Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет комп’ютерних наук
(повна назва)
Кафедра програмної інженерії
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА

Пояснювальна записка

рівень вищої освіти другий (магістерський)

Дослідження алгоритмів управління кешованими даними в Redis для підвищення продуктивності застосунків
(тема)

Виконав:
Здобувач 2 року навчання
групи ІПЗм-23-1
Матвій КУЧАПІН
(власне ім’я, прізвище)
Спеціальність 121 – Інженерія програмного забезпечення
(код і повна назва спеціальності)
Тип програми освітньо-наукова
Освітня програма Інженерія програмного забезпечення
(повна назва освітньої програми)
Керівник доц. Олена ШЕВЧЕНКО
(посада, власне ім’я, прізвище)

Допускається до захисту
Зав. кафедри ____________ Кирило СМЕЛЯКОВ
(підпис) (власне ім’я, прізвище)

2025 р

Харківський національний університет радіоелектроніки

Факультет комп’ютерних наук

Кафедра програмної інженерії

Рівень вищої освіти другий (магістерський)

Спеціальність 121– Інженерія програмного забезпечення
(код і повна назва)

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

Освітня програма Інженерія програмного забезпечення
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри ____________
(підпис)
«___»________ 2025р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

здобувачеві Кучапіну Матвію Юрійовичу
(прізвище, ім’я, по батькові)

1. Тема роботи «Дослідження алгоритмів управління кешованими даними в Redis для підвищення продуктивності застосунків»

Затверджена наказом по університету від 15.04.2025р. № 290 Ст

2. Термін подання здобувачем роботи до екзаменаційної комісії 11.06.2025р.

3. Вихідні дані до роботи: *електронні ресурси за обраною тематикою, вимоги до реалізації алгоритмів управління кешованими даними, база даних Redis, середовище Visual Studio 2022, мова C#, платформа .NET, бібліотека StackExchange.Redis.*

4. Перелік питань, що потрібно опрацювати в роботі: *вступ, актуальність теми дослідження та визначення мети роботи, аналіз предметної галузі та огляд механізмів управління кешованими даними, огляд літературних джерел, сучасні підходи до управління кешем, проблеми та перспективи розвитку, постановка задачі, визначення основних обмежень існуючих підходів до управління кешем та потреби у вдосконаленні, аналіз існуючих алгоритмів, детальний розгляд механізмів TTL, LRU та LFU, їх недоліків при роботі зі складними структурами даних, розробка нових підходів, теоретичне обґрунтування запропонованих рішень, програмна реалізація запропонованих підходів, розробка програмного забезпечення для проведення заміврів, проведення експерименту, порівняння отриманих результатів, формування рекомендацій, висновки, перелік джерел посилань, додатки.*

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Терміни виконання етапів роботи	Примітка
1	Отримання завдання	16.04.2025	*виконано*
2	Аналіз предметної галузі і постановка задачі	16.04.2025 - 17.04.2025	*виконано*
3	Аналіз існуючих механізмів управління кешем в Redis	17.04.2025 – 20.04.2025	*виконано*
4	Аналіз обмежень в існуючих механізмах	20.04.2025 – 21.04.2025	*виконано*
5	Реалізація нових підходів	21.04.2025 – 25.04.2025	*виконано*
6	Розробка розширення для Redis	25.04.2025 – 31.04.2025	*виконано*
7	Підготовка до апробації результатів дослідження. Публікація матеріалів	16.04.2025 – 24.04.2025	*виконано*
8	Проведення експерименту	01.05.2025 – 02.05.2025	*виконано*
9	Аналіз експерименту	03.05.2025	*виконано*
10	Підготовка пояснювальної записки	03.05.2025 – 15.05.2025	*виконано*
11	Підготовка презентації та доповіді	16.05.2025 – 19.05.2025	*виконано*
12	Перевірка на плагіат	21.05.2025	*виконано*
13	Нормоконтроль	29.05.2025	*виконано*
14	Рецензування	02.05.2025	*виконано*
15	Попередній захист	05.06.2025	*виконано*
16	Занесення диплома в електронний архів	06.06.2025	*виконано*
17	Допуск до захисту у зав. кафедри	07.06.2025	*виконано*

Дата видачі завдання 16 квітня 2025р.

Здобувач ______________________ Матвій КУЧАПІН
(підпис)

Керівник роботи ______________ доц. Олена ШЕВЧЕНКО
(підпис) (посада, власне ім’я, прізвище)

РЕФЕРАТ / ABSTRACT

Пояснювальна записка до кваліфікаційної роботи: 139 стор., 25 рис., 1 табл., 26 джерел.

АДАПТИВНЕ КЕШУВАННЯ, АЛГОРИТМИ, БАЗА ДАНИХ, КЕШ, КЕШУВАННЯ, ОПТИМІЗАЦІЯ, C#, LFU, LRU, NOSQL, Redis, TTL.

Об’єктом дослідження є механізми управління кешованими даними у Redis, що впливають на ефективність використання пам’яті та швидкодію системи.

Предметом дослідження виступають механізми динамічного керування кешем, зокрема динамічне визначення TTL та адаптивний алгоритм заміщення кешу на базі LRU та LFU, спрямованими на оптимізацію видалення даних у складних структурах для підвищення ефективності роботи Redis.

Метою роботи є проведення дослідження існуючих методів управління кешованими даними та розробка модифікацій і впровадження покращень для їх адаптації в Redis. Це дозволить забезпечити більш ефективне використання пам’яті та підвищити швидкодію системи за рахунок оптимізації процесів видалення та зберігання даних.

Методи розробки базуються на використанні мови програмування C# на платформі .NET, а також бібліотеки StackExchange.Redis, яка забезпечує інтеграцію з Redis. На основі цього буде розроблено розширення з новими підходами для керування складними кешованими структурами даних.

У результаті проведених досліджень та експериментів було проаналізовано ефективність існуючих механізмів, запропоновано новий підхід до динамічного визначення TTL, а також розроблено та експериментально підтверджено ефективність адаптивного алгоритму заміщення кешу на основі поєднання LRU та LFU, що забезпечує покращення показників швидкодії та використання пам'яті у Redis.

ADAPTIVE CACHING, ALGORITHMS, DATABASE, CACHE, CACHING, OPTIMIZATION, C#, LFU, LRU, NOSQL, Redis, TTL.

The object of research is the mechanisms for managing cached data in Redis, which affect the efficiency of memory usage and system performance.

The subject of the study is the mechanisms of dynamic cache management, in particular, dynamic TTL determination and adaptive cache replacement algorithm based on LRU and LFU, aimed at optimizing data deletion in complex structures to improve Redis performance.

The aim of the work is to study existing methods of managing cached data and develop modifications and implement improvements to adapt them to Redis. This will ensure more efficient use of memory and improve system performance by optimizing data deletion and storage processes.

The development methods are based on the use of the C# programming language on the .NET platform, as well as the StackExchange.Redis library, which provides integration with Redis. Based on this, an extension with new approaches for managing complex cached data structures will be developed.

As a result of the research and experiments, we analyzed the effectiveness of existing mechanisms, proposed a new approach to dynamic TTL determination, and developed and experimentally confirmed the effectiveness of an adaptive cache replacement algorithm based on a combination of LRU and LFU, which improves performance and memory usage in Redis.

Завідувачу кафедри ПІ
проф. Кирилу СМЕЛЯКОВУ

ЗАЯВА
щодо самостійності виконання кваліфікаційної роботи та можливості її публікації (та/або публікації анотації кваліфікаційної роботи) в електронному архіві відкритого доступу ElAr KhNURE

Я, Кучапін Матвій Юрійович, здобувач вищої освіти на другому (магістерському) рівні вищої освіти академічної групи ІПЗм-23-1 кафедра програмної інженерії, заявляю: моя кваліфікаційна робота на тему «Дослідження алгоритмів управління кешованими даними в Redis для підвищення продуктивності застосунків», що буде представлена в екзаменаційну комісію для публічного захисту, виконана самостійно, в ній не містяться елементи плагіату і вона може бути опублікована в репозиторії "ElArKhNURE". Погоджуюся з авторським договором, відповідно до Положення про репозиторій ХНУРЕ "ElArKhNURE". Всі запозичення з друкованих та електронних джерел мають відповідні посилання.

Я ознайомлений з вимогами академічної доброчесності, згідно з якими виявлення плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту та застосування дисциплінарних заходів.

20.05.2025 Матвій КУЧАПІН

ЗМІСТ

Перелік умовних скорочень ... 9
Вступ ... 10
1 Аналіз предметної галузі ... 12
1.1 Аналіз проблемної галузі дослідження ... 12
1.2 Огляд існуючих підходів та обмежень ... 14
1.3 Тенденції та перспективи розвитку ... 16
2 Огляд й аналіз літературних, наукових джерел ... 17
2.1 Огляд літературних джерел ... 17
2.2 Аналіз літератури ... 19
2.3 Оцінка актуальності та новизни ... 21
2.4 Висновки з огляду ... 23
3 Постановка задачі ... 25
4 Теоретичне дослідження ... 28
4.1 Аналіз існуючих механізмів управління кешем в Redis ... 28
4.1.1 Механізм TTL ... 28
4.1.2 Алгоритм LRU ... 30
4.1.3 Алгоритм LFU ... 31
4.2 Обмеження стандартних підходів для складних структур даних ... 34
4.3 Пропоновані поліпшення та нові підходи ... 36
4.3.1 Архітектурна концепція системи та інтеграція підходів ... 36
4.3.2 Адаптивне управління часом життя даних (TTL) ... 41
4.3.3 Адаптивний алгоритм заміщення кешу ... 44
5 Опис експериментального дослідження ... 50
5.1 Розробка програмного забезпечення ... 50
5.2 Тестування програмного забезпечення ... 58
5.3 Проведення експерименту ... 60
5.4 Аналіз отриманих результатів ... 63
5.4.1 Cache Hit/Miss Rates метрики ... 64
5.4.2 Evictions метрика ... 67

5.4.3 Average TTL метрика 70
5.4.4 Average Write/Read Times метрики 72
5.4.5 Memory Usage Delta метрика 75
5.4.6 Process Cpu Usage метрика 77
5.4.7 Process Memory Delta метрика 79
5.5 Висновки експериментального дослідження та рекомендації 81
Висновки 84
Перелік джерел посилання 86
Перелік джерел посилання за науковими напрямами керівника та науковців кафедри програмної інженерії 90
Додаток А Звіт результатів перевірки на унікальність тексту в базі ХНУРЕ 91
Додаток Б Код адаптивного кеш клієнта 92
Додаток В Код адаптивного менеджеру для заміщення кешу 104
Додаток Г Презентаційний матеріал до кваліфікаційної роботи 116
Додаток Д Апробація результатів роботи на конференції «13-та Міжнародна науково-технічна конференція «Інформаційні системи та технології ІСТ-2024» 26 - 28 листопада 2024 р» 129
Додаток Е Апробація результатів роботи на конференції «9th Open International Conference "Electrical, Electronic and Information Sciences“ eStream 2025» 133
Додаток Ж Експертний висновок результатів перевірки кваліфікаційної роботи на відповідність оформлення вимогам ДСТУ 3008: 2015 139

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

RDB – Redis Database Backup

AOF – Append-Only File

NoSQL – Not Only SQL

E-commerce – Electronic Commerce

IoT – Internet of Things

TTL – Time-to-Live

LRU – Least Recently Used

LFU – Least Frequently Used

ARC – Adaptive Replacement Cache

ВСТУП

У сучасному світі інформаційних технологій, де обсяги даних постійно зростають, з'являються нові виклики щодо ефективної обробки та зберігання інформації. Великі обсяги даних вимагають від розробників програмних систем та баз даних забезпечення швидкого доступу до інформації, мінімізації затримок та оптимальне використання ресурсів. З цієї причини розробка алгоритмів управління кешованими даними та підвищення ефективності роботи систем становиться пріоритетним завданнями в галузі інформаційних технологій.

Одним з ключових методів оптимізації роботи інформаційних систем є кешування. Кешування це механізм тимчасового зберігання часто використовуваних або важливих даних в оперативній пам'яті для швидкого доступу до них. Кешування дозволяє знизити навантаження на основну базу даних, прискорити обробку запитів та зменшити час відгуку системи. Завдяки цьому даний підхід широко використовується у високонавантажених веб-додатках, системах аналітики та обробки даних, а також у багатьох інших галузях.

Актуальність теми дослідження полягає в необхідності оптимізації механізмів управління кешованими даними у Redis, враховуючи специфіку сучасних систем, де часто використовуються складні структури даних, такі як хеші, множини, списки тощо. Традиційні підходи передбачають видалення всієї структури даних наприкінці її життя, навіть якщо деякі елементи все ще активні, що призводить до надмірного видалення корисних даних і, як наслідок, додаткових витрат ресурсів пам'яті та системи для відтворення кешованих елементів.

Метою роботи є розробка нових алгоритмів до управління кешованими даними в Redis, щоб забезпечити більш ефективне використання пам'яті та підвищення швидкодії системи за рахунок оптимізації видалення та зберігання даних. Для досягнення цієї мети необхідно виконати наступні завдання:

– проаналізувати існуючі підходи до управління кешем у Redis;

– визначити їхні недоліки при роботі зі складними структурами даних;

- запропонувати нові методи та механізми управління кешем, такі як динамічне визначення TTL та розробити адаптивний алгоритм заміщення кешу;
- оцінка ефективності розроблених підходів за допомогою експериментальних тестів, з використанням метрик частоти попадання в кеш (Cache Hit Rate), частоти промахів (Cache Miss Rate), часу виконання операцій читання та запису, залишкового TTL, кількості видалених даних, використання пам’яті, та навантаження на процесор та оперативну пам’ять.

Об’єктом дослідження є механізми управління кешованими даними у Redis, що впливають на ефективність використання пам’яті та швидкодію системи.

Предметом дослідження виступає механізми динамічного керування кешем, зокрема динамічне визначення TTL та адаптивний алгоритм заміщення кешу на базі LRU та LFU, спрямованими на оптимізацію видалення даних у складних структурах для підвищення ефективності роботи Redis.

Методи розробки базуються на використанні мови програмування C# на платформі .NET, а також бібліотеки StackExchange.Redis, яка забезпечує інтеграцію з Redis. На основі цього буде розроблено розширення з новими підходами для керування складними кешованими структурами даних.

Наукова новизна роботи полягає у розробці нового підходу для управління кешованими даними в Redis на основі динамічного визначення TTL та адаптивного алгоритму заміщення кешу на базі LRU та LFU, який враховує особливості роботи зі складними структурами даних. Запропоновані підхід дозволить ефективно управляти пам’яттю, мінімізувати надмірні видалення та забезпечити більш стабільну продуктивність у порівнянні з існуючими підходами.

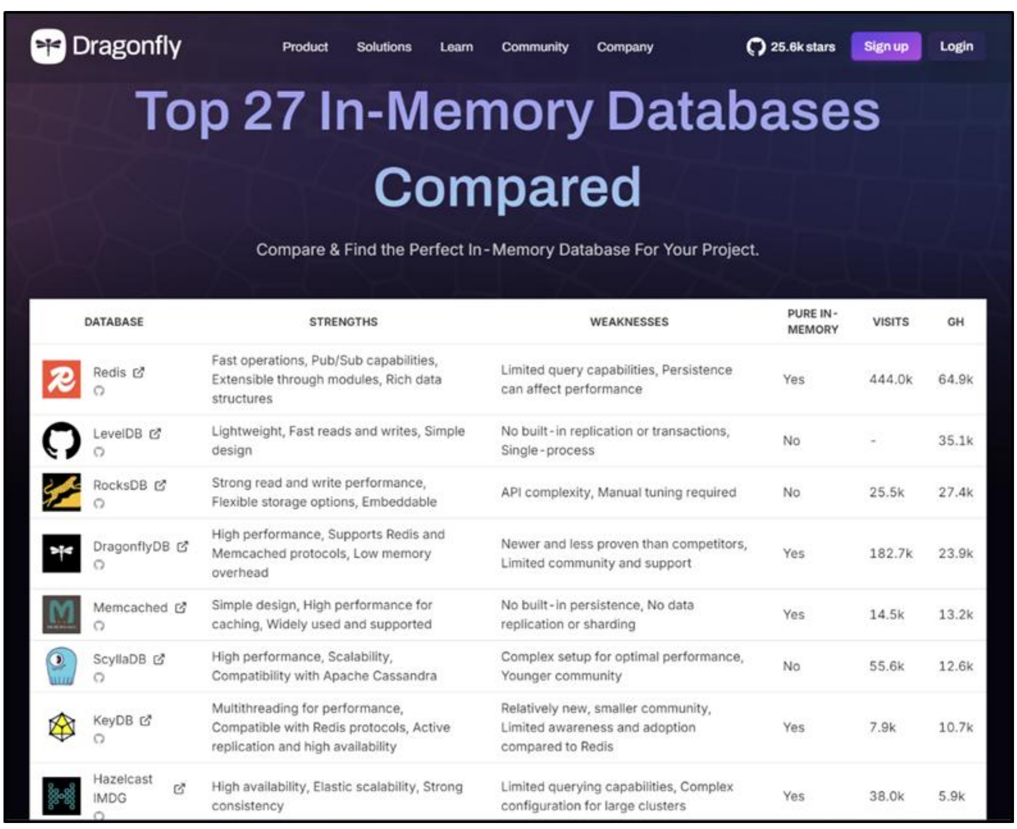
Розроблений підхід є готовим до застосування у системах з високим навантаженням, що використовують Redis для кешування великих обсягів даних.

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ

1.1 Аналіз проблемної галузі дослідження

Кешування – це процес тимчасового зберігання даних у швидкодоступній пам’яті, що дозволяє системам швидше отримувати доступ до часто використовуваної інформації та зменшувати навантаження на основну базу даних [1]. Бази даних, які використовуються для кешування, є важливими компонентами забезпечення швидкодії та продуктивності сучасних програмних систем [2]. Завдяки кешуванню можна значно зменшити затримки при доступі до даних, що особливо важливо для веб-застосунків та систем з великим навантаженням. Тому вибір відповідного інструменту для кешування є важливим кроком для забезпечення стабільної роботи та високої продуктивності таких систем.

Аналізуючи сучасні та популярні in-memory бази даних, що забезпечують швидкий доступ до даних та стабільну роботу систем під значним навантаженням (див. рис. 1.1), для дослідження було обрано Redis як основний інструмент для кешування.



DATABASE	STRENGTHS	WEAKNESSES	PURE IN-MEMORY	VISITS	GH
Redis	Fast operations, Pub/Sub capabilities, Extensible through modules, Rich data structures	Limited query capabilities, Persistence can affect performance	Yes	444.0k	64.9k
LevelDB	Lightweight, Fast reads and writes, Simple design	No built-in replication or transactions, Single-process	No	-	35.1k
RocksDB	Strong read and write performance, Flexible storage options, Embeddable	API complexity, Manual tuning required	No	25.5k	27.4k
DragonflyDB	High performance, Supports Redis and Memcached protocols, Low memory overhead	Newer and less proven than competitors, Limited community and support	Yes	182.7k	23.9k
Memcached	Simple design, High performance for caching, Widely used and supported	No built-in persistence, No data replication or sharding	Yes	14.5k	13.2k
ScyllaDB	High performance, Scalability, Compatibility with Apache Cassandra	Complex setup for optimal performance, Younger community	No	55.6k	12.6k
KeyDB	Multithreading for performance, Compatible with Redis protocols, Active replication and high availability	Relatively new, smaller community, Limited awareness and adoption compared to Redis	Yes	7.9k	10.7k
Hazelcast IMDG	High availability, Elastic scalability, Strong consistency	Limited querying capabilities, Complex configuration for large clusters	Yes	38.0k	5.9k

Рисунок 1.1 – Рейтинг in-memory баз даних (за даними [3])

Це зумовлено його високою швидкодією та використанням оперативної пам’яті для зберігання даних, що забезпечує мінімальну затримку при обробці запитів. Redis є ідеальним рішенням для застосунків із критичними вимогами до швидкості обробки інформації [4]. Підтримка широкого спектра складних структур даних, таких як строки, хеші, множини, списки, дозволить гнучка працювати з різними типами інформації, а можливість збереження на диску за допомогою RDB та AOF забезпечує стійкість даних навіть у разі збою системи. Завдяки своїй швидкодії, продуктивності та підтримці різних політик керування пам’яттю, Redis стабільно займає провідні позиції в рейтингах NoSQL баз даних і є оптимальним рішенням для кешування в сучасних інформаційних системах, таких як реальна аналітика, e-commerce та IoT.

Однак, незважаючи на всі переваги, управління складними структурами даних у Redis пов’язане з низкою труднощів [5]. Проблеми виникають під час роботи з такими структурами, як хеші, множини і списки, де окремі елементи відрізняються за важливістю та частотою використання. У таких випадках надмірне видалення все ще корисних даних або, навпаки, зберігання старих елементів, які споживають ресурси пам’яті, може негативно позначитися на продуктивності системи. Це призводить до збільшення частоти видалення важливих даних і зниження загальної продуктивності через необхідність відтворювати або завантажувати дані повторно. Водночас зберігання застарілої інформації призводить до нераціонального використання пам’яті та знижує ефективність кешування.

Основною проблемою в контексті складних структур даних є відсутність гнучких механізмів управління на рівні окремих елементів усередині цих структур. Це обмежує можливість оптимізації роботи над великими об’єктами, які можуть містити як рідко використовувані, так і часто запитувані елементи. Неповнота наявних механізмів управління призводить до того, що алгоритми не можуть ефективно оцінювати важливість кожного елемента й ухвалювати відповідні рішення щодо їхнього видалення або збереження.

Тому ефективне управління великими та складними структурами кешованих даних потребує розроблення нових адаптивних підходів, що можуть динамічно оцінювати важливість даних, регулювати термін життя елементів та ухвалювати рішення щодо видалення або збереження, враховуючи як частоту, так і актуальність інформації.

1.2 Огляд існуючих підходів та обмежень

У сучасних системах управління кешованими даними ключову роль відіграють алгоритми видалення та оновлення даних. Redis, як одна з найпопулярніших систем кешування, реалізує декілька підходів для управління даними: TTL, LRU та LFU. Однак, кожен із цих методів має свої обмеження при роботі зі складними структурами, що впливає на ефективність та продуктивність системи.

TTL є стандартним механізмом керування терміном життя даних у Redis [6]. Він дозволяє призначати певний термін існування кожному ключу, після закінчення якого дані автоматично видаляються з кешу. Це забезпечує контроль над свіжістю даних та автоматичне очищення кешу від застарілих записів. Проте, у випадку великих структур, таких як хеші або множини, TTL застосовується до всього об’єкта. Це означає, що після завершення терміну життя видаляється вся структура, навіть якщо частина елементів усе ще актуальна. В результаті виникає проблема надмірного видалення корисних даних та збільшення навантаження на систему через необхідність повторного створення цих елементів. Відсутність гнучкого управління TTL на рівні окремих елементів у великих структурах обмежує можливості адаптації системи до різних рівнів важливості та частоти використання даних.

Алгоритм LRU видаляє найдавніше використані елементи, щоб звільнити місце під нові дані [7]. Це дозволяє зберігати у кеші найчастіше використовувані елементи. Однак, при роботі зі складними структурами Redis не завжди коректно визначає, які елементи потрібно видаляти. У випадках, коли деякі елементи у структурі використовуються рідко, але залишаються важливими, LRU може

видалити їх, що призводить до втрати критично важливих даних і зниження ефективності кешування. Відсутність розмежування важливості елементів у великих структурах робить цей алгоритм менш ефективним для складних випадків.

LFU орієнтується на частоту використання даних і видаляє найменш часто запитувані елементи [7]. Він більш ефективний для сценаріїв, де частота використання є важливішим критерієм, ніж час останнього доступу. Проте LFU також має свої обмеження при роботі зі складними структурами. Даний алгоритм не враховує зв'язки між елементами у структурах, що може призводити до видалення важливих даних, якщо вони запитуються нечасто, але є критичними для програми. Крім того, цей алгоритм потребує додаткових ресурсів для відстеження частоти використання, що ускладнює його реалізацію та знижує продуктивність при високих навантаженнях.

Існуючі підходи управління кешем у Redis мають кілька спільних обмежень, зокрема:

- неефективне використання пам'яті при застосуванні TTL до великих структур даних. Весь об'єкт видаляється після закінчення терміну життя, незалежно від активності окремих елементів, що призводить до перевитрати оперативної пам'яті та необхідності повторного створення кешованих даних;
- відсутність можливості гнучкого управління TTL на рівні окремих елементів у великих структурах, що обмежує адаптивність системи до різних рівнів важливості та частоти використання даних. Це не дозволяє коректно видаляти застарілі елементи з великих структур, які можуть мати різну важливість;
- рідко використовувані елементи у великих структурах продовжують займати пам'ять навіть після того, як їх використання стає неактуальним. Це негативно впливає на загальне використання ресурсів системи та може призвести до зниження продуктивності.

Існуючі алгоритми управління кешем у Redis ефективно виконують основні функції, але мають суттєві обмеження під час роботи зі складними структурами даних. Ці недоліки наголошують на необхідності розроблення нових підходів, що забезпечать кращу адаптивність та ефективність під час роботи з великими обсягами даних і складними структурами.

1.3 Тенденції та перспективи розвитку

Сучасний розвиток технологій управління кешованими даними зосереджується на створенні більш адаптивних алгоритмів, здатних гнучка реагувати на зміни в структурі та частоті використання даних. Одним із перспективних напрямків є розробка підходу для динамічного визначення TTL, що дозволить контролювати час життя даних на рівні окремих елементів складних структур, враховуючи їхню активність та важливість. Це дозволить мінімізувати ризик видалення актуальної інформації та зменшити навантаження на систему шляхом автоматичного керування терміном зберігання даних.

Крім того, сучасні дослідження орієнтуються на створення алгоритмів, що поєднують можливості LRU та LFU, але мають покращену гнучкість і здатність адаптуватися до змінних умов роботи системи [8]. Такий підхід дозволяє краще управляти пам’яттю, зберігаючи часто запитувані дані та своєчасно видаляючи рідко використовувані елементи, знижуючи загальну затримку доступу до кешованих даних.

Перспективи розвитку полягають у впровадженні комбінованих алгоритмів, які автоматично підлаштовуються під поточне навантаження та обсяг кешованої інформації [8]. Подальша інтеграція таких адаптивних алгоритмів у кешуючі системи, зокрема Redis, дозволить ефективніше вирішувати проблеми масштабування кешованих даних, підвищувати продуктивність та забезпечувати високу швидкість доступу до даних у системах із високим навантаженням.

2 ОГЛЯД Й АНАЛІЗ ЛІТЕРАТУРНИХ, НАУКОВИХ ДЖЕРЕЛ

2.1 Огляд літературних джерел

У рамках дослідження алгоритмів управління кешованими даними в Redis було проведено комплексний огляд літератури та наукових джерел з метою виявлення сучасних підходів та технологій у сфері кешування, управління пам'яттю та оптимізації продуктивності систем баз даних. Відбір джерел здійснювався за ретельно визначеними критеріями, що забезпечували високу якість та актуальність інформації для поставлених завдань дослідження.

По-перше, для забезпечення надійності джерел, що використовуються, було відібрано матеріали відомих дослідників та експертів у галузі комп'ютерних наук. Особлива увага приділялася публікаціям, які були опубліковані у рецензованих наукових журналах та представленими на міжнародних конференціях. Це забезпечує високу наукову цінність та достовірність представлених даних. Крім того, важливу роль зіграла офіційна документація Redis [9], де міститься детальна інформація про внутрішні механізми роботи системи, що необхідно для глибокого розуміння теми дослідження.

Актуальність літературних джерел була забезпечена відбором матеріалів, опублікованих за останні 10 років. Це дозволило охопити найсучасніші підходи та інновації в галузі управління кешованими даними та відобразити останні тенденції й технологічні досягнення. Використання відповідних джерел є ключем до відображення поточного стану досліджуваної проблеми та визначення напрямку подальших досліджень.

Об'єктивність та достовірність інформації перевірялися за допомогою кількох рівнів аналізу. Перший рівень включав оцінку методології досліджень, представлених у джерелах, що дозволяло визначити надійність та валідність отриманих результатів. Другий рівень передбачав порівняння результатів різних досліджень для виявлення консенсусу або протиріч між ними, що допомагало сформувати об'єктивну картину існуючих підходів та їх ефективності.

Групування джерел за темами стало важливим кроком в організації інформації. Джерела були поділені на кілька основних категорій: механізми

управління кешем в Redis, проблеми управління складними структурами даних, алгоритми адаптивного кешування, а також практичний досвід оптимізації системи. Такий підхід дозволив структуровано охопити всі ключові аспекти дослідження та забезпечити логічну послідовність викладу матеріалу.

Для пошуку літературних джерел використовувалися різноманітні наукові бази даних, такі як IEEE Xplore [10], ACM Digital Library [11], ResearchGate [12] та Google Scholar [13]. Це забезпечило доступ до широкого спектра матеріалів, включаючи статті, конференційні матеріали, дисертації та технічні звіти. Використання різних джерел дозволило зібрати різнобічну інформацію та забезпечити всебічний підхід до аналізу теми. Крім того, для забезпечення повноти огляду літератури були включені матеріали з технічних блогів та публікацій провідних компаній у сфері інформаційних технологій. Ці джерела надали практичний контекст до теоретичних досліджень, демонструючи реальні виклики та рішення, застосовувані на практиці для оптимізації систем кешування.

При відборі джерел також застосовувалися чіткі критерії включення та виключення. До критеріїв включення належали актуальність теми дослідження, висока наукова цінність джерела та внесок у розвиток обраної теми. Критерії виключення включали застарілу інформацію, відсутність експертної оцінки та низьку наукову якість вихідного матеріалу. Особливу увагу було приділено синтезу інформації з різних джерел. Це дало нам змогу виявити загальні тенденції, порівняти різні підходи та визначити прогалини в наявних дослідженнях. А також дало змогу чітко зрозуміти поточний стан питання та визначити галузі для подальших досліджень.

Таким чином, огляд літератури забезпечив базу для аналізу та подальшого дослідження алгоритмів управління кешованими даними у Redis. Систематичний підхід до вибору та аналізу джерел дозволив нам підготувати повний та обґрунтований огляд сучасних підходів, що буде основою для розробки нових методів оптимізації та підвищення ефективності систем кешування.

2.2 Аналіз літератури

Аналіз літератури є важливим етапом дослідження, оскільки дозволяє глибше зрозуміти існуючі підходи до управління кешованими даними в Redis, оцінити їхні переваги та недоліки, а також виявити прогалини, які можуть стати основою для подальших досліджень. У цьому розділі розглядаються основні дослідження, опубліковані з 2015 року, які висвітлюють різні аспекти кешування, управління пам’яттю та оптимізації продуктивності систем баз даних на основі Redis.

Одним із важливих напрямків у сучасних дослідженнях є адаптивні алгоритми кешування, які здатні динамічно адаптуватися до динамічних умов доступу до даних. Наприклад, у статті «Adaptive TTL-Based Caching for Content Delivery» розроблено адаптивний підхід до TTL, який використовує методи машинного навчання для прогнозування популярності даних та динамічного регулювання параметрів кешу [14]. Це дозволяє значно покращити показники продуктивності порівняно з традиційними статичними методами та гнучкіше управляти життєвим циклом даних.

Інше дослідження «Research on Adaptive Cache Mechanism Based on TTL» [15] доповнює ці результати, пропонуючи нові методики адаптації TTL для складних структур даних. Автори дослідження фокусуються на оптимізації часу життя окремих елементів у складних структурах, що дозволяє зберігати активні дані довше та видаляти лише неактивні, підвищуючи тим самим ефективність використання пам’яті та знижуючи навантаження на систему. Це дослідження підтверджує важливість гнучких підходів до TTL та демонструє їхню ефективність у реальних умовах експлуатації.

У дослідженні Shah та Siddiqui [16] розглядається покращена стратегія видалення кешу, яка поєднує алгоритми LRU та LFU. Такий гібридний підхід дозволяє краще адаптуватися до різноманітних шаблонів доступу до даних, зменшуючи кількість помилок кешу та підвищуючи загальну продуктивність системи. Результати показали, що комбінування LRU та LFU забезпечує більш

ефективне використання пам'яті та покращує швидкодію порівняно з використанням кожного з алгоритмів окремо.

Іншим значущим внеском у розвиток адаптивних алгоритмів є робота Shen та ін. [17], де запропоновано політику кешування ARC-learning, яка автоматично налаштовується під час роботи системи відповідно до динамічної популярності даних. ARC-learning використовує методи самонавчання для оптимізації розміру та параметрів кешу, що дозволяє досягати високої продуктивності без необхідності ручного налаштування. Це дослідження демонструє потенціал адаптивних систем кешування у забезпеченні ефективності та адаптивності до змінних умов навантаження.

У роботі Ait-Oucheggou та ін. [18] досліджуються багаторівневі та QoS-орієнтовані стратегії кешування на основі ARC. Автори пропонують мульти-тірні рішення, які враховують якість обслуговування при управлінні кешем, забезпечуючи гнучкість у відповідності до специфічних вимог застосунків. Це дозволяє системам більш ефективно керувати кешованими даними, забезпечуючи баланс між швидкістю доступу та витратами на інфраструктуру.

Дослідження впливу алгоритму видалення кешу в Redis, які представлені у статті «What Are the Impacts of the Redis Expiration Algorithm?» [19], розкривають суттєві аспекти роботи TTL у Redis та їхній вплив на продуктивність системи. Автори аналізують різні сценарії використання TTL та їхні наслідки для управління пам'яттю, що дозволяє краще зрозуміти оптимальні налаштування для різних типів даних та навантажень.

Практичний досвід реалізації адаптивних алгоритмів кешування відображений у документації компанії Microsoft, яка описує оптимізацію Redis у хмарних сервісах Azure. У матеріалі «Azure Cache for Redis and Operational Excellence» детально розглядаються методи забезпечення високої доступності та стійкості до відмов у Redis [20]. Microsoft пропонує стратегії для підвищення продуктивності, включаючи такі методи, як налаштування багаторівневої реплікації, зонова надлишковість та автоматичний перехід на резервні сервери. Крім того, для оптимального управління пам'яттю використовуються гібридні

алгоритми та функції масштабування, які адаптують кеш до мінливих робочих навантажень. Це дозволяє системам забезпечувати низьку затримку та високу доступність навіть під час пікових навантажень.

Таким чином, аналіз літератури показує, що управління кешем та даними в Redis є активною галуззю досліджень, де постійно розробляють нові підходи та методи для підвищення продуктивності, ефективності та безпеки системи. Виявлені тенденції та прогалини вказують на подальші напрямки досліджень, спрямованих на створення гнучкіших, ефективніших і надійніших рішень для керування кешем у сучасних інформаційних системах.

2.3 Оцінка актуальності та новизни

Проведений огляд та аналіз літературних джерел підтверджують високу актуальність дослідження в сфері управління кешованими даними в Redis. Сучасні інформаційні системи, що оперують великими обсягами даних у режимі реального часу, вимагають ефективних механізмів кешування для забезпечення швидкого доступу до інформації та оптимального використання ресурсів пам’яті. Зростаючі вимоги до продуктивності, масштабованості та надійності систем роблять питання управління кешем критично важливим для забезпечення їхньої конкурентоспроможності та ефективності.

Аналіз літератури виявив, що існуючі механізми управління кешем, такі як TTL, LRU та LFU мають значні обмеження при роботі зі складними структурами даних. Зокрема, застосування TTL на рівні всього об’єкта призводить до одночасного видалення всіх його елементів, незалежно від їхньої актуальності. Це негативно впливає на ефективність використання пам’яті та загальну продуктивність системи, особливо при роботі зі складними структурами даних, де різні елементи можуть мати різну частоту доступу та важливість. Така проблема підкреслює необхідність удосконалення існуючих підходів до управління кешем, що стало основою для нашого дослідження.

Новизна нашого дослідження полягає в розробці та впровадженні адаптивних алгоритмів кешування, які здатні динамічно реагувати на змінні

умови доступу до даних та характер навантаження. Використання адаптивного TTL на рівні окремих елементів складних структур даних дозволяє більш гнучка керувати терміном життя даних, зберігаючи активні елементи довше та видаляючи лише неактивні. Це суттєво підвищує ефективність використання пам'яті та знижує навантаження на систему, особливо при роботі з нерівномірно розподіленими та часто змінними даними. Такий підхід відповідає потребам сучасних систем, де різні елементи можуть мати різну частоту доступу та важливість, забезпечуючи більш ефективне управління життєвим циклом даних.

Крім того, наше дослідження зосереджене на комбінуванні різних стратегій заміни кешу, таких як LRU та LFU, для досягнення оптимального балансу між простотою реалізації та ефективністю управління кешем. Розроблені гібридні підходи дозволяють системі більш точно адаптуватися до різних патернів доступу, зменшуючи кількість промахів кешу та покращуючи загальну продуктивність. Це особливо важливо для великих та розподілених систем, де різні компоненти можуть мати різні вимоги до швидкодії та доступу до даних. Гібридні алгоритми, що поєднують переваги LRU та LFU, забезпечують більш ефективне використання пам'яті та покращують швидкодію порівняно з використанням кожного з алгоритмів окремо.

Інноваційним аспектом нашого дослідження є впровадження адаптивних політик кешування, які автоматично регулюють параметри кешу на основі аналізу динаміки популярності даних. Такий підхід дозволяє зменшити потребу в ручному налаштуванні системи, підвищуючи її адаптивність та здатність ефективно працювати в умовах змінних навантажень. Це забезпечує високу продуктивність та стабільність роботи системи навіть при різких змінах у шаблонах доступу до даних, що є критично важливим для забезпечення надійності та ефективності сучасних інформаційних систем.

Практичний аспект нашого дослідження підтверджується впровадженням адаптивних алгоритмів кешування у реальних умовах, зокрема в хмарних сервісах таких великих компаній, як Microsoft. Використання Redis для забезпечення високої доступності та стійкості до відмов демонструє ефективність

запропонованих алгоритмів у реальних умовах експлуатації. Інтеграція з іншими системами кешування та використання гібридних алгоритмів для покращення управління пам’яттю свідчить про високий практичний потенціал цих підходів. Це також підтверджує, що запропоновані рішення можуть бути успішно застосовані в реальних інформаційних системах, забезпечуючи їхню надійність та ефективність.

Оцінка актуальності нашого дослідження також обґрунтована тим, що існуючі дослідження в основному зосереджені на покращенні продуктивності та ефективності алгоритмів заміни кешу, а також на інтеграції адаптивних алгоритмів із різними архітектурами систем кешування. Це свідчить про необхідність подальших досліджень, спрямованих на розробку більш гнучких та ефективних методів управління кешем, що здатні адаптуватися до різних умов та забезпечувати високу продуктивність у складних та динамічних середовищах.

2.4 Висновки з огляду

Проведена оцінка літературних джерел підтверджує, що наше дослідження є актуальним у сфері управління кешованими даними в Redis. Розробка адаптивних та гібридних алгоритмів кешування відкриває нові можливості для оптимізації продуктивності систем, забезпечуючи їхню гнучкість та здатність ефективно працювати в умовах високої динаміки та різноманітних шаблонів доступу до даних. Впровадження автоматичних політик кешування дозволяє суттєво підвищити ефективність використання пам’яті та знизити навантаження на систему, що є ключовими аспектами для сучасних інформаційних технологій.

Дослідження також демонструє високий практичний потенціал запропонованих підходів через їх впровадження в реальних умовах експлуатації, що підтверджується кейсами великих компаній, таких як Microsoft. Інтеграція з іншими системами кешування та акцент на безпеку даних додають цінності нашому дослідженню, роблячи його комплексним та практично значущим. Водночас, виявлені прогалини в існуючих дослідженнях підкреслюють необхідність подальших досліджень у цьому напрямку. Наше дослідження

спрямоване на подолання цих недоліків, пропонуючи нові методи, що враховують як продуктивність, так і ефективність управління кешем.

Таким чином, дослідження створює міцний фундамент для розвитку передових методів оптимізації та підвищення ефективності систем кешування в Redis, відповідаючи сучасним вимогам до інформаційних систем та відкриваючи нові напрями для подальших наукових досліджень.

3 ПОСТАНОВКА ЗАДАЧІ

Після аналізу предметної галузі та виявлення основних проблем, пов’язаних із управлінням кешованими даними у Redis, було визначено необхідність розробки нових підходів, які б враховували специфіку роботи зі складними структурами даних, такими як хеші, множини та списки, а також забезпечували ефективне використання пам’яті. Поставлена задача передбачає створення нових підходів, здатних динамічно управляти часом життя окремих елементів у таких структурах та ефективно видаляти рідко використовувані дані, поєднуючи підходи LRU та LFU для адаптивного менеджеру кешу.

Метою дослідження є розробка механізмів управління кешованими даними, які оптимізують продуктивність Redis при роботі з великими обсягами даних та складними структурами.

Основні проблеми, які пов’язані з управлінням кешованими даними у Redis, виникають через обмеження стандартного механізму TTL та алгоритмів заміщення кешу, як LRU та LFU. Вони застосовуються до всього об’єкта, не враховуючи активність або важливість окремих елементів. Це призводить до ситуацій, коли видаляються всі елементи складної структури, навіть якщо частина з них активно використовується, або навпаки – застарілі дані продовжують займати пам’ять. Ці недоліки знижують продуктивність системи, особливо у високонавантажених сценаріях, де важливим є швидкий доступ до часто використовуваних даних.

Для вирішення цих проблем необхідно розробити підхід для застосування та динамічного визначення TTL, який би дозволяв задавати час життя окремих елементів складних структур, зменшуючи ймовірність надмірного видалення корисних даних та перевантаження пам’яті. Окрім того, запропоновані підходи повинні включати адаптивний алгоритм заміщення кешу на базі LRU та LFU, які враховували б частоту доступу до елементів та їхню актуальність. Це дозволить знизити частоту видалення важливих даних через низьку частоту звернення. Розробка запропонованих підходів та модифікацій повинна враховувати також особливості реалізації в Redis, зокрема використання оперативної пам’яті для

зберігання даних, що забезпечує високу швидкість доступу, але потребує оптимального управління пам'яттю для запобігання зниженню продуктивності системи.

Розробка програмного забезпечення для реалізації запропонованих підходів та модифікацій для управління кешем має здійснюватися на платформі .NET з використанням мови програмування C# та бібліотеки StackExchange.Redis. Це дозволить інтегрувати їх безпосередньо в існуючі додатки, що використовують Redis як основну платформу для кешування даних. Планується створення окремого модуля, який працюватиме як розширення для бібліотеки StackExchange.Redis, що забезпечить легке впровадження розроблених підходів та модифікацій без змін у основному коді додатків.

Після реалізації програмного забезпечення необхідно створити тестове середовище для проведення експериментів, яке включатиме різні типи даних та структури, що відповідають реальним умовам експлуатації системи. Тестове середовище має включати складні структури даних у Redis, такі як великі хеші, множини та списки з різними обсягами та частотою доступу. Це дозволить оцінити ефективність модифікованих підходів в умовах, максимально наближених до реальних сценаріїв. Для збору даних щодо продуктивності планується розробити набір запитів різної складності, що імітують типові сценарії роботи додатків, які використовують Redis для кешування. Основні метрики, що будуть використовуватися для оцінки ефективності розроблених алгоритмів, включають частоту попадання в кеш (Cache Hit Rate), частоту промахів (Cache Miss Rate), час виконання операцій читання та запису, залишковий TTL, кількість видалених даних, використання пам'яті, та навантаження на процесор та оперативну пам'ять.

Проведення тестування з використанням стандартних алгоритмів та розроблених адаптивних алгоритмів дозволить порівняти їхню ефективність на різних обсягах даних та під різними навантаженнями. Після збору результатів тестування планується провести детальний аналіз ефективності кожного алгоритму та зробити висновки щодо його доцільності у різних сценаріях

використання. Важливим аспектом є визначення, наскільки нові алгоритми дозволяють знизити частоту видалення активних даних, зменшити загальне навантаження на систему та покращити показники продуктивності.

Обґрунтування вибору інструментів дослідження базується на популярності та практичності використання платформи .NET та мови програмування C#, які дозволяють швидко розробляти високопродуктивні додатки для роботи з Redis. Бібліотека StackExchange.Redis надає широкий набір можливостей для роботи з кешованими даними, що робить її ідеальним вибором для інтеграції нових алгоритмів. Redis обрано як основну платформу для кешування завдяки його популярності, високій продуктивності та можливості гнучкого налаштування алгоритмів управління пам’яттю. Крім того, відкритий код Redis дозволяє дослідити його внутрішню архітектуру та адаптувати нові підходи та алгоритми до специфічних потреб дослідження, що є важливим фактором для ефективної реалізації нових підходів.

Очікувані результати включають розробку підходу для застосування та динамічного визначення TTL, який дозволить більш гнучка управляти часом життя окремих елементів складних структур даних у Redis, а також створення адаптивного алгоритму заміщення кешу на базі LRU та LFU. Очікується, що розроблені адаптивні алгоритми забезпечать покращення метрик продуктивності, таких як Cache Hit Rate, зниження Cache Miss Rate та оптимізація використання пам’яті. Результати дослідження можуть бути корисними як для наукової спільноти, так і для практичного використання в індустрії, надаючи розробникам інструменти для ефективнішого управління кешованими даними та підвищення продуктивності додатків, які використовують Redis.

4 ТЕОРЕТИЧНЕ ДОСЛІДЖЕННЯ

4.1 Аналіз існуючих механізмів управління кешем в Redis

У сучасних високопродуктивних системах управління кешем є критично важливим для забезпечення швидкого доступу до даних та ефективного використання ресурсів. Redis, як одна з найпопулярніших систем кешування та баз даних типу «ключ-значення», пропонує різноманітні механізми управління кешем. До основних з них належать механізм TTL та алгоритми заміщення кешу LRU та LFU. У цьому розділі розглянемо їхні принципи роботи, переваги та обмеження.

4.1.1 Механізм TTL

Механізм TTL у Redis дозволяє встановлювати час життя для кожного ключа, після закінчення якого ключ автоматично видаляється з бази даних. Це полегшує управління неактуальними або тимчасовими даними без необхідності написання додаткового коду для їх ручного видалення. TTL працює на рівні ключа і може бути встановлений за допомогою кількох команд, таких як EXPIRE для задання часу в секундах, PEXPIRE – в мілісекундах, а також EXPIREAT і PEXPIREAT для визначення точного часу закінчення життя ключа на основі UNIX-часу. На рисунку 4.1 зображена команда, яка налаштовує збереження сесії користувача протягом 3600 секунд, 1 година.

```
SET session:user:1001 "session_data"
EXPIRE session:user:1001 3600
```

Рисунок 4.1 – Встановлення сесії користувача (рисунок виконаний самостійно)

Redis використовує два механізми для видалення прострочених ключів: ліниве та періодичне видалення. Ліниве видалення відбувається у момент, коли клієнт звертається до ключа. Redis перевіряє його час життя (TTL), і якщо він закінчився, то ключ видаляється, а користувач отримує відповідь про те, що ключ більше не існує. Такий підхід дозволяє уникнути зайвого споживання ресурсів на

видалення ключів, які більше не використовуються. Однак ліниве видалення має недолік, що прострочені ключі залишаються у пам'яті до тих пір, доки до них не відбудеться звернення. Така робота механізму може призводити до зайвого використання ресурсів, особливо якщо такі ключі рідко використовуються.

Періодичне видалення працює шляхом регулярних перевірок ключів з встановленим TTL. За замовчуванням кожні 100 мілісекунд Redis випадковим чином перевіряє певну кількість ключів та видаляє ключі у яких час життя вже закінчився. Цей механізм допомагає уникнути накопичення великої кількості прострочених ключів, що могло б призвести до зниження продуктивності системи. Водночас періодичне видалення потребує додаткових обчислювальних ресурсів, оскільки його робота передбачає регулярне виконання перевірок у фоновому режимі. Поєднання цих двох механізмів дозволяє Redis ефективно управляти простроченими ключами, зберігаючи баланс між продуктивністю системи та ефективним використанням пам'яті.

Проте, основним обмеженням механізму TTL є те, що він діє тільки на рівні ключа, а не на рівні окремих елементів всередині складних структур даних, таких як хеші, списки або множини. Це призводить до низки проблем, зокрема до видалення актуальних даних. Наприклад, якщо встановити TTL на хеш, що зберігає інформацію про користувача (рис. 4.2), то всі поля цього хешу будуть видалені одночасно після закінчення терміну дії, навіть якщо деякі з них, як name та email, залишаються актуальними.

```
HMSET user:1001 name "Alice" email "alice@example.com" last_login "2024-10-14"
EXPIRE user:1001 86400
```

Рисунок 4.2 – Зберігання інформації про користувача (рисунок виконаний самостійно)

Такий підхід може призвести до втрати важливих даних, які ще можуть бути корисними, а також до неефективного використання ресурсів, оскільки відновлення великих структур даних вимагає значних витрат на систему й збільшує затримки для користувачів. Наприклад, якщо у хеші зберігаються дані

користувача, такі як name, email, last_login, то видалення цього хешу через TTL призведе до втрати всієї цієї інформації одночасно, незалежно від того, що лише деякі поля можуть бути актуальними для подальших операцій.

4.1.2 Алгоритм LRU

Алгоритм LRU в Redis використовується для керування кешем, видаляючи ключі, до яких не було звернень найдовший час. Даний механізм активується, коли використання пам’яті перевищує встановлений ліміт і тоді, Redis застосовує одну з політик витіснення на основі LRU. Зокрема, використовуються такі політики:

– volatile-lru – видаляються найменш недавно використовувані ключі, які мають встановлений TTL;
– allkeys-lru – видаляються найменш недавно використовувані ключі незалежно від наявності TTL.

Для реалізації алгоритму LRU Redis використовує комбінацію двостороннього зв’язного списку та хеш-таблиці. Двосторонній зв’язний список зберігає ключі у порядку їх останнього доступу, де найновіші звернення знаходяться на одному кінці списку, а найстаріші – на іншому. Хеш-таблиця забезпечує швидкий доступ до елементів списку, дозволяючи оперативна оновлювати позицію ключа у списку при кожному зверненні до нього. Кожного разу, коли ключ звертається, його позиція у списку оновлюється, переміщуючи його до початку списку, що позначає його як найновіше використаний.

Принцип роботи алгоритму LRU полягає в тому, що система відстежує порядок доступу до кожного ключа, забезпечуючи зберігання найбільш часто використовуваних даних у кеші. Коли ліміт пам’яті перевищено, Redis видаляє ключ з кінця списку, тобто той, до якого не було звернень найдовший час. Це дозволяє оптимізувати використання пам’яті, зберігаючи актуальні дані та автоматизуючи процес очищення кешу без необхідності ручного втручання. Алгоритм LRU підвищує загальну ефективність системи, зменшуючи час

відповіді на запити користувачів за рахунок швидшого доступу до часто використовуваних даних.

Однією з основних переваг алгоритму LRU є його здатність оптимізувати використання пам'яті, зберігаючи у кеші ті ключі, до яких звертаються найчастіше, та автоматично видаляючи ті, що рідко використовуються. Це забезпечує зменшення часу відповіді системи та підвищує її продуктивність. Крім того, автоматизація управління кешем знижує потребу в ручному очищенні, що спрощує адміністрування системи та зменшує ризик людських помилок.

Однак у деяких сценаріях ефективність LRU може бути знижена. Наприклад, якщо обсяг нових даних швидко збільшується, то нові ключі можуть витіснити старі, але важливі дані, які ще можуть знадобитися. Крім того, періодичні схеми доступу можуть призводити до видалення даних у періоди бездіяльності, що тягне за собою додаткові витрати на перезавантаження. Крім того, LRU можуть не враховувати важливість рідко використовуваних, але критично важливих даних, що може негативно позначитися на системі. Ось один із таких прикладів, у системі прогнозування погоди доступ до історичних даних здійснюється дуже рідко, але вони використовуються для моделювання. LRU може видалити ці дані з кешу, і під час наступного звернення до них система витратить більше часу на їх повторне завантаження, що знизить ефективність роботи.

4.1.3 Алгоритм LFU

Алгоритм Least Frequently Used у Redis призначений для управління кешем на основі частоти використання ключів. Основна ідея полягає в тому, що ключі, до яких звертаються найрідше, з більшою ймовірністю будуть видалені з кешу. Таким чином, LFU забезпечує збереження в пам'яті найпопулярніших даних, які найбільше потрібні користувачам.

У Redis LFU реалізований через дві основні політики витіснення:

– volatile-lfu – видаляються найменш часто використовувані ключі, що мають встановлений TTL;

– allkeys-lfu – видаляються найменш часто використовувані ключі незалежно від того, чи мають вони TTL.

Для відстеження частоти використання кожного ключа Redis використовує спеціальний лічильник, який збільшується при кожному зверненні до ключа. Цей лічильник не зростає лінійно, а коригується за допомогою певної функції, що враховує час, щоб уникнути необмеженого зростання значень і зберегти баланс між частотою використання та актуальністю даних. Таким чином, Redis може ефективно визначати, які ключі слід видаляти, базуючись на їхній частоті використання протягом певного періоду часу.

Алгоритм LFU ефективний у ситуаціях, коли популярність даних стабільна в часі. Наприклад, у системах, де певні дані завжди користуються високим попитом, LFU може забезпечити оптимальне управління кешем. Прикладами таких систем є:

– системи з постійною популярністю даних, наприклад, інтернет-магазини, де деякі товари завжди мають високий попит;
– контент-провайдери, де користувачі постійно взаємодіють із певним популярним контентом, таким як відео чи статті;
– аналітичні системи, в яких важливо швидко отримувати доступ до даних, що часто запитуються для забезпечення ефективності аналітики.

Однак алгоритм LFU має недоліки, які можуть обмежити його ефективність у деяких сценаріях. Один із головних недоліків – повільна адаптація до змін популярності даних. Якщо популярність певного ключа різко змінюється, LFU може не встигнути швидко відреагувати. Відновлення таких даних вимагає додаткових ресурсів і часу, що може негативно позначитися на продуктивності системи. Наприклад, в аналітичних системах або системах прогнозування погоди, де доступ до історичних даних здійснюється нечасто, але вони необхідні для моделювання, LFU може видалити ці дані, що призведе до збільшення часу на повторне завантаження та аналіз.

Щоб краще зрозуміти, як працюють алгоритми LRU та LFU, розглянемо їхні основні відмінності. Обидва підходи використовуються для управління

кешем, але відрізняються критеріями вибору ключів для видалення. LRU віддає пріоритет нещодавно використаним ключам та видаляє ключі до яких давно не зверталися, без урахування частоти доступу. LFU, навпаки, орієнтується на частоту запитів та зберігає ключі, що використовуються найчастіше, а ключі, які використовуються рідко будуть видалені.

У таблиці 4.1 наведено порівняльний аналіз, який допоможе зрозуміти ключові відмінності між цими алгоритмами та їхні оптимальні сфери застосування.

Таблиця 4.1 – Порівняльна таблиця алгоритмів LRU та LFU (за даними [21])

Параметр	LRU	LFU
Критерій видалення	Час останнього доступу до ключа	Частота доступу до ключа
Переваги	Простота реалізації	Зберігає найпопулярніші ключі
Недоліки	Може видаляти часто використовувані, але нещодавно неактивні ключі	Повільна адаптація до змін у популярності
Підходить для	Динамічних патернів доступу	Стабільних патернів доступу

Дані характеристики вказують на те, що вибір між LFU та LRU залежить від конкретної системи та вимог до кешування. LRU є більш ефективним у системах з динамічними патернами доступу, де актуальність даних змінюється швидко. LFU, навпаки, підходить для систем зі стабільною популярністю даних, де певні ключі постійно користуються попитом. Різні патерни доступу до даних можуть впливати на ефективність кожного з цих алгоритмів, тому вибір алгоритму управління кешем повинен базуватися на аналізі поведінки користувачів та характеру даних у системі.

4.2 Обмеження стандартних підходів для складних структур даних

При розробці високопродуктивних систем, які працюють з великими та складними структурами даних, ефективне управління кешем є критично важливим для забезпечення швидкодії та стабільності. Стандартні підходи до управління кешем, такі як Time-to-Live, Least Recently Used та Least Frequently Used, широко застосовуються в Redis. Однака при роботі зі складними структурами даних ці методи мають суттєві недоліки, що можуть негативно впливати на продуктивність системи та роботу з даними користувачів.

Механізм TTL дозволяє встановлювати час життя для ключів, після закінчення якого ключ автоматично видаляється з бази даних. Однак TTL застосовується на рівні ключа, а не на рівні окремих елементів складних структур даних, таких як хеші, списки чи множини. Це означає, що після закінчення TTL вся структура даних видаляється повністю, навіть якщо деякі її елементи залишаються актуальними та активно використовуються [22].

Наприклад розглянемо хеш user:session на рисунку 4.3, який містить інформацію про сесії користувачів. Поле cart_items може оновлюватися рідко, тоді як поле last_activity оновлюється при кожній активності користувача. Якщо встановити TTL на 1 годину, то через цю годину вся інформація про сесію буде видалена, навіть якщо користувач продовжує бути активним. Це призводить до втрати актуальних даних та необхідності повторного створення сесії, що є неефективним використанням пам'яті та ресурсів системи. Така неефективність може спричинити зайві витрати ресурсів на часті видалення та повторне створення великих структур даних, погіршити продуктивність через зростання часу відповіді системи.

```
HMSET user:session:user123 cart_items "item1,item2" last_activity "2024-10-14T12:34:56Z"
EXPIRE user:session:user123 3600
```

Рисунок 4.3 – Хеш user:session (рисунок виконаний самостійно)

Least Recently Used – це політика витіснення, яка видаляє з кешу найдовше невикористовувані елементи. Цей підхід добре працює для простих ключів, але

має обмеження при роботі зі складними структурами даних [23]. Алгоритм працює на рівні ключа і не враховує, як часто звертаються до окремих елементів у складних структурах. Наприклад, якщо в хеші є кілька полів, деякі з яких активно використовуються, а інші ні, то LRU може видалити весь хеш, якщо до нього рідко звертаються. Це призводить до втрати потрібних даних і додаткових витрат на їх відновлення.

Least Frequently Used видаляє з кешу найменш часто використовувані елементи; як і LRU, LFU працює на рівні ключів і не враховує частоту доступу до окремих елементів складних структур даних [24]. Якщо деякі поля хешу активно використовуються, а загальна частота доступу до хешу низька, то алгоритм може видалити весь хеш. Такі дії призводять до тієї ж проблеми, що у випадку з LRU. Фактичні дані губляться, що призводить до їх повторного завантажування та обчислювання.

Загалом, використання стандартних механізмів управління кешем при роботі зі складними структурами даних може призвести до наступних негативних наслідків:

- старі та неактуальні дані можуть залишатися в пам’яті та займати місце, яке могло б бути використане для зберігання актуальної інформації;
- часте видалення та повторне створення великих структур даних збільшує навантаження на систему та призводить до збільшення часу відповіді;
- видалення актуальних даних через недосконалість політик витіснення може призвести до неконсистентності даних та негативно вплинути на користувацький досвід;
- при зростанні обсягів даних та кількості користувачів обмеження пам’яті та недостатня гнучкість політик витіснення стають більш помітними, що може обмежити можливості системи.

Недосконалість стандартних підходів також проявляється у неможливості тонкого налаштування під конкретні потреби додатків. Стандартні політики витіснення не враховують особливості доступу до даних у різних частинах

складних структур. Це може обмежити ефективність кешування та призвести до нераціонального використання ресурсів.

Таким чином, стандартний механізм TTL та алгоритми заміщення кешу LRU та LFU мають суттєві недоліки при роботі зі складними структурами даних у високопродуктивних системах. Для подолання цих проблем необхідно розробляти більш гнучкі та адаптивні механізми управління кешем, які враховують специфіку складних структур даних та характер доступу до них, забезпечуючи ефективне використання ресурсів та високу продуктивність системи.

4.3 Пропоновані поліпшення та нові підходи

4.3.1 Архітектурна концепція системи та інтеграція підходів

Теоретична концепція додатку базується на адаптивному підході до управління кешованими даними, який враховує динамічну природу запитів, різнорідність структур даних та необхідність ефективного використання пам’яті. Архітектура системи має забезпечувати інтеграцію з Redis для роботи зі складними структурами, такі як хеші, списки, множини, підтримуючи індивідуальне управління часом життя елементів через механізм TTL. Це дозволить гнучка адаптувати час зберігання кожного елемента залежно від частоти доступу та важливості даних.

Для зберігання та обробки складних структур використовується механізм серіалізації, який трансформує об’єкти у пари ключ-значення. Наприклад, {Name: "John", Address: {City: "NY"}} зберігається як object.Name = "John" та object.Address.City = "NY". При отриманні даних з Redis, передбачається їх розгортка у початкову структуру, забезпечуючи повноту та цілісність інформації для користувача.

Менеджер кешу та механізм TTL концептуально повинні взаємодіяти у тісній співпраці. Менеджер кешу здійснює постійний моніторинг ресурсів пам’яті, аналізує частоту та давність доступу до даних, визначаючи їх актуальність та необхідність у подальшому зберіганні. Механізм TTL забезпечує адаптивну зміну часу життя даних, орієнтуючись на частоту використання,

останній час доступу та інші параметри. Завдяки такій інтегрованій роботі забезпечується баланс між збереженням важливих даних і своєчасним видаленням непотрібної інформації.

Робота системи детально представлена на UML-діаграмі активностей, яка для зручності поділена на дві частини: фоновий процес очищення та основний потік обробки запитів.

На рисунку 4.4 представлений фоновий процес очищення даних. Він регулярно перевіряє, чи отримано сигнал на скасування роботи процесу. Якщо такого сигналу немає, здійснюється пошук прострочених ключів та властивостей у кеші. При виявленні прострочених елементів відбувається їх автоматичне видалення з подальшим оновленням лічильників вилучень для підтримки актуальної статистики. Після завершення кожної ітерації процес переходить в очікування на заданий інтервал часу перед наступною перевіркою. Такий механізм підтримує ефективне очищення пам’яті та запобігає її перевантаженню.

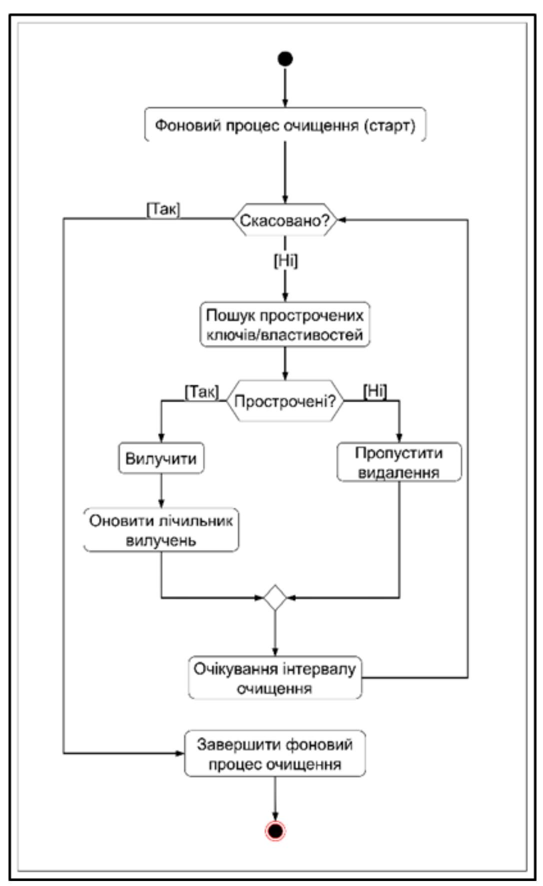


Рисунок 4.4 – UML-діаграма активності для фонового процесу очищення даних (рисунок виконаний самостійно)

На рисунку 4.5 представлений основний потік роботи системи з кешем, який відповідає за оперативну обробку запитів користувачів. Основне завдання цього потоку полягає у забезпеченні швидкого доступу до даних з одночасним контролем їх актуальності та оптимальним використанням пам’яті.

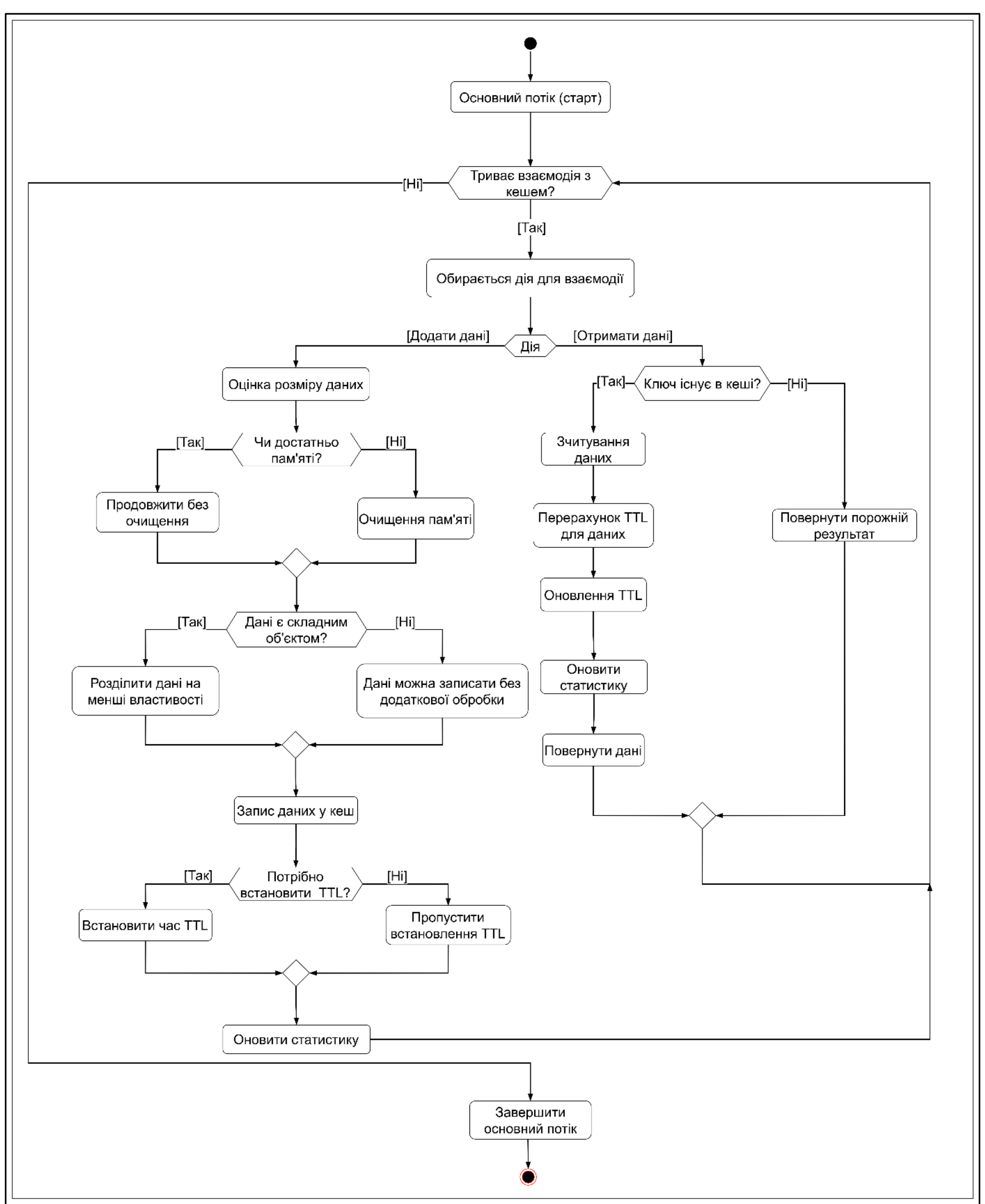


Рисунок 4.5 – UML-діаграма активності для основного процесу роботи з кешем (рисунок виконаний самостійно)

Основний потік передбачає постійну взаємодію з кешем, опрацьовуючи запити на читання або запис. При отриманні запиту на читання перевіряється наявність ключа в кеші, і якщо ключ знайдено, система витягує відповідні дані, перераховує та оновлює TTL, оновлює метрики доступу та повертає результат користувачу. У випадку відсутності ключа повертається порожній результат.

При запиті на запис даних система оцінює необхідний обсяг пам’яті. Якщо пам’яті недостатньо, запускається адаптивне очищення пам’яті з вилученням елементів з низькою актуальністю. Складні об’єкти додатково розбиваються на менші елементи з індивідуальним керуванням TTL, а прості дані записуються без додаткових дій. Після запису проводиться перевірка на необхідність встановлення TTL та оновлення відповідних статистичних показників.

Після розгляду послідовності операцій у системі, представленої на діаграмах активностей, доцільно перейти до огляду архітектурних компонентів, що забезпечують реалізацію описаних процесів.

На рисунку 4.6 наведена схема взаємодії компонентів системи адаптивного управління кешем.

Центральним компонентом є Adaptive Cache Client, що отримує запити на читання, запис або встановлення TTL від Client Application. Цей компонент відповідає за пряме звернення до Redis Client на основі StackExchange.Redis, здійснюючи базові операції з кешем та передачу інформації про запити для подальшого аналізу.

Adaptive Cache Manager аналізує запити на заміщення кешу, приймаючи рішення щодо видалення менш актуальних елементів. Він здійснює постійний моніторинг та оцінку стану кешу, спираючись на інформацію, зібрану компонентом Metrics Collector, який відповідає за накопичення статистичних даних про використання кешу.

Policy Manager, ґрунтуючись на аналізі метрик, періодично оновлює правила кешування, що дозволяє системі адаптивно реагувати на зміну характеру навантаження та структури даних. В результаті цього система забезпечує

ефективну роботу з пам’яттю та оптимальний рівень продуктивності, підтримуючи гнучке налаштування TTL та стратегії заміщення даних.

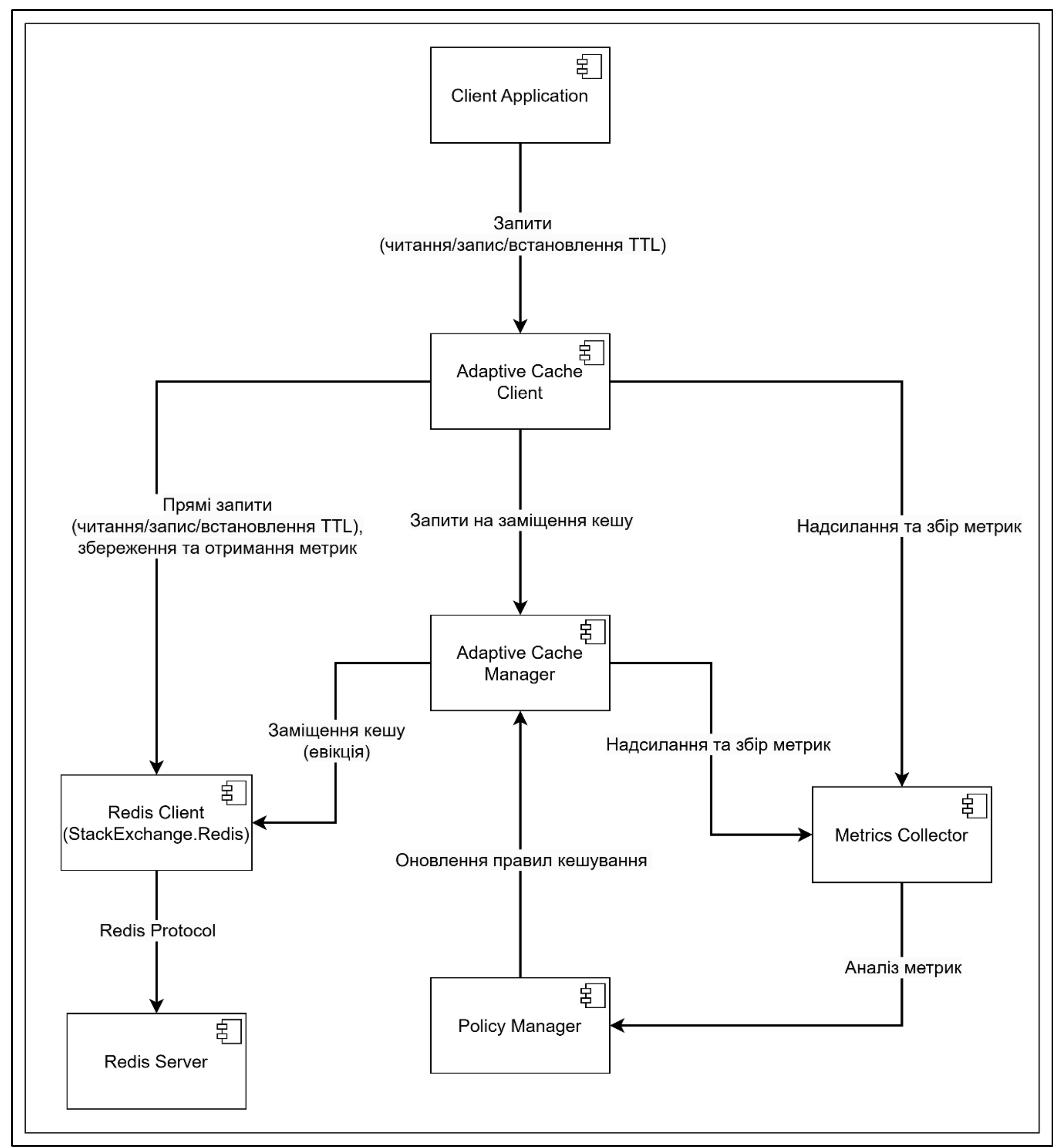


Рисунок 4.6 –Схема взаємодії компонентів системи адаптивного управління кешем (рисунок виконаний самостійно)

Таким чином, концептуальна архітектура системи, представлена двома частинами UML-діаграми активностей, забезпечує адаптивність, високу продуктивність та гнучкість, створюючи міцну основу для подальшої реалізації та практичного застосування розробленого рішення.

4.3.2 Адаптивне управління часом життя даних (TTL)

У класичному Redis час життя даних TTL визначається статично, що призводить до фіксованого терміну зберігання, незалежно від частоти доступу чи важливості даних. Це створює ризик видалення важливих даних або збереження рідко використовуваних ключів, що займають пам'ять. Адаптивний підхід до TTL динамічно визначає час життя для кожного елемента залежно від частоти доступу до нього. Це дозволяє оптимізувати використання пам'яті, забезпечуючи ефективне збереження важливих даних.

Для реалізації динамічного визначення TTL використовується багатоступеневий підхід, у якому кожен етап відповідає за певний аспект адаптивного управління часом життя. Цей підхід забезпечує точність розрахунків, а також гнучкість у налаштуванні параметрів кешу під конкретні задачі системи.

Першим етапом адаптивного підходу є визначення кількості звернень до кожного ключа в кеші. Однак у класичному підході використовується лише загальна кількість звернень до елемента, незалежно від того, коли ці звернення відбувалися. Це призводить до того, що ключ, який був популярним у минулому, може отримувати високий пріоритет навіть тоді, коли він вже втратив свою актуальність. Щоб усунути цю проблему, вводиться поняття звернення до ключа з експоненційним згасанням.

Експоненційне згасання враховує час, що минув з моменту останнього звернення до ключа, та поступово зменшує вагу старих звернень. Такий підхід дозволяє знизити вплив застарілих даних та зосередити ресурси на актуальних ключах. Розрахунок звернень до ключа з експоненційним згасанням наведений у формулі 4.1:

$$f_{decayed} = f * e^{-\lambda * (t_{current} - t_{last_access})} \quad (4.1)$$

де $f_{decayed}$ – кількість звернень до ключа з урахуванням експоненційного згасання,

f – кількість звернень до ключа за час його життя в кеші,

e – основа натурального логарифма (≈ 2.718),

λ – коефіцієнт згасання, що визначає швидкість зменшення «ваги» старих звернень ($\lambda \geq 0$),

$t_{current}$ – поточний час (секунди),

t_{last_access} – час останнього звернення до ключа (секунди).

Коефіцієнт затухання λ визначає, наскільки швидко старі звернення втрачають свій вплив. Вищі значення λ сприяють швидшому «старінню», тоді як нижчі значення забезпечують більш тривале збереження популярності ключа.

Після розрахунку скоригованої кількості доступів з урахуванням експоненційного згасання $f_{decayed}$, наступним етапом є нормалізація цього значення. Нормалізація необхідна для приведення кількості доступів до ключа до уніфікованого масштабу, що робить обчислення стабільними та забезпечує коректне врахування популярності ключа у фінальному розрахунку TTL. Без нормалізації значення $f_{decayed}$ може бути надто великим або малим, залежно від частоти доступу та активності ключа. Це ускладнює порівняння між різними ключами та може призвести до некоректних результатів у подальших обчисленнях.

Нормалізація дозволяє звести кількість доступів до діапазону [0,1], де:

- 0 означає ключ із мінімальною кількістю доступів або відсутністю звернень;
- 1 означає ключ із максимальною кількістю доступів у системі.

Цей підхід дозволяє пропорційно масштабувати час життя ключа, враховуючи його важливість, навіть за значних змін у популярності даних. Нормалізована кількість доступів розраховується за формулою 4.2:

$$f_{normalized} = \min\left(\frac{f_{decayed}}{f_{max}}, 1\right) \quad (4.2)$$

де $f_{normalized}$ – нормалізована кількість доступу до ключа ($0 \leq f_{normalized} \leq 1$),

f_{max} – максимальна кількість доступу серед ключів.

Формула 4.3 описує механізм розрахунку адаптивного часу життя ключа (TTL), що визначає тривалість його зберігання у пам'яті залежно від популярності. У цьому підході використовуються два основні параметри: базовий гарантований час життя T_{base} та максимальний час життя T_{max}. Базовий час життя T_{base} слід обирати з урахуванням типу даних та частоти доступу до них. У системах із високою динамікою даних T_{base} може бути нижчим, щоб забезпечити швидке звільнення пам'яті. Натомість у системах із довготривалими даними T_{base} слід встановлювати вищим, щоб уникнути передчасного видалення ключів. Максимальний час життя T_{max} забезпечує верхню межу для популярних ключів, але його значення також має бути збалансованим, щоб уникнути надмірного використання пам'яті найпопулярнішими даними. Наприклад, занадто високе значення T_{max} може призвести до збереження ключів, які втратили свою актуальність.

Час життя ключа розраховується пропорційно, перебуваючи між цими межами. Завдяки такому підходу популярні ключі отримують більше часу для зберігання, тоді як малопопулярні ключі видаляються швидше. Формула виглядає наступним чином:

$$TTL = T_{base} + (T_{max} - T_{base}) * f_{normalized} \quad (4.3)$$

де TTL – підсумковий адаптивний час життя ключа (секунди),

T_{base} – базовий гарантований час життя ключа (секунди),

T_{max} – максимальний базовий час життя (секунди).

Запропонований механізм дозволяє динамічно адаптувати TTL для кожного ключа, що знижує навантаження на пам'ять, підвищує продуктивність системи та забезпечує збереження актуальних даних у кеші. Нижче наведені основні переваги цього рішення:

– динамічне оновлення часу життя дозволяє зберігати важливі та часто використовувані дані в кеші довше;

– зменшення TTL для рідко використовуваних даних сприяє звільненню ресурсів для більш пріоритетних запитів;
– підвищення продуктивності, завдяки більш ефективному управлінню пам’яттю.

4.3.3 Адаптивний алгоритм заміщення кешу

Наступним запропонованим поліпшенням є адаптивний алгоритм заміщення кешу, який поєднує переваги концепцій алгоритмів LRU та LFU для ефективного управління даними в кеші. Він враховує три основні параметри: давність доступу, кількість звернень та обсяг пам’яті, який займає ключ. Це дозволяє адаптувати поведінку кешу до реальних умов використання, забезпечуючи баланс між продуктивністю та ефективністю використання ресурсів.

Для досягнення цієї мети використовуються кілька ключових етапів обчислення, які формалізовані у вигляді формул. На кожному етапі враховуються відповідні метрики та механізми адаптації, що забезпечують гнучкість системи.

Першим кроком є нормалізація основних метрик: давності доступу (a) та кількості звернень (f). Нормалізація виконується для приведення цих параметрів до спільного масштабу, що запобігає домінуванню однієї метрики над іншою через різницю в їхніх розмірностях або значеннях. Це забезпечує коректність багатокритеріального аналізу при подальших розрахунках.

Нормалізація кількості звернень (f) вже була розглянута в попередньому розділі (див. формулу 4.2). У цьому ж розділі буде розглянуто нормалізацію давності доступу (a), що дозволяє оцінювати давність у пропорційному масштабі.

Давність доступу (a) визначається як різниця між поточним часом та часом останнього доступу до ключа. Значення розраховується за формулою 4.4:

$$a = t_{current} - t_{last_access} \tag{4.4}$$

де a – давність доступу до ключа (секунди).

Цей параметр відображає, скільки часу минуло з моменту останнього використання ключа. Менші значення a вказують на більш «свіжі» ключі, які були використані нещодавно, тоді як більші значення характеризують ключі, які не використовувалися найдовше.

Оскільки значення a може змінюватися в широкому діапазоні залежно від активності системи та обсягу даних, необхідно виконати нормалізацію цього параметра. Нормалізація переводить абсолютні значення давності у відносний масштаб від 0 до 1, що дозволяє коректно порівнювати ключі незалежно від часових характеристик системи. Нормалізація здійснюється за формулою 4.5:

$$a_{normalized} = \min\left(\frac{a}{a_{max}}, 1\right) \tag{4.5}$$

де $a_{normalized}$ – нормалізована давність доступу до ключа ($0 \leq a_{normalized} \leq 1$),

a_{max} – максимальна давність доступу серед ключів (секунди).

Нормалізація давності доступу (a) переводить цей параметр у діапазон від 0 до 1, де кожне значення має чітку інтерпретацію:

- $a_{normalized} = 0$. Ключ є найактивнішим, оскільки його давність доступу (a) дорівнює 0 або близька до цього. Такий ключ був використаний нещодавно та має мінімальний пріоритет для видалення;
- $a_{normalized} = 1$. Ключ має максимальну давність доступу ($a \approx a_{max}$), тобто він не використовувався найдовше серед усіх ключів і має високий пріоритет для видалення.

Значення $a_{normalized}$ між 0 та 1 вказують на відносну давність доступу ключа: чим ближче значення до 1, тим довше ключ перебував у кеші без доступу, і тим вищий його пріоритет для видалення. Ця нормалізація дозволяє порівнювати ключі за давністю доступу незалежно від абсолютного масштабу часу (a) у системі.

На наступному етапі обчислюється комбінована метрика видалення (S), яка враховує три ключові параметри: нормалізовану давність доступу, нормалізовану

частоту доступу та обсяг пам’яті, який займає ключ. Ця метрика дозволяє виконувати багатокритеріальний аналіз та приймати обґрунтовані рішення щодо видалення даних із кешу. Використання комбінованого підходу забезпечує баланс між збереженням важливих даних та ефективним звільненням ресурсів пам’яті для нових записів. Розрахунок комбінованої метрики наведений у формулі 4.6:

$$S = w_a * \frac{1}{a_{normalized} + \varepsilon} + w_f * f_{normalized} + w_m * \frac{m_{max}}{m} \quad (4.6)$$

де S – оцінка видалення,

w_a – ваговий коефіцієнт давності доступу ($0 \leq w_a \leq 1$),

w_f – ваговий коефіцієнт частоти доступу ($0 \leq w_f \leq 1$),

w_m – ваговий коефіцієнт для пам’яті ($0 \leq w_m \leq 1$),

m – обсяг пам’яті, яку використовує ключ (байти),

m_{max} – максимальний обсяг пам’яті серед усіх ключів (байти),

ε – невелике значення, яке запобігає діленню на дуже малі значення.

Метрика S об’єднує три ключові параметри, які забезпечують багатокритеріальний підхід до вибору ключів для видалення. Перевернута нормалізована давність доступу ($\frac{1}{a_{normalized} + \varepsilon}$) використовується для того, щоб підвищити пріоритет видалення ключів, які давно не використовувалися. Інверсія цього параметра забезпечує, що ключі з малою давністю доступу, які використовувалися нещодавно, залишатимуться в кеші. Тоді як ключі з великою давністю доступу матимуть вищий пріоритет для видалення. Додавання невеликого значення ε гарантує стабільність розрахунків та запобігає діленню на нуль.

Нормалізована частота доступу $f_{normalized}$ додається у прямій пропорції, оскільки вона показує, наскільки популярний ключ. Більш популярні ключі отримують вищий пріоритет для збереження, що мінімізує негативний вплив на продуктивність системи при зверненні до кешу.

Обсяг пам'яті ($\frac{m_{max}}{m}$) враховується у перевернутому вигляді, оскільки ключі, що займають більше пам'яті мають вищий пріоритет для видалення. Чим більше значення m, тим меншим стає внесок цього параметра у загальну метрику S.

Такий підхід до використання інверсії у метриці S забезпечує правильну інтерпретацію параметрів: більші значення давності доступу та обсягу пам'яті знижують важливість ключа для збереження, тоді як частота доступу, обчислена у прямій пропорції, підвищує її.

Сума вагових коефіцієнтів повинна постійно дорівнювати 1 (див. формулу 4.7). Це необхідно для збереження балансу впливу кожної метрики на загальну оцінку S та забезпечення коректної інтерпретації результатів. Якщо сума вагових коефіцієнтів не дорівнює 1, це може призвести до непропорційного впливу однієї з метрик та некоректної роботи алгоритму.

$$w_a + w_f + w_m = 1 \tag{4.7}$$

Вагові коефіцієнти w_a, w_f, w_m встановлюються залежно від пріоритетів системи. Наприклад, у системах, де важливість даних визначається їхньою актуальністю, пріоритет надається w_a. Якщо ж важлива популярність даних, більша вага надається w_f. Значення коефіцієнтів повинні бути підібрані експериментально для конкретних сценаріїв використання. Наприклад, для систем із високою частотою запитів, але низькими вимогами до актуальності, можна встановити вагові коефіцієнти як $w_a = 0.2, w_f = 0.7, w_m = 0.1$. У системах, де важлива свіжа інформація, перевага надається w_a, наприклад $w_a = 0.6, w_f = 0.3, w_m = 0.1$

Для визначення, чи слід видалити ключ із кешу, використовується оцінка видалення S. Ця оцінка дозволяє визначати значущість кожного ключа, враховуючи такі параметри, як давність доступу, частота використання та обсяг пам'яті, який займає об'єкт. Видалення елементів із кешу виконується у двох ключових сценаріях. Перший сценарій передбачає видалення ключів, якщо

використання пам’яті перевищує встановлений поріг, наприклад, 70% від загального доступного обсягу пам’яті. У цьому випадку необхідно визначити ключі, які займають найменш пріоритетну позицію за значенням S, та видалити їх до того моменту, поки використання пам’яті не повернеться до рівня, що не перевищує порогу. Порогове значення використання пам’яті визначається за формулою 4.8:

$$M_{threshold} = P_{memory} * M_{max} \tag{4.8}$$

де $M_{threshold}$ – порогове значення використання пам’яті, при перевищенні якого запускається очищення кешу,

P_{memory} – частка максимально допустимого використання пам’яті ($0 < P_{memory} < 1$),

M_{max} – максимальний обсяг пам’яті кешу, визначений системою.

Наприклад, якщо кеш має загальний обсяг пам’яті $M_{max} = 100\ MB$, поріг використання пам’яті становить $P_{memory} = 0.7$, тобто $M_{threshold} = 70\ MB$. Якщо пам’ять перевищує цей поріг, наприклад, $M = 80\ MB$, необхідно почати видаляти ключі з найменшими значеннями S, доки використання пам’яті M не стане меншим за $M_{threshold}$.

Другий сценарій видалення передбачає очищення пам’яті під час спроби додавання нового об’єкта до кешу. Якщо пам’яті недостатньо для його розміщення, необхідно автоматично видаляти інші елементи з низьким значенням S, звільняючи місце для нового запису. У цьому випадку видалення триває доти, поки загальний обсяг пам’яті не стане достатнім для розміщення нового елемента.

Метрика S забезпечує баланс між ефективністю використання пам’яті та збереженням важливих даних, оскільки враховує одразу кілька факторів: давність доступу, частоту використання та розмір об’єкта. Видаляються лише ті елементи, які мають найменший пріоритет, що дозволяє зберігати найбільш потрібні та запитувані дані. Такий підхід забезпечує адаптивність та ефективне управління кешем у різноманітних умовах роботи.

Таким чином, запропонований адаптивний алгоритм заміщення кешу дозволяє ефективно управляти даними, враховуючи три ключові параметри: давність доступу, частоту звернень та обсяг пам’яті, який займає кожен ключ. Нормалізація метрик забезпечує узгодженість розрахунків незалежно від різниці у розмірностях параметрів. Комбінована метрика S, яка враховує давність доступу, частоту використання та обсяг пам’яті, забезпечує гнучке та об’єктивне визначення елементів для видалення. Це дозволяє досягти балансу між збереженням найбільш важливих даних та звільненням пам’яті для нових записів. Запропоновані два сценарії видалення забезпечують адаптивність алгоритму. Видалення елементів при перевищенні порогового значення пам’яті дозволяє зберігати стабільність та ефективність роботи кешу, тоді як очищення пам’яті під час додавання нового об’єкта запобігає переповненню та гарантує простір для нових даних. Універсальність алгоритму досягається за рахунок можливості налаштування вагових коефіцієнтів w_a, w_f, w_m та параметра P_{memory}, що дозволяє адаптувати його до різних сценаріїв використання. Такий підхід забезпечує не лише високу продуктивність, але й ефективне використання ресурсів пам’яті, що робить його придатним для систем із різними вимогами до управління кешем.

5 ОПИС ЕКСПЕРИМЕНТАЛЬНОГО ДОСЛІДЖЕННЯ

5.1 Розробка програмного забезпечення

У рамках даного дослідження було розроблено програмне забезпечення, яке спрямоване на підвищення ефективності управління кешованими даними в Redis. Головна мета розробки полягає у впровадженні динамічного управління часом життя даних (TTL) та адаптивної оптимізації кешу, що базується на алгоритмах LRU та LFU. Такий підхід дозволяє не лише оптимізувати використання пам’яті, але й зберігати стабільну продуктивність системи навіть за умов високих навантажень. Для досягнення цих цілей програмне забезпечення було реалізовано на платформі .NET із використанням мови програмування C# та бібліотеки StackExchange.Redis, що забезпечує зручну інтеграцію з Redis.

Ключовим компонентом програмного забезпечення є механізм динамічного управління TTL, який дозволяє задавати індивідуальні значення часу життя для кожного елемента складних структур даних, таких як хеші, списки та множини. Це стало можливим завдяки використанню динамічного перерахунку TTL на основі частоти доступу до даних, часу останнього звернення та інших важливих параметрів. Методика дозволяє уникати видалення активних елементів, зберігаючи їх у пам’яті для подальшого використання. Наприклад, метод StoreObjectAsync забезпечує збереження складних об’єктів у вигляді хеш-структур, що спрощує їх подальшу обробку (див. рис. 5.1). Під час цього процесу об’єкти серіалізуються у вигляді пар «ключ-значення», що дозволяє ефективно використовувати Redis для роботи з різнорідними даними. Окрім цього, система підтримує автоматичне оновлення метрик доступу, що є основою для адаптивного обчислення TTL. Це забезпечується викликами методу UpdateKeyAccessAsync, який оновлює частоту доступу, зберігає час останнього звернення до ключа та забезпечує адаптивне коригування TTL. Якщо виявлено, що ключ використовується часто, система збільшує його час життя, запобігаючи передчасному видаленню. Водночас метод GetObjectAsync дозволяє отримувати та відновлювати об’єкти з Redis у їх початковій формі, підтримуючи актуальність метрик доступу до даних (див. рис. 5.1). Завдяки цьому підходу забезпечується

збереження гнучкості та масштабованості при роботі з великими обсягами даних, що є важливим для сучасних високонавантажених систем.

```
10 references | mgtippet33, 18 days ago | 2 authors, 6 changes
public async Task StoreObjectAsync<T>(string key, T obj)
{
    await HandleInsertWithMemoryManagementAsync(key, obj, async () =>
    {
        var hashEntries = ObjectSerializer.FlattenObject(obj)
            .Select(kvp => new HashEntry(kvp.Key, kvp.Value))
            .ToArray();

        await _database.HashSetAsync(key, hashEntries);
        await UpdateKeyAccessAsync(key, false);
    });
}

8 references | mgtippet33, 18 days ago | 1 author, 4 changes
public async Task<T> GetObjectAsync<T>(string key) where T : new()
{
    var hashEntries = await _database.HashGetAllAsync(key);
    if (hashEntries.Length == 0)
        return default;

    var properties = hashEntries.ToDictionary(entry => (string)entry.Name, entry => (string)entry.Value);

    var obj = ObjectSerializer.UnflattenObject<T>(properties);

    await UpdateKeyAccessAsync(key, true);

    return obj;
}
```

Рисунок 5.1 – Методи для серіалізації та десеріалізації складних об’єктів (рисунок виконаний самостійно)

Метод SetPropertyTTLAsync дозволяє встановлювати індивідуальні значення часу життя для властивостей об’єктів, що є ключовим для гнучкого управління складними структурами даних у Redis. Цей метод використовує впорядковану множину для збереження інформації про властивості та їхній час завершення, що дозволяє швидко ідентифікувати елементи з простроченим TTL. Під час виконання методу розраховується час завершення TTL на основі поточного часу та заданого TTL, після чого властивість додається до множини за допомогою команди SortedSetAddAsync. Додатково, метод забезпечує механізм автоматичного оновлення часу життя властивостей у випадку повторного доступу, що дозволяє динамічно подовжувати збереження важливих елементів.

Реалізація методу оптимізована для ефективного управління ресурсами навіть у високонавантажених системах. Завдяки використанню асинхронної обробки та багатопотокового виконання, метод мінімізує затримки у доступі до

даних і забезпечує швидке оновлення TTL без значного навантаження на систему. Крім того, механізм управління TTL у SetPropertyTTLAsync інтегрується з глобальними метриками доступу, що дозволяє оптимізувати розподіл пам’яті та уникати перевантаження кешу неактуальними даними. Реалізація методу представлена на рисунку 5.2.

```
public async Task SetPropertyTTLAsync(string key, string propertyPath, TimeSpan ttl)
{
    var sortedSetKey = key + Constants.TtlSortedSetSuffix;
    var now = DateTimeOffset.UtcNow.ToUnixTimeSeconds();
    var expiryTime = now + (long)ttl.TotalSeconds;

    await _database.SortedSetAddAsync(sortedSetKey, propertyPath, expiryTime);

    Log($"TTL for property '{propertyPath}' is set to {ttl.TotalSeconds} seconds.");
}
```

Рисунок 5.2 – Метод SetPropertyTTLAsync для встановлення індивідуального значення часу життя (рисунок виконаний самостійно)

Обчислення динамічного TTL реалізоване у методі ComputeAdaptiveTTLAsync, який враховує нормалізовану частоту доступу, використовуючи масштабуючий коефіцієнт. Цей коефіцієнт періодично оновлюється для відображення актуальних змін у навантаженні. Завдяки цьому TTL може автоматично збільшуватися для часто використовуваних даних, тоді як менш значущі елементи отримують коротший час життя. Метод виконує експоненційне згасання ваги частоти доступу, що дозволяє забезпечити адаптивне регулювання часом життя даних відповідно до змін у патернах доступу. Під час обчислення TTL враховуються глобальні метрики доступу до даних, такі як максимальна частота звернень та мінімальний час останнього доступу. Використання цих параметрів у поєднанні з експоненційним згасанням дозволяє системі динамічно пристосовуватися до навантаження та уникати передчасного видалення важливих об’єктів. Крім того, метод ComputeAdaptiveTTLAsync підтримує захист даних від видалення у разі їх активного використання, що забезпечує додаткову надійність у збереженні критично важливих кешованих записів. Реалізація методу представлена на рисунку 5.3.

```
private async Task<TimeSpan> ComputeAdaptiveTTLAsync(string metaKey, string accessKey)
{
    string prefix = (accessKey == "key") ? "" : (accessKey + ":");
    string totalAccessesField = $"{prefix}totalAccesses";
    string lastAccessField = $"{prefix}lastAccess";

    var totalAccessesValue = await _database.HashGetAsync(metaKey, totalAccessesField);
    var lastAccessValue = await _database.HashGetAsync(metaKey, lastAccessField);

    long f = totalAccessesValue.IsNullOrEmpty ? 0 : (long)totalAccessesValue;
    long tLastAccess = lastAccessValue.IsNullOrEmpty
        ? DateTimeOffset.UtcNow.ToUnixTimeSeconds()
        : (long)lastAccessValue;

    double now = DateTimeOffset.UtcNow.ToUnixTimeSeconds();
    double f_decayed = f * Math.Exp(-_options.DecayLambda * (now - tLastAccess));

    double maxTotalAccesses = 0;
    var totalAccessSetRange = await _database.SortedSetRangeByRankWithScoresAsync(
        Constants.GlobalFrequencySetKey,
        0,
        0,
        Order.Descending);

    if (totalAccessSetRange != null && totalAccessSetRange.Length > 0)
    {
        maxTotalAccesses = totalAccessSetRange[0].Score;
    }

    double f_normalized = (maxTotalAccesses > 0)
        ? Math.Min(f_decayed / maxTotalAccesses, 1.0)
        : 0.0;

    var ttlSec = _options.BaseTTL + (_options.MaxTTL - _options.BaseTTL) * f_normalized;
    return ttlSec;
}
```

Рисунок 5.3 – Метод ComputeAdaptiveTTLAsync для обчислення динамічного TTL (рисунок виконаний самостійно)

Реалізація механізму автоматичного очищення прострочених властивостей здійснюється через метод CleanupExpiredPropertiesAsync, який періодично перевіряє впорядковані множини TTL та видаляє дані, що втратили актуальність. Метод виконується у строго послідовному режимі, що гарантує уникнення конкурентного доступу до даних та забезпечує стабільність роботи. Процес перевірки здійснюється у кілька етапів. Спочатку отримується список ключів, що містять TTL, далі зчитуються прострочені властивості та після цього виконується їхнє безпосереднє видалення. У випадку великої кількості ключів система може виконувати очищення поступово, оптимізуючи навантаження на Redis. Реалізація методу представлена на рисунку 5.4.

```
private async Task CleanupExpiredPropertiesAsync()
{
    await _cleanupSemaphore.WaitAsync();
    try
    {
        var server = _redis.GetServer(_redis.GetEndPoints()[0]);
        var keys = server.Keys(pattern: "*" + Constants.TtlSortedSetSuffix);

        var now = DateTimeOffset.UtcNow.ToUnixTimeSeconds();

        foreach (var key in keys)
        {
            var dataKey = key.ToString().Replace(Constants.TtlSortedSetSuffix, "");
            var expiredProperties = await _database.SortedSetRangeByScoreAsync(key, 0, now);

            if (expiredProperties.Length > 0)
            {
                foreach (var propertyPath in expiredProperties)
                {
                    Log($"The '{propertyPath}' property for the key '{dataKey}' was deleted after the TTL expired.");

                    await RemovePropertyAndSubPropertiesAsync(dataKey, key, propertyPath);
                    //ExpiredKeys++;
                    Interlocked.Increment(ref _expiredKeys);
                }
            }
        }
    }
    catch (Exception ex)
    {
        Log("Error during CleanupExpiredProperties: " + ex.Message);
    }
    finally
    {
        _cleanupSemaphore.Release();
    }
}
```

Рисунок 5.4 – Метод CleanupExpiredPropertiesAsync для очищення прострочених властивостей (рисунок виконаний самостійно)

Механізм адаптивного менеджеру кешу реалізує декілька основних методів. Метод ComputeEvictionScore використовується для розрахунку балу видалення кожного ключа в кеші на основі віку даних, частоти доступу та використаної пам'яті. Розрахунок здійснюється через вагові коефіцієнти, які адаптивно налаштовуються для знаходження оптимального балансу між цими параметрами. Ключі з найнижчим балом видаляються першими у разі необхідності звільнення пам’яті. Метод використовує нормалізовані значення віку, частоти та пам’яті, що забезпечує стабільний розподіл ресурсів у кеші та підвищує ефективність роботи системи. Реалізація методу наведена на рисунку 5.5.

Метод CleanupForRequiredMemoryAsync аналізує стан пам’яті Redis та ініціює очищення кешу, якщо використана пам’ять перевищує допустимий рівень. Для цього він визначає необхідний обсяг пам’яті для звільнення, отримує список

ключів з найнижчим балом значущості та поступово їх видаляє, поки не досягне безпечного рівня використання ресурсів.

```
private double ComputeEvictionScore(double age, double frequency, long memorySize, double aMax, double fMax, long mMax)
{
    double aNormalized = 0.0;
    if (aMax > 0)
    {
        aNormalized = Math.Min(age / aMax, 1.0);
    }

    double fNormalized = 0.0;
    if (fMax > 0)
    {
        fNormalized = Math.Min(frequency / fMax, 1.0);
    }

    double memInverted = (mMax > 0 && memorySize > 0)
        ? (mMax / (double)memorySize)
        : 1.0;

    double sAge = _options.AgeWeight * (1.0 / (aNormalized + EPSILON));
    double sFreq = _options.FrequencyWeight * fNormalized;
    double sMem = _options.MemoryWeight * memInverted;

    double S = sAge + sFreq + sMem;
    return Math.Max(S, 0.0);
}
```

Рисунок 5.5 – Метод ComputeEvictionScore для розрахунку балу видалення (рисунок виконаний самостійно)

Метод також контролює, щоб очищення не було надмірним, підтримуючи баланс між продуктивністю та доступністю даних. Крім того, під час виконання очищення він аналізує частоту доступу до кожного ключа та розподіляє навантаження на основі адаптивного коефіцієнта важливості. Використання багатопотокового підходу дозволяє паралельно перевіряти стан пам’яті та обробляти кандидати на видалення, що значно підвищує швидкість виконання операції. У разі якщо очищення не вдається виконати повністю, метод логікує залишковий дефіцит пам’яті та може повторно викликати себе із скоригованими параметрами евікції. Реалізація методу наведена на рисунку 5.6.

```
public async Task CleanupForRequiredMemoryAsync(long requiredSize)
{
    var server = _redis.GetServer(_redis.GetEndPoints()[0]);
    long usedMemory = await GetUsedMemoryAsync(server);
    long maxMemory = await GetMaxMemoryAsync() ?? _options.MaxCacheMemory;
    long maxAllowedMemory = CalculateMaxAllowedMemory(maxMemory);

    if (usedMemory + requiredSize <= maxAllowedMemory)
    {
        return;
    }

    long memoryToFree = (usedMemory + requiredSize) - maxAllowedMemory;
    long freedSoFar = await FreeMemoryAsync(memoryToFree);

    LogMemoryStatus(freedSoFar, requiredSize, memoryToFree);
}
```

Рисунок 5.6 – Метод CleanupForRequiredMemoryAsync для видалення ключа при перевищенні допустимого рівня (рисунок виконаний самостійно)

Метод CleanupExpiredKeysAsync відповідає за автоматичне очищення прострочених ключів. Він періодично перевіряє всі ключі, що мають асоційовані TTL, та визначає, які з них перевищили свій термін життя. Якщо ключ прострочений, метод видаляє його з кешу, звільняючи пам'ять для нових даних. Очищення виконується за допомогою ітеративного аналізу впорядкованих множин, що містять інформацію про TTL ключів. Метод використовує семафори для запобігання одночасному виконанню кількох потоків очищення, що дозволяє підтримувати стабільну продуктивність системи навіть у високонавантажених середовищах. Реалізація методу наведена на рисунку 5.7.

```
private async Task CleanupExpiredKeysAsync()
{
    await _cleanupSemaphore.WaitAsync();
    try
    {
        var server = _redis.GetServer(_redis.GetEndPoints()[0]);
        long usedMemory = await GetUsedMemoryAsync(server);
        long maxMemory = await GetMaxMemoryAsync() ?? _options.MaxCacheMemory;
        long memoryThreshold = (long)(_options.PMemoryThreshold * maxMemory);
        if (usedMemory <= memoryThreshold) return;

        long memoryToFree = usedMemory - memoryThreshold;
        long freedSoFar = await FreeMemoryAsync(memoryToFree);
        LogMemoryStatus(freedSoFar, 0, memoryToFree);
    }
    finally
    {
        _cleanupSemaphore.Release();
    }
}
```

Рисунок 5.7 – Метод CleanupExpiredKeysAsync для автоматичного очищення прострочених ключів (рисунок виконаний самостійно)

Метод UpdateEvictionScoreAsync оновлює значення балу видалення для кожного ключа в глобальному впорядкованому списку Redis. Він аналізує останній час доступу до ключа, частоту звернень та обсяг використовуваної пам'яті, нормалізує ці показники та оновлює значення у відповідному множині. Це дозволяє системі оперативно адаптуватися до змін у навантаженні та правильно розподіляти пріоритетність збереження даних у кеші. Реалізація методу наведена на рисунку 5.8.

```
public async Task UpdateEvictionScoreAsync(string dataKey)
{
    var metaKey = (RedisKey)(dataKey + Constants.TtlHashSuffix);
    var tracker = await GetObjectAccessTrackerAsync(metaKey);

    double now = DateTimeOffset.UtcNow.ToUnixTimeSeconds();

    double freqGeneral = 0.0;
    double ageGeneral = 0.0;
    if (tracker.Frequency > 0)
    {
        double t = now - tracker.LastFrequencyUpdateTime;
        freqGeneral = tracker.Frequency * Math.Exp(-_options.DecayLambda * t);
        ageGeneral = now - tracker.LastAccess;
    }

    var memoryUsageResult = await _database.ExecuteAsync("MEMORY", "USAGE", dataKey);
    if (!long.TryParse(memoryUsageResult.ToString(), out var memorySize) || memorySize <= 0)
    {
        memorySize = 1;
    }

    double fMax = await GetFMaxAsync();
    double aMax = await GetAMaxAsync();
    long mMax = _options.MaxCacheMemory;

    double sKey = ComputeEvictionScore(ageGeneral, freqGeneral, memorySize, aMax, fMax, mMax);

    lock (_updateEvictionScoreLock)
    {
        _database.SortedSetAdd(Constants.GlobalEvictionSetKey, dataKey, sKey);
    }

    int propCount = tracker.PropertyAccessState.Count;
    if (propCount > 0)
    {
        double propertyMemoryShare = memorySize / (propCount + 1.0);

        foreach (var kvp in tracker.PropertyAccessState)
        {
            string propertyName = kvp.Key;
            var propTracker = kvp.Value;

            double freqProp = 0.0;
            double ageProp = 0.0;
            if (propTracker.Frequency > 0)
            {
                double tProp = now - propTracker.LastFrequencyUpdateTime;
                freqProp = propTracker.Frequency * Math.Exp(-_options.DecayLambda * tProp);
                ageProp = now - propTracker.LastAccess;
            }

            double sProp = ComputeEvictionScore(ageProp, freqProp,
                                                (long)propertyMemoryShare,
                                                aMax, fMax, mMax);

            string zsetMember = $"{dataKey}::{propertyName}";

            lock (_updateEvictionScoreLock)
            {
                _database.SortedSetAdd(Constants.GlobalEvictionSetKey, zsetMember, sProp);
            }
        }
    }
}
```

Рисунок 5.8 – Метод UpdateEvictionScoreAsync для оновлення балу видалення (рисунок виконаний самостійно)

Таким чином, у процесі розробки програмного забезпечення були реалізовані всі ключові аспекти, які необхідні для забезпечення стабільної та ефективної роботи системи. Впроваджені механізми динамічного управління кешем та часу життя даних, а також адаптивна оптимізація кешу гарантують не лише коректність обробки даних, але й оптимальне використання ресурсів пам’яті. Завдяки інтеграції з Redis та використанню сучасних підходів до управління даними система забезпечує високу продуктивність, актуальність кешу

та гнучкість в умовах змінюваних навантажень, що робить її потужним рішенням для сучасних розподілених систем.

5.2 Тестування програмного забезпечення

Тестування програмного забезпечення є невід'ємною частиною життєвого циклу розробки, спрямованою на перевірку якості та функціональності системи. Воно дозволяє виявляти та усувати помилки, перевіряти виконання очікуваних сценаріїв роботи, забезпечувати стабільність та надійність програмного продукту. Зокрема, тестування дозволяє оцінити поведінку системи в реальних умовах експлуатації, перевірити її адаптивність до змінного навантаження та переконатися у відсутності критичних уразливостей.

У процесі реалізації практичної частини дослідження тестування було організовано таким чином, щоб охопити всі ключові функції розробленої системи, забезпечити перевірку коректної інтеграції між її компонентами та оцінити продуктивність запропонованих механізмів управління кешем. Для цього використовувалися як модульні, так і інтеграційні тести, що дало змогу оцінити як роботу окремих компонентів, так і їхню взаємодію в цілісній системі. Окрему увагу приділили навантажувальному тестуванню для оцінки ефективності роботи кешу в умовах високої інтенсивності запитів та значного обсягу збережених даних.

Основний акцент у тестуванні був зроблений на перевірці функціональності адаптивного клієнта кешу, який забезпечує динамічний TTL, управління метаданими та взаємодію з Redis. Було розроблено серію тестових кейсів, які охоплювали різні сценарії роботи, включаючи збереження та читання об'єктів, оновлення часу життя даних, захист ключів, очищення прострочених елементів, а також взаємодію з різними структурами даних Redis, такими як хеші, списки, множини та впорядковані множини. Особливу увагу приділили перевірці коректного застосування TTL до складних структур, щоб підтвердити відповідність результатів очікуваним характеристикам продуктивності та ефективності використання пам'яті.

Крім того, було протестовано механізм автоматичного очищення даних, термін дії яких завершився. Це перевірялося через різні сценарії, включаючи ситуації, коли ключі повинні автоматично видалятися після закінчення їх TTL, а також сценарії, де захищені ключі повинні залишатися доступними навіть після завершення часу життя. Тестування підтвердило, що такі ключі залишаються у сховищі, а їхній статус правильно відображається у метаданих. Окрім цього, була проведена перевірка очищення залишкових метаданих, що дозволило усунути потенційні проблеми, пов'язані з накопиченням непотрібної інформації у великих сховищах даних.

Окрім клієнта кешу, було протестовано роботу адаптивного менеджера кешу, який відповідає за автоматичне очищення застарілих або неактуальних даних, контроль використання пам'яті та адаптивне налаштування параметрів кешу на основі вагових коефіцієнтів, таких як частота доступу, вік даних, обсяг пам'яті. У процесі тестування менеджера кешу було перевірено його ефективність у видаленні прострочених ключів, забезпечення збереження часто використовуваних даних, а також захист критично важливих ключів. Зокрема, підтверджено, що менеджер коректно видаляє прострочені ключі, оновлює eviction бал у реальному часі та забезпечує балансування пам'яті без значних втрат продуктивності.

Результати тестування були автоматизовані та представлені за допомогою Test Explorer, інтегрованого у Visual Studio 2022. Усі тести виконувалися на локальній тестовій базі Redis, розгорнутій у середовищі Docker, що дозволило імітувати реальні умови експлуатації та забезпечити ізоляцію тестового середовища від зовнішніх факторів. Використання Docker дозволило гнучко змінювати конфігурацію Redis для тестування різних сценаріїв, таких як обмеження ресурсів пам'яті або зміна політики видалення даних. Результати виконання тестів у Test Explorer можна побачити на рисунку 5.9.

Таким чином, результати тестування підтвердили коректність роботи основних компонентів системи керування кешем.

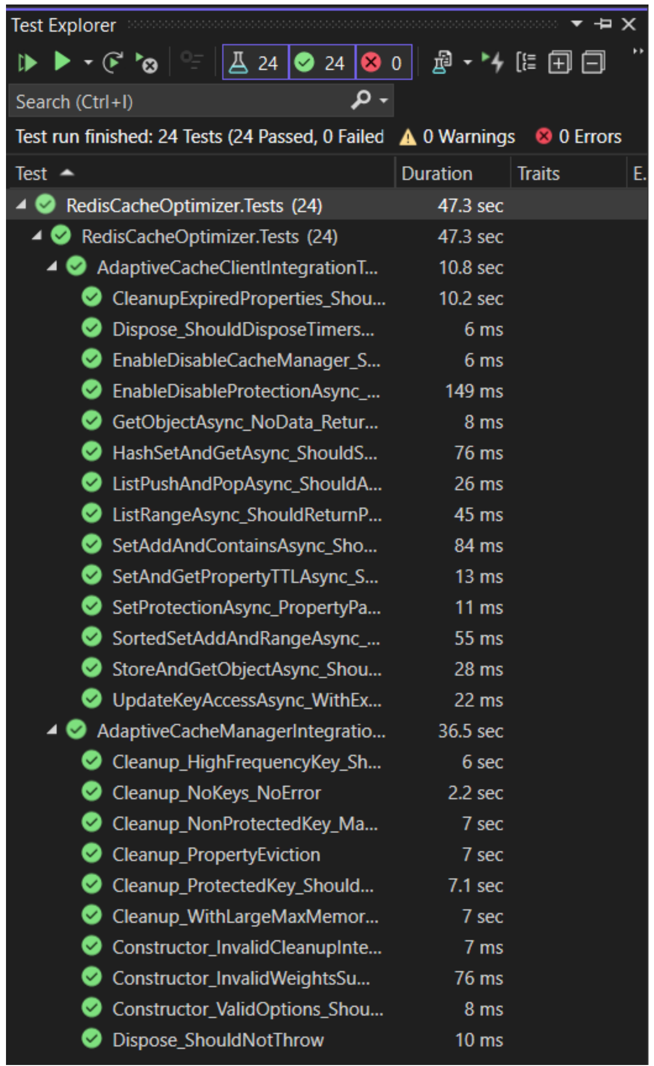


Рисунок 5.9 – Результати тестування програмного забезпечення (рисунок виконаний самостійно)

Зокрема, перевірено, що динамічний TTL ефективно адаптується до змінної частоти доступу до даних, забезпечуючи оптимальне використання пам'яті, а адаптивний менеджер кешу коректно виконує автоматичне очищення неактуальних даних, балансування навантаження та захист критично важливих ключів.

5.3 Проведення експерименту

Для проведення експериментального дослідження було створено спеціальне тестове середовище, яке максимально точно відтворює реальні умови роботи систем із високим навантаженням на кеш Redis. Експеримент виконувався на

пристрої з наступними характеристиками: процесор Intel Core i7-12700H (2.70 ГГц), 32 ГБ оперативної пам'яті та швидкий накопичувач SSD, що забезпечують достатню обчислювальну потужність та мінімальні затримки при доступі до даних. Програмне забезпечення розроблено на платформі .NET із застосуванням мови програмування C# та бібліотеки StackExchange.Redis, що дозволяє зручно інтегруватися з сервером Redis. Тестове середовище розгорнуто у контейнерах Docker, що гарантує ізоляцію експериментальних баз даних та можливість точного налаштування параметрів, таких як максимальний обсяг пам'яті, політики заміщення (allkeys-lru, allkeys-lfu, allkeys-random або noeviction). Для обмеження використання пам'яті застосовувався параметр maxmemory, який динамічно розраховувався залежно від обсягу даних. Середній розмір об'єкта, близько 3 КБ, множився на кількість елементів у датасеті, після чого отримане значення зменшувалося до 90 %, щоб алгоритми витіснення активізувалися при наближенні кешу до заповнення.

Вхідними даними для експерименту були синтетично згенеровані набори даних, які імітували складні об'єкти. Генерація даних здійснювалася за допомогою спеціального класу, де кожен об'єкт містив поля Id, Name, Timestamp та великий рядок (Payload) з приблизно 3000 символів. Ключі створювалися циклічно у форматі «key:{номер}», а значення об'єктів формувалися із використанням поточного часу, випадкового числа та константного текстового рядка для Payload. Для кожного запису встановлювався початковий час життя (TTL) із випадковим значенням від 60 до 120 секунд, що дозволяло змоделювати як рівномірний розподіл звернень, так і сценарії з концентрацією «гарячих» ключів. При моделюванні доступу до кешу перші 20% ключів вважалися «гарячими», а решта – «холодними». За допомогою псевдовипадкового вибору здійснювався доступ до «гарячих» ключів з ймовірністю 80% і до «холодних» з ймовірністю 20%. Окрім цього, проводилися експерименти із змінним співвідношенням доступу (наприклад, 80:20 та 50:50), щоб оцінити вплив патерну запитів на ефективність роботи системи.

На етапі підготовки експерименту було розроблено набір методів, що забезпечують повну ініціалізацію та налаштування системи для проведення тестування. Спочатку база даних заповнювалася синтетично згенерованими об’єктами, який призначав кожному ключу індивідуальне значення TTL. Для обробки великої кількості записів використовувалася пакетна обробка, що гарантувало послідовне виконання операцій із контролем рівня паралелізму. Для моделювання реалістичного навантаження застосовувалися власні скрипти, що реалізовували псевдовипадковий доступ до «гарячих» та «холодних» ключів, а також для перевірки реакції системи на переповнення кешу та активації механізмів видалення даних згідно з встановленою політикою заміщення.

Кожен тестовий сценарій проводився тричі, після чого результати усереднювалися з розрахунком стандартного відхилення для підвищення достовірності отриманих даних. Співвідношення операцій читання до запису не було рівномірним: приблизно 80% звернень спрямовувалися до «гарячих» ключів, що забезпечувало високу ймовірність попадання в кеш, а решта 20% – до «холодних» ключів, де частіше виникали кеш-промахи та наступне повторне записування даних. У деяких експериментах проводилися варіації цього співвідношення (наприклад, 50:50), що дозволяло дослідити вплив патерну доступу на продуктивність системи.

Під час експериментів було здійснено комплексний збір метрик, що дозволило детально оцінити ефективність роботи системи кешування. До основних показників відносилися показники Cache Hit Rate та Cache Miss Rate, які відображають відсоткове співвідношення успішних звернень до кешу до загальної кількості запитів, що дає змогу оцінити якість збереження та відновлення даних. Окрім цього, вимірювався середній час виконання операцій запису та читання. Ці дані використовувалися для порівняння продуктивності різних стратегій роботи з кешем при обробці великої кількості операцій.

Також здійснювався збір даних про залишковий час життя даних (TTL) у кеші. Для цього визначалися середній, мінімальний та максимальний залишкові TTL, що дозволяло аналізувати, наскільки адаптивно система регулює час життя

записів та зберігає актуальні дані. Окрім цього, відслідковувалася кількість видалених записів (Eviction Count), що свідчило про частоту активації механізмів витіснення даних у разі заповнення кешу відповідно до встановлених політик заміщення.

Моніторинг системних ресурсів також є важливою складовою експерименту. За допомогою стандартних команд Redis, таких як INFO memory та MEMORY USAGE, проводився аналіз використання пам'яті для оцінки ефективності очищення застарілих даних, а також розраховувався коефіцієнт фрагментації пам'яті. Для оцінки навантаження на процесор збиралися дані щодо середнього відсотка використання CPU протягом експерименту та використання оперативної пам'яті.

Отже, комплексний збір цих метрик – показників cache hit та cache miss, часу операцій, залишкового TTL, кількості евікшенів, використання пам'яті та навантаження на процесор забезпечує повну оцінку ефективності адаптивного підходу порівняно з традиційними алгоритмами заміщення кешу. Отримані дані експортувалися у CSV-файли, що дозволяло проводити подальший детальний аналіз та порівняння результатів при різних умовах експерименту.

Таким чином, проведення експерименту включало точне налаштування апаратного та програмного середовища, детальну генерацію синтетичних даних із заданими характеристиками та специфічним патерном доступу з домінуванням «гарячих» ключів, встановлення оптимальних параметрів використання пам'яті, а також реалізацію автоматизованої методики тестування з визначенням співвідношення операцій читання до запису, повторюваністю тестів (кожен сценарій виконувався тричі) та попереднім прогріванням системи для досягнення стабільного стану кешу. Це забезпечило відтворюваність умов, точність вимірювань і комплексну оцінку ефективності запропонованих рішень.

5.4 Аналіз отриманих результатів

Отримані результати експериментів дозволили детально проаналізувати ефективність адаптивного підходу порівняно з базовими алгоритмами заміщення

кешу. У дослідженні розглядалися шість рівнів датасету: 10 000, 20 000, 30 000, 50 000, 80 000 та 100 000 записів, що дало змогу виявити закономірності змін у продуктивності кешування для алгоритмів Adaptive, LRU, LFU та Random Eviction. Результати були обчислені як середні значення за трьома повтореннями тестових сценаріїв, при цьому для кожної метрики було визначено стандартне відхилення, що дозволяє оцінити варіативність та надійність отриманих даних. Далі буде проведено детальний аналіз усіх зібраних метрик.

5.4.1 Cache Hit/Miss Rates метрики

Аналіз метрики Cache Hit Rate та Cache Miss Rate за розміром датасету дозволяє оцінити ефективність різних стратегій кешування при зміні обсягу даних. У дослідженні розглядалися шість рівнів датасету: 10000, 20000, 30000, 50000, 80000 та 100000 записів для алгоритмів Adaptive, LRU, LFU та Random Eviction, що дозволило виявити закономірності змін у продуктивності кешування.

На етапі роботи з малими датасетами 10000 записів всі стратегії демонструють схожі високі показники Cache Hit Rate: Adaptive – 90,36%, LRU – 90,80%, LFU – 91,25%, Random Eviction – 90,61%. Низьке стандартне відхилення від 2,03% до 3,75% підтверджує стабільність кешування, що пояснюється ефективним розміщенням малого обсягу даних у пам’яті. Це сприяє мінімізації кеш-промахів, а також зменшенню звернень до основного сховища даних.

При збільшенні датасету до 20000 записів відзначається позитивна динаміка у Adaptive Cache Hit Rate зростає до 90,71% та LFU до 93,06%. Тоді як LRU та Random Eviction починають демонструвати деякі коливання в ефективності, маючи результати 89,93% та 92,22% відповідно. Це може бути пов’язано з тим, що LRU видаляє дані лише на основі часу останнього звернення, не враховуючи частоту використання, а випадкове видалення у Random Eviction починає створювати додаткові кеш-промахи. Варто зазначити, що стандартне відхилення для Adaptive зменшується до 1,38%, що свідчить про його стабільність у роботі.

При 30000 записів показники Adaptive дорівнюють 90,78% та залишаються майже незмінними, а LFU – 93,88% та Random Eviction – 93,11% демонструють

ще вищий рівень попадання в кеш. Проте LRU починає відставати – 89,58%, що свідчить про зростаючий вплив обмеження кешу на його ефективність. Дуже мале стандартне відхилення Adaptive ≈0,27 – 0,28% вказує на передбачуваність його роботи, що може бути корисним для забезпечення стабільної продуктивності.

При 50000 записах всі алгоритми стикаються із зниженням Cache Hit Rate. Adaptive падає до 88,82%, LRU – до 86,49%, а LFU та Random Eviction – до 89,91% та 88,59% відповідно. Це пояснюється тим, що кеш не вміщує весь набір даних та часті промахи стають неминучими. Зростання стандартного відхилення вказує на підвищення варіативності кеш-промахів, що ускладнює прогнозованість роботи кешу.

При 80000 записах ефективність усіх стратегій продовжує знижуватися: Adaptive до 82,75%, LFU – 82,09%, Random Eviction – 81,71%, а LRU – 78,52%. Це підтверджує тенденцію, що при значному розмірі датасету кеш більше не може ефективно зберігати популярні дані, що призводить до збільшення кількості кеш-промахів. Водночас, зростання стандартного відхилення ≈3,69–4,24% вказує на зниження стабільності результатів.

При 100000 записах всі стратегії досягають найнижчих значень попадання в кеш: Adaptive – 73,59%, LRU – 72,06%, LFU – 73,86%, а Random Eviction – 74,42%. Це свідчить про критичний вплив обмеження кешу на продуктивність усіх підходів. Високий рівень Cache Miss Rate та збільшене стандартне відхилення ≈4,46–5,03% підтверджують, що система не здатна ефективно утримувати часто використовувані дані, через що зростає навантаження на основне сховище.

Аналіз результатів свідчить, що для малих обсягів даних, 10000 – 30000 записів, LFU та Adaptive демонструють найкращі показники Cache Hit Rate, оскільки вони краще адаптуються до характеру звернень до кешу. LRU, хоча і працює ефективно, починає втрачати позиції при збільшенні обсягу даних, що вказує на його обмеження у сценаріях із великою кількістю записів. Random Eviction дає непогані результати, однак його випадковий підхід до видалення даних може бути менш ефективним у довготривалій перспективі.

Для великих датасетів, 50000 – 100000 записів, спостерігається зниження ефективності всіх стратегій, що підтверджується зростанням Cache Miss Rate. Adaptive демонструє меншу варіативність у результатах до 50000 записів, однак при подальшому збільшенні навантаження його ефективність також падає. Усі алгоритми при 100000 записах показують схожі результати, що свідчить про те, що обмеження кешу стає вирішальним фактором продуктивності.

Графіки на рисунках 5.10 та 5.11 ілюструють динаміку змін Cache Hit Rate та Cache Miss Rate, підтверджуючи, що правильний вибір стратегії кешування залежить від розміру датасету та характеристик звернень.

Таким чином, аналіз показує, що ефективність стратегій кешування суттєво залежить від розміру датасету та характеристик звернень.

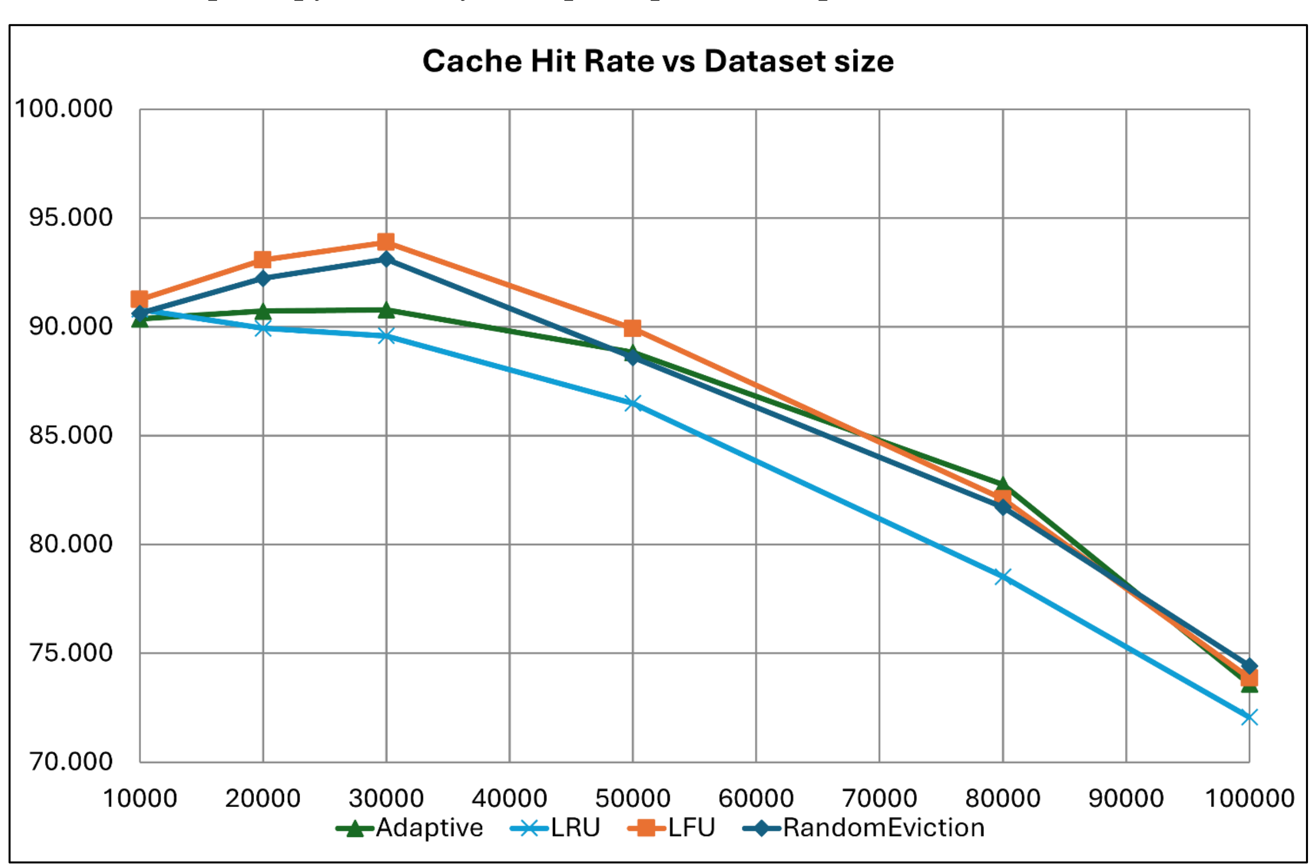


Рисунок 5.10 – Графік залежності Cache Hit Rate від розміру набору даних (рисунок виконаний самостійно)

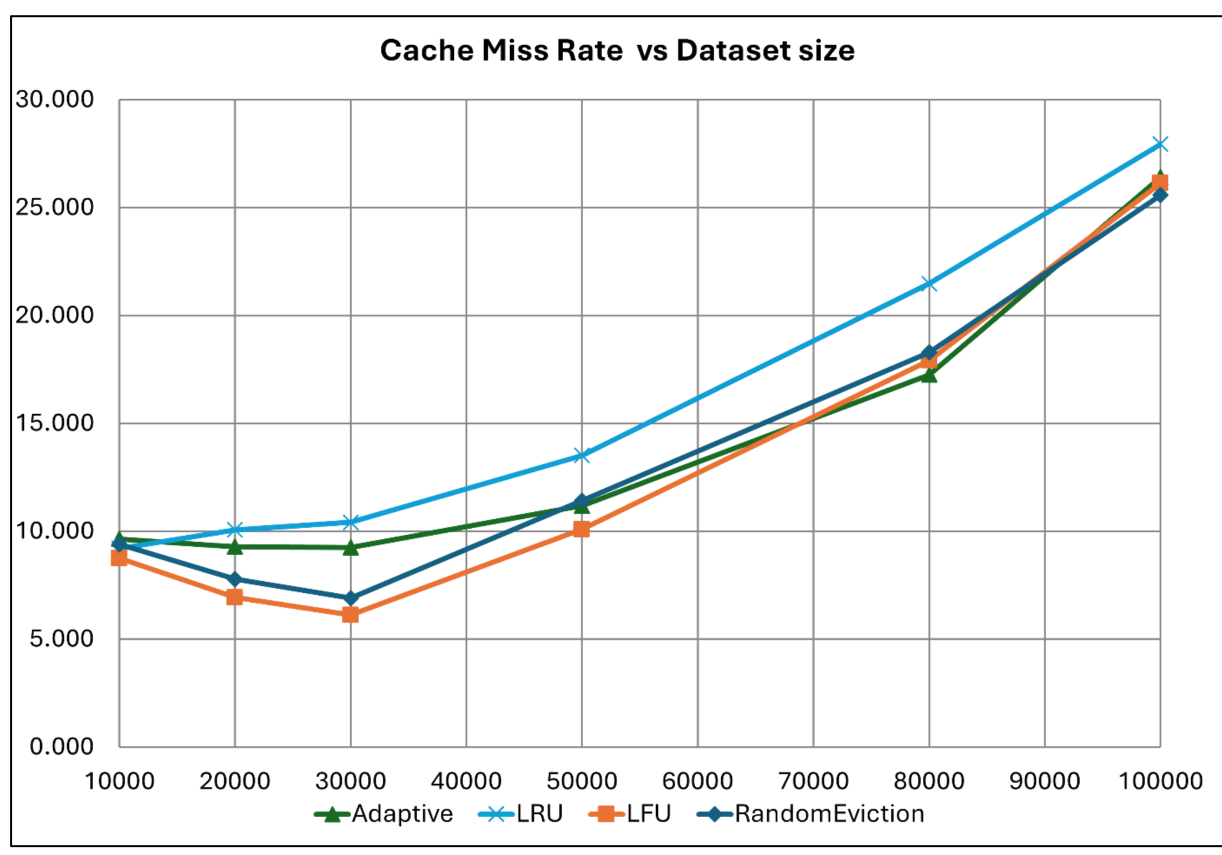


Рисунок 5.11 – Графік залежності Cache Miss Rate від розміру набору даних (рисунок виконаний самостійно)

Отримані результати дозволяють зробити висновок, що Adaptive та LFU є більш стабільними для малих та середніх обсягів даних, тоді як при значному збільшенні датасету всі стратегії стикаються із зниженням продуктивності через обмеження кешу.

5.4.2 Evictions метрика

Метрика Evictions є важливим показником ефективності кешування, що демонструє, як часто дані видаляються з кешу через обмеження пам’яті та політику заміщення. Висока кількість евікшенів може свідчити про надмірну ротацію даних, що призводить до повторного завантаження інформації та збільшення затримок. Оптимальна стратегія має підтримувати баланс між оновленням кешу та збереженням актуальних даних.

На малих наборах даних 10000 записів всі стратегії демонструють приблизно однакову кількість евікшенів: Adaptive – 2261, LRU – 2452, LFU –

2403, Random Eviction – 2455. Проте ключовою відмінністю є варіативність видалень. Adaptive демонструє найнижчий рівень варіативності, ≈28,88, тоді як у LRU вона значно вища ≈216,18, а у LFU – ≈174,25. Це свідчить про те, що Adaptive більш передбачуваний та стабільний у роботі на початкових рівнях навантаження, що допомагає уникати надмірного видалення даних.

Зі збільшенням датасету до 20 000 записів загальна кількість евікшенів зростає. Adaptive досягає 4211 видалень, що трохи менше, ніж у LRU – 4458, але більше, ніж у Random Eviction – 4003. Однак варто відзначити, що варіативність Adaptive ≈285,00 є дещо більшою, ніж у LFU ≈250,99, проте практично співпадає з Random Eviction ≈285,16, що свідчить про схожу стабільність його роботи та прогнозованість поведінки.

При 30000 записів загальна кількість видалень для Adaptive зростає до 6209, а LRU до 6399, тоді як LFU – 5113 та Random Eviction – 5344 демонструють менші показники. Попри це, Adaptive продовжує демонструвати найнижчу варіативність ≈168,54, що вказує на стабільну частоту видалень. Натомість LRU та Random Eviction мають вищу варіативність ≈420,69 та 346,68 відповідно, що може свідчити про нерівномірний розподіл видалень у кеші. Це особливо важливо для систем, які потребують передбачуваного кешування без різких стрибків ефективності.

При 50000 записів спостерігається значне зростання кількості евікшенів для всіх стратегій. Adaptive демонструє 8248 видалень, що менше за LRU 9754, але більше, ніж у LFU 7803 та Random Eviction 8076. Важливо зазначити, що, хоча загальна кількість видалень збільшується, Adaptive продовжує демонструвати нижчу варіативність ≈402,24 у порівнянні з LRU ≈423,63. Це підтверджує, що Adaptive залишається стабільним навіть при зростанні навантаження, забезпечуючи передбачувану частоту видалень.

При 80000 записів кількість евікшенів для Adaptive досягає 10979, що є наближеним до LFU – 10944 та Random Eviction – 11195, проте значно нижчим за LRU – 13803. Варіативність Adaptive ≈283,45 свідчить про його здатність підтримувати стабільну частоту видалень даних, навіть у складних сценаріях,

коли обсяг даних значно перевищує кеш. Це підтверджує його ефективність для систем із великими наборами даних, у яких важливо зберігати передбачувану продуктивність та мінімізувати випадкові втрати важливої інформації.

Нарешті, при 100000 записів усі стратегії стикаються зі значним зростанням кількості евікшенів. Adaptive має 12697 видалень, що наближено до LFU – 12776, проте значно нижче, ніж у LRU 16548 та Random Eviction 13165. Це підтверджує, що LRU схильний до надмірної ротації кешу, що може негативно впливати на його ефективність у великих системах. Random Eviction має найменшу кількість евікшенів ≈301,16, за нею слідує LRU ≈310,77, потім LFU ≈325,76, а Adaptive – найвищу ≈348,33. Це свідчить про те, що Adaptive застосовує більш агресивний підхід до управління кешем, що може впливати на його ефективність у великих системах.

Графік на рисунку 5.12 наочно ілюструє динаміку змін Evictions залежно від розміру датасету.

Рисунок 5.12 – Графік залежності Evictions від розміру набору даних (рисунок виконаний самостійно)

Отже, метрика Evictions свідчить, що зростання розміру датасету веде до збільшення кількості заміщень даних, причому на малих та середніх наборах

стратегії Adaptive та LFU демонструють кращу стабільність із нижчою варіативністю, ніж LRU та Random Eviction. Проте при великих датасетах хоча загальна кількість евікшенів для Adaptive залишається конкурентоспроможною, її зростаюча варіативність може впливати на передбачуваність роботи системи, що свідчить про необхідність врахування компромісу між абсолютними значеннями заміщень та їх стабільністю при виборі оптимальної стратегії кешування.

5.4.3 Average TTL метрика

Метрика Average TTL, що відображає середній час життя даних у кеші, є важливим показником ефективності стратегії управління кешем. Вона демонструє, як довго дані залишаються в кеші перед видаленням, що безпосередньо впливає на продуктивність системи. Вищий середній TTL свідчить про те, що дані довше зберігаються, зменшуючи навантаження на основне джерело, проте занадто високий TTL може призводити до накопичення застарілих записів, а занадто низький до частого видалення корисної інформації. Слід зазначити, що TTL для ключів встановлюється на основі базового значення, а алгоритми заміщення лише впливають на процес видалення даних.

На малих наборах даних 10000 записів Adaptive демонструє найвищий середній TTL – 82,26 секунд, що суттєво перевищує результати LRU – 56,53 секунд, LFU – 54,01 секунд та Random Eviction – 55,54 секунд. Це свідчить про здатність адаптивного підходу утримувати популярні дані у кеші та зменшувати кількість повторних запитів до основного сховища. Варіативність показників також є важливою: стандартне відхилення для Adaptive складає ≈0,87 секунд, тоді як для LRU – ≈1,57 секунд, LFU – ≈0,35 секунд та Random Eviction – ≈0,39 секунд.

При збільшенні обсягу датасету до 20000 записів, Adaptive демонструє середній TTL 85,08 секунд, що є незначним покращенням порівняно з попереднім рівнем, тоді як при використанні традиційних алгоритмів відбувається зниження TTL: LRU – 55,50 секунд, LFU – 49,78 секунд, Random Eviction – 51,15 секунд. Стандартне відхилення для Adaptive становить ≈2,54 секунд, що трохи перевищує

показники LFU ≈2,19 секунд та Random Eviction ≈2,51 секунд, проте залишається досить стабільним.

При середньому розмірі датасету 30000 записів Adaptive забезпечує середній TTL на рівні 85,04 секунд, що значно перевищує показники при роботі традиційних стратегій LRU – 53,15 секунд, LFU – 46,64 секунд, Random Eviction – 47,83 секунд. Варіативність роботи Adaptive трохи перевищує її аналоги ≈2,52 секунд порівняно з LRU ≈2,07 секунд, LFU ≈1,14 секунд та Random Eviction ≈0,38 секунд.

При 50000 записів Adaptive підтримує високий середній TTL – 85,36 секунд при стабільному стандартному відхиленні ≈0,57 секунд, тоді інші демонструють значно менші значення TTL: LRU – 49,31 секунд, LFU – 43,40 секунд, Random Eviction – 44,29 секунд із подібною варіативністю.

При 80000 записів Adaptive демонструє середній TTL 84,29 секунд, що трохи нижче, ніж при 50000 записів, але все ще суттєво перевищує показники LRU – 51,76 секунд, LFU – 46,36 секунд та Random Eviction – 46,85 секунд. Варіативність Adaptive тут становить ≈0,36 секунд, тоді як для традиційних алгоритмів виходить наступне: LRU – ≈0,23 секунд, LFU – ≈1,10 секунд, Random Eviction – ≈1,45 секунд.

При 100000 записів Adaptive підтримує середній TTL 84,75 секунд, що є конкурентоспроможним результатом у порівнянні з традиційними методами LRU – 55,58 секунд, LFU – 51,87 секунд, Random Eviction – 51,72 секунд. Стандартне відхилення для Adaptive складає ≈0,55 секунд, що залишається нижчим, ніж у LRU ≈0,78 секунд, LFU ≈1,168 секунд та Random Eviction ≈1,55 секунд, що дозволяє підтримувати стабільність збереження даних.

Графік на рисунку 5.13 наочно ілюструє динаміку змін Average TTL залежно від розміру датасету.

Отже, результати дослідження підтверджують, що адаптивне управління TTL дозволяє ефективніше зберігати корисні дані в кеші, мінімізуючи кеш-промахи та зменшуючи навантаження на основне сховище.

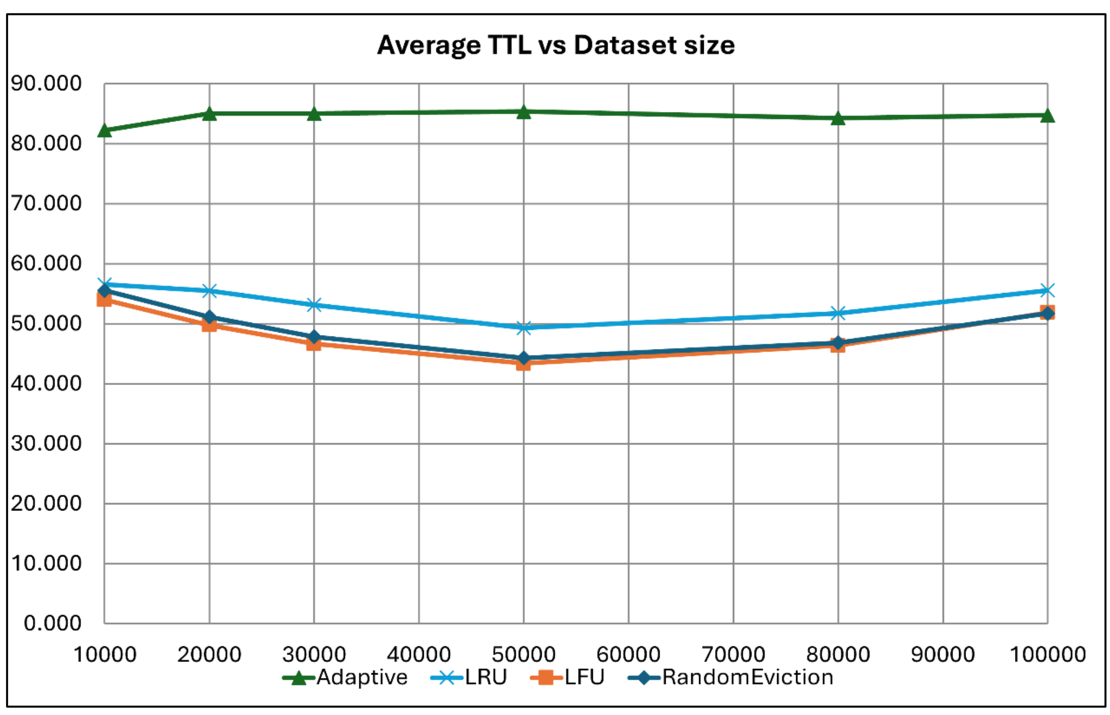


Рисунок 5.13 – Графік залежності Average TTL від розміру набору даних (рисунок виконаний самостійно)

Адаптивна стратегія демонструє стабільний середній TTL незалежно від розміру датасету, тоді як традиційні алгоритми схильні до швидшого видалення даних, що може негативно впливати на продуктивність у сценаріях із частими повторними запитами.

5.4.4 Average Write/Read Times метрики

Метрики Average Write Time (середній час запису) та Average Read Time (середній час читання) є ключовими показниками ефективності стратегій кешування, адже вони безпосередньо впливають на загальну продуктивність системи та швидкість обробки запитів.

На малих наборах даних 10000 записів Adaptive демонструє середній час запису 6,855 мс/ключ та середній час читання 5,408 мс/ключ, що суттєво перевищує показники традиційних методів кешування. Наприклад, LRU має 0,452 мс/ключ для запису та 0,436 мс/ключ для читання, LFU – 0,455 мс/ключ і 0,459 мс/ключ відповідно, а Random Eviction – 0,475 мс/ключ і 0,458 мс/ключ відповідно.

Зі збільшенням датасету до 20000 та 30000 записів час запису для Adaptive змінюється незначно 7,115 мс/ключ та 7,061 мс/ключ, тоді як середній час читання помітно знижується з 3,257 мс/ключ до 1,550 мс/ключ. У той же час LRU, LFU та Random Eviction зберігають стабільно низькі показники близько 0,38 – 0,46 мс/ключ для запису та читання, демонструючи високу передбачуваність та мінімальні накладні витрати незалежно від розміру датасету.

При 50000 записів середній час запису Adaptive становить уже 6,206 мс/ключ, тоді як середній час читання знижується до 0,836 мс/ключ. На цьому етапі алгоритм досягає певної стабільності в управлінні кешем, що прискорює отримання збережених даних. Водночас традиційні алгоритми зберігають свої показники на рівні 0,38 – 0,42 мс/ключ, підтверджуючи стабільність у масштабованих середовищах.

При 80000 записів час читання Adaptive далі зменшується до 0,786 мс/ключ, що наближається до значень традиційних алгоритмів, проте час запису 5,993 мс/ключ залишається суттєво вищим. LRU, LFU та Random Eviction, як і раніше, демонструють майже незмінну продуктивність із часом запису в межах 0,39 – 0,40 мс/ключ і часом читання на рівні 0,38 – 0,39 мс/ключ.

Нарешті, при 100000 записів Adaptive досягає найнижчого середнього часу читання 0,783 мс/ключ, що приблизно у 7 разів нижче, ніж при 10000 записів, підтверджуючи ефективність адаптивної оптимізації для великих наборів даних. Водночас час запису залишається досить високим 6,059 мс/ключ, що може обмежувати застосування Adaptive у сценаріях із великою кількістю операцій запису. У той же час LRU, LFU та Random Eviction продовжують демонструвати стабільні показники ≈0,39 мс/ключ для запису та читання, підкреслюючи їхню передбачуваність у масштабованих системах.

Графіки на рисунках 5.14 та 5.15 наочно ілюструють тенденції змін Average Write Time та Average Read Time залежно від розміру датасету, дозволяючи порівняти ефективність різних стратегій у динаміці.

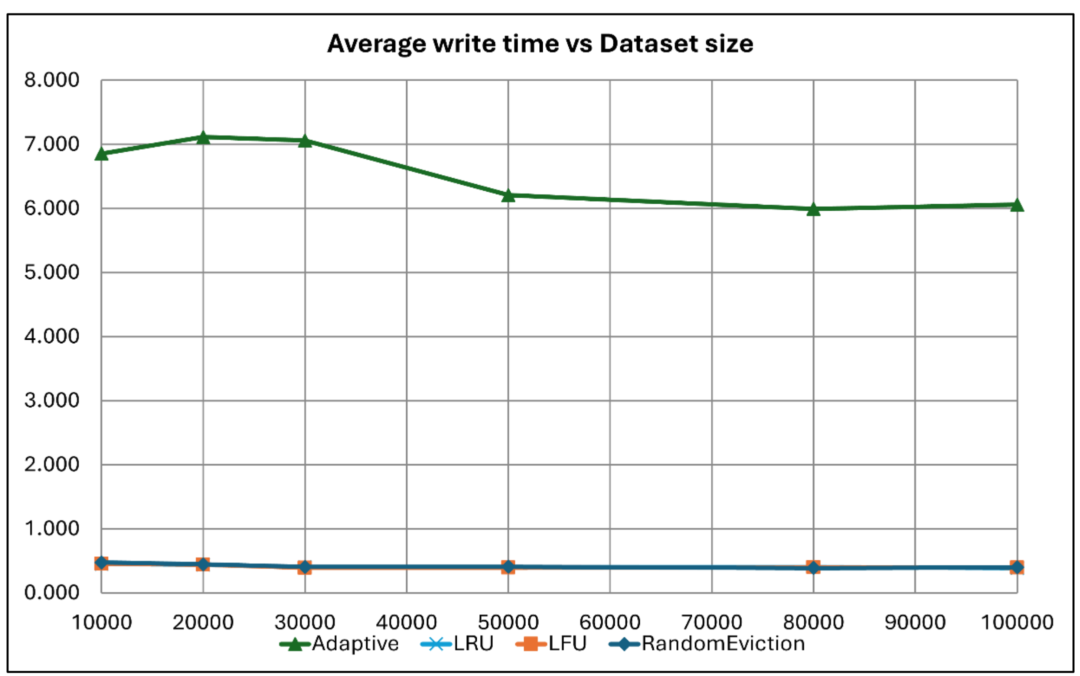


Рисунок 5.14 – Графік залежності Average Write Time від розміру набору даних (рисунок виконаний самостійно)

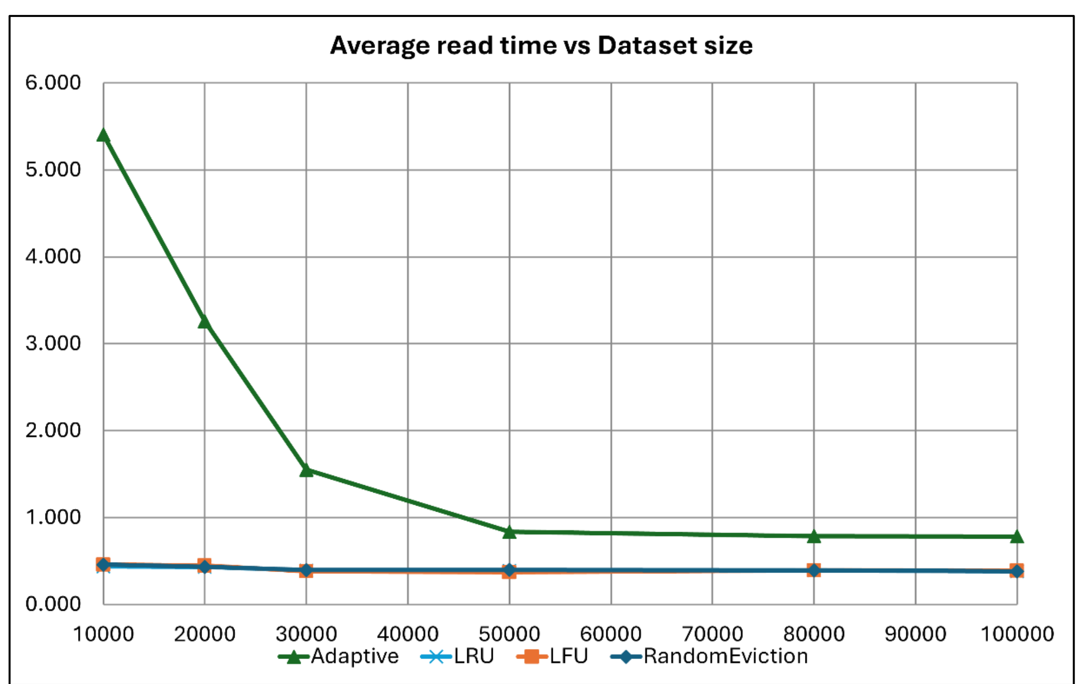


Рисунок 5.15 – Графік залежності Average Read Time від розміру набору даних (рисунок виконаний самостійно)

Отже, результати дослідження показують, що традиційні алгоритми кешування (LRU, LFU, Random Eviction) забезпечують стабільно низький час запису та читання, що робить їх ефективними для сценаріїв із високою частотою

запитів та мінімальними затримками. Adaptive, хоча й має значно вищі накладні витрати на запис, демонструє суттєве покращення часу читання при збільшенні розміру датасету, що підтверджує його ефективність у довгостроковій перспективі.

5.4.5 Memory Usage Delta метрика

Метрика Memory Usage Delta дозволяє оцінити, як змінюється використання пам’яті для кешування при застосуванні різних стратегій кешування. Цей показник відображає різницю між використаною пам’яттю до та після операцій кешування, що дозволяє оцінити ефективність механізмів евікшену та загальну продуктивність алгоритмів.

На малих наборах даних 10000 записів Adaptive демонструє стабільне зниження використання пам’яті з Memory Usage Delta – -4292 байти, що свідчить про активне очищення кешу та видалення застарілих записів. При 100000 записів це значення збільшується до -11368 байт, що вказує на інтенсивніше звільнення пам’яті зі збільшенням обсягу даних. Це підтверджує, що Adaptive активно звільняє кеш, адаптуючись до зміни навантаження, хоча не завжди зі строго монотонною динамікою. Наприклад, при 80000 записів значення -9082 байти, що дещо вище, ніж при 50000 записів – -1674 байти.

Для LRU значення Memory Usage Delta коливаються у відносно малому діапазоні, що вказує на стабільність використання пам’яті. Наприклад, при 10000 записів LRU має +757 байт, при 50000 – -1 533 байти, а при 100 000 – +1219 байт. Це свідчить про те, що LRU не вносить значних змін у загальне використання пам’яті, зберігаючи баланс між звільненням та додаванням нових елементів.

Стратегія LFU демонструє нестабільну поведінку щодо використання пам’яті. При 10000 записів Memory Usage Delta становить +1504 байти, а при 30000 записів – лише +189 байт. Однак при 80 000 записів спостерігається різке зростання використання пам’яті +9293 байти, що може бути пов’язано з пороговими змінами у стратегії заміщення даних. При 100000 записів Memory

Usage Delta зменшується до +651 байта, що може вказувати на стабілізацію після аномального стрибка.

У Random Eviction значення Memory Usage Delta коливаються без чіткої тенденції. Наприклад, при 10000 записів воно становить – -779 байт, при 20000 – +1787 байт, при 30000 – -736 байт, при 50000 – +165 байт, а при 80000 спостерігається зниження – -2603 байти. Така осцилююча поведінка пояснюється випадковим характером алгоритму видалення, що може призводити як до збільшення використання пам’яті, так і до його зменшення залежно від конкретного набору запитів.

Графік на рисунку 5.16 наочно ілюструє залежність Memory Usage Delta від розміру датасету.

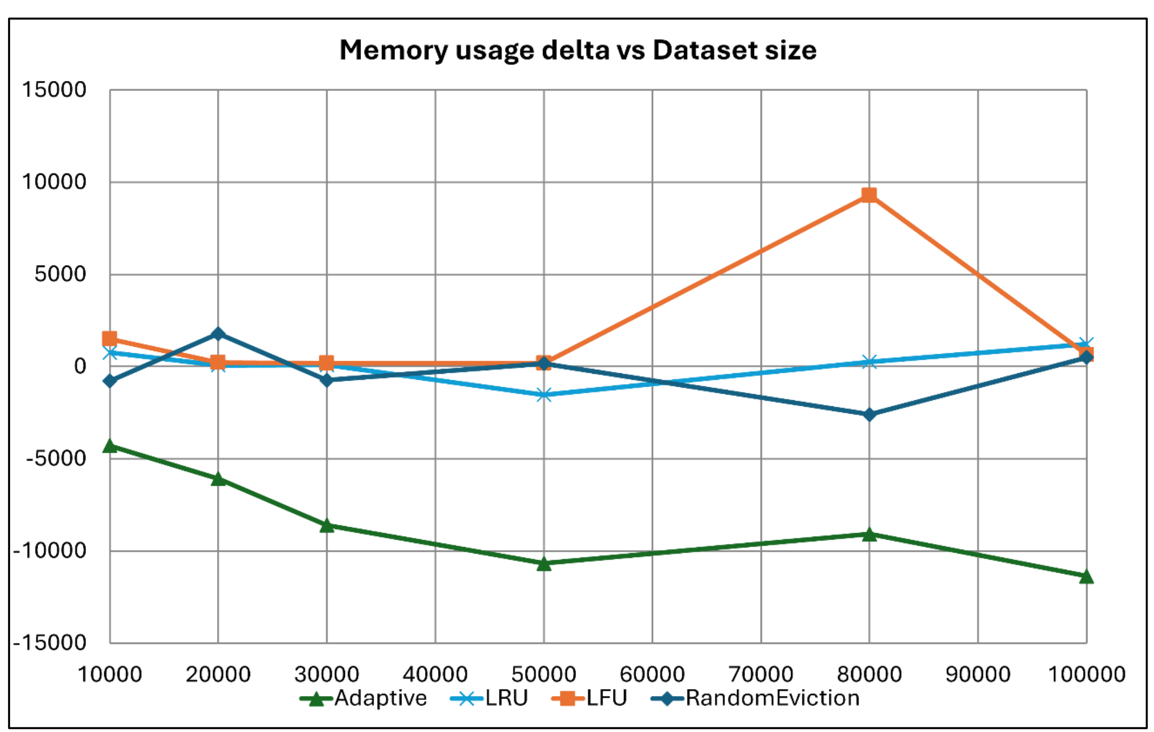


Рисунок 5.16 – Графік залежності Memory Usage Delta від розміру набору даних (рисунок виконаний самостійно)

Таким чином, Adaptive загалом найбільш інтенсивно звільняє кеш, проте це може впливати на стабільність продуктивності. LRU вирізняється найпередбачуванішим використанням пам’яті, що робить його привабливим для систем із прогнозованим навантаженням. LFU та Random Eviction мають вищу

варіативність, що може призводити до непередбачуваних змін у використанні пам'яті при великих обсягах даних.

5.4.6 Process CPU Usage метрика

Метрика Process CPU Usage відображає рівень використання процесорних ресурсів у Redis при застосуванні різних стратегій кешування. Вона дозволяє оцінити, наскільки ефективно кожен алгоритм управляє процесорним навантаженням при зміні обсягу даних.

Аналіз отриманих експериментальних даних показує, що стратегія Adaptive демонструє найвищий рівень використання CPU, який зростає майже лінійно зі збільшенням розміру датасету: від 6.547% при 10000 ключів до 11.345% при 100000. Це свідчить про те, що алгоритм використовує додаткові ресурси для аналізу та оптимізації внутрішніх структур кешу. Стратегія LRU демонструє більш рівномірне зростання використання CPU: від 3.628% при 10000 ключів до 9.286% при 100000. При цьому на малих наборах даних алгоритм працює більш ефективно, але з ростом обсягу спостерігається поступове підвищення обчислювального навантаження. Стратегія LFU має схожу динаміку до LRU, однак початкові значення CPU є дещо нижчими: 3.161% при 10000 ключів та 9.553% при 100000. Це може вказувати на менший початковий обчислювальний наклад, проте зі зростанням обсягу кешованих даних продуктивність поступово наближається до LRU. RandomEviction демонструє найнижче використання процесорних ресурсів серед усіх алгоритмів від 2.182% при 10000 ключів до 8.827% при 100000. Це пояснюється простотою механізму випадкового видалення, який не потребує додаткових обчислень для визначення, які елементи кешу варто замінити.

Аналіз стандартного відхилення показує, що Adaptive має найбільші коливання на малих обсягах даних ≈2.306% при 10000 ключів, однак зі збільшенням датасету стабільність покращується ≈0.565% при 100000. Це свідчить про те, що адаптивний алгоритм вимагає більше ресурсів на початкових етапах роботи, але стабілізується з ростом датасету. LRU демонструє відносно

сталі значення стандартного відхилення, які варіюються від 1.042% до 1.256% на малих обсягах даних, а на великих наборах знижуються до 0.628 – 0.686%. Це свідчить про стабільне використання CPU при великих розмірах кешу. Для LFU стандартне відхилення залишається невеликим у межах 0.472 – 0.828% на малих наборах, але підвищується до 1.169% при 100000 ключів. Це може вказувати на дещо непередбачувану поведінку алгоритму при великих обсягах кешу. RandomEviction демонструє найменші коливання, оскільки стандартне відхилення поступово зменшується від 1.369% для 10000 ключів до 0.0657% для 100000 ключів, що свідчить про його стабільну продуктивність навіть при великих датасетах.

Графік на рисунку 5.17 наочно ілюструє залежність Process CPU Usage від розміру датасету.

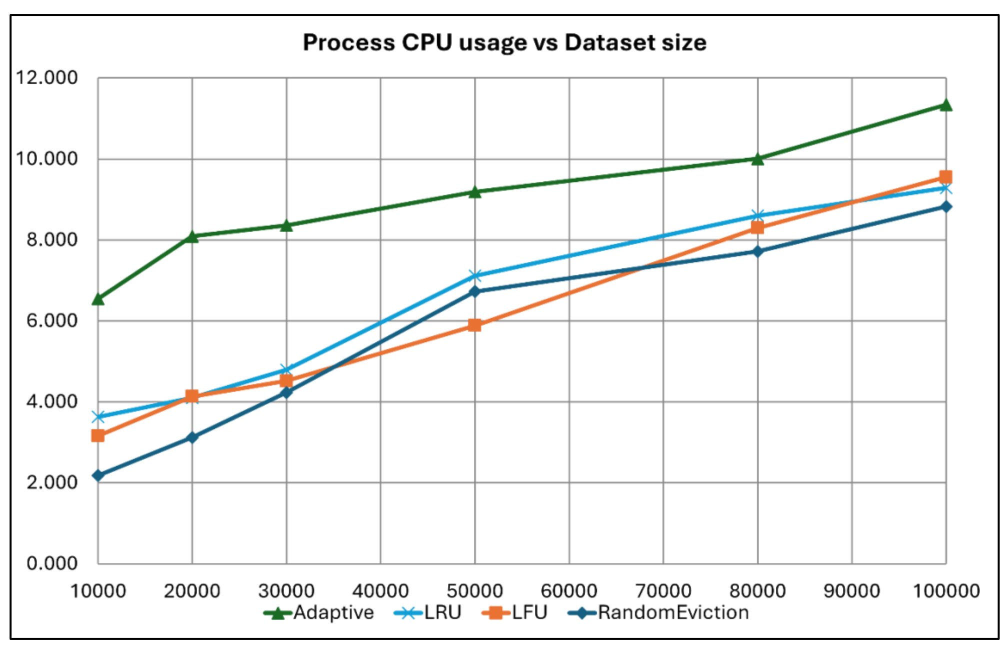


Рисунок 5.17 – Графік залежності Process CPU Usage від розміру набору даних (рисунок виконаний самостійно)

Отже, аналіз метрики Process CPU Usage показує, що RandomEviction є найменш ресурсоємним алгоритмом ідеальним для середовищ з жорсткими обмеженнями на CPU. LRU та LFU забезпечують збалансоване використання процесорних ресурсів, що робить їх оптимальними для компромісу між продуктивністю кешування та навантаженням на систему. Adaptive, хоча і

пропонує високу адаптивність, вимагає більше CPU-ресурсів і доцільний лише там, де доступна висока обчислювальна потужність.

5.4.7 Process Memory Delta метрика

Метрика Process Memory Delta відображає зміну використання оперативної пам’яті при виконанні операцій кешування в Redis, що дозволяє оцінити ефективність управління пам’яттю для різних стратегій кешування.

Стратегія Adaptive демонструє тенденцію до зростання використання пам’яті із збільшенням обсягу даних. При розмірі датасету 10000 ключів її Memory Delta становить 5312512 байт, а при 100000 ключів — вже 25643691 байт. Незважаючи на періодичні незначні коливання, загальна тенденція свідчить про значні накладні витрати на управління кешем, які можуть бути пов’язані з обробкою додаткової метаінформації або частішими внутрішніми реорганізаціями пам’яті. Зі збільшенням розміру кешу навантаження продовжує зростати, що вказує на необхідність оптимізації підхід Adaptive у високонавантажених середовищах.

Алгоритм LRU демонструє значні коливання у використанні пам’яті. При 10000 записах Memory Delta становить 2222763 байти, знижується до 1758549 байт при 20000 записах, потім зростає до 5018965 байт при 30000, різко падає до 845141 байт при 50000 і знову зростає до 14792021 байт при 100000 записах. Така нестабільність може бути наслідком внутрішніх механізмів очищення кешу, коли алгоритм змінює поведінку залежно від наповненості пам’яті. Найнижчі значення при 50000 записах можуть свідчити про оптимальний рівень кешування для цієї стратегії, тоді як різке зростання при 100000 вказує на можливі проблеми з масштабуванням.

У стратегії LFU спостерігається порівняно більш стабільне використання пам’яті, хоча теж з певними коливаннями. При 10000 записах Memory Delta становить 3896661 байт, при 20000 спостерігається невелике зниження до 3620864 байт, після чого показник поступово зростає до 8844629 байт при 100000 записах. Найнижче використання пам’яті фіксується при 50000 записах – 1530539 байт, що

вказує на можливу оптимізацію механізму заміщення ключів у цьому масштабі. Загалом LFU показує помірне використання пам’яті та кращу стабільність порівняно з LRU та Adaptive, що робить її привабливим варіантом для сценаріїв з високою частотою звернень до обмеженого набору ключів.

Стратегія Random Eviction демонструє найменші витрати пам’яті серед усіх алгоритмів, однак теж має коливання. При 10000 записах Memory Delta становить 3831125 байт, при 20000 – 3303211 байт, після чого значення продовжують змінюватися, досягаючи 7201156 байт при 100000 записах. Незважаючи на випадковий характер видалення даних, Random Eviction загалом демонструє більш ефективне управління пам’яттю в порівнянні з Adaptive та LRU. Простота алгоритму дозволяє зменшити накладні витрати, але може призводити до видалення важливих ключів, що може негативно впливати на продуктивність у певних сценаріях.

Аналіз стандартного відхилення показує, що Adaptive має найбільші коливання: від $2{,}1\times10^{6}$ байт при 10000 записах до $3{,}2\times10^{6}$ байт при 100000 записах, що свідчить про значну варіативність використання пам’яті на різних рівнях масштабування. LRU, навпаки, демонструє стабільність при 50000 записах, де стандартне відхилення становить лише 161485 байт, але при 100000 записах різко зростає до $10{,}6\times10^{6}$ байт, що підтверджує його нестабільність у великих масштабах. LFU демонструє порівняно низькі значення стандартного відхилення, що варіюються від $1{,}4\times10^{6}$ до $2{,}5\times10^{6}$ байт, що свідчить про відносно передбачуване використання пам’яті. Random Eviction показує найнижчі значення стандартного відхилення при малих розмірах датасету, 252719 байт при 20000 записах та дещо вищі при 100000 записах – $3{,}7\times10^{6}$ байт, що все ж залишається меншим за Adaptive та LRU.

Аналіз кореня стандартного відхилення додатково підтверджує тенденції. Для Adaptive він знижується від $1{,}2\times10^{6}$ байт при 10000 записах до $1{,}8\times10^{6}$ байт при 100000 записах, що підтверджує зменшення варіативності на великих наборах даних. Для LRU корінь стандартного відхилення коливається від 93233 байт при 50000 записах до $6{,}1\times10^{6}$ байт при 100000 записах, що підтверджує значну

непередбачуваність у великих масштабах. LFU та Random Eviction демонструють стабільніші значення, що варіюються в межах 25 – 40% від середнього використання пам'яті, що вказує на їхню більш передбачувану поведінку.

Графік на рисунку 5.18 наочно ілюструє залежність Process Memory Delta від розміру датасету.

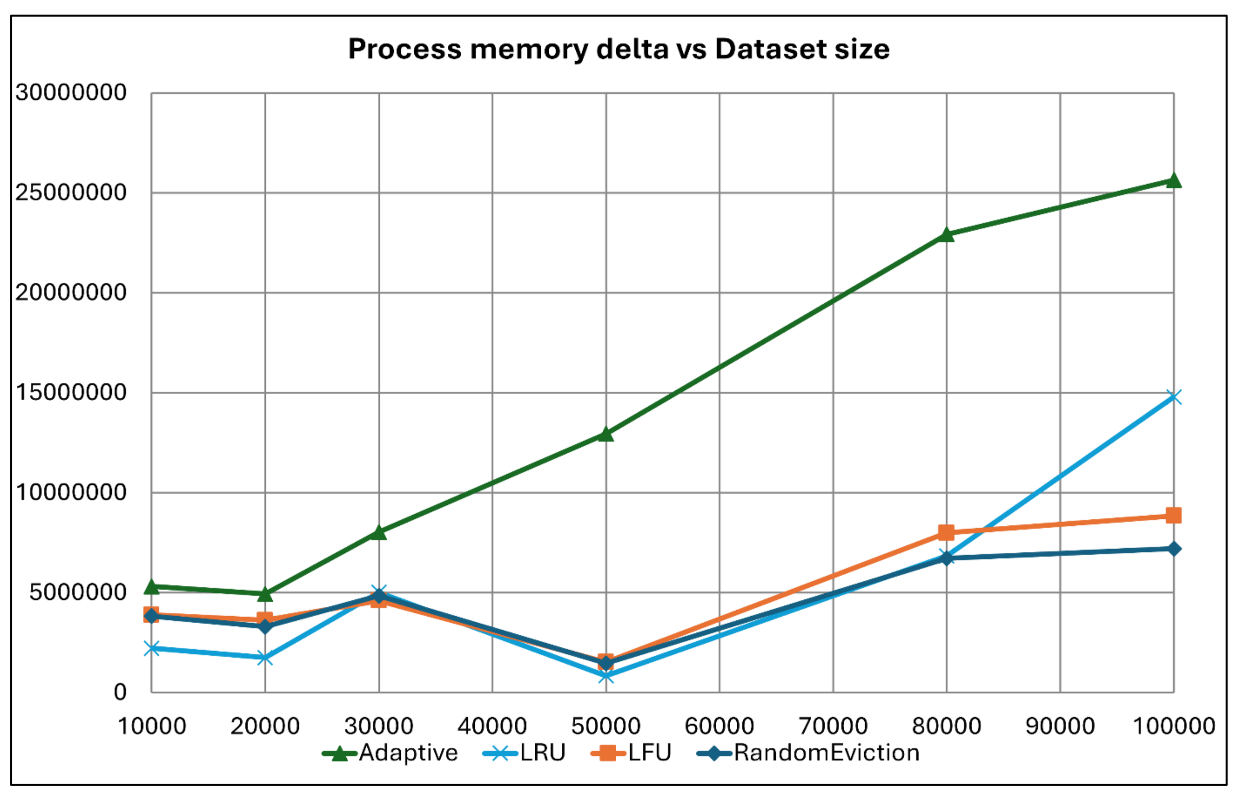


Рисунок 5.18 – Графік залежності Process Memory Delta від розміру набору даних (рисунок виконаний самостійно)

Отже, аналіз Process Memory Delta показує, що Random Eviction демонструє найнижчі витрати пам'яті та відносну стабільність, тоді як LFU забезпечує більш передбачуване використання пам'яті порівняно з Adaptive та LRU. Adaptive, незважаючи на свою ефективність, характеризується значним зростанням накладних витрат і високими коливаннями, що може ускладнювати його застосування у високонавантажених середовищах.

5.5 Висновки експериментального дослідження та рекомендації

На основі експериментального дослідження були зроблені висновки щодо ефективності різних стратегій керування кешованими даними: Adaptive, LRU,

LFU та Random Eviction у залежності від розміру набору даних та умов експлуатації.

На малих та середніх обсягах (10 000 – 30 000 записів) найкращі результати показали стратегії LFU та Adaptive. Обидва підходи забезпечують високу частоту попадання в кеш (Cache Hit Rate), при цьому Adaptive додатково демонструє перевагу за рахунок гнучкого визначення TTL окремих елементів, що сприяє ефективному збереженню активних даних та своєчасному видаленню менш важливих.

При збільшенні розміру набору даних до 50 000 – 100 000 записів стають очевидними обмеження обсягу кешу. У таких умовах усі досліджувані алгоритми зазнають певних обмежень, проте Adaptive та LFU залишаються найбільш конкурентоспроможними. LRU демонструє помітне зростання кількості кеш-промахів через базування виключно на часі останнього доступу. Стратегії LRU та Random Eviction характеризуються нижчими витратами на запис та читання, однак Adaptive стратегія суттєво зменшує час читання даних при роботі з великими обсягами, що є критичним для високонавантажених систем.

Отримані результати мають практичну цінність для розробників та адміністраторів систем. При виборі стратегії кешування слід враховувати особливості навантаження та вимоги до продуктивності:

- для малих та середніх систем (10 000 – 30 000 записів) оптимальним вибором є Adaptive або LFU завдяки високій ефективності та стабільності роботи;
- якщо пріоритетом є мінімізація накладних витрат на обчислення та запис, варто розглядати LRU або Random Eviction;
- для великих датасетів (50 000 – 100 000 записів і більше) перевагу має Adaptive підхід, оскільки він суттєво скорочує час читання та зменшує навантаження на основне сховище, підвищуючи загальну продуктивність.

Подальші дослідження можуть бути спрямовані на зниження обчислювального навантаження та покращення продуктивності адаптивного алгоритму шляхом застосування додаткових оптимізаційних підходів, наприклад,

адаптації до специфіки апаратного забезпечення чи використання алгоритмів прогнозування запитів. Перспективним також є дослідження інтегрованих підходів, які будуть динамічно обирати або поєднувати кілька стратегій залежно від поточних умов роботи та характеру запитів.

Окрім того, актуальним напрямком може стати розробка гібридних алгоритмів, що поєднуватимуть переваги кількох стратегій (наприклад, LRU та LFU з адаптивним TTL), а також впровадження механізмів динамічного налаштування параметрів кешування на основі аналізу профілю запитів та зміни навантаження. Такі гібридні підходи мають потенціал для забезпечення ще вищих показників продуктивності та більш ефективного використання пам’яті, що є особливо важливим для високонавантажених систем та додатків з високими вимогами до продуктивності та стабільності.

Таким чином, результати дослідження не лише дають змогу об’єктивно оцінити ефективність кожної стратегії кешування, а й окреслюють напрями для подальшого вдосконалення підходів до керування кешем.

Вихідний код розробленого застосунку доступний для завантаження з [25]. Основні положення роботи були представлені у тезах доповіді [26].

ВИСНОВКИ

Проведене дослідження підтвердило актуальність задачі оптимізації управління кешованими даними у Redis, особливо для складних структур, таких як хеші, списки та множини. Аналіз існуючих підходів, як TTL, LRU, LFU, показав їх обмеження щодо ефективності використання пам'яті та продуктивності при роботі зі складними структурами. Зокрема, традиційні методи мають суттєві недоліки, пов'язані з недостатньою гнучкістю TTL, що застосовується до всієї структури даних, та не враховують частоту доступу та актуальність окремих елементів, що призводить до надмірного видалення активних даних та нераціонального використання пам’яті.

Для вирішення цих проблем було запропоновано новий адаптивний підхід до управління кешем, який базується на динамічному визначенні TTL для окремих елементів складних структур та комбінованому алгоритмі заміщення кешу, що враховує частоту звернень, давність використання та обсяг пам’яті, яку займають дані. Запропоновані методи дозволяють ефективно регулювати час життя даних у кеші, уникаючи як надмірного зберігання неактуальної інформації, так і передчасного видалення важливих елементів.

Експериментальне дослідження підтвердило переваги адаптивного підходу. Він показав стабільно високі показники попадання в кеш (Cache Hit Rate) на рівні 88 – 90% для малих та середніх обсягів даних до 50 000 записів, значно перевершуючи традиційні LRU та Random Eviction алгоритми, які демонстрували зниження ефективності вже при середніх навантаженнях. Особливо помітні переваги адаптивного підходу були при великих обсягах даних 80 000 – 100 000 записів, де його ефективність перевищувала показники стандартних алгоритмів. Крім того, адаптивний механізм забезпечив більш прогнозовану поведінку кешу, що відображалося у низькому стандартному відхиленні отриманих результатів.

Отже, розроблені адаптивні алгоритми забезпечують суттєве покращення продуктивності, ефективне використання оперативної пам’яті та стабільність роботи системи навіть за умов значних навантажень. Практична реалізація запропонованих підходів у вигляді розширення до бібліотеки StackExchange.Redis

показала їх високу придатність для реальних високонавантажених інформаційних систем. Подальшими напрямками досліджень можуть бути додаткове покращення адаптивних механізмів, інтеграція з методами машинного навчання для більш точного прогнозування поведінки даних, а також розширення функціональних можливостей для підтримки інших структур Redis.

У додатках Б та В наведено програмний код з реалізацією роботи зі складними даними, динамічним визначенням TTL та комбінований алгоритм заміщення кешу.

Результати роботи були апробовані на наукових конференціях: «13-та Міжнародна науково-технічна конференція «Інформаційні системи та технології ІСТ-2024» 26 - 28 листопада 2024 р» (див. додаток Д) та «9th Open International Conference "Electrical, Electronic and Information Sciences“ eStream 2025» (див. додаток Е).

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Data Caching in Edge Computing: A Survey Kara, M. Ç., Elamine Benlakehal, M., Shayea, I., Tussupov, A., Rzayeva, L., 2024 IEEE 4th International Conference on Smart Information Systems and Technologies (SIST), Astana, Kazakhstan, 2024, pp. 433-439, doi: 10.1109/SIST61555.2024.10629324 (дата звернення 16.04.2025).
2. Free DBMSs Performance for an Inventory Management System based on Spring Boot Guzhov, V., Smelyakov, K., 2024 IEEE Open Conference of Electrical, Electronic and Information Sciences (eStream), Vilnius, Lithuania, 2024, pp. 1-5, doi: 10.1109/eStream61684.2024.10542597 (дата звернення 16.04.2025).
3. Top 27 In-Memory Databases Compared. 2024. URL: https://www.dragonflydb.io/guides/in-memory-databases (дата звернення 16.04.2025).
4. Analyzing and Comparison of NoSQL DBMS Kuzochkina, A., Shirokopetleva, M., Dudar, Z., 2018 International Scientific-Practical Conference Problems of Infocommunications. Science and Technology (PIC S&T), Kharkiv, Ukraine, 2018, pp. 560-564, doi: 10.1109/INFOCOMMST.2018.8632133 (дата звернення 16.04.2025).
5. Redis++: A High Performance In-Memory Database Based on Segmented Memory Management and Two-Level Hash Index Zhang, P., Xing, L., Yang, N., Tan, G., Liu, Q., Zhang, C., 2018 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Ubiquitous Computing & Communications, Big Data & Cloud Computing, Social Computing & Networking, Sustainable Computing & Communications (ISPA/IUCC/BDCloud/SocialCom/SustainCom), Melbourne, VIC, Australia, 2018, pp. 840-847, doi: 10.1109/BDCloud.2018.00125 (дата звернення 17.04.2025).
6. Research on Adaptive Cache Mechanism Based on TTL Liu, J., Wan, X., Zhu, Q., Peng, T., Hu, X., 2022 2nd International Conference on Networking, Communications and Information Technology (NetCIT), Manchester, United Kingdom, 2022, pp. 507-511, doi: 10.1109/NetCIT57419.2022.00125 (дата звернення 17.04.2025).

7. EPP-LFU: An Efficient Producer Popularity-based LFU Policy for the Applications of Named-Data Network Burhan, M., Arsalan, A., Rehman, R. A., 2022 24th International Multitopic Conference (INMIC), Islamabad, Pakistan, 2022, pp. 1-6, doi: 10.1109/INMIC56986.2022.9972919 (дата звернення 17.04.2025).

8. LearnedCache: A Locality-Aware Collaborative Data Caching by Learning Model Ma, W., Zhu, Y., Wang, S., Bao, Y., 2019 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom), Xiamen, China, 2019, pp. 718-726, doi: 10.1109/ISPA-BDCloud-SustainCom-SocialCom48970.2019.00109 (дата звернення 17.04.2025).

9. Redis Docs. 2024. URL: https://redis.io/docs/latest/ (дата звернення 18.04.2025).

10. IEEE Xplore. 2024. URL: https://ieeexplore.ieee.org/Xplore/home.jsp (дата звернення 18.04.2025).

11. ACM Digital Library. 2024. URL: https://dl.acm.org/ (дата звернення 18.04.2025).

12. ResearchGate. 2024. URL: https://www.researchgate.net/ (дата звернення 18.04.2025).

13. Google Scholar. 2024. URL: https://scholar.google.com/ (дата звернення 18.04.2025).

14. Basu, S. Adaptive TTL-Based Caching for Content Delivery / Basu, S., Sundarrajan, A., Ghaderi, J., Shakkottai, S., Sitaraman, R., in IEEE/ACM Transactions on Networking, vol. 26, no. 3, pp. 1063-1077, June 2018, doi: 10.1109/TNET.2018.2818468 (дата звернення 19.04.2025).

15. Research on Adaptive Cache Mechanism Based on TTL Liu, J., Wan, X., Zhu, Q., Peng, T., Hu, X., 2022 2nd International Conference on Networking, Communications and Information Technology (NetCIT), Manchester, United Kingdom, 2022, pp. 507-511, doi: 10.1109/NetCIT57419.2022.00125 (дата звернення 19.04.2025).

16. An Improved Cache Eviction Strategy: Combining Least Recently Used and Least Frequently Used Policies Shah, J., Siddiqui, A. A., 2023 6th International Conference on Advances in Science and Technology (ICAST), Mumbai, India, 2023, pp. 1-6, doi: 10.1109/ICAST59062.2023.10454976 (дата звернення 19.04.2025).

17. ARC-learning: A Self-tuning Cache Policy under Dynamic Popularity Shen, Z., Jiang, B., Wang, X., Zhou, C., 2022 IEEE 8th International Conference on Computer and Communications (ICCC), Chengdu, China, 2022, pp. 610-615, doi: 10.1109/ICCC56324.2022.10065823 (дата звернення 20.04.2025).

18. Investigating Multi-Tier and QoS-Aware Caching Based on ARC Ait-Oucheggou, L., Rubini, S., Battou, A., Boukhobza, J., 2023 31st International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), Stony Brook, NY, USA, 2023, pp. 1-4, doi: 10.1109/MASCOTS59514.2023.10387601 (дата звернення 20.04.2025).

19. What Are the Impacts of the Redis Expiration Algorithm?. 2024. URL: https://redis.io/kb/doc/1fqjridk8w/what-are-the-impacts-of-the-redis-expiration-algorithm (дата звернення 21.04.2025).

20. Azure Cache for Redis and operational excellence. Microsoft Learning. 2023. URL: https://learn.microsoft.com/en-us/azure/well-architected/service-guides/azure-cache-redis/operational-excellence (дата звернення 21.04.2025).

21. Аналіз алгоритмів кешування в Redis Кучапін, М., Широкопетлєва, М., Шевченко, О., Інформаційні системи та технології: матеріали 13-ї Міжнародної науково-технічної конференції. Частина 1. [Електронний ресурс], Харків, 26 - 28 листопада 2024 року / наук. ред. Ю.О. Романенков, В.В. Безкоровайний, S. Yakovlev, L. Petryshyn, З.В. Дудар, В.Г. Кобзєв. – Х.: ХНУРЕ, 2024. – с.127-130 (дата звернення 21.04.2025).

22. TTL-based Cloud Caches Carra, D., Neglia, G., Michiardi, P., EEE INFOCOM 2019 - IEEE Conference on Computer Communications, Paris, France, 2019, pp. 685-693, doi: 10.1109/INFOCOM.2019.8737546 (дата звернення 21.04.2025).

23. Zheng, Q. On the Analysis of Cache Invalidation With LRU Replacement / Zheng, Q., Yang, T., Kan, Y., Tan, X., Yang, J., Jiang, X., in IEEE Transactions on Parallel and Distributed Systems, vol. 33, no. 3, pp. 654-666, 1 March 2022, doi: 10.1109/TPDS.2021.3098459 (дата звернення 21.04.2025).

24. An arbitration on cache replacements based on frequency — Recency product values Das, S., Banerjee, A., 2016 International Conference on VLSI Systems, Architectures, Technology and Applications (VLSI-SATA), Bengaluru, India, 2016, pp. 1-6, doi: 10.1109/VLSI-SATA.2016.7593031 (дата звернення 21.04.2025).

25. Github – Архівація. URL: https://github.com/mmatvii34/master-work-archive (дата звернення 04.06.2025).

26. Enhancing Redis Cache Efficiency Based on Dynamic TTL and Adaptive Eviction Mechanism Shevchenko, O., Kuchapin, M., Dudar, Z., Shirokopetleva, M., 2025 IEEE Open Conference of Electrical, Electronic and Information Sciences (eStream), Vilnius, Lithuania, 2025, pp. 1-6, doi: 10.1109/eStream66938.2025.11016870 (дата звернення 07.06.2025).

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ ЗА НАУКОВИМИ НАПРЯМАМИ КЕРІВНИКА ТА НАУКОВЦІВ КАФЕДРИ ПРОГРАМНОЇ ІНЖЕНЕРІЇ

2. Free DBMSs Performance for an Inventory Management System based on Spring Boot Guzhov, V., Smelyakov, K., 2024 IEEE Open Conference of Electrical, Electronic and Information Sciences (eStream), Vilnius, Lithuania, 2024, pp. 1-5, doi: 10.1109/eStream61684.2024.10542597 (дата звернення 16.04.2025).

4. Analyzing and Comparison of NoSQL DBMS Kuzochkina, A., Shirokopetleva, M., Dudar, Z., 2018 International Scientific-Practical Conference Problems of Infocommunications. Science and Technology (PIC S&T), Kharkiv, Ukraine, 2018, pp. 560-564, doi: 10.1109/INFOCOMMST.2018.8632133 (дата звернення 16.04.2025).

21. Аналіз алгоритмів кешування в Redis Кучапін, М., Широкопетлєва, М., Шевченко, О., Інформаційні системи та технології: матеріали 13-ї Міжнародної науково-технічної конференції. Частина 1. [Електронний ресурс], Харків, 26 - 28 листопада 2024 року / наук. ред. Ю.О. Романенков, В.В. Безкоровайний, S. Yakovlev, L. Petryshyn, З.В. Дудар, В.Г. Кобзєв. – Х.: ХНУРЕ, 2024. – с.127-130 (дата звернення 21.04.2025).

26. Enhancing Redis Cache Efficiency Based on Dynamic TTL and Adaptive Eviction Mechanism Shevchenko, O., Kuchapin, M., Dudar, Z., Shirokopetleva, M., 2025 IEEE Open Conference of Electrical, Electronic and Information Sciences (eStream), Vilnius, Lithuania, 2025, pp. 1-6, doi: 10.1109/eStream66938.2025.11016870 (дата звернення 07.06.2025).