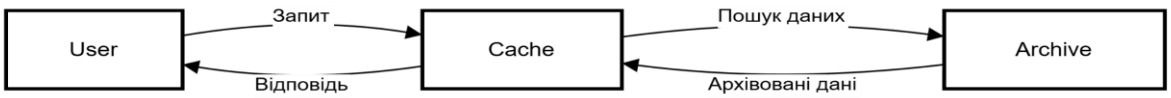


Рис. 2.10. Схема взаємодії компонентів

При архівуванні даних необхідно забезпечити їх цілісність. Для цього використовується механізм контрольних сум та цифрових підписів.

Алгоритм архівування з перевіркою цілісності:

1. Обчислення хеш-значення.
2. Збереження даних та хешу.
3. Відновлення даних.
4. Перевірка цілісності.
5. Вплив на доступність даних.

Архівування може збільшувати час доступу до даних. Середній час доступу з урахуванням архіву [26]. Попередньо вже було представлено формулу (2.15), T_{avg}.

Для забезпечення високої доступності даних пропонуються наступні кроки:

- регулярний аналіз та перенесення часто запитуваних даних з архіву до кешу.
- архівування даних у періоди низького навантаження на систему.
- застосування SSD або NVMe-дисків.

Ймовірність доступу до даних можна змоделювати за допомогою закону Парето або розподілу Зіпфа (2.32), що відображають нерівномірність запитів [32]:

$$P(i) = \frac{1}{i^a} / \sum_{k=1}^{N} \frac{1}{k^a}, \quad (2.32)$$

де $P(i)$ – ймовірність запиту до i елемента; a –параметр розподілу; N - загальна кількість елементів.

Для забезпечення узгодженості між кешем та архівом використовується протокол узгодження версій[33]:

1. Кожен елемент даних має мітку часу останньої модифікації.
2. При доступі до даних порівнюються мітки часу кешу та архіву.
3. Якщо версія в архіві новіша, кеш оновлюється.

Для розробки безпечної та надійної системи кешування з інтегрованим архівуванням необхідний комплексний і цілісний підхід. Реалізована

автентифікація, шифрування, перевірка цілісності та механізм постійного моніторингу гарантують повний захист даних від доступу та підробки. Математичні моделі для аналізу ризиків і оптимізації доступності даних підвищують загальну ефективність системи. Регулярні аудити безпеки та оновлення політики безпеки щодо переважаючих загроз будуть рекомендовані як захід принаймні для постійного оновлення рівня захисту.

Розроблення архітектура гібридного кешування, можна розглянути на рисунку 3.1, яка включає основні модулі та їх взаємодію.

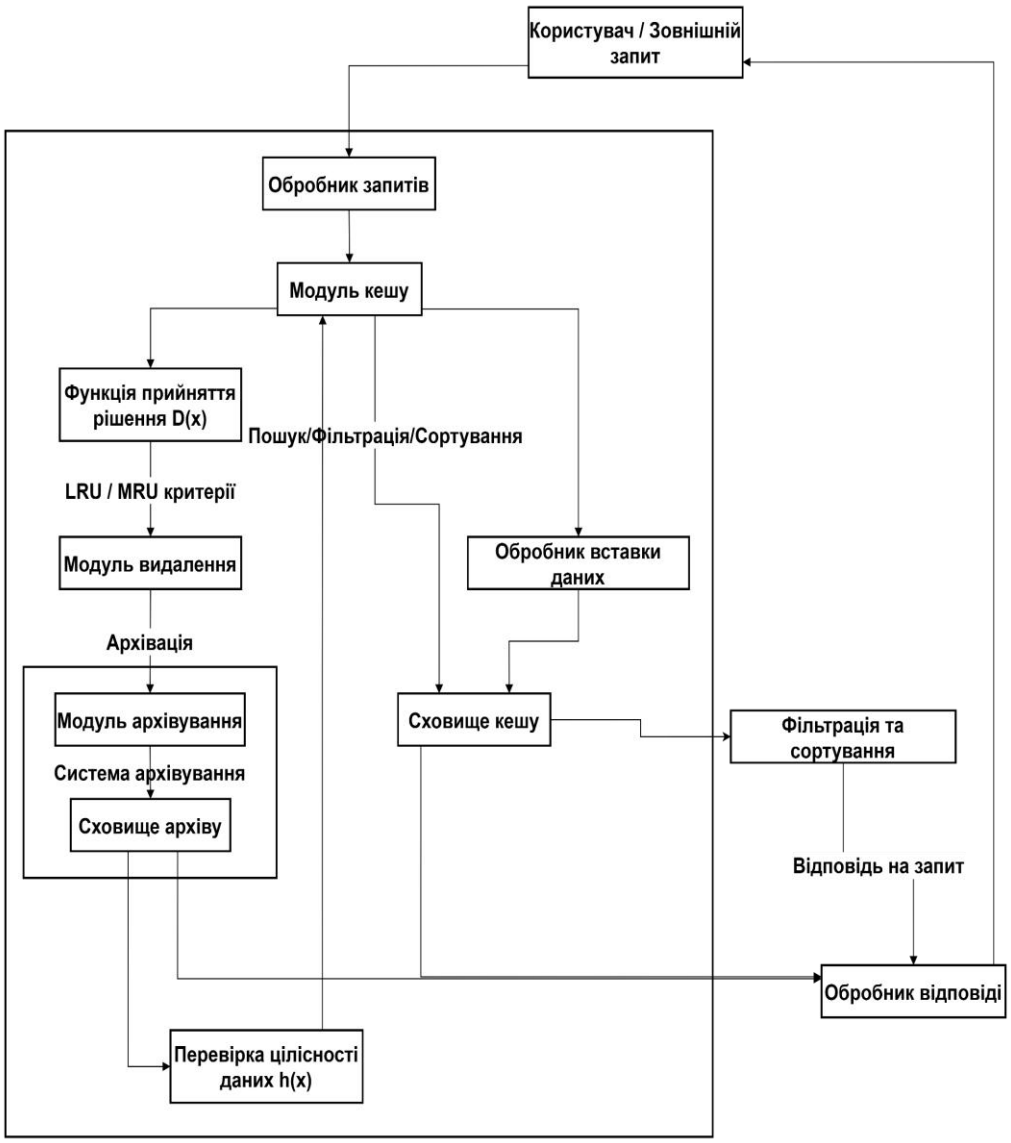


Рис. 2.11. Архітектура гібридного методу кешування

2.6 Висновок до розділу

У роботі розглянуто методи поєднання LRU і MRU в залежності від таких метаданих, як частота доступу, дисперсія шаблону, пріоритети, час перебування кешу. Розроблена адаптивна модель динамічного вибору алгоритму яка адаптується до змін шаблонів доступу, що у свою чергу, забезпечує краще управління кеш-пам'яттю та підвищення продуктивності системи.

Було запропоновані моделі для оцінки частоти доступу через дисперсію та для адаптація зміни алгоритму через функцію Лагранжа, що дозволяє автоматично приймати рішення про те, коли видаляти або архівувати дані, тим самим знижуючи ризик перевантаження кешу та експоненціальне згладжування, яке дозволяє системі надавати більш значущу вагу останнім запитам до даних, одночасно зберігаючи вплив на попередні.

Запропоновані критерії архівування даних повинні ґрунтуватися на частоті доступу, часу перебування та контрольних сумах таким чином, щоб підтримувалася цілісність даних, але їх доступність підвищувалася в довгостроковій перспективі.

Для реалізації запропонованого методу буде використано мову програмування C++ разом із бібліотекою OpenSSL, яка надає модуль SHA для створення криптографічних хеш-функцій. Ці хеш-функції необхідні для забезпечення цілісності та безпеки даних у кеші, оскільки вони дозволяють перевіряти, чи не були дані змінені або пошкоджені. Крім того, буде застосовано бібліотеку zlib для стиснення даних, що дозволить архівувати інформацію в меншому об'ємі. Поєднання C++ з OpenSSL та zlib забезпечить високу продуктивність та надійність розробленого методу, оскільки C++ дозволяє оптимізувати код для системного рівня з використанням асемблерних вставок для інструкцій процесора AVX-512, а OpenSSL та zlib надають необхідні інструменти для безпечного та ефективного управління даними.

3 РОЗРОБКА ГІБРИДНОГО МЕТОДУ ТА ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ ТА ТЕСТУВАННЯ ГІБРИДНОГО МЕТОДУ

3.1 Постановка експерименту

Розглянемо вплив апаратного забезпечення, та як паралелізм впливає на загальну продуктивність програми та визначення оптимальних апаратних ресурсів для її виконання. Особливу увагу приділимо оцінці теоретичного та реального прискорення програми при використанні багато поточності.

Метою розділу є визначення оптимального апаратного забезпечення в контексті, процесора та його паралелізму. Це дасть розуміння, яку конфігурацію вибрати для подальшого тестування та аналізу

Типи кешу має великий вплив на продуктивність, та затримки відповіді наприклад In-Memory Cache, кеш який зберігається в оперативній пам'яті, забезпечує найнижчу затримку через використання оперативної пам’яті для зберігання даних, але має обмеження об'єму та надійності збереженості даних як це реалізовно в такий програмних забезпеченнях Redis, Memcached [2;3]. Дисковий кеш, або кеш який зберігає постійно запам'ятовуючий пристрій, хоч затримки відповідно зворотні від ОЗП, але все залишається нижчими ніж при звернення до основного джерела даних.

3.1.1 Апаратне середовище та його вплив

Для досягнення поставленої в кваліфікаційній роботі мети, важливо зосередитися на оцінці алгоритмічної ефективності існуючих методів кешування та аналізі впливу апаратного середовища на реальну продуктивність системи. Нижче потрібно розглянути ключові аспекти, які слід враховувати під час аналізу.

На вихідні дані аналізу можуть вплинути апаратні характеристики системи. Один с цих факторів – це процесор та кількість ядер, що дає робити паралелізм.

Алгоритм кешування може бути налаштований для обробки декількох запитів одночасно. Ефективність багатопоточності залежить від структури задачі, використання ресурсів і накладних витрат.

Сучасні процесори можуть мати від одного до десятків ядер. Кожне ядро здатне виконувати потік інструкцій незалежно від інших:

- одноядерні процесори підходять переважно для послідовного виконання. Висока тактова частота та ефективна архітектура ядра сприяють швидкому виконанню таких програм;
- багатоядерні процесори, дозволяють виконувати кілька потоків одночасно, що потенційно може прискорити програми з високою часткою паралельного коду.

Проміжним етапом обчислення процесора є кеш-пам'ять Вона призначена для зберігання часто використовуваних даних та інструкцій, зменшуючи затримки при доступі до них. Їх розбивають на три рівня :

- L1-кеш: найшвидший та найменший за обсягом, розташований найближче до ядра процесора.
- L2-кеш: має більший обсяг, але повільніший за L1.
- L3-кеш: спільний для всіх ядер процесора, має найбільший обсяг серед кешів, але найповільніший з них.

3.1.2 Закон Амдала та теоретичні обмеження

Для визначення ефективності розпаралелювання процесів застосовується закон Амдала, запропонований Джином Амдалом у 1967 році. Цей закон використовується для оцінки теоретичного максимального прискорення виконання завдання в системі при оптимізації частини її ресурсів [34]. Як зазначено в законі, навіть якщо значна частина завдання може бути виконана паралельно, частина, яку не можна розпаралелити, встановлює межу на теоретичне прискорення (3.1) [34]:

$$S(N) = \frac{1}{(1-P) + \frac{P}{N}}, \tag{3.1}$$

де $S(N)$ – прискорення при використанні N потоків або процесорів; P – частка програми, яка може бути виконана паралельно (від 0 до 1); $1 - P$ – частка програми, яка виконується послідовно; N — кількість потоків або процесорів.

Теоретичні обмеження (3.2) може досягатися при максимальному прискорені $N \to \infty$ [34]:

$$S_{max} = \frac{1}{(1-P)}, \tag{3.2}$$

3.1.3 Проведення експерименту та результати

Було проведено практичне застосування багатопоточності, на рисунку 3.1 наведено інформацію, щодо одного потоку N=1. За результатами видно, що при використанні одного потоку загальний час виконання програми становить 61,776 мікросекунд, з яких 2,062 мікросекунди витрачено на паралельні операції, а 59,713 мікросекунд – на послідовні операції. Частка паралельного коду зіставляє $P = \frac{T_p}{T_N} = \frac{2.062}{61.776} \approx 0.03338$.

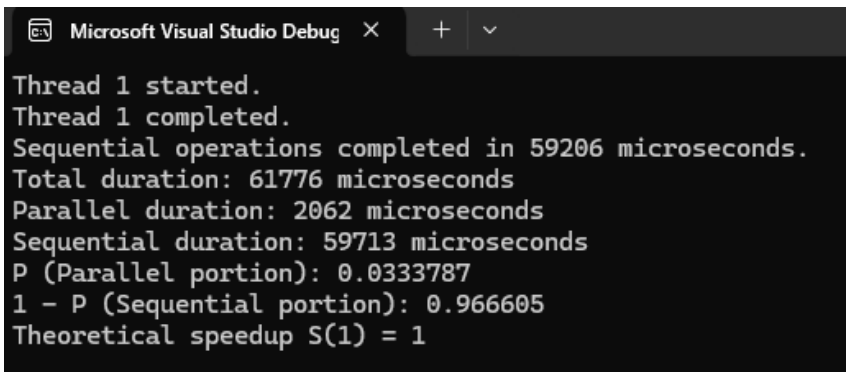


Рис. 3.1. Виконання програми з одним потоком

У таблиці 1 представлені значення теоретичного прискорення $S(N)$, розраховані за законом Амдала для різної кількості потоків N. Частка паралельного коду P обчислена на основі експериментальних даних та відображає пропорцію програми, яка може бути розпаралеленна.

Таблиця 3.1.

Теоретичне прискорення залежно від кількості потоків

Кількість потоків N	Теоретичне прискорення $S(N)$	Частка програми P
1	1,00	0,03338
2	1,0186	0,03655
4	1,0383	0,03984
8	1,0497	0,04147

У таблиці 2 містить результати вимірювань загального часу виконання програми T_N, паралельного часу T_p та послідовного часу T_S для різної кількості потоків N. Фактичне прискорення $S(N)$ обчислено як відношення $S(N) = \frac{T_i}{T_N}$, де T_i – загальний час виконання при $N = i$.

Таблиця 3.2.

Фактичне прискорення та часові показники програми

Кількість потоків N	Фактичне прискорення $S(N)$	Частка програми P	Загальний час виконання (T_N, мкс)	Паралельний час (T_p, мкс)	Послідовний час (T_S, мкс)
1	1,00	0,03338	61,776	2,062	59,713
2	1,0008	0,03655	61,727	2,256	59,470
4	1,0036	0,03984	61,551	2,452	59,099
8	1,0047	0,04147	61,487	2,550	58,937

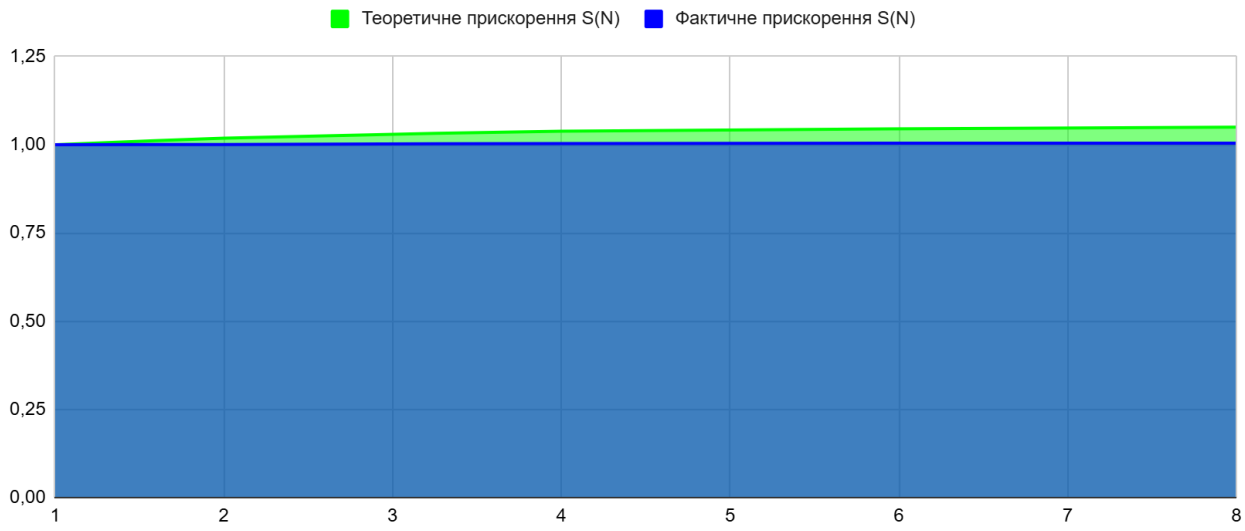


Рис. 3.2. Порівняння теоретичного та фактичного прискорення

3.1.4 Аналіз результатів та рекомендації

Згідно з законом Амдала, теоретичне прискорення зростає з 1.00 до 1.0497 при збільшенні кількості потоків з 1 до 8. Проведене практичне застосування багатопоточності продемонструвало не значне прискорення при збільшенні кількості потоків. Основна причина низького масштабування полягає в тому, що частка програми, яка може бути розпаралелена P , є дуже малою, яка становить приблизно від 3.3% до 4,1%, що фактично залишається на рівні 1.00 до 1.0047, що свідчить про відсутність суттєвого покращення продуктивності при додаванні потоків. Закон Амдала демонструє обмеження на максимальне прискорення виконання завдання, пов'язані з послідовними частинами його обробки. Все тому, що послідовна частина даної розробки займає понад 95% загального часу виконання.

Низька частка програми, яка виконується послідовно, обмежує максимальне теоретичне прискорення, незалежно від кількості потоків.

Аналіз показав, що здебільшого вистачить одноядерного процесора, оскільки збільшення кількості ядер та потоків не призводить до суттєвого покращення продуктивності. А гонка потоків може навпаки зменшити продуктивність. Вибір

процесорів з високою тактовою частотою та великим кешем рівня L3 дозволить забезпечити максимальну продуктивність для послідовних операцій.

Один з шляхів для покращення продуктивності полягає в оптимізації послідовного коду та збільшенні частки паралельного коду. Також важливо враховувати апаратні особливості при використанні та оптимізації програмного забезпечення.

3.2 Метрики оцінювання

Для об'єктивної оцінки ефективності розробленого гібридного методу кешування та порівняння її з існуючими підходами, необхідно визначити набір метрик, які допоможуть показати продуктивність використання. Описання основних метрик, дасть можливість тестувати даний метод не лише в даний момент, а через деякий час, щоб потім оцінювати результати та співставляти з іншими методами, механізмами тощо. В цьому розділі детально розглянемо основні метрики, які будуть використовуватися під час тестування та аналізу, як застосовувати математичні моделі з розділу 1 для їхнього обчислення та інтерпретації результатів.

3.2.1 Пропускна здатність

Пропускна здатність – це продуктивність яка визначає кількість операцій, виконаних системою за одиницю часу. Компанії, такі як IBM, активно використовують ці стандарти для оцінки та сертифікації своїх розробок [33]. У контексті системи кешування це може бути кількість успішно оброблених запитів на читання або запис за секунду.

Пропускна здатність вимірює кількість операцій або транзакцій, які система може обробити за одиницю часу. Зазвичай вона вимірюється в [35]:

- TPS (Transactions Per Second) – транзакцій за секунду;
- TPM (Transactions Per Minute) – транзакцій за хвилину.

Фактори, що впливають на пропускну здатність, які були визначено в попередньому пункті, стосовно процесора та його паралелізму, та ще обсяг оперативної пам'яті, тип постійно запам’ятовуючого пристрою – жорсткого диск або SSD [36]. Накладні витрати програмного забезпечення, що стосуються виконання алгоритмів, запитів, та сама оптимізації коду. Також на це може вплинути складність транзакцій (читання/запис, обчислення, пошук) та інші типи операцій. Методи вимірювання пропускної здатності які можна використати :

1. Вбудований механізм логування:
    a. Програма фіксує часові мітки (timestamps) для кожної транзакції.
    b. Аналіз журналів дозволяє оцінити швидкість виконання.
2. Використання тестових сценаріїв (benchmark):
    a. Створення типових трансакційних сценаріїв для порівняння.
    b. Автоматизація вимірювання за допомогою інструментів моніторингу.

Схема роботи пропускної здатності під час передачі даних зображена на рисунку 3.3 [41], де продемонстровано вимірювання кількості даних, що можуть бути передані та отримані між компонентами системи (сервером, мережею, модемом і кінцевим пристроєм) за одиницю часу.

Рис. 3.3. Схема роботи пропускної здатності під час передачі даних [41]

Для оцінки продуктивність системи та порівняти її з іншими методами кешування потрібно використати формулу (1.35) яку було застосовано для моделі аналізу. Щоб порахувати це значення припустимо, що за $T_{total} = 10$ секунд та $N = 5000$ запитів та всі вони були успішні, то розрахунок відбувається таким чином:

$$T = \frac{5000}{10} = 500 \text{ запитів}/c$$

Пропускна здатність T показник показує що система може обробляти в середньому 500 запитів за секунду. Ці значення будуть підраховуватися для кожного методу, алгоритму тощо. Щоб визначити, який з них забезпечує найвищу продуктивність

3.2.2 Затримка

Затримка - це час, який необхідний для вилучення інформації порівняно з отриманням із основної бази знань [37]. Оскільки сам механізм розроблений, щоб зменшити затримку, потрібно контролювати цей параметр, щоб не сумувати затримку двох систем , потрібно анулювати похибку нульової затримки та проаналізувати їх враховуючи, що система запущено локально [37;38].

Чому затримка важлива для кешування. Цей параметр дозволяє зменшити час доступу, ніж при отриманні даних із основного сховища, також це критично для високонавантажених систем, де кожна мілісекунда впливає на продуктивність. Оптимізація взаємодії користувача дає можливість забезпечити швидкий відгук системи та покращує досвід користувача.

Це все разом призводить до зменшення навантаження на основне джерело даних, що дозволяє уникати перевантажень і високої затримки через конкуренцію за ресурси.

На схемі, представленій на рисунку 3.4 [39], ілюструється затримка (latency) як час, необхідний для передачі пакета даних від сервера до кінцевого пристрою через інтернет та модем. У контексті кешування це важливо, оскільки зменшення затримки через використання кешу (локального чи розподіленого) дозволяє мінімізувати час передачі даних та покращити продуктивність системи.

Рис. 3.4. Схема затримки передачі даних [41]

Пропускною здатністю та затримки доступу пов'язані кількістю оброблених операцій за секунду, які потрібні на виконання кожної операції. Це випливає так що чим більше затримки тим менше операції може виконати система.

Низька затримка дозволяє обробляти більше запитів за одиницю часу, підвищуючи пропускну здатність. Висока затримка знижує ефективність системи, обмежуючи кількість запитів, які можуть бути оброблені. Пропускна здатність максимізується за умов мінімізації затримок. Оптимізація затримки (через локальний кеш, CDN, стиснення) безпосередньо покращує продуктивність.

Затримка є показником продуктивності для її оцінки потрібно вимірювати середній час операції та максимальну затримку.

Для оцінки середнього часу – W та середнього часу відповіді – R, виконання однієї операції, як приклад : вставка; отримання; видалення. Застосуємо формулу (1.34) та (1.38). Заповнено формули умовними одиницями $\lambda = 5000$ запитів/с – інтенсивність запитів, $\mu = 500$ тоді можна вирахувати дані:

$$W = \frac{5000}{500(5000-500)} = 0.002 \text{ секунд}$$

$$R = 0.002 + \frac{1}{5000} = 0.022 \text{ секунд}$$

Це означає, що середній час відповіді системи складає приблизно 0.22 секунд.

3.2.3 Коефіцієнт кеш-промаху та кеш-потраплянням

Ефективність алгоритму хешування найкраще оцінювати за коефіцієнтом кеш-промаху та коефіцієнтом кеш-потраплянням, які вказують на ступінь

кешування даних і, отже, на те, як часто виникає потреба звертатися до повільного основного сховища даних.

Взаємодія між частотою звернення до кешування та частотою промахів, які служать критичними показниками для оцінки ефективності систем кешування [39]. Швидкість звернень до кешу, це метрика, яка відображає частку запитів, які успішно витягуються з кешу без необхідності доступу до повільніших ієрархій пам'яті [40]. Висока частота звернень до кешу часто виражена у відсотках, чим нижче відсоток тим краще, оскільки це означає, що більшість запитів даних виконується кеш-пам'яттю [40]. І навпаки, високий відсоток промахів кешу вказує на неефективність [40].

Отже, існує запит на стратегічне керування кеш-пам'яттю та дизайном, який мінімізує промахи кеш-пам'яті для підвищення ефективності системи. Таким чином, для забезпечення високої продуктивності в сучасних обчислювальних середовищах наголошується на стратегічному управлінні кеш-пам'яттю та дизайні.

Частота промахів кешу обчислюється як частина запитів, для яких елементів не було знайдено в кеші, отже, потрібно було отримати доступ до повного основного сховища даних. Щоб отримати коефіцієнт кеш-промахів H' потрібно модифікувати формулу (1.36) таким чином, щоб n – кількість звернень до кешу та n_r – загальна кількість запитів віднімалась та ділилася на кількість звернень до кешу:

$$H' = \frac{5000-800}{5000} = 0{,}84 \times 100\% = 84\%$$

Кеш-потрапляння відбувається коли файл запитується, то, якщо він уже закешований, його можна повернути без звернення до оригінального сховища, яке називається "кеш-хіт". Наприклад, коли користувач просить переглянути веб-сторінки, що містять дані про продукти, які вже є в кеші, даний текст просто відображатиметься знову без звернень в основну базу, як приклад, ці дані можуть зберігатися на розподіленій системі CDN (Content delivery network)[41]. Це зменшує час очікування, а також навантаження на основну базу знань, тощо.

Відсутність кешу відбувається, коли запитані дані немає у кеші. Представлена схема CDN на рисунку 3.5, яка ілюструє принципи роботи мережі [41].

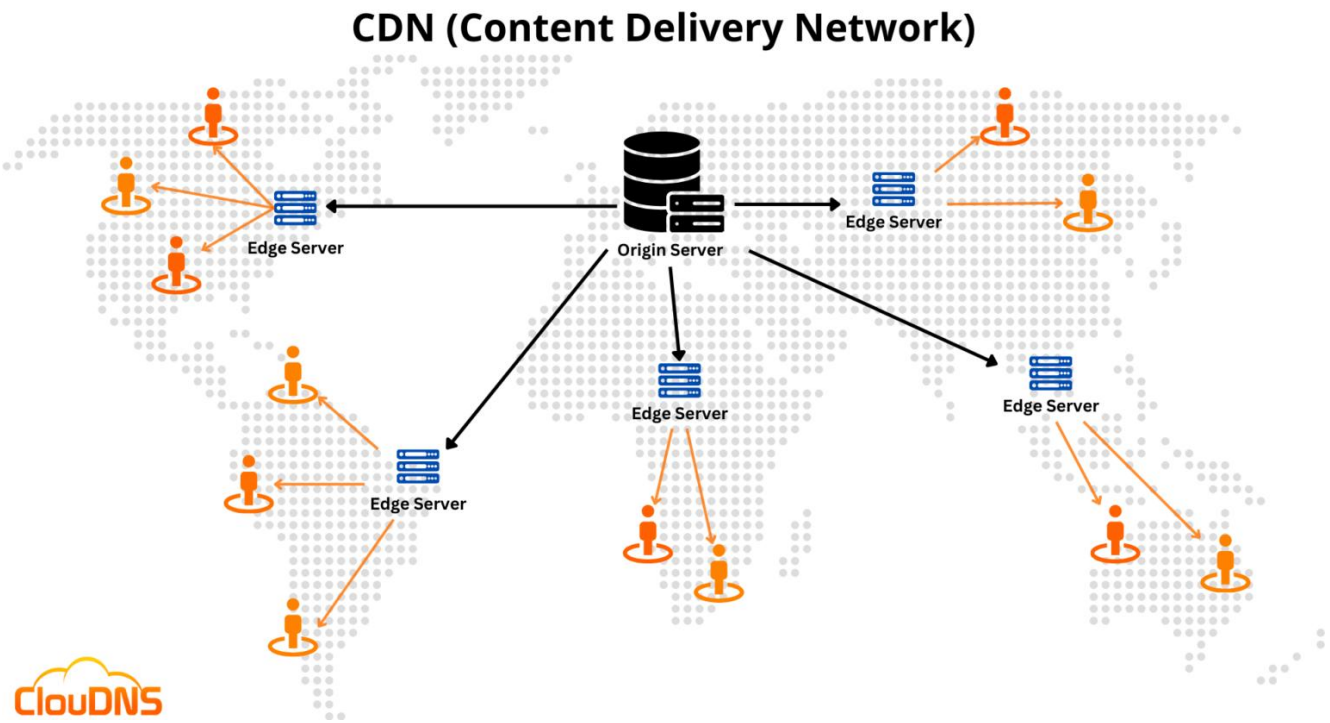


Рис. 3.5. Принципи роботи мережі CDN [41]

Отже, у цьому випадку він передасть запит на вихідний сервер із запитом потрібного вмісту. Наприклад, система зберегла в кеші зображення кота, який грає на піаніно [41]. Якщо якийсь інший кінцевий користувач попросить це конкретне зображення, воно буде подано з кешу. В іншому випадку система отримує його з вихідного сервера, а потім кешує копію.

Оптимальна частота звернень до кешу, як заявляють cloudflare, CDN: частота звернень до кешу в регіоні 95-99% буде ідеальною для сайтів, які мають переважно статичний вміст [41]. Проте дивіденди можуть бути трохи нижчими у випадку динамічного вмісту, хоча сучасні технології дозволяють кешувати досить велику частку такого контенту[41].

Коефіцієнт кеш-потрапляння визначає частку запитів, які були успішно обслуговувані безпосередньо з кешу (1.36). Припустимо, що під час експерименту було зафіксовано 800 кеш-хітів з 5000 запитів :

$$H = \frac{800}{5000} = 0{,}16 \times 100\% = 16\%$$

Можна бачити, що цей показник відображає, що він не є ефективний та збільшує кількість звернень до основного сховища даних. А якщо звернутися до частоти промахів то цей показник вказує 84%. Тому, це ще раз показує, що ці метрики є взаємопов'язані, та знайти відсоток можна лише знаючі один с коефіцієнтів $H + H' = 100\%$.

Таким чином, під час перевірки та оцінки алгоритмів, методів тощо ці показники повинні бути враховані в результати тестування.

3.2.4 Узгодженість даних

Узгодженість даних – це ступінь, яка описує гарантію актуальність та цілісність даних при конкурентному доступі або після відмов [42]. Розглянути схему даного механізму можна на рисунку 3.6. проілюстровано реплікацію даних які з’єднані між двома центрами обробки даних та їх невідповідність. Забезпечення узгодженості даних у механізмах кешування має ключове значення для сучасних програм, оскільки це безпосередньо впливає на цілісність даних і надійність системи [42]. Основною проблемою є виявлення неузгоджених даних і керування ними під час процесів передачі даних.

Однак, якщо виявлено невідповідності, система забезпечує гнучкість обробки цих аномалій. Користувачі можуть або припинити копіювання, або продовжити процес, використовуючи налаштування відмовостійкості, що дозволяє пропускати неузгоджені файли. Це забезпечує стійкість системи до аномалій і підтримує цілісність даних навіть у випадках помилок або збоїв.

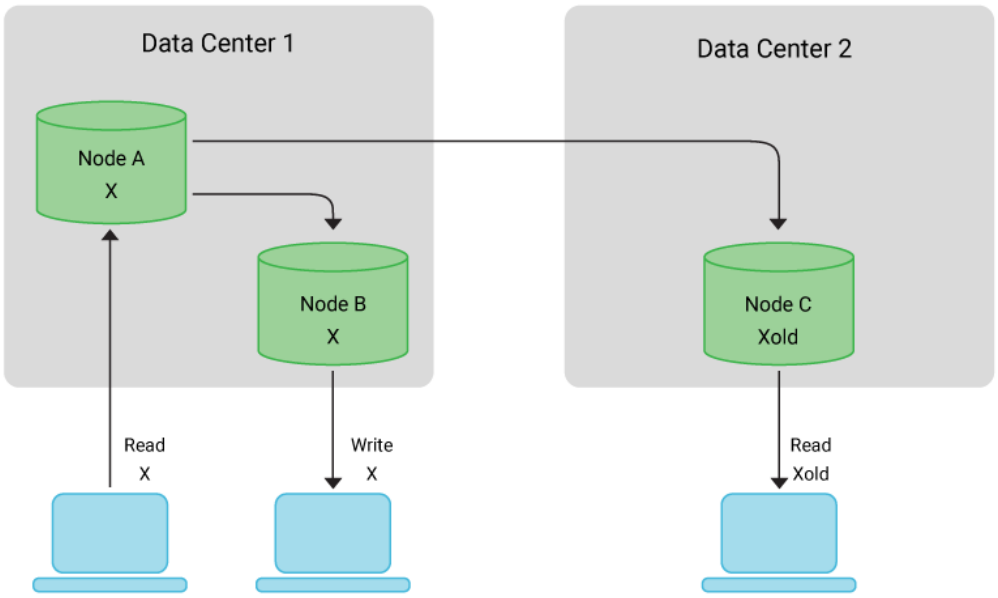


Рис. 3.6. Схема реплікації даних між двома центрами обробки даних

Узгодженість даних показує стан даних при одночасних операціях читання та запису. Гарантуючі, що всі користувачі бачать однакові та актуальні дані незалежно від порядку та часу їхнього доступу[40].

Для оцінки узгодженості даних використовуються наступні метрики:

1. Гонка даних (англ. Data Races).
2. Цілісність даних (англ. Data Integrity).
3. Атомарність операцій (англ. Operation Atomicity).
4. Сильна консистентність (англ. Strong Consistency).

Гонка даних виникає, коли два або більше потоків здійснюють паралельний доступ до одного й того ж ресурсу, причому хоча б один із них виконує операцію запису [43]. Наявність недостатньо ефективних механізмів синхронізації, призначених для захисту спільного ресурсу, зазвичай призводить до виникнення не актуального стану даних або до непередбачуваних ситуацій у програмному забезпеченні [43].

Кожного разу, коли кілька завдань або потоків отримують доступ до спільного ресурсу без належного захисту одночасно, вони створюють невизначену

або не передбачувану поведінку програми. Такі помилки можуть викликати в системі серйозні втрати даних, які важко знайти й виправити.

Для ілюстрації феномену гонки даних розглянемо простий приклад, представлений псевдокодом на рисунку 3.7. У цьому прикладі два потоки одночасно інкрементують спільну змінну “sharedVar” 100,000 разів кожен, щоб це значення в кінці було 200,000 але через те, що операції читання та запису виконуються некоректно без захисту, значення буде меншим.

```
// Ініціалізація спільної змінної
shared Var ← 0
// Опис функції для збільшення спільної змінної
FUNCTION increment()
    FOR i FROM 1 TO 100,000 DO
        shared Var ← sharedVar + 1
    END FOR
END FUNCTION

// Виконання основної програми
BEGIN
    // Створення двох потоків
    thread1 ← CREATE_THREAD(increment)
    thread2 ← CREATE_THREAD(increment)
    // Очікування завершення потоків
    JOIN_THREAD(thread1)
    JOIN_THREAD(thread2)
    // Виведення результату
    PRINT "Final value of sharedVar: " + sharedVar
END
```

Рис. 3.7 Приклад гонки даних

Цілісність даних – це забезпечення точності та повноти наданої інформації протягом усього їхнього життєвого циклу. Вона також гарантує, що дані залишаються незмінними незалежно від виконуваних операцій.

Атомарність операцій є одним принцип управління транзакціями в базах даних та багато поточному програмуванні. Вона гарантує, що кожна операція або виконується повністю, або взагалі не виконується, не залишаючи дані в проміжному стані.

Ключові аспекти атомарності операцій:

1. Неподільність (англ. Indivisibility): операція не може бути поділена.

2. Умовністьність (англ. Conditionality): відновлюється стан до початку операції.

3. Цілісність при відмовах : не впливають на загальний стан даних.

Атомарність операцій є одним з методів, що використовується для запобігання гонкам даних. Інтерпретуючі її на код, можна сказати так , що певний блок коду виконується повністю без переривань іншими потоками, забезпечуючи таким чином консистентний стан спільних ресурсів. Розглянувши рисунок 3.7 та механізми синхронізації даних, було виправлено помилку гонки даних та преставив псевдокод операції на рисунку 3.8.

Консистентність визначає, наскільки узгодженими є дані на різних вузлах системи після виконання операцій запису. Сильна консистентність гарантує, що всі вузли бачать однаковий стан після завершення операції запису, що є важливим для точності даних, які мають пріоритетне значення. Для забезпечення консистентності використовуються різні механізми синхронізації та управління транзакціями, що дозволяють підтримувати однорідний стан у всіх реплікаціях.

```
// Ініціалізація спільної змінної
sharedVar ← 0
// Ініціалізувати м'ютекс для синхронізації доступу до sharedVar
mutex ← CREATE_MUTEX()

//Опис функції для збільшення спільної змінної з синхронізацією
FUNCTION increment()
    FOR i FROM 1 TO 100,000 DO
        LOCK(mutex)
        sharedVar ← sharedVar + 1
        UNLOCK(mutex)
    END FOR
END FUNCTION

// Виконання основної програми
BEGIN
    // Створення двох потоків
    thread1 ← CREATE_THREAD(increment)
    thread2 ← CREATE_THREAD(increment)
    // Очікування завершення потоків
    JOIN_THREAD(thread1)
    JOIN_THREAD(thread2)
    // Виведення результату
    PRINT "Final value of sharedVar: " + sharedVar
END
```

Рис. 3.8 Приклад атомарності операцій

Наведений приклад (рис 3.9) демонструє, що будь-яке оновлення даних у кеші відразу відображається на всіх репліках, якщо уявний користувач оновлює дані через один вузол, то всі інші вузли повинні отримати оновлення.

```
// Спільні змінні для кешу
sharedVar1 = 0
sharedVar2 = 0

// Мютекс для захисту спільних змінних
mutex ← CREATE_MUTEX()

// Функція для оновлення кешу
FUNCTION updateCache(val1, val2)
    acquire lock on mtx
    sharedVar1 = val1
    sharedVar2 = val2
    //  Необов'язково: Додати синхронізацію з іншими вузлами
    release lock on mtx
END FUNCTION

// Функція для читання з кешу
FUNCTION readCache()
    acquire lock on mtx
    print "sharedVar1:", sharedVar1, ", sharedVar2:", sharedVar2
    release lock on mtx
END FUNCTION

// Виконання основної програми
BEGIN
    // Створення потоків для оновлення кешу
    thread1 ← CREATE_THREAD(updateCache, 10, 20)
    thread2 ← CREATE_THREAD(updateCache, 30, 40)
    // Очікування завершення потоків
    JOIN_THREAD(thread1)
    JOIN_THREAD(thread2)


    // Читання з кешу
    readCache()

    return 0

END
```

Рис. 3.9 Приклад сильної консистентності

3.3 Порівняння з існуючими підходами та аналіз даних

У цьому розділі буде проведено аналіз запропонованого гібридного методу з алгоритмами LRU, MRU та ARC які базуються на подібних принципах управління кешем. Основною метою аналізу є визначення переваг та недоліків кожного методу в контексті продуктивності системи.

Для об'єктивного порівняння алгоритмів використовуватимуться раніше розглянуті метрики продуктивності, такі як:

- пропускна здатність (Throughput);
- затримка (Latency);
- коефіцієнт кеш-потрапляння (Cache Hit Rate);
- коефіцієнт кеш-промаху (Cache Miss Rate).

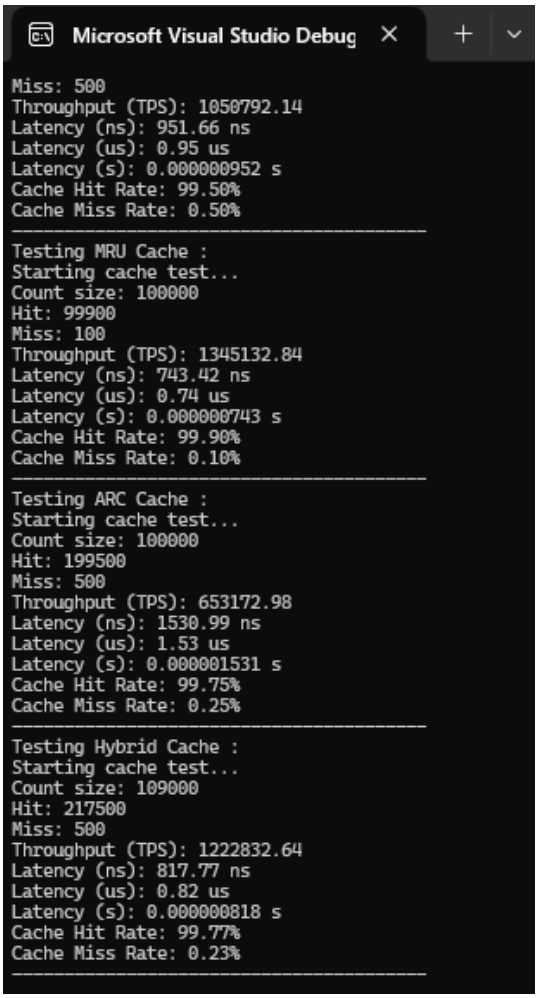


Рис. 3.10 Результати тестування

У тестуванні використовувалася однакова кількість запитів для всіх типів кешів, за винятком “Hybrid” обробляє 100000 звернень, і серед них близько 9% потребують звернення до MRU після невдачі в LRU, то загальна кількість транзакцій буде 109000.

Під час тестування (рис. 3.10) ARC та Hybrid спостерігалися відмінності у значеннях “Miss”. У ARC менше пропусків через логіку де враховується доступ до архівованих списків (B1/B2), а у Hybrid при перемиканні

В таблиці 3.3 наведено всі показники с рисунка 3.10, кожна метрика була розрахована відповідно вище сказаних моделей. Незважаючи на те, що дані підтверджують відповідність очікуваним результатам, слід звернути увагу на те,

що дані, які були використані, мають патерн доступу, це означає, що вони передбачені.

Таблиця 3.3.

Показники тестування

Тип	LRU	MRU	ARC	Hybrid
Кількість запитів	100000	100000	100000	109000
Hit	99500	99900	199500	217500
Miss	500	100	500	500
Throughput (TPS)	1228934,53	1154077,41	707407,19	1195285,53
Затримки (μs)	0,95	0,74	1,53	0,82
Загальний час виконання (S)	0,08	0,10	0,13	0,08
Кеш-потрапляння (H)	99,50%	99,90%	99,75%	99,25%
Кеш-промаху (H')	0,50%	0,10%	0,25%	0,23%

Під час тестування (рис. 3.11) різні алгоритми продемонстрували суттєві відмінності у співвідношенні “Hit” та пропускної здатності (Throughput, TPS). Для LRU низький показник “Hit” (99500) пояснюється тим, що алгоритм орієнтований на збереження даних, до яких зверталися найдавніше. У MRU ситуація кардинально інша, його “Hit” (99900) є найвищим серед усіх, оскільки алгоритм зосереджений на останніх використаних даних, які мають найбільшу ймовірність повторного доступу навіть у випадкових патернах. На відміну від попередніх, ARC представляв середні “Hit” (199 500) і нижчу пропускну здатність (707 407 TPS) через зміщення логіки до різноманітних списків (T1, T2, B1, B2) з урахуванням як гарячого, так і холодного доступу до даних. Hybrid поєднує переваги LRU та MRU,

що дає баланс між кількістю “Hit” (217500) та пропускною здатністю (1,195,285 TPS).

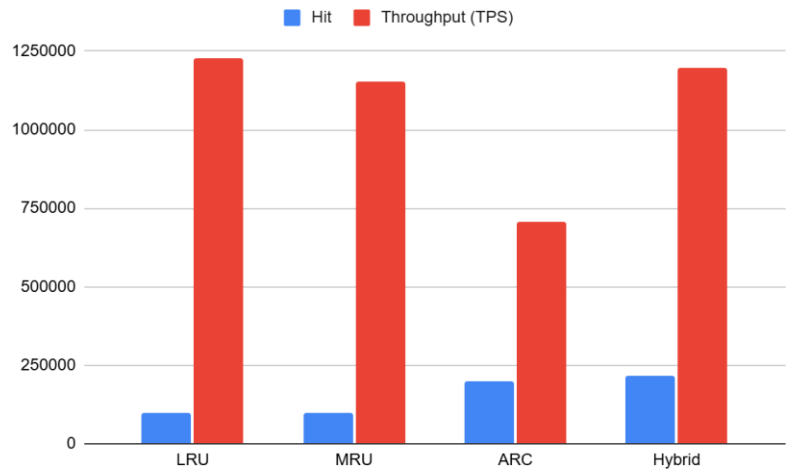


Рис. 3.11 Порівняльне відношення пропускної здатності та “Hit”

Порівняння затримок (Latency) та загального часу виконання (рис. 3.12) для різних типів алгоритмів, при тестуванні продемонстровано співвідношення і лінію тренду значення часу для кожного алгоритму та, як вони реагують на передбачуваність доступу до даних.

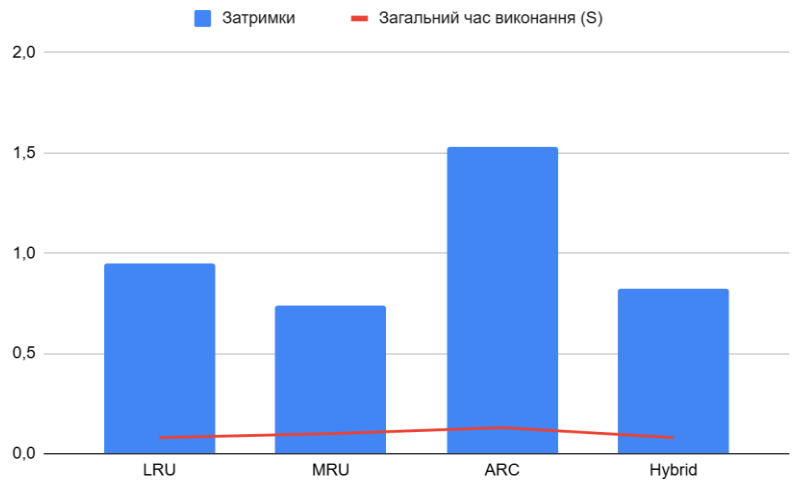


Рис. 3.12 Порівняння затримок та загального часу виконання

На рисунку 3.13 представлена динаміка затримок та загального часу виконання алгоритмів, затримка значно збільшується у ARC, що пов'язано із складністю його внутрішньої логіки. Hybrid утримує стабільність, маючи затримку, близьку до MRU.

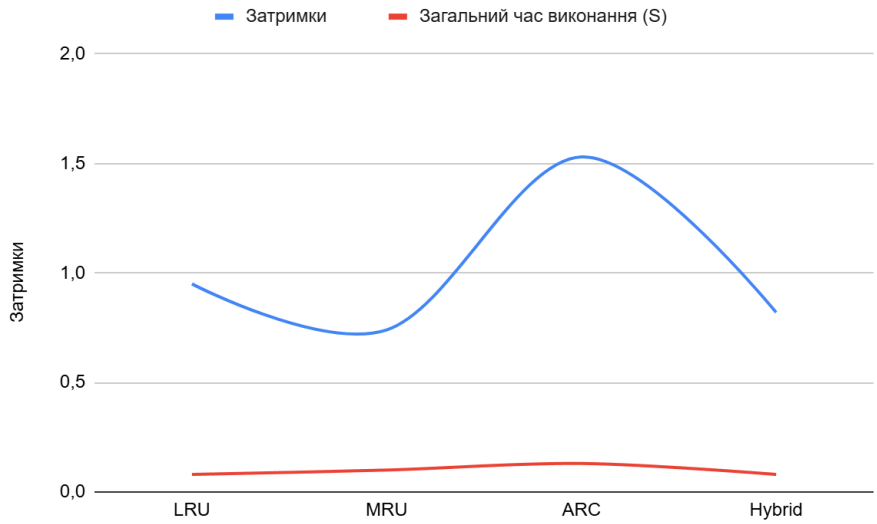


Рис. 3.13 Динаміка затримок та загальний час виконання

Для оцінки ефективності алгоритмів кешування у різних сценаріях, було впроваджено в тестування, симуляцію, псевдовипадкову генерацію шумів для даних. Що забезпечує унікальність ключів для кожного запиту, це означає, що кожен новий запит створює новий ключ, невідомий іншим алгоритмам кешування. Після тестування було визначено алгоритми, які здатні ефективно справлятися з такими умовами.

Алгоритм MRU влаштований таким чином, що кеш зберігає останні використані дані, що дозволяє забезпечувати повторний доступ до найбільш актуальних ключів.

Гібридний метод, крім поєднання принципів LRU та MRU, включає додаткові сценарії обробки даних, такі як хешування даних та ключів. Це дозволяє захистити дані від спаму, проте може призводити до більшої кількості кеш-промахів через складність внутрішньої логіки алгоритму. Завдяки хешуванню,

гібридний метод здатний ефективніше управляти просторовою організацією кешу. У свою чергу, алгоритм LRU зберігає дані, до яких зверталися найдавніше. При появі нового запиту видаляються найстаріші дані, що дозволяє ефективно управляти ресурсами кешу та забезпечувати доступ до більш актуальних даних.

Результати тестування можна розглянути на рисунку 3.14.

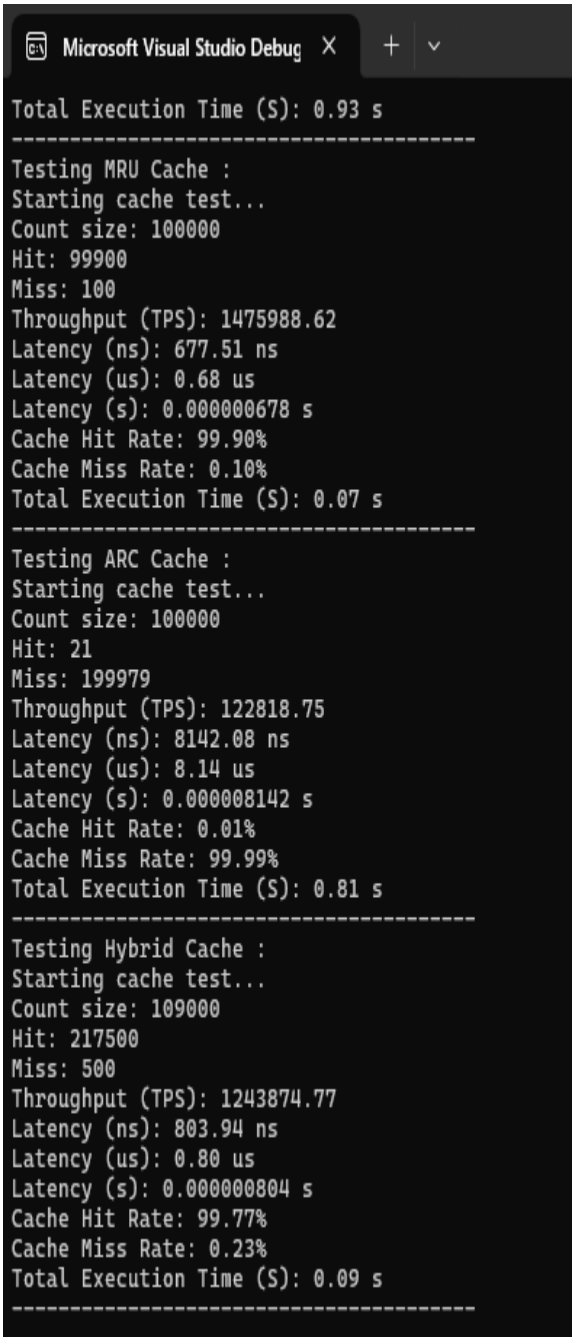


Рис. 3.14 Результати тестування у випадкових патернах

Таблиця 3.4.

Показники тестування у випадкових патернах

Тип	LRU	MRU	ARC	Hybrid
Кількість запитів	100000	100000	100000	109000
Hit	7	99900	21	217500
Miss	99993	100	199979	500
Throughput (TPS)	107612,90	1433519,12	125794,85	1372904,90
Затримки (µs)	9,29	0,70	7,95	0,73
Загальний час виконання (S)	0.93	0,07	0.81	0.09
Кеш-потрапляння (H)	0,01%	99,90%	0,01%	99,77%
Кеш-промаху (H')	99,99%	0,10%	99,99%	0,23%

Показники тестування у випадкових патернах можна розглянути у таблиці 3.4 вона ілюструє, що випадкові патерни доступу суттєво ускладнюють роботу деяким алгоритмом, які залежать від передбачуваності доступу до даних.

На графіку (рис 3.15) ще більш наочно показана розбіжність у кількості “Hit” між шаблонними та випадковими патернами. Для випадкових даних, ефективність LRU падає. MRU залишається ефективним у обох патернах. ARC втрачає свою ефективність. Hybrid практично не втратив ефективності.

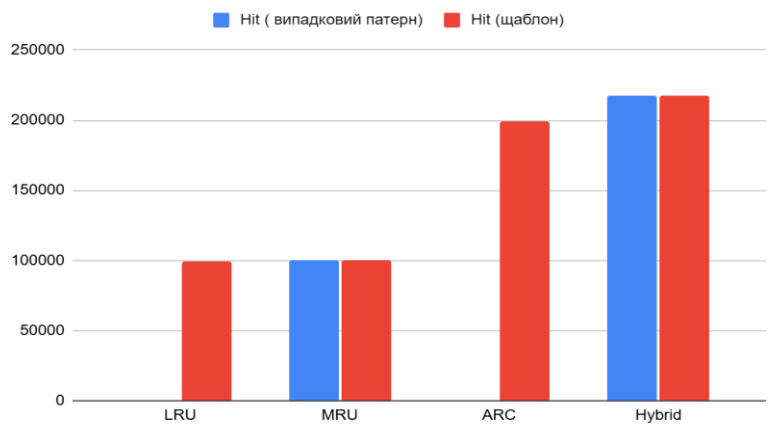


Рис. 3.15 Пряме порівняння у випадкових та шаблонних патернах

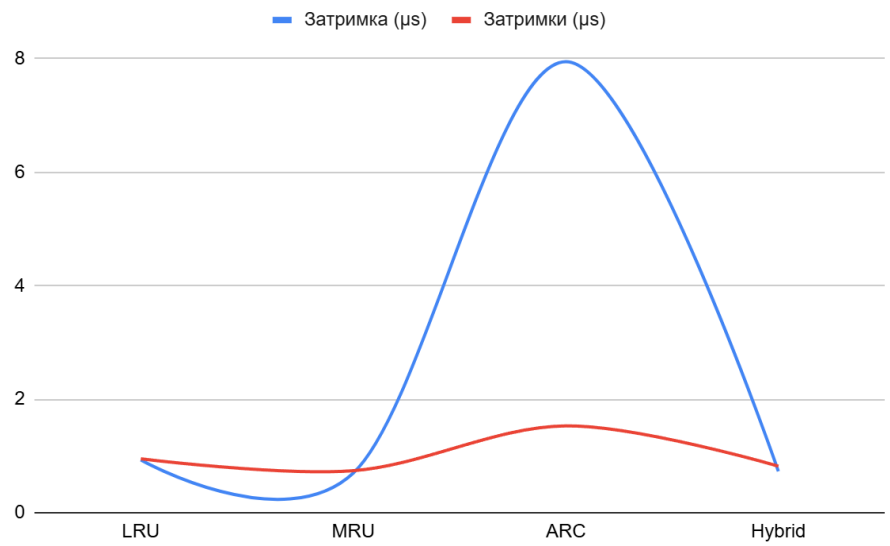


Рис. 3.16 Порівняння затримки у випадкових та шаблонних патернах

Розглянути затримки у випадкових та шаблонних патернах можна на рисунку 3.16, де демонструється залежність у мікросекундах та пропускна здатність (рис 3.17) у запитах на секунду (TPS).

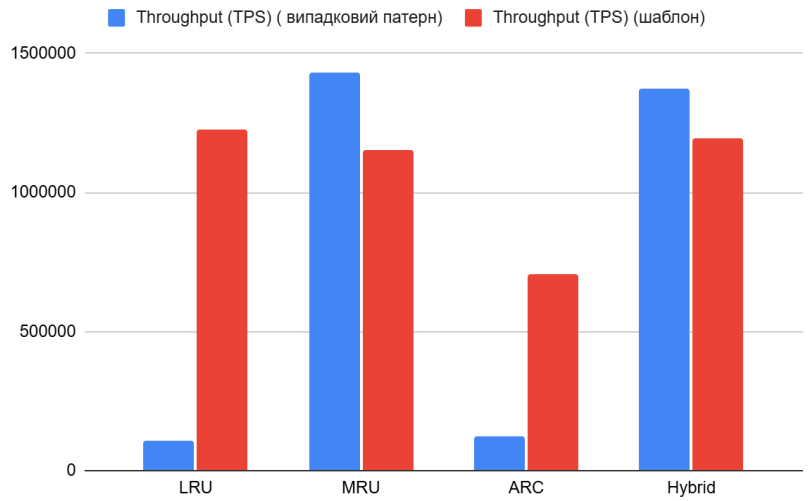


Рис. 3.17 Порівняння пропускну здатність у випадкових та шаблонних патернах

3.4 Висновок до розділу

Було розглянуто та проведено експериментальне дослідження впливу апаратного забезпечення, стосовно багато поточності та вплив паралізму на розроблений метод. Також продемонстровано, що – це не сильно впливає на продуктивність роботи, краще вибрати процесор з більш тактовою частотою та більшою кількістю кеш-пам'яті.

Проведено порівняння гібридного методу з іншими вибірковими алгоритмами та методами, показало, що запропонований підхід забезпечує гарний баланс між пропускною здатністю та затримко, також добре адаптується під нешаблонні дані.

Інші підходити, також мають високу перевагу, особливо потрібно звернути увагу на MRU, продемонстрував перфоманс та не поступався. Але є свої недоліки, що порозводять до звертання у патернах із частим зверненням до старих даних.

ВИСНОВКИ

В кваліфікаційній роботі було досліджено методи та алгоритми кешування даних LRU, MRU, LFU, FIFO та ARC, що дозволило визначити ключові переваги та недоліки кожного алгоритму. Це стало основою для подальшої розробки більш ефективного підходу.

Розроблено методику кешування даних на основі ключових ідентифікаторів та метаданих, що дає змогу аналізувати частоту доступу до даних та їхню дисперсію. Обраний підхід забезпечує вибір між алгоритмами LRU та MRU для підвищення продуктивності системи. Побудовано математичні моделі процесів адаптації та архівації даних, які, зокрема, гарантують зменшення затримок та зростання коефіцієнта кеш-потрапляння. Для забезпечення безпеки та довгострокового зберігання було застосовано криптографічне шифрування, контрольні суми і архівування. Шифрування унеможливлює несанкціоноване прочитання інформації без потрібного ключа. Контрольні суми перевіряють достовірність файлів, адже найменша зміна вмісту викличе розбіжність хеш-значень. Архівування з контрольними сумами забезпечує довгострокове зберігання, та звільняє ресурси основного кешу й полегшує відновлення, зберігаючи цілісність даних.

Реалізовано програмне забезпечення гібридного кешування мовою програмування C++, яке дозволило провести порівняльне тестування з іншими алгоритмами. Експериментальні результати показали, що гібридний підхід забезпечує необхідний баланс між пропускною здатністю та затримкою. У випадкових патернах доступу метод зменшилася затримка та загальний час виконання системи порівняно з алгоритмами LRU та ARC до даних у 0,73 мікросекунд. Це було досягнуто завдяки динамічному перемиканню між алгоритмами LRU та MRU, що дозволило швидко адаптуватися до змінних патернів доступу. Крім того, гібридний метод підвищив коефіцієнт кеш-потрапляння на 9% більше за ARC, без зниження пропускної здатності системи, яка досягла 1 372 904,90 транзакцій за секунду (TPS).

ПЕРЕЛІК ПОСИЛАНЬ

1. Hafeez U. U., Wajahat M., Gandhi A. ElMem: Towards an Elastic Memcached System. 2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS), Vienna, 2–6 July 2018. 2018.

2. Memcached - memcached.org – [Електронний ресурс] – Режим доступу: https://memcached.org/ (дата звернення: 16.05.2024)

3. Redis - redis.io – [Електронний ресурс] – Режим доступу: https://redis.io/

4. Rocksdb – rocksdb.org – [Електронний ресурс] – Режим доступу: https://rocksdb.org/ (дата звернення: 16.05.2024)

5. Couchbase – couchbase.com – [Електронний ресурс] – Режим доступу: https://www.couchbase.com/ (дата звернення: 16.05.2024)

6. Hazelcast -- hazelcast.com – [Електронний ресурс] – Режим доступу: https://hazelcast.com/ (дата звернення: 16.05.2024)

7. Apache Ignite – ignite.apache.org – [Електронний ресурс] – Режим доступу: https://ignite.apache.org/ (дата звернення: 16.05.2024)

8. Ehcache – ehcache.org – [Електронний ресурс] – Режим доступу: https://www.ehcache.org/ (дата звернення: 16.05.2024)

9. Couchdb – couchdb.apache – [Електронний ресурс] – Режим доступу: https://couchdb.apache.org/ (дата звернення: 16.05.2024)

10. Fusion Middleware Developing Applications with Oracle Coherence - docs.oracle.com – [Електронний ресурс] – Режим доступу: http://surl.li/uighg (дата звернення: 18.05.2024)

11. Худік Б.О., Гавор А.С. Огляд методів кешування даних для розробки ефективної методики кешування даних. // Всеукраїнська науково-технічна конференція «Застосування програмного забезпечення в ІКТ» – Київ: ДУІКТ , 2024. – С.135-137.

12. Худік Б.О., Гавор А.С. Модель оптимізації алгоритмів гібридного методу кешування. // ІІ Всеукраїнській науково-технічній конференції «Технологічні

горизонти: дослідження та застосування інформаційних технологій для технологічного прогресу України і світу» - подана до друку. – Київ: ДУІКТ , 2024. – С.320-321.

13. Client-side storage – developer.mozilla.org – [Електронний ресурс] – Режим доступу: http://surl.li/uighl (дата звернення: 18.05.2024)

14. Local Cache - docs.gigaspaces.com – [Електронний ресурс] – Режим доступу: http://surl.li/uighr (дата звернення: 18.05.2024)

15. Киричек Г.Г., Тягунова М.Ю., Братчиков В.В. Система кешування даних в розгалуженій мікросервісній архітектурі. Вчені записки Таврійського національного університету ім. В.І. Вернадського. Серія: Технічні науки. 2024. 141–146.

16. Jelenković P. R., Radovanović A. Least-recently-used caching with dependent requests. Theoretical Computer Science. 2004. Vol. 326, no. 1-3. P. 293–327.

17. Silberschatz A., Gagne G., Galvin P. B. Operating System Concepts. Wiley & Sons, Incorporated, John, 2017.

18. On the existence of a spectrum of policies that subsumes the least recently used (LRU) and least frequently used (LFU) policies / D. Lee et al. ACM SIGMETRICS Performance Evaluation Review. 1999. Vol. 27, no. 1. P. 134–143.

19. FIFO can be Better than LRU: the Power of Lazy Promotion and Quick Demotion / J. Yang та ін. HOTOS '23: 19th Workshop on Hot Topics in Operating Systems, м. Providence RI USA. New York, NY, USA, 2023.

20. Nimrod M., S M. D. ARC: A Self-Tuning, Low Overhead Replacement Cache. 2nd USENIX Conference on File and Storage Technologies, San Francisco, CA, 31 March 2003. P. 115–130.

21. Megiddo N., Modha D. S. Outperforming LRU with an adaptive replacement cache algorithm. Computer. 2004. Vol. 37, no. 4. P. 58–65.

22. What is MRU (Most Recently Used)? - magnetforensics.com – [Електронний ресурс] – Режим доступу: http://surl.li/uighw (дата звернення: 22.05.2024).

23. Gu X., Ding C. On the theory and potential of LRU-MRU collaborative cache management. ACM SIGPLAN Notices, 2011, Vol. 46, no. 11, pp. 43–54.

24.Wang Y. et al. LR-LRU: A PACS-Oriented Intelligent Cache Replacement Policy. IEEE Access, 2019, Vol. 7, pp. 58073–58084.

25. Vanstone S. A., Menezes A. J., Oorschot P. C. Handbook of Applied Cryptography. CRC Press, 1996.

26. Wu L., Zhang W. Cache-Aware SPM Allocation Algorithms for Performance and Energy Optimization on Hybrid SPM-Cache Architecture. Journal of Computing Science and Engineering, 2018, Vol. 12, no. 3, pp. 91–105.

27. Yang J., Yue Y., Rashmi K. V. A Large-scale Analysis of Hundreds of In-memory Key-value Cache Clusters at Twitter. ACM Transactions on Storage, 2021, Vol. 17, no. 3, pp. 1–35.

28. Hu Z. et al. A Collaborative Cache Allocation Strategy for Performance and Link Cost in Mobile Edge Computing. SSRN Electronic Journal, 2022.

29. Sun S., Tang Y., Zhou F. An Index with Caching Mechanism for Real-time Processing System. Proceedings of the 2019 4th International Conference, Guangzhou, China, 2019.

30. Shostack A. Threat Modeling: Designing for Security. Wiley & Sons, Incorporated, John, 2014. 624 p.

31. William S. Cryptography and Network Security: Principles and Practice. Pearson Education, Limited, 2002. 681 c.

32. Web caching and Zipf-like distributions: evidence and implications / L. Breslau та ін. IEEE INFOCOM '99. Conference on Computer Communications. Proceedings. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. The Future is Now (Cat. No.99CH36320), м. New York, NY, USA, 25 берез. 1999 р. 1999.

33. Jacob K. P., Preetha T. J. Cooperative caching architecture for mobile ad hoc networks. International Journal of Information and Communication Technology. 2017. Vol. 11, no. 3. P. 334.

34. Закон Амдала. LibreTexts - Ukrayinska. – ukrayinska.libretexts.org – [Електронний ресурс] – Режим доступу: http://surl.li/reqljp (дата звернення: 20.11.2024).

35. Developing a basic approach to performance measurement and tuning. IBM - United States. – ibm.com – [Електронний ресурс] – Режим доступу: http://surl.li/sokhqp (дата звернення: 17.11.2024).

36. Performance basics. IBM - United States. – ibm.com – [Електронний ресурс] – Режим доступу: http://surl.li/qxzigx (дата звернення: 17.11.2024).

37. IBM. What Is Latency? | IBM. IBM - United States. – ibm.com – [Електронний ресурс] – Режим доступу: http://surl.li/oqbcap (дата звернення: 20.11.2024).

38. Understanding latency - Web performance | MDN. MDN Web Docs. – developer.mozilla.org – [Електронний ресурс] – Режим доступу:http://surl.li/qduciz (дата звернення: 17.11.2024).

39. Raju H. Understanding System Design: Key Concepts Explained with Real-World Examples. *Medium*. – javascript.plainenglish.io – [Електронний ресурс] – Режим доступу: http://surl.li/uvluaq (дата звернення: 18.11.2024).

40. Cache Miss - Everything you need to know. – redis.io – [Електронний ресурс] – Режим доступу: http://surl.li/bcgxvb (дата звернення: 18.11.2024).

41. What is a cache hit ratio?. – cloudflare.com – [Електронний ресурс] – Режим доступу: http://surl.li/ruuhyq (дата звернення: 18.11.2024).

42. Arnold J. Data Consistency vs Data Integrity: Similarities and Differences | IBM. IBM - United States. – ibm.com – [Електронний ресурс] – Режим доступу: http://surl.li/hjqtle (дата звернення: 18.11.2024).

43. What Are Data Races and How to Avoid Them During Software Development. MathWorks - Maker of MATLAB and Simulink - MATLAB & Simulink. – mathworks.com – [Електронний ресурс] – Режим доступу: http://surl.li/sbluju (дата звернення: 18.11.2024).

ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

НАВЧАЛЬНО–НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Магістерська робота

ГІБРИДНИЙ МЕТОД КЕШУВАННЯ НА ОСНОВІ КЛЮЧОВИХ ІДЕНТИФІКАТОРІВ ТА МЕТАДАНИХ

Виконав: студент групи ПДМ-62 Гавор Артур Станіславович

Керівник: доктор філософії, старший викладач кафедри ІПЗ Худік Богдан Олександрович

Київ - 2025

МЕТА, ОБ'ЄКТ ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

Мета роботи: оптимізація кешування даних за рахунок використання гібридного методу на основі алгоритмів LRU та MRU з застосуванням механізму архівування, ключових ідентифікаторів та метаданих.

Об'єкт дослідження: процес кешування даних.

Предмет дослідження: методи та алгоритми кешування даних.

АКТУАЛЬНІСТЬ РОБОТИ

1. Проблеми існуючих алгоритмів кешування

- Недоліки популярних підходів:
  - LRU (Least Recently Used) – неефективний при випадкових запитах;
  - MRU (Most Recently Used) – при перевантаженнях видає аномальні дані;
  - ARC (Adaptive Replacement Cache) має високу рівневу складність по списках.
- Динамічний вибір алгоритму LRU та MRU на основі аналізу частоти доступу через дисперсію. Переміщення рідко використовуваних даних до архіву для звільнення місця в кеші.

2. Запропонована ідея покращення кешування

- Динамічний вибір алгоритму LRU та MRU на основі аналізу частоти доступу через дисперсію.
- Переміщення рідко використовуваних даних до архіву для звільнення місця в кеші.

АЛГОРИТМ ГІБРИДНОЇ АДАПТАЦІЇ

Обчислення дисперсії для алгоритму LRU та MRU :

$$\sigma^2_{LRU} = \frac{1}{T}\sum_{i=1}^{n}(f_{lry}(t_i) - \bar{f})^2 \Delta t_i$$

$$\sigma^2_{MRU} = \frac{1}{T}\sum_{i=1}^{n}(f_{mry}(t_i) - \bar{f})^2 \Delta t_i$$

$$\bar{f} = \frac{\sum_{i=1}^{n}(f(i) \times \Delta t_i)}{\sum_{i=1}^{n}\Delta t_i}$$

$$T = \sum_{i=1}^{n}\Delta t_i$$

Вибір алгоритму по патерну запитів:

$$S(t) = \begin{cases} LRU & \sigma^2_{LRU} < \sigma^2_{MRU} \\ MRU & \sigma^2_{MRU} \le \sigma^2_{LRU} \end{cases}$$

Де:

σ^2_{LRU} – дисперсія частоти доступу для алгоритму LRU;

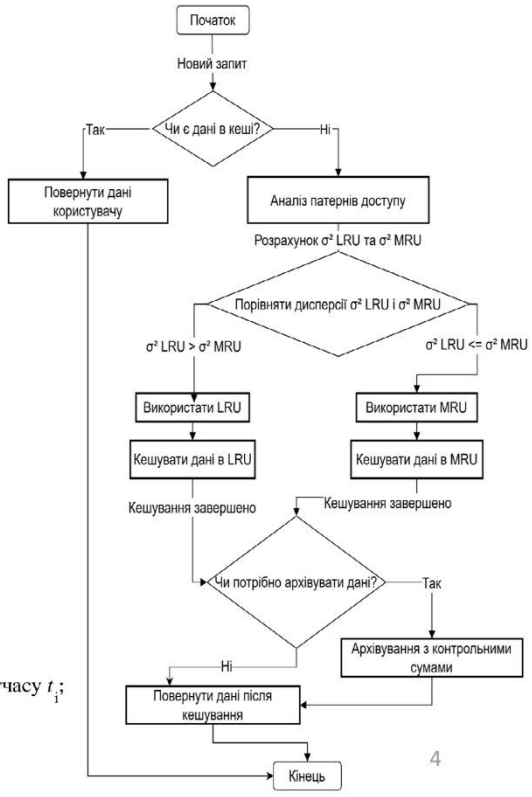
σ^2_{MRU} – дисперсія частоти доступу для алгоритму MRU;

T – загальний час;

$\bar{f}$ – середнє значення частоти доступу для кожного алгоритму за суму всіх часових інтервалів;

$f_{lry}(t_i)$ та $f_{mry}(t_i)$ – частота доступа допевного ключа у кеші, на моментчасу t_i;

n – кількість окремих моментів часу t_i, на яких здійснюється вимірювання показників.



АЛГОРИТМ УЗГОДЖЕННЯ ДАНИХ РЕПЛІКАЦІЙ

Критерій архівування $A(x)$:

$$A(x)\begin{cases} 1, \left(f(x) < f^{alg}_{threshold}\right) \vee \left(t_{cache}(x) > t^{alg}_{max}\right) \wedge p(x) = 0 \\ 0, \text{інакше} \end{cases}$$

Де:

$f(x)$ – частота звернень до даних x;

$f^{alg}_{threshold}$ – порогова частота доступу

для алгоритмів LRU або MRU;

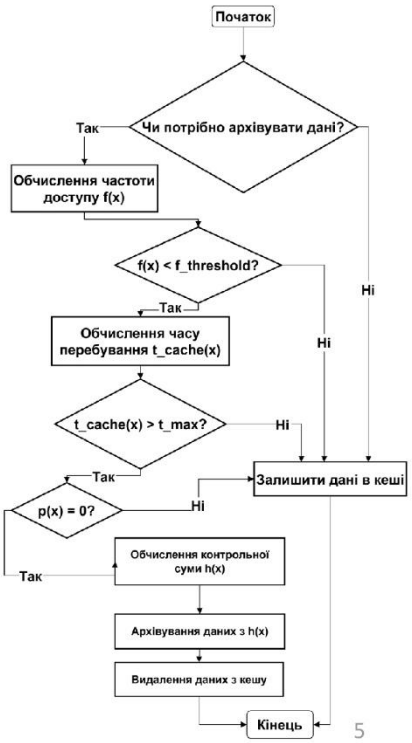
t^{alg}_{max} – максимальне значення життєвого циклу

алгоритму LRU або MRU;

$t_{cache}(x)$ – час перебування в кеші, x елемента;

$p(x)$ – пріоритет x елмента;

$h(x)$ – криптографічна хеш – функкція.



5

АЛГОРИТМ ПЕРЕВІРКИ ЦІЛІСНОСТІ ДАНИХ

Обчислення хеш-функції:

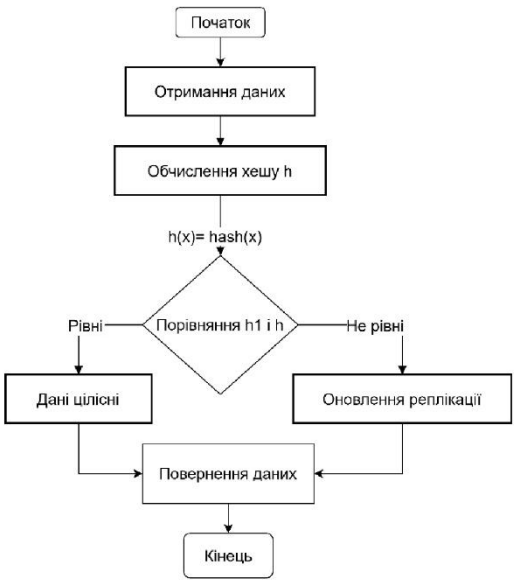
$$h(x)=hash(x)$$

Де:

$h(x)$ – криптографічна хеш-функція;

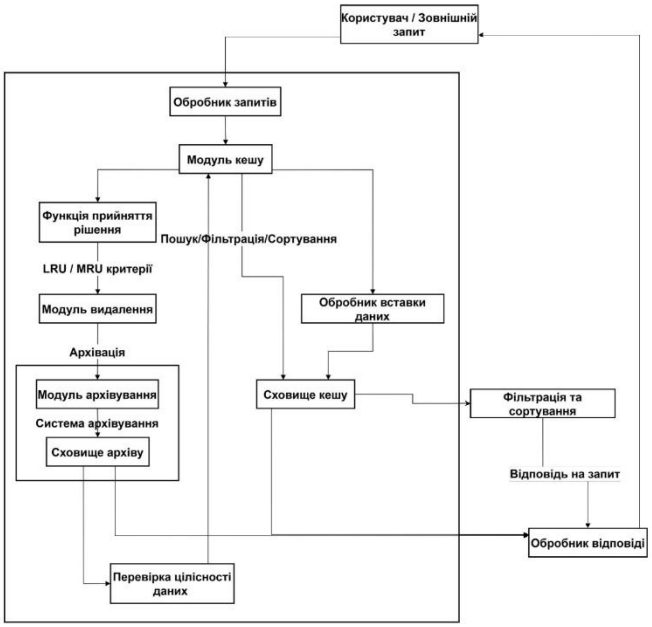
$h1$ – очікуваний хеш;

h – обчислений хеш.



6

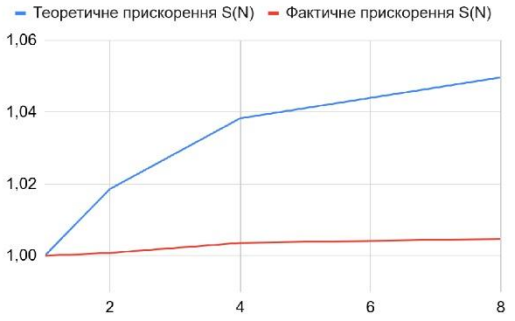
АРХІТЕКТУРА ГІБРИДНОГО КЕШУВАННЯ



7

РЕЗУЛЬТАТИ ТЕСТУВАННЯ ЗА ЗАКОНОМ АМДАЛА

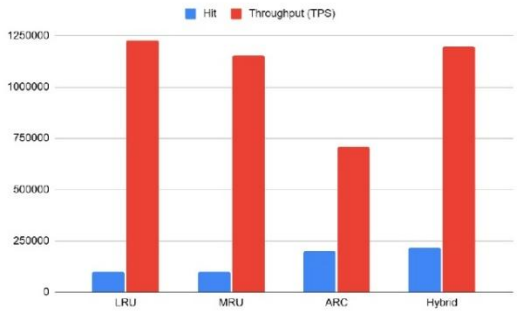
Кількість потоків	Фактичне (s)	Теоретичне (s)	Частка програми	Загальний час виконання (µs)	Паралельний час (µs)	Послідовний час (µs)
1	1,00	1,00	0,03338	61,776	2,062	59,713
2	1,0008	1,0186	0,03655	61,727	2,256	59,470
4	1,0036	1,0383	0,03984	61,551	2,452	59,099
8	1,0047	1,0497	0,04147	61,487	2,550	58,937



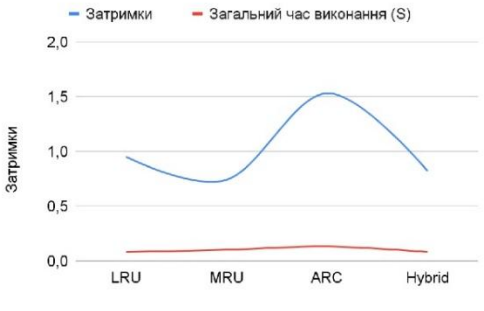
Порівняння теоретичного та фактичного прискорення за законом Амдала

8

РЕЗУЛЬТАТИ ТЕСТУВАННЯ ТА ПОРІВНЯННЯ У ШАБЛОННИХ ПАТЕРНАХ ДАНИХ

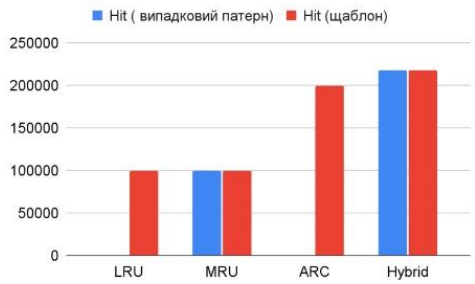


Порівняльне відношення пропускної здатності TPS (Transactions Per Second) та "Hit"(звернень)

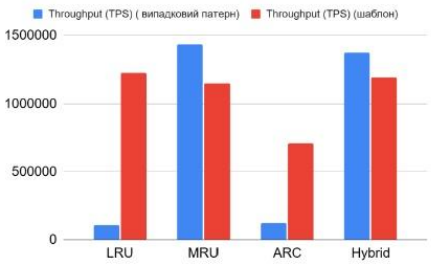


Динаміка затримок та загальний час виконання

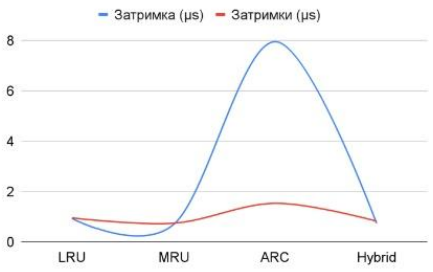
РЕЗУЛЬТАТИ ТЕСТУВАННЯ ТА ПОРІВНЯННЯ У ВИПАДКОВИХ ПАТЕРНАХ ДАНИХ



Пряме порівняння у випадкових та шаблонних патернах



Порівняння TPS (Transactions Per Second) пропускна здатність у випадкових та шаблонних патернах



Порівняння затримок у випадкових та шаблонних патернах

ВИСНОВКИ

1. Проведено аналіз алгоритмів кешування LRU, MRU, LFU, FIFO та ARC, що дозволило визначити їхні ключові переваги та обмеження.
2. Розроблено алгоритми керування кешем з використанням ключових ідентифікаторів та метаданих, які забезпечують динамічний аналіз частоти доступу та розподілу патернів даних.
3. Впроваджено механізми та алгоритми шифрування та перевірки цілісності даних за допомогою алгоритму що гарантує захист кешованої інформації від несанкціонованого доступу та підробки.
4. Реалізовано програмне забезпечення гібридного кешування мовою програмування C++, що дозволило провести порівняльне тестування з іншими алгоритмами.
5. Результати експериментального аналізу гібридного методу ілюструють оптимальний баланс між пропускною здатністю та затримкою, досягаючи затримки доступу до даних у 0,73 мікросекунди, коефіцієнта кеш-потрапляння 99,25% порівняно з алгоритмами LRU та ARC, без зниження пропускної пропускної здатності 1 372 904,90 транзакцій за секунду (TPS).

ПУБЛІКАЦІЇ ТА АПРОБАЦІЯ РОБОТИ

Тези доповідей:

1. Худік Б.О., Гавор А.С. Огляд методів кешування даних для розробки ефективної методики кешування даних. // Всеукраїнська науково-технічна конференція «Застосування програмного забезпечення в ІКТ» – Київ: ДУІКТ , 2024. – С.135-137.
2. Худік Б.О., Гавор А.С. Модель оптимізації алгоритмів гібридного методу кешування. // II Всеукраїнській науково-технічній конференції «Технологічні горизонти: дослідження та застосування інформаційних технологій для технологічного прогресу України і світу» - подана до друку. – Київ: ДУІКТ , 2024. – С.320-321.

ДОДАТОК Б. ЛІСТИНГИ ПРОГРАМНИХ МОДУЛІВ

```
#include "AdaptiveCache.hpp"
#include <iostream>
#include <algorithm>
#include <utility>
#include "ConcreteCacheStrategy.hpp"
#include <cmath>
#include <openssl/sha.h>
#include <fstream>

namespace cache_library {

    adaptive_cache::adaptive_cache(std::shared_ptr<i_cache_strategy> strategy)
        : cacheStrategy(std::move(strategy)) {}

    adaptive_cache::adaptive_cache(const std::shared_ptr<i_cache>& lru_cache_ptr, const std::shared_ptr<i_cache>&
                                   mru_cache_ptr) {
        // Створюємо стратегію за замовчуванням
        cacheStrategy = std::make_shared<concrete_cache_strategy>(lru_cache_ptr, mru_cache_ptr);
    }

    void adaptive_cache::insert(const int key, const int value) const
    {
        // Вибираємо відповідний кеш
        const auto cache = cacheStrategy->select_cache(key);

        // Вставляємо в обраний кеш
        cache->insert(key, value);

        // Оновлюємо стратегію з ключем
        cacheStrategy->update_strategy(key);
    }



    int adaptive_cache::get(const int key) const
    {
        int value = -1;

        const auto lru_cache = cacheStrategy->lruCache();
        const auto mru_cache = cacheStrategy->mruCache();

        if (lru_cache->contains(key)) {
            value = lru_cache->get(key);

        }
        else if (mru_cache->contains(key)) {
            value = mru_cache->get(key);

        }
        else {
            // Спробувати відновити з архіву, якщо реалізовано
        }

        if (value != -1) {
            // Оновлюємо стратегію з ключем
            cacheStrategy->update_strategy(key);
```

```
    }

    return value;
}

std::vector<int> adaptive_cache::filter(const std::function<bool(int)>& predicate) const {
    std::vector<int> result;

    const auto lru_cache = cacheStrategy->lruCache();
    const auto mru_cache = cacheStrategy->mruCache();

    // Фільтрація в LRU кеші
    for (const int key : lru_cache->get_keys()) {
        if (predicate(key)) {
            result.push_back(key);
        }
    }

    // Фільтрація в MRU кеші
    for (const int key : mru_cache->get_keys()) {
        if (predicate(key)) {
            result.push_back(key);
        }
    }

    return result;
}

std::vector<int> adaptive_cache::sort(std::function<bool(int, int)> comparator) const {
            const auto lru_cache = cacheStrategy->lruCache();
            const auto mru_cache = cacheStrategy->mruCache();

    std::vector<int> all_keys = lru_cache->get_keys();
    std::vector<int> mru_keys = mru_cache->get_keys();
    all_keys.insert(all_keys.end(), mru_keys.begin(), mru_keys.end());

    std::ranges::sort(all_keys, std::move(comparator));

    return all_keys;
}
double adaptive_cache::calculate_sha256(const std::string& data) {
    unsigned char hash[SHA256_DIGEST_LENGTH];
    SHA256(reinterpret_cast<const unsigned char*>(data.c_str()), data.size(), hash);
    std::string hashStr(reinterpret_cast<char*>(hash), SHA256_DIGEST_LENGTH);

    double checksum = 0;
    for (unsigned char ch : hash) {
        checksum += static_cast<double>(ch);
    }
    return checksum;
}

void adaptive_cache::write_to_archive_file(int key, double checksum) const
{
    std::ofstream archiveFile(archiveFilePath_, std::ios::app);
    if (archiveFile.is_open())  {
        archiveFile << "Key: " << key << ", Checksum: " << checksum << "\n";
        archiveFile.close();
    }
    else {
```

```
            std::cerr << "Unable to open archive file!" << std::endl;
        }
    }
    void adaptive_cache::display_cache_status() const {
        std::cout << "Cache Status:\n";
        cacheStrategy->lruCache()->display_status();
        cacheStrategy->mruCache()->display_status();
    }

} // namespace cache_library

#include "ConcreteCacheStrategy.hpp"
#include <cmath>
#include <iostream>

namespace cache_library {

    concrete_cache_strategy::concrete_cache_strategy(std::shared_ptr<i_cache> lru_cache, std::shared_ptr<i_cache>
mru_cache)
        : lru_cache_(std::move(lru_cache)), mru_cache_(std::move(mru_cache)), dispersionLRU_(0.0),
dispersionMRU_(0.0){}

    std::shared_ptr<i_cache> concrete_cache_strategy::select_cache(int key) {
        // Вибираємо кеш на основі дисперсії

        if (dispersionLRU_ < dispersionMRU_) {
            return lru_cache_;

        }
        else {
            return mru_cache_;
        }
    }

    void concrete_cache_strategy::update_strategy(int key) {

                constexpr double start = 1.0;
        // Update access frequency
                const auto now = std::chrono::steady_clock::now();

        if (!accessFrequency_.contains(key)) {
            // Ініціалізуємо частоту доступу
            accessFrequency_[key] = start;
            insertionTime_[key] = now;
        }
        else {
            // Оновлюємо частоту доступу з експоненціальним згладжуванням
            accessFrequency_[key] = alpha_ * start + (1.0 - alpha_) * accessFrequency_[key];
        }

     /*   std::cout << accessFrequency_[key];*/
        // Recalculate dispersions
        calculate_dispersions();
    }

 void concrete_cache_strategy::calculate_dispersions() {
    auto now = std::chrono::steady_clock::now();

    double weighted_sum_lru = 0.0, weighted_time_lru = 0.0;
    double weighted_sum_mru = 0.0, weighted_time_mru = 0.0;
```

```
    // Середнє значення f для LRU та MRU з урахуванням часу
    for (const auto& [key, freq] : accessFrequency_) {
        auto it = insertionTime_.find(key);
        if (it == insertionTime_.end()) continue;

        double dt = std::chrono::duration<double>(now - it->second).count();
        dt = (dt <= 0.0) ? 1.0 : dt;

        if (lru_cache_->find(key) != lru_cache_->end()) {
            weighted_sum_lru += freq * dt;
            weighted_time_lru += dt;
        } else if (mru_cache_->find(key) != mru_cache_->end()) {
            weighted_sum_mru += freq * dt;
            weighted_time_mru += dt;
        }
    }

    double mean_lru = (weighted_time_lru > 0.0) ? (weighted_sum_lru / weighted_time_lru) : 0.0;
    double mean_mru = (weighted_time_mru > 0.0) ? (weighted_sum_mru / weighted_time_mru) : 0.0;

    double weighted_sum_dev_lru = 0.0, weighted_sum_dev_mru = 0.0;

    for (const auto& [key, freq] : accessFrequency_) {
        auto it = insertionTime_.find(key);
        if (it == insertionTime_.end()) continue;

        double dt = std::chrono::duration<double>(now - it->second).count();
        dt = (dt <= 0.0) ? 1.0 : dt;

        if (lru_cache_->find(key) != lru_cache_->end() && weighted_time_lru > 0.0) {
            double diff = freq - mean_lru;
            weighted_sum_dev_lru += diff * diff * dt;
        } else if (mru_cache_->find(key) != mru_cache_->end() && weighted_time_mru > 0.0) {
            double diff = freq - mean_mru;
            weighted_sum_dev_mru += diff * diff * dt;
        }
    }

    dispersionLRU_ = (weighted_time_lru > 0.0) ? (weighted_sum_dev_lru / weighted_time_lru) : 0.0;
    dispersionMRU_ = (weighted_time_mru > 0.0) ? (weighted_sum_dev_mru / weighted_time_mru) : 0.0;
}

    // Реалізація методів доступу до кешів
    std::shared_ptr<i_cache> concrete_cache_strategy::lruCache() const {
        return lru_cache_;
    }

    std::shared_ptr<i_cache> concrete_cache_strategy::mruCache() const {
        return mru_cache_;
    }

} // namespace cache_library
```