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

# КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Аналіз підходів кешування даних на основі високонавантаженого Python веб-додатку»

на здобуття освітнього ступеня магістра

зі спеціальності 126 Інформаційні системи та технології
*(код, найменування спеціальності)*

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

_____ Антон Ритов
*(підпис)* *Ім'я, ПРІЗВИЩЕ здобувача*

Виконав: здобувач вищої освіти гр. __
Антон Ритов
Ім'я, ПРІЗВИЩЕ

Керівник: Вадим ВЛАСЕНКО
*науковий ступінь, вчене звання* кандидат технічних наук

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

**Київ – 2024**

**ДЕРЖАВНИЙ УНІВЕРСИТЕТ**
**ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ**
**Навчально-науковий інститут інформаційних технологій**

Кафедра Інженерії програмного забезпечення автоматизованих систем

Ступінь вищої освіти Магістр

Спеціальність 126 Інформаційні системи та технлогії

Освітньо-професійна програма 126 Інформаційні системи та технлогії

**ЗАТВЕРДЖУЮ**

Завідувач кафедрою _____
___Сторчак К.П.
«____» _____ 2024 р.

**ЗАВДАННЯ**
**НА КВАЛІФІКАЦІЙНУ РОБОТУ**
_____Ритов Антон Андрійович_____
*(прізвище, ім'я, по батькові здобувача)*

1. Тема кваліфікаційної роботи: Аналіз підходів кешування даних на основі високонавантаженого Python веб-додатку
керівник кваліфікаційної роботи Вадим Власенко Кандидат технічних наук,
*(Ім'я, ПРІЗВИЩЕ, науковий ступінь, вчене звання)*
затверджені наказом Державного університету інформаційно-комунікаційних технологій від «19» 10.2023 р. № 145
2. Строк подання кваліфікаційної роботи «29» грудня 2023р.
3. Вихідні дані до кваліфікаційної роботи: науково-технічна література, алгоритми кешування даних в розподілених системах, метрики продуктивності, сценарії тестування продуктивності
4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
4.1. Аналіз предметної області
4.2. Алгоритми, методи та технології кешування даних в розподілених системах
4.3. Проектування та розробка програмної системи з використанням Python
4.4 Порівняльний аналіз підходів кешування
5. Перелік ілюстративного матеріалу: презентація
6. Дата видачі завдання«19» жовтня 2023 р.

# КАЛЕНДАРНИЙ ПЛАН

| № з/п | Назва етапів кваліфікаційної роботи | Строк виконання етапів роботи | Примітка |
|---|---|---|---|
| 1 | Аналіз наявної науково-технічної літератури | 19.10 - 05.11.23 | виконано |
| 2 | Аналіз існуючих технологій та методів кешування даних | 05.11 - 12.11.23 | виконано |
| 3 | Дослідження шаблонів доступу до даних при кешуванні | 13.11 - 19.11.23 | виконано |
| 4 | Дослідження існуючих технологій серверного кешування даних | 20.11 - 25.11.23 | виконано |
| 5 | Проєктування, розробка та тестування програмної системи для порівняння продуктивності підходів кешування | 27.11 - 03.12.23 | виконано |
| 6 | Формування результатів проведення тестувань підходів кешування | 04.12 - 10.12.23 | виконано |
| 7 | Оформлення роботи: вступ, висновки, реферат | 11.12 - 20.12.23 | виконано |
| 8 | Розробка демонстраційних матеріалів | 21.12 - 29.12.23 | виконано |

Здобувач вищої освіти _____ Антон Ритов
*(підпис)* *(Ім'я, ПРІЗВИЩЕ)*

Керівник
кваліфікаційної роботи _____ Вадим Власенко
*(підпис)* *(Ім'я, ПРІЗВИЩЕ)*

# РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня магістра: 74 стор., 3 табл., 52 рис., 29 джерел.

*Мета роботи* – аналіз та оцінка різних підходів до кешування даних у високонавантажених веб-додатках та ідентифікації оптимальних стратегій кешування для забезпечення швидкої та надійної роботи веб-додатків.

*Об'єкт дослідження* – високонавантажені Python веб-додатки.

*Предмет дослідження* – методи та підходи кешування даних у контексті оптимізації продуктивності високонавантажених Python веб-додатків.

*Короткий зміст роботи*: У роботі проведено дослідження серверних технологій кешування. Проаналізовано основні алгоритми та політики систем кешування даних. Проведено порівняльний аналіз існуючих технологій кешування. Спроєктовано та розроблено Python високонавантажений веб-додаток. Протестовано та порівняно продуктивність технологій кешування.

КЛЮЧОВІ СЛОВА: КЕШУВАННЯ, IN-MEMORY, MEMCACHED, REDIS, ОПТИМІЗАЦІЯ, ПРОДУКТИВНІСТЬ ВЕБ-ДОДАТКІВ.

## ABSTRACT

The text part of the qualification work for obtaining a master's degree: 74 pages, 3 tables, 52 figures, 29 sources.

*The purpose of the work* is to analyze and evaluate different approaches to data caching in highly loaded web applications and to identify optimal caching strategies to ensure fast and reliable operation of web applications.

*The object of the research* is highly loaded Python web applications.

*The subject of research* is data caching methods and approaches in the context of optimizing the performance of highly loaded Python web applications.

*Summary of the work:* The research of server caching technologies is carried out in the work. The main algorithms and policies of data caching systems are analyzed. A comparative analysis of existing caching technologies was conducted. Designed and developed a Python high-load web application. The performance of caching technologies is tested and compared.



KEY WORDS: CACHING, IN-MEMORY, MEMCACHED, REDIS, OPTIMIZATION, WEB APPLICATION PERFORMANCE.

**ЗМІСТ**

## ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

API – Application Programming Interface (інтерфейс програмування застосунків)

LRU – Least Recently Used (найменш нещодавно використаний)

MRU – Most Recently Used (нещодавно використаний)

TTL – Time to Live (час життя)

LFU – Least Frequently Used (найменш часто використаний)

HTTP – HyperText Transfer Protocol (протокол передачі гіпертекстових документів)

IOPS – Input-output operations per second (операції вводу-виводу в секунду)

CRUD – Create, Read, Update, Delete (операції створення, зчитування, оновлення, видалення)

FIFO – First-In-First-Out (Перший прийшов - перший вийшов)

CDN – Content Delivery Network (мережа доставки контенту)

# ВСТУП

**Актуальність дослідження.** У сучасну епоху інформаційного суспільства значний акцент приділяється проведенню численних досліджень, спрямованих на підвищення якості обслуговування користувачів інформаційних технологій. Зокрема, дослідження [1, 2], зосереджені на веб-технологіях, виявили цікаву кореляцію: коли веб-сайти завантажуються швидше, користувачі прагнуть досліджувати більше сторінок. Отже, у міру розширення бази користувачів стає обов'язковим вивчення методологій, які можуть оптимізувати процеси обробки запитів і доставки даних.

Для задовлення цієї потреби було впроваджено нові технології та підходи. Сюди входить створення швидкої інфраструктури та використання високошвидкісних служб хостингу, механізми балансування трафіку, розподілені мережі доставки вмісту, методи оптимізації зображень та ефективне кешування HTTP заголовків. Кешування даних стало переважаючим рішенням для вирішення цієї проблеми в наш час.

Концепція передбачає тимчасове зберігання даних у проміжному буфері, таким чином значно підвищуючи швидкість доступу до цих даних.

У галузі інформаційних технологій кеш-пам'ять служить швидким і тимчасовим шаром зберігання, який містить певний набір даних. Його призначення полягає в тому, щоб пришвидшити процеси пошуку даних, дозволяючи швидший доступ порівняно з основним місцем зберігання. Кеш – це життєво важливий компонент, який використовується на багатьох технологічних рівнях, включаючи операційні системи, мережеві рівні, CDN, DNS, інтернет-програми та бази даних. Його використання дає помітні переваги, такі як значне зменшення затримки та покращена продуктивність IOPS, особливо для робочих навантажень, які значною мірою покладаються на програми, що інтенсивно читають. Приклади таких навантажень включають портали запитань і відповідей, ігрові ресурси, портали розповсюдження медіа

та портали соціальних мереж. У рамках кешування існує безліч елементів, які можна кешувати для оптимізації продуктивності.

У сучасному світі високонавантажені веб-додатки є необхідною складовою багатьох сфер життя, починаючи від електронної комерції і закінчуючи соціальними мережами та фінансовими сервісами. Однією з ключових вимог до таких додатків є висока швидкодія та надійність, яка вимагає ефективного керування обробкою даних. В цьому контексті кешування даних стає важливим інструментом для оптимізації роботи веб-додатків. Наприклад, результати запитів до бази даних, ресурсомісткі обчислення, запити та відповіді API можна кешувати. Крім того, робочі навантаження, які передбачають значну обчислювальну потужність, такі як високопродуктивне обчислювальне моделювання або сервіси рекомендацій, можуть ефективно використовувати рівень даних у пам’яті як кеш. Ці програми часто вимагають доступу в режимі реального часу до масивних наборів даних, поширених у кластерах із сотень машин. Використання кешу є корисним для таких програм, оскільки керування наборами даних і їх зберігання на дисковому сховищі зазвичай створює вузьке місце через відносно повільну швидкість основного базового обладнання.

Отже, на основі вищезазначеного можна зробити висновок, що тема дослідження магістерскої кваліфікаційної роботи є актуальною.

**Метою магістерскої кваліфікаційної роботи** є аналіз та оцінка різних підходів до кешування даних у високонавантажених веб-додатках та ідентифікації оптимальних стратегій кешування для забезпечення швидкої та надійної роботи веб-додатку на мові Python в умовах високого навантаження.

**Об'єкт дослідження** – високонавантажені Python веб-додатки.

**Предмет дослідження** – методи та підходи кешування даних у контексті оптимізації продуктивності високонавантажених Python веб-додатків.

.

# 1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

## 1.1 Теоретичні основи кешування даних

Одним із ключових атрибутів, які зазвичай зустрічаються в комп’ютеризованих системах, є здатність швидко отримувати інформацію, необхідну кінцевому користувачеві, на основі конкретних критеріїв, а також мінімізувати навантаження на взаємопов’язані системи.

Кешування є критичним фактором для досягнення оптимальної продуктивності в сучасних системах. Його ефективне впровадження відіграє ключову роль у досягненні високого рівня продуктивності. Навіть процесори, які виконують інструкції, мають декілька рівнів пам’яті, які відрізняються за швидкістю та ємністю. Ці рівні пам’яті багаторівневі, причому кожен наступний рівень повільніший, але здатний зберігати більші обсяги даних.

Кешування можна описати як процес тимчасового зберігання часто використовуваних даних у спеціальній області зберігання, відомої як кеш-пам'ять. Основною метою кешування є підвищення продуктивності як програм, так і систем шляхом скорочення часу доступу до даних.

Замість того, щоб отримувати дані з вихідного джерела, що може бути повільнішим процесом, система може спочатку перевірити кеш, коли буде зроблено запит даних. Якщо запитувані дані вже збережені в кеші, їх можна швидко отримати та надати користувачеві. З іншого боку, якщо дані не знайдено в кеші, система отримає їх із вихідного джерела та збереже в кеші для подальшого використання. Таким чином, наступного разу, коли дані запитуються, вони можуть бути подані безпосередньо з кешу, що забезпечує швидший доступ порівняно з отриманням із вихідного джерела.

Підсумовуючи, кешування оптимізує доступ до даних, використовуючи область тимчасового зберігання, яка зберігає часто використовувані дані, тим самим покращуючи загальну продуктивність системи та програм.

Існують різні форми кешування, які використовуються в різних системах. До них належать кешування пам’яті, кешування в пам’яті та кешування диска. Кешування пам’яті передбачає збереження даних у системному кеші, що є значно швидшим, ніж доступ до даних із дискового сховища. З іншого боку, кешування в пам'яті перевершує кешування пам'яті з точки зору швидкості, оскільки воно зберігає дані безпосередньо в оперативній пам'яті системи. Хоча кешування диска повільніше, ніж кешування пам’яті, воно має перевагу в тому, що може вмістити більшу кількість даних. Кешування може відбуватися на кількох рівнях інфраструктури, таких як веб-браузер, веб-сервер, CDN (Content Delivery Network, мережа доставки вмісту) і вихідний сервер. Веб-браузери, наприклад, використовують методи кешування для зберігання HTML, зображень і коду, щоб зменшити кількість запитів, зроблених до веб-сервера [2].

Подібним чином веб-сервери використовують механізми кешування, щоб зменшити навантаження на процесор і підвищити загальну продуктивність програми. CDN, з іншого боку, використовують кешування, щоб зменшити затримку та забезпечити покращену взаємодію з користувачем.

Вихідні сервери також використовують кешування, щоб зменшити навантаження на внутрішні сервери та оптимізувати продуктивність програми. API також використовують кешування для підвищення продуктивності. Коли ініціюється запит API, система може перевірити кеш, щоб визначити, чи бажана відповідь уже зберігається там.

Якщо відповідь знайдено в кеші, система може отримати та надати її без необхідності повторної обробки вихідного запиту. Цей механізм кешування значно покращує загальну ефективність і швидкість реагування API [3].

### 1.2 Технології кешування даних

Технології кешування можна розділити на чотири різні групи, а саме: кешування в пам’яті, кешування проксі, кешування CDN і кешування

браузера. Кешування в пам’яті передбачає зберігання даних, до яких часто звертаються, в енергозалежній пам’яті, як-от DRAM, щоб мінімізувати час отримання з повільніших пристроїв зберігання. Ця конкретна технологія знаходить застосування в різних сценаріях, включаючи керування сеансами, сховища даних ключ-значення та бази даних NoSQL. Реалізація кешування в пам’яті дає значні переваги, оскільки значно скорочує час відповіді програми та покращує загальну взаємодію з користувачем. Завдяки тимчасовому зберіганню часто запитуваних даних у пам’яті швидкого доступу потреба отримувати їх із повільніших носіїв пам’яті мінімізується, що призводить до швидшого пошуку даних і покращення продуктивності програми. На рисунку 1.1 представлена архітектура застосунку з використанням In-Memory Caching.

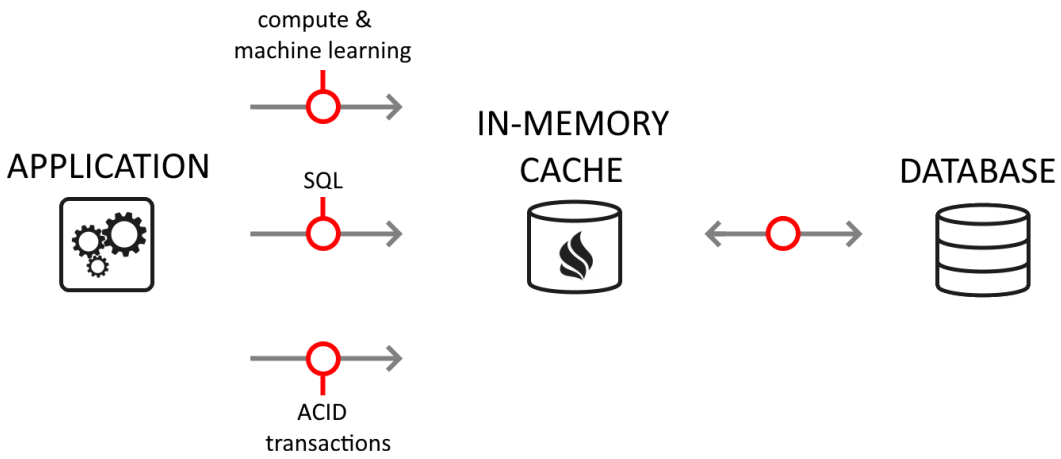



Рисунок 1.1 – Архітектура застосунку з використанням In-Memory Cache [4]

Проксі-кешування – це техніка, яка передбачає зберігання даних, до яких часто звертаються, на проміжному сервері, відомому як проксі. Цей сервер знаходиться між клієнтом, який ініціює запити даних, і основним сервером, який зберігає запитувані дані. Коли клієнт надсилає запит на дані, проксі-сервер спочатку перевіряє свій кеш, щоб визначити, чи збережена запитана

інформація. Якщо дані доступні в кеші, проксі-сервер безпосередньо доставляє їх клієнту, усуваючи необхідність пересилати запит на головний сервер. Основною метою кешування проксі-сервера є підвищення продуктивності програм шляхом мінімізації споживання пропускної здатності та скорочення часу відповіді сервера. Зберігаючи дані, які часто запитуються, завжди доступними на проксі-сервері, це позбавляє від необхідності обслуговувати наступні запити основним сервером, зменшуючи трафік.

Кешування CDN – це техніка, за якої дані, до яких часто звертаються, зберігаються на численних серверах, стратегічно розташованих у різних місцях по всьому світу. Коли користувач запитує дані, сервер мережі доставки вмісту (CDN), розташований найближче до користувача, відповідає запитуваною інформацією. Основною перевагою кешування CDN є його здатність підвищувати продуктивність програми за рахунок скорочення часу відповіді та мінімізації використання пропускної здатності сервера [4].

Кешування веб-браузера – це метод, який дозволяє зберігати веб-ресурси, такі як HTML-сторінки, зображення, стилі та сценарії, на комп’ютері користувача під час доступу до веб-сайтів.

На рисунку 1.2 представлений алгоритм кешування даних у браузері.

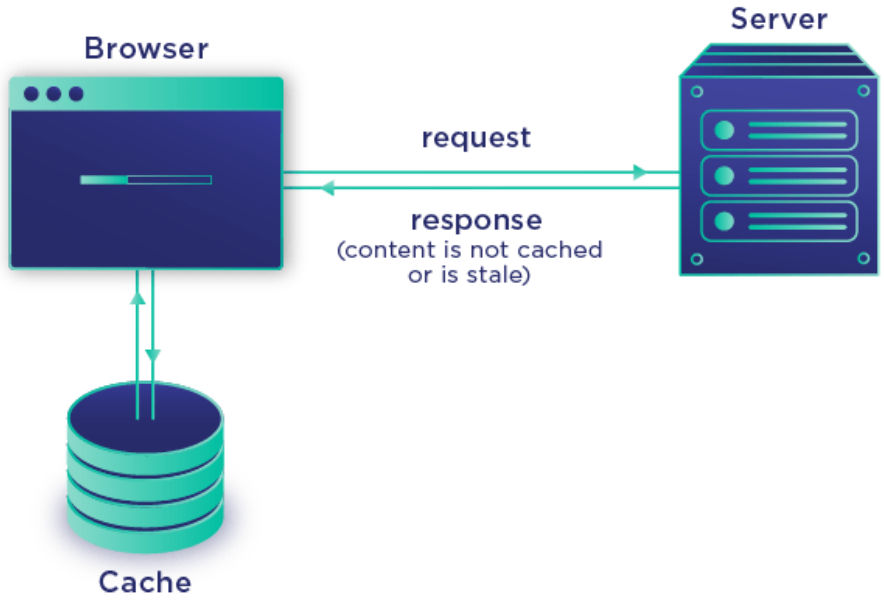



Рисунок 1.2 – Алгоритм кешування даних у веб-браузері [4]

Ця техніка корисна для прискорення процесу завантаження веб-сторінок і ресурсів під час повторного відвідування веб-сайтів, оскільки браузер може використовувати локально збережені копії замість того, щоб завантажувати їх знову з сервера. Браузери виділяють визначений простір для зберігання кешованих ресурсів, який зазвичай має обмеження щодо розміру та тривалості зберігання. Ідентифікація Кешовані ресурси ідентифікуються за допомогою ключів, які можуть бути URL-адресами або іншими ідентифікаторами, і пов’язані з конкретними запитами ресурсів. Кешовані ресурси мають тривалість життя або період очікування, після якого вони вважаються застарілими та потребують оновлення з сервера. Сервери можуть надсилати заголовок HTTP "Cache-Control", щоб надати браузеру інструкції щодо кешування певного ресурсу. Наприклад, «no-cache» означає, що ресурс не слід кешувати. Кешування браузера може бути організоване в ієрархічній структурі, де деякі ресурси зберігаються локально на комп’ютері користувача, а інші зберігаються на проксі-серверах або в мережах доставки вмісту (CDN).

### 1.3 Актуальність теми дослідження

Кешування даних має величезний потенціал для програм, оскільки пропонує безліч переваг, покращуючи продуктивність, мінімізуючи накладні витрати та збільшуючи пропускну здатність. Важливість кешування полягає в його здатності значно оптимізувати роботу програми. Це досягається за рахунок використання переваги швидкості читання даних із кешу в пам’яті, що перевершує отримання даних із дискового сховища. Зберігаючи дані, до яких часто звертаються, в оперативній пам’яті, кешування ефективно зменшує затримку, пов’язану з доступом до повільніших довготривалих пристроїв зберігання даних. Зрештою, це не тільки покращує взаємодію з користувачем, але й підвищує ефективність критичних бізнес-операцій, що призводить до покращення загальної продуктивності та продуктивності. Кешування відіграє вирішальну роль у мінімізації навантаження на бази даних. Він досягає цього,

зберігаючи дані, до яких часто звертаються, у пам’яті, тим самим зменшуючи потребу в повторному вилученні з бази даних. Отже, це зменшує навантаження на сервер бази даних, що призводить до зниження використання бази даних і пов’язаних з цим витрат. Крім того, кешування допомагає підвищити пропускну здатність, яка вказує на обсяг даних, які система може обробити протягом певного періоду часу. Зберігаючи дані, до яких часто звертаються, у пам’яті, кешування скорочує час, необхідний для отримання даних із бази даних або інших пристроїв зберігання. Це, у свою чергу, підвищує загальну пропускну здатність програми, забезпечуючи оптимальну продуктивність. Коли мова заходить про кешування, існують різні форми, такі як веб-кеш, розподілений кеш і кеш-пам’ять. Відомі рішення для кешування, такі як Redis і Memcached, є кращим вибором. Крім того, мережі доставки контенту (CDN) також використовують методи кешування, зберігаючи контент, до якого часто звертаються, у географічно рознесених місцях. Ця стратегія не тільки сприяє швидкому завантаженню, але й забезпечує захист від кібератак. Таким чином, кешування є потужною технікою, яка пропонує численні переваги для програм. Це покращує продуктивність, знижує витрати та максимізує використання пропускної здатності, забезпечуючи таким чином оптимальну швидкість, ефективність і надійність роботи програмних систем

На основі наведеного в першому розділі аналізу можна зробити висновки, що кешування – це важливий атрибут комп'ютеризованих систем, який допомагає підвищити продуктивність та зменшити навантаження на системи. Це процес тимчасового зберігання часто використовуваних даних у спеціальній області пам'яті, що дозволяє швидше отримувати доступ до них. Кешування може бути застосоване на різних рівнях інфраструктури, включаючи веб-браузери, веб-сервери, CDN та API, що збільшує продуктивність програм, зменшує витрати і підвищує пропускну здатність.

# 2 АЛГОРИТМИ, МЕТОДИ ТА ТЕХНОЛОГІЇ КЕШУВАННЯ ДАНИХ В РОЗПОДІЛЕНИХ СИСТЕМАХ

## 2.1 Шаблони доступу до даних при кешуванні

Використання певного шаблону для доступу до даних відіграє вирішальну роль у встановленні зв’язку між джерелом даних і рівнем кешування. Тому вкрай важливо переконатися, що цей аспект обробляється належним чином, оскільки він може мати значний вплив на продуктивність кешування. Існують різні підходи до кешування даних, такі як Cache-Aside, Read-Through, Write-Back і Write-Through, кожен із яких має свої переваги та недоліки. Важливо розуміти, що ці підходи можна використовувати окремо або в комбінації, залежно від бажаного рівня ефективності. Далі, буде детальніше розглянуто кожен з вищезазначених шаблонів доступу до даних.

### 2.1.1 Шаблон Cache-Aside

У шаблоні Cache-Aside дані кешуються вибірково, лише коли це необхідно. Це означає, що дані тимчасово зберігаються в кеші та використовуються, коли це потрібно. Коли робиться запит для певного елемента даних і він уже існує в кеші, дані витягуються з кешу та надаються клієнту, який зробив запит. Однак, якщо запитані дані не знайдено в кеші, програма потім отримує дані з призначеного джерела даних, наприклад бази даних, і повертає їх клієнту. Згодом ці дані зберігаються в кеш-пам’яті для подальшого використання, що забезпечує швидший час відповіді на наступні запити, які потребують тих самих даних.

На рисунку 2.1 представлена візуалізація робота шаблону Cache-Aside.

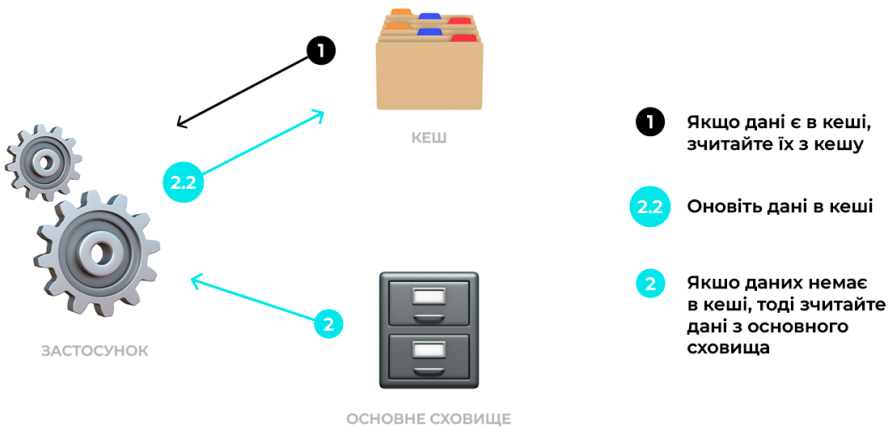



Рисунок 2.1 – Приклад роботи шаблону Cache-Aside [4]

Шаблон Cache-Aside пропонує кілька переваг. По-перше, це гарантує, що в кеші зберігаються лише запитані дані, уникаючи накопичення невикористаних даних. Цей шаблон особливо добре підходить для робочих процесів, які передбачають інтенсивні операції читання, де дані переважно читаються кілька разів, перш ніж відбуваються будь-які оновлення, якщо взагалі відбуваються. Однією з ключових переваг шаблону Cache-Aside є його стійкість до промахів кешу. Якщо кеш-пам’ять недоступна, система безперебійно отримує дані з базового сховища даних. Однак важливо зазначити, що тривалий період збою кешу може призвести до збільшення затримки. Ще одна перевага полягає в тому, що модель даних у кеші не обов’язково має відображати модель у базі даних. Це забезпечує гнучкість зберігання результатів кількох запитів під одним ідентифікатором у кеші, оптимізуючи використання пам’яті.

З іншого боку, є кілька недоліків, які слід враховувати. Коли відбувається пропуск кешу, це може призвести до збільшення затримки, оскільки потрібно виконати більше роботи порівняно з прямим запитом даних. Крім того, шаблон Cache-Aside не гарантує негайної узгодженості між базою даних і кешем. Якщо дані оновлюються в базі даних, може виникнути

затримка у відображенні цих змін у кеші, що може призвести до того, що програма обслуговуватиме застарілі дані.

Щоб пом’якшити цю проблему, шаблон «Cache-Aside» часто поєднується зі стратегією «наскрізного запису». Ця стратегія передбачає одночасне оновлення даних як у базі даних, так і в кеші, гарантуючи, що кешовані дані залишаються актуальними та відповідають БД. Застосовуючи цей підхід, ризик надання застарілих даних із кешу значно зменшується.

### 2.1.2 Шаблон Read-Through

Підхід Read-Through подібний до Cache-Aside, його основна ціль – допомогти при частому читанні з основного сховища [6].

Однак між ними є дві чіткі відмінності. По-перше, у підході Read-Through програма постійно отримує доступ до кешу. Це означає, що зв’язок із основним сховищем перенаправляється на рівень провайдера кешу, який зазвичай є окремою бібліотекою. По-друге, і кеш, і основне сховище містять ту саму модель даних, що забезпечує узгодженість між ними. На рисунку 2.2 представлений приклад роботи шаблону доступу до даних Read-Through.

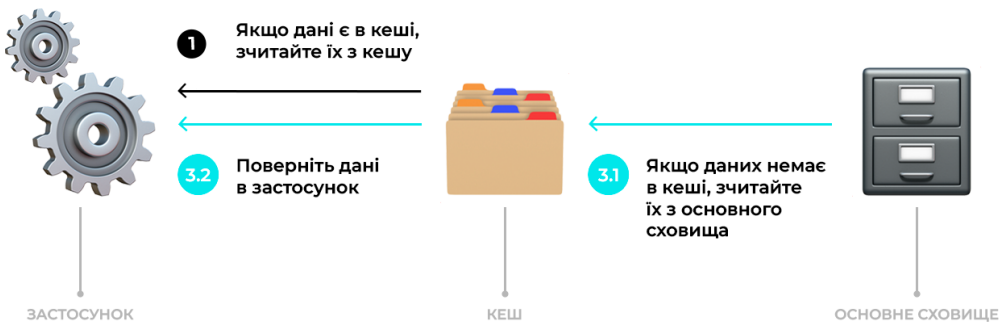



Рисунок 2.2 – Приклад роботи шаблону Read-Through [4]

### 2.1.3 Шаблон Write-back Cache

Кеш-пам'ять зазвичай пов'язують із пошуком даних. Однак у цьому конкретному підході (Write-Back Cache) це служить додатковій меті сприяння ефективнішому запису в основне сховище. Одна відома техніка, яка використовується в кешуванні даних, відома як зворотний запис. За допомогою зворотного запису будь-які зміни, внесені в кеш, не зберігаються відразу в основній пам’яті. Замість цього вони зберігаються в кеші, доки не буде зроблено конкретний запит на збереження цих змін в основну пам’ять. Такий підхід забезпечує кращу ефективність загальної системи зберігання. На рисунку 2.3 представлений приклад шаблону доступу до даних Write-Back.

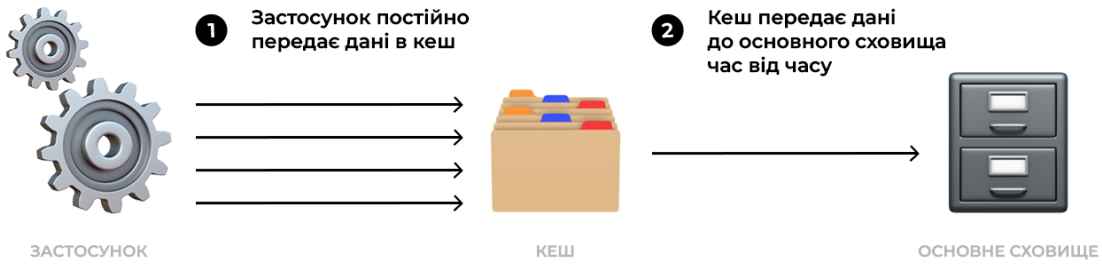



Рисунок 2.3 – Приклад роботи шаблону Write-Back Cache [4]

Цей підхід має кілька переваг. По-перше, дані в кеші завжди актуальні, оскільки вони синхронізуються з базою даних після кожної операції запису. Це особливо корисно в системах, які не можуть терпіти застарілі або застарілі дані кешу. Однак є також деякі недоліки, які слід враховувати. Під час запису даних за допомогою техніки зворотного запису виникає додаткова затримка,

оскільки вимагається більше роботи. Дані спочатку повинні бути записані в базу даних, а потім у кеш, що може призвести до затримки. Крім того, якщо рівень кешу стає недоступним, будь-яка операція запису не вдасться. Іншим потенційним недоліком є накопичення непотрібних даних у кеші, які можуть ніколи не бути прочитаними. Це може призвести до марної витрати ресурсів. Щоб пом’якшити цю проблему, рекомендується поєднати шаблон зворотного запису з шаблоном «Cache-Aside» або застосувати політику часу життя (TTL).

### 2.1.4 Шаблон Write-Through

У контексті стратегії Write-Through рівень кешу бере на себе роль основного сховища даних для програми. Це означає, що будь-які нові або змінені дані безпосередньо додаються або оновлюються в кеші, звільняючи основне сховище даних від цієї відповідальності. На рисунку 2.4 представлений приклад шаблону доступу до даних Write-Through.

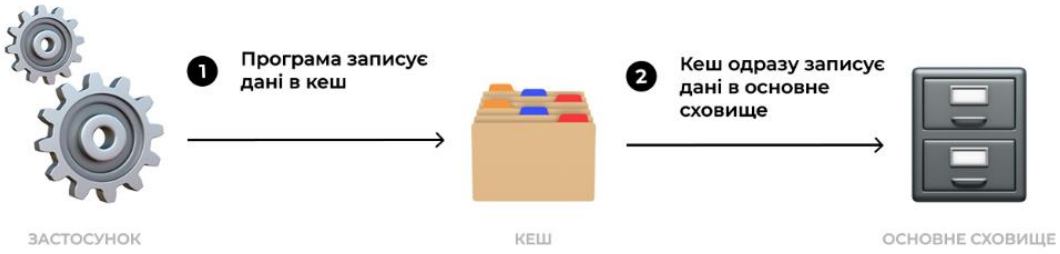



Рисунок 2.4 – Приклад роботи шаблону Write-Through Cache [4]

Щоб забезпечити узгодженість, обидві операції запису виконуються в одній транзакції, запобігаючи будь-яким конфліктам між кешованими даними та основною базою даних. Цей підхід забезпечує ефективне керування даними та зменшує навантаження на основне сховище даних, оскільки рівень кешу бере на себе завдання синхронізації даних із базою даних.

Однією з переваг цього підходу є те, що дані в кеші завжди залишатимуться актуальними завдяки синхронізації з базою даних після кожної операції запису. Цей конкретний шаблон виявляється корисним для систем, які не можуть дозволити собі мати застарілу інформацію кеша.

Однак слід враховувати деякі недоліки використання Write-Through. По-перше, процес запису даних викликає затримку, оскільки вимагає додаткової роботи: спочатку дані потрібно записати в базу даних, а потім у кеш. Крім того, операція запису не вдасться виконати, якщо рівень кешу стає недоступним, що створює потенційну проблему. Іншим недоліком є те, що кеш може накопичувати дані, які насправді ніколи не читаються, що призводить до марної витрати ресурсів. Щоб вирішити цю проблему, доцільно поєднати цей шаблон із шаблоном «Cache-Aside» або реалізувати політику на основі часу для даних кешу, відому як «Time-to-Live» (TTL). Впроваджуючи ці заходи, можна уникнути непотрібного накопичення невикористаних даних.

### 2.1.5 Шаблон Refresh-Ahead

У шаблоні Refresh-Ahead практика оновлення кешованих даних, до яких часто звертаються, відбувається до закінчення терміну їх дії. Цей процес відбувається асинхронно, що означає, що програма не зазнає негативного впливу затримок під час отримання об’єкта зі сховища даних у разі закінчення терміну дії. Проактивно оновлюючи кешовані дані, програма гарантує, що вона завжди матиме доступ до актуальної інформації, запобігаючи будь-яким потенційним уповільненням або перервам у процесі пошуку. На рисунку 2.5 представлений приклад шаблону доступу до даних Refresh-Ahead.

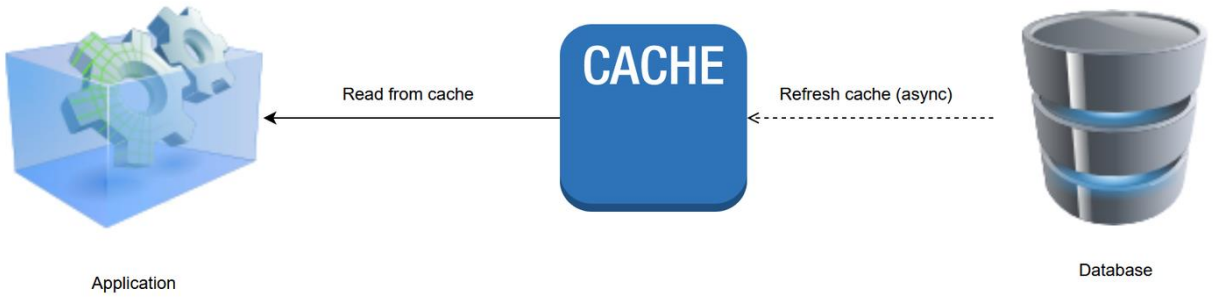



Рисунок 2.5 – Приклад роботи шаблону Refresh-Ahead Cache [5]

У реалізації цього шаблону є кілька переваг. По-перше, це особливо корисно, коли читання даних зі сховища даних потребує ресурсів. Завдяки активному оновленню кешу мінімізується потреба у дорогому пошуку бази даних. Крім того, шаблон Refresh-Ahead Cache допомагає підтримувати синхронізацію часто використовуваних записів кешу, гарантуючи, що вони завжди актуальні. Це особливо корисно в робочих навантаженнях, де низька затримка має вирішальне значення, наприклад веб-сайти, які відображають результати спортивних змагань у прямому ефірі, або інформаційні панелі фінансових бірж. Однак є також деякі недоліки, які слід враховувати. Передбачення того, які елементи кешу знадобляться в майбутньому, є важливим для ефективного функціонування цього шаблону. Якщо кеш неточно передбачає елементи, які будуть потрібні, це може призвести до непотрібних читань із бази даних, що може вплинути на продуктивність.

## 2.2 Алгоритми та політики кешування даних

Алгоритми кешування, також відомі як алгоритми виключення або алгоритми/політики заміни, відіграють вирішальну роль в оптимізації виконання інструкцій. Вони реалізовані як спеціальні комп’ютерні програми або апаратно підтримувані структури, які ефективно керують інформацією, що

зберігається в кеш-пам’яті комп’ютерної системи. Коли кеш-пам’ять заповнюється, ці алгоритми повинні приймати рішення про те, які дані потрібно видалити, щоб отримати нову та більш релевантну інформацію.

Частота звернень до кешу означає частоту, з якою потрібні дані знаходяться в кеші. Щоб підвищити відсоток звернень у заданому розмірі кешу, більш ефективні політики вигнання відстежують доступ до інформації, яка найчастіше використовується. З іншого боку, «затримка» кешу визначає швидкість, з якою кеш може негайно повернути запитувані дані у разі потрапляння в кеш. Стратегії швидшого вилучення зазвичай зосереджуються на відстеженні інформації, що використовується найменше, або, у випадку кеша з прямим відображенням, на виявленні відсутньої інформації, щоб мінімізувати час, необхідний для оновлення даних. Важливо зазначити, що кожна стратегія виселення передбачає компроміс між частотою звернень і затримкою. Ретельно вибираючи та впроваджуючи найбільш відповідний алгоритм, практики можуть знайти баланс між цими двома факторами та оптимізувати загальне функціонування кеш-системи. Вирішуючи, яку політику видалення застосувати, важливо пам’ятати, що універсальний підхід не завжди підходить для кожного запису в кеші. У випадках, коли отримання кешованого об’єкта зі сховища даних є дорогою операцією, може виявитися корисним зберегти цей елемент у кеші, незалежно від необхідності його видалення. Крім того, може знадобитися поєднати кілька політик видалення, щоб отримати найефективніше рішення для конкретного випадку використання. Ретельно враховуючи ці фактори, ви можете переконатися, що ваша політика видалення відповідає вимогам і оптимізує продуктивність.

Підводячи підсумок, алгоритми кешування, які також називають політиками виключення, допомагають ефективно керувати кешами. Їхня мета – визначити, які дані слід вилучити з кешу, щоб звільнити місце для нової інформації, беручи до уваги такі фактори, як частота звернень і затримка [6].

### 2.2.1 Алгоритм LRU

Алгоритм видалення елементів LRU (Least Recently Used), які використовувалися найменше, працює за пріоритетом видалення елементів, до яких нещодавно не зверталися. Однак ефективне впровадження цього алгоритму передбачає накладні витрати на відстеження використання кожного елемента, що може бути дорогим, якщо метою є постійне видалення елемента, який використовувався найменше. Загалом, загальний підхід передбачає підтримку «вікових бітів» для кеш-рядків і визначення найменш використовуваного кеш-рядка на основі цих вікових бітів. Ця реалізація гарантує, що щоразу, коли здійснюється доступ до рядка кешу, відповідним чином регулюється вік усіх інших рядків кешу. Використовуючи цю техніку, алгоритм успішно визначає та відкидає елемент, до якого останнім часом зверталися найменше. На рисунку 2.6 представлений приклад роботи алгоритму LRU, де послідовність доступу є наступною: A B C D E D F.

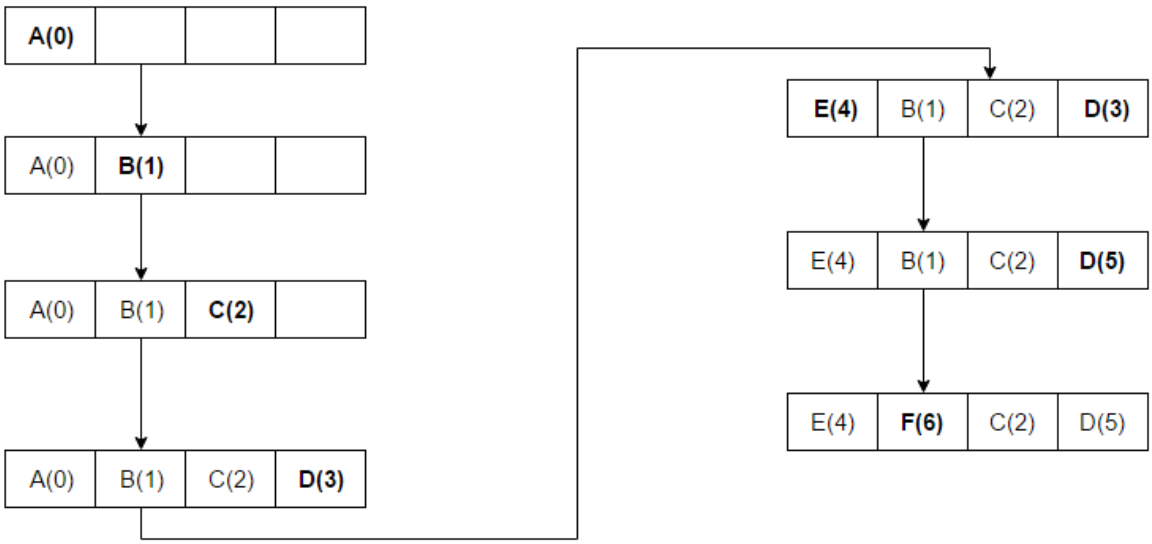



Рисунок 2.6 – Приклад роботи алгоритму LRU (Least Recently Used) [7]

Алгоритм працює наступним чином: спочатку блоки A, B, C і D встановлюються в пам'ять з порядковими номерами, присвоєними кожному блоку (збільшення порядкового номера для кожного нового доступу). Коли здійснюється доступ до блоку E, виявляється, що він є промахом, що вказує на те, що його на даний момент немає в пам’яті. Таким чином, згідно з алгоритмом LRU, блок із найнижчим рангом, яким у даному випадку є A з рангом A(0), буде замінено блоком E. Рухаючись вперед, на передостанньому кроці блок D здійснюється доступ, що призводить до оновлення його порядкового номера. Нарешті здійснюється доступ до блоку F, який замінює блок B, який мав найнижчий ранг, зокрема B(1), на той момент.

### 2.2.2 Алгоритм MRU

На відміну від алгоритму найменшого використання (LRU), алгоритм найновішого використання (MRU) дотримується іншого підходу, спочатку відкидаючи елементи, до яких зверталися останнім часом. Цю відмінність підкреслили Чоу та ДеВітт під час їхньої презентації на 11-й конференції VLDB [10]. Їхні дослідження показали, що у випадках, коли файл багаторазово сканується в циклічному послідовному шаблоні посилання, алгоритм MRU виявляється найефективнішим алгоритмом заміни.

Спираючись на це, інші дослідники представили свої висновки на 22-й конференції VLDB [11], показавши, що алгоритми кешу MRU демонструють вищу частоту звернень, ніж алгоритми LRU, для шаблонів довільного доступу та повторного сканування великих наборів даних, які зазвичай називають шаблонами циклічного доступу. Цю перевагу можна пояснити тенденцією MRU зберігати старі дані, що дозволяє підвищити продуктивність у сценаріях, коли ймовірність доступу до елемента зростає з його віком.

Важливо відзначити, що алгоритми MRU особливо корисні в ситуаціях, коли частота доступу до елемента прямо корелює з його віком. Це означає, що чим старіший елемент, тим більша ймовірність доступу до нього.

Щоб проілюструвати цю концепцію, розглянемо надану послідовність доступу: A, B, C, D, E, C, D, B, яка представлена на рисунку 2.7.

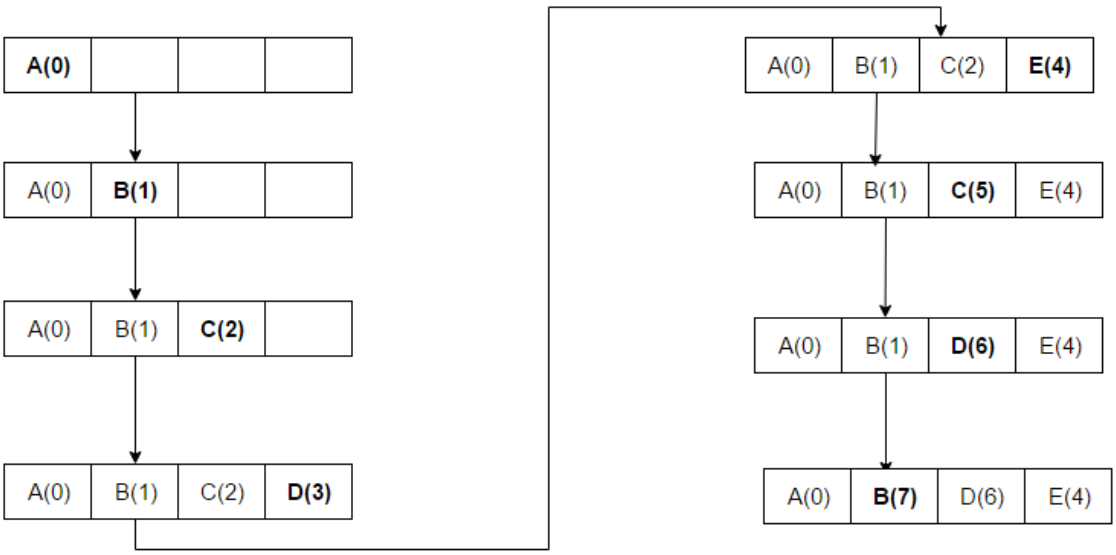



Рисунок 2.7 – Приклад роботи алгоритму MRU (Most Recently Used) [10]

У кеші елементи A, B, C і D спочатку зберігаються, оскільки ще є вільний простір. Однак після 5-го доступу вміст блоку D замінюється на E, оскільки E є останнім використовуваним елементом. Згодом, коли є інший доступ до C, він потім замінюється на D, оскільки C був елементом, до якого зверталися безпосередньо перед D. Цей процес продовжується, причому блок, до якого був доступ раніше, замінюється в міру доступу до нових елементів.

### 2.2.3 Алгоритм Pseudo-LRU

Для кеш-пам’яті процесора з високим рівнем асоціативності, який зазвичай перевищує 4 канали, вартість реалізації політики найменшого використання (LRU) стає непрактичною та непомірно високою. У результаті багато розробників ЦП обирають алгоритм псевдо LRU (PLRU), який

пропонує альтернативний підхід, який вимагає лише одного біта на елемент кешу для ефективного функціонування. Хоча PLRU може незначно впливати на коефіцієнт пропусків кешу, він пропонує покращену затримку, зменшене енергоспоживання та менші витрати порівняно з політикою LRU. Щоб краще зрозуміти, як працює PLRU, розглянемо приклад. PLRU використовує двійкову структуру дерева, де кожен вузол містить 1-бітний вказівник, що вказує на піддерево, яке використовувалося рідше. Дотримуючись ланцюжка вказівників від кореневого вузла до кінцевого вузла, можна визначити елемент кешу, який буде замінено. Коли відбувається доступ, усі вказівники в ланцюжку, починаючи з листового вузла, що відповідає елементу, до якого здійснюється доступ, будуть оновлені, щоб вказувати на піддерево, яке не включає елемент, до якого здійснюється доступ. Цей механізм забезпечує ефективну заміну та керування елементами кешу, демонструючи практичність та ефективність алгоритму PLRU у розробці кешу ЦП.

На рисунку 2.8 представлений принцип роботи алгоритму Pseudo-LRU.

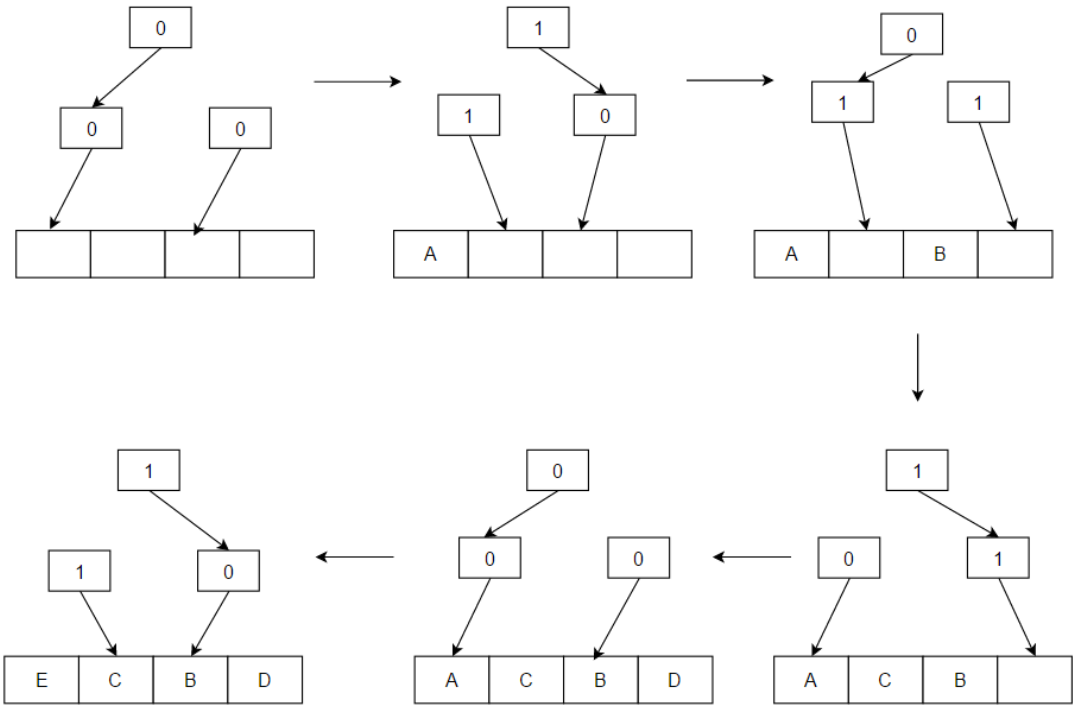



Рисунок 2.8 – Принцип роботи алгоритму кешування Pseudo-LRU [11]

Принцип, що лежить в основі цієї концепції, можна легко зрозуміти, спостерігаючи за покажчиками-стрілками. Процес наступний: щоразу, коли потрібно отримати доступ до значення, наприклад «A», і не можна знайти його в кеші, отримуємо його з пам’яті та розміщуємо в блоці, куди зараз спрямовані стрілки, рухаючись зверху вниз. Коли значення розміщено в блоці, змінюємо напрямок тих самих стрілок. У наданому прикладі спостерігаємо послідовне розміщення «A», а потім «B», «C» і «D». Коли кеш-пам’ять досягає свого об’єму, «E» замінює «A», тому що цей блок було вказано стрілками в той конкретний час, а стрілки, які спочатку вказували на «A», були перевернуті, щоб вказати протилежний напрямок. Згодом стрілки спрямовуються до «B», який буде наступним блоком, який буде замінено у випадку промаху кешу.

### 2.2.4 Алгоритми FIFO та LIFO

Алгоритм FIFO, що розшифровується як First In, First Out, функціонує шляхом видалення з кешу даних, які були додані раніше, коли виникає необхідність створити місце для нових даних. Це означає, що елемент даних, який першим було вставлено в кеш, буде першим видалено, коли кеш досягне своєї ємності. Однак важливо відзначити, що цей алгоритм має певні недоліки. Одним із помітних недоліків є те, що це не завжди найефективніший підхід, оскільки найстаріші дані можуть мати значну важливість і корисність. Крім того, алгоритм FIFO не враховує релевантність даних, а це означає, що інформація, що зберігається в кеші, може з часом застаріти або застаріти.

LIFO, що означає Last-In-First-Out, – це алгоритм, який використовується в системах кешування. У цьому алгоритмі найновіші дані, додані до кешу, мають пріоритет для видалення, коли кеш заповнюється. Це означає, що останній доданий елемент буде видалений першим. Використання алгоритму LIFO має кілька переваг. По-перше, його відносно легко реалізувати та зрозуміти, що робить його доступним для розробників. Крім

того, це може бути особливо корисним у ситуаціях, коли найновіші дані мають найбільшу важливість або коли структура даних нагадує стек. Однак важливо також враховувати недоліки алгоритму LIFO. Оскільки він не враховує актуальність даних, це може призвести до ненавмисного видалення корисної інформації з кешу перед застарілими даними. Це може бути проблематичним у сценаріях, коли актуальність даних не обов'язково є єдиним визначальним фактором важливості. Таким чином, ефективність алгоритму LIFO може змінюватися залежно від конкретних випадків використання та вимог.

### 2.2.5 Політика закінчення терміну дії (TTL)

Тривалість, протягом якої зберігається кешований елемент, визначається політикою терміну дії, яку використовує кеш. Коли об'єкт додається до кешу, йому, як правило, призначається політика терміну дії, спеціально створена для його конкретного типу. Основний підхід передбачає призначення фіксованого часу завершення для кожного об'єкта після його додавання до кешу. Після закінчення цього часу запис кешу вважається застарілим і видаляється, або оновлюється. Вибір терміну дії залежить від вимог клієнта, включаючи частоту змін даних і толерантність системи до застарілих даних. Ще одна популярна стратегія – ковзаючий термін дії, який дозволяє зробити кешовані об'єкти недійсними. За допомогою цієї стратегії для записів, до яких часто звертаються, встановлюється пріоритет, подовжуючи час їх закінчення на вказаний інтервал кожного разу, коли до них звертаються. Наприклад, якщо елемент має ковзний термін дії 15 хвилин, його буде видалено з кешу лише в тому випадку, якщо він залишатиметься недоторканим протягом періоду, що перевищує 15 хвилин.

Після початкового впровадження кешу стає обов'язковим уважний моніторинг продуктивності вибраних значень TTL. Цей процес моніторингу дозволяє вчасно переглядати, якщо це необхідно. Важливо пам'ятати, що багато фреймворків кешування не видаляють прострочені елементи з метою

оптимізації продуктивності. Замість цього вони використовують алгоритм збирання сміття, який запускається під час доступу до кешу. Цей алгоритм шукає прострочені записи та видаляє їх відповідно. Застосовуючи цей підхід, можна уникнути постійного відстеження подій закінчення терміну дії, таким чином ефективно визначаючи, коли елементи слід видалити зі сховища.

## 2.3 Існуючі технології серверного кешування даних

У наступних підрозділах будуть розглянуті характеристики, переваги та недоліки наступних систем: Redis та Memcached (підхід збереження в пам’яті).

### 2.3.1 Redis

Redis – це розподілена система зберігання даних, яка зберігає пари ключ-значення переважно в оперативній пам’яті, що дозволяє користувачам запитувати збереження даних, коли це необхідно. Він пропонує функціональність, подібну до Memcached, надаючи можливість зберігати дані у форматі ключ/значення. Однак Redis виходить за рамки простого зберігання простих даних і пропонує підтримку таких структурованих типів даних, як списки, хеші та набори. Однією з ключових переваг Redis перед Memcached є те, що він забезпечує постійне зберігання даних на диску навіть у разі несподіваного завершення роботи. На додаток до можливостей зберігання, Redis також підтримує транзакції, які дозволяють виконувати групу команд як одну атомарну операцію. Це забезпечує узгодженість та ізоляцію виконання команд, запобігаючи втручанню команд, виданих іншими клієнтами [12].

У разі будь-яких проблем Redis також дозволяє повертати зміни, забезпечуючи безпеку для операцій з даними. За допомогою Redis усі дані ефективно кешуються в оперативній пам’яті, що забезпечує швидкий доступ і пошук. Він пропонує набір команд для керування даними, наприклад збільшення/зменшення значень, виконання стандартних операцій над

списками та наборами (наприклад, об’єднання та перетин), перейменування ключів і виконання кількох функцій вибору та сортування. Redis пропонує два режими зберігання: періодична синхронізація даних із диском і ведення журналу змін на диску. Це забезпечує гнучкість у плані збереження даних. В останньому режимі Redis гарантує збереження всіх змін, внесених до набору даних. Крім того, Redis дозволяє встановлювати реплікацію даних у форматі master-slave на кількох серверах, забезпечуючи доступність даних і масштабованість без блокування. Крім того, Redis підтримує режим обміну повідомленнями публікації/підписки, де клієнти можуть підписатися на певні канали та отримувати повідомлення, що розповсюджуються через ці канали. Цей режим обміну повідомленнями полегшує спілкування в реальному часі та забезпечує ефективний розподіл інформації між підключеними клієнтами.

На рисунку 2.9 представлений алгоритм роботи системи з Redis.

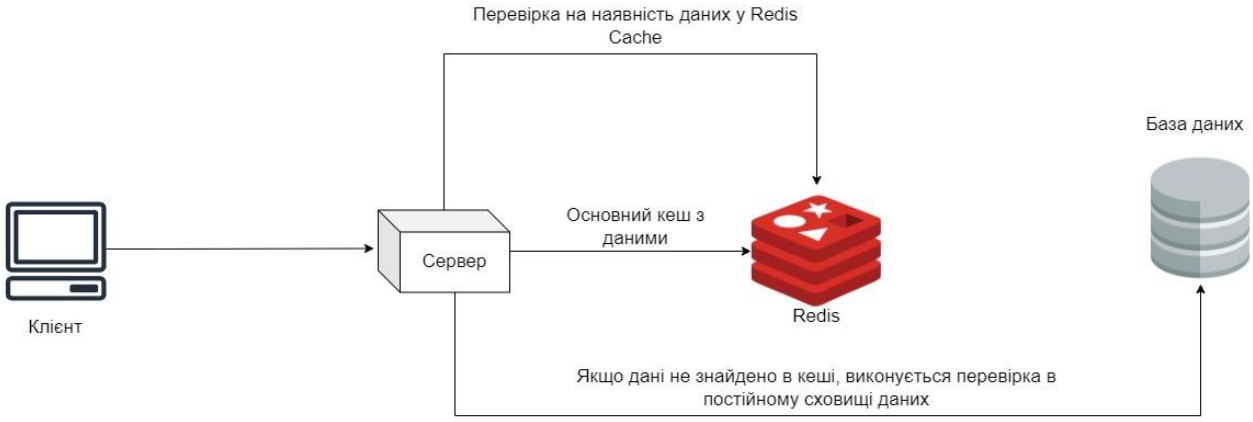



Рисунок 2.9 – Алгоритм роботи застосунку з використанням Redis

### 2.3.2 Memcached

Memcached – програмне забезпечення з відкритим кодом, яке працює з хеш-таблицями та підтримує всі основні мови програмування. Memcached служить технологією зберігання ключ-значення, яка в основному використовується для кешування сеансу, але також здатна зберігати інші типи

даних, наприклад невеликі розділи HTML-коду. Процес кешування передбачає, що запит клієнта спочатку направляється до сховища Memcached, де він шукає попередньо скомпільовану відповідь [13, 14].

Якщо потрібної відповіді не знайдено, запит пересилається на сервер, генерується сторінка, а результат негайно зберігається в оперативній пам’яті за допомогою служби Memcached. Подальші запити на подібні сеанси можуть бути швидко отримані з кешу без оподаткування основних дисків. Цей ефективний підхід значно скорочує час відповіді до мілісекунд, ефективно зменшуючи навантаження на потужності сервера та сприяючи швидшому завантаженню веб-сайту. Memcached працює виключно з рядковими ключами та не має складних типів даних, що додатково оптимізує використання пам’яті.

Важливо зазначити, що Memcached зберігає дані виключно в оперативній пам’яті та не зберігає їх після перезавантаження системи. У випадках, коли пам'яті недостатньо, нові дані перезаписують старі. Крім того, Memcached є багатопоточним і ефективно масштабується вертикально за рахунок розширення пам’яті та збільшення кількості ядер [15].

На рисунку 2.10 представлений алгоритм роботи системи з Memcached.

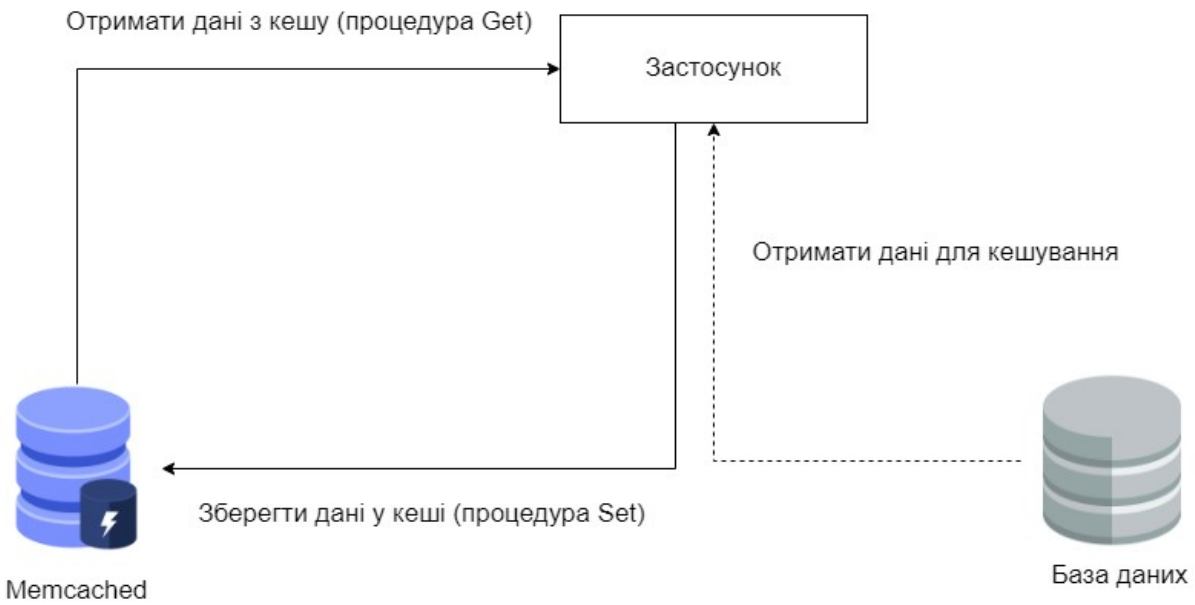



Рисунок 2.10 – Алгоритм роботи застосунку з використанням Memcached

### 2.3.3 Порівняння існуючих технологій кешування

На основі наведеної інформації в попередніх підрозділах, проведемо порівняльний аналіз технологій кешування Redis та Memcached. В таблиці 2.1 представлений порівняльний аналіз технологій Redis та Memcached.

Таблиця 2.1 – Порівняльний аналіз технологій Redis та Memcached

| **Особливість** | **Redis** | **Memcached** |
|---|---|---|
| Підтримувані типи даних | Рядки, списки, набори, відсортовані набори, хеші, бітові масиви, геопросторові дані | Рядки |
| Управління пам'яттю | Може зберігати дані на диску, коли фізична пам'ять повністю заповнена | Тільки In-Memory, але має підтримку збереження даних на диск з використанням зовнішніх драйверів |
| Обмеження розміру даних | 512 МБ (для рядків) | 1 МБ |
| Субмілісекундна затримка | Підтримує | Підтримує |
| Збереження даних | Підтримує використання RDB і політик збереження журналу AOF | Не підтримує збереження постійних даних |
| Режим кластера (розподілене кешування) | Підтримує | Не підтримує. Можливо досягти на стороні клієнта за допомогою узгодженого хешування |
| Багатопотоковість | Використовує один потік для GET/SET | Підтримує |
| Масштабування | Підтримує горизонтальне масштабування | Підтримує тільки вертикальне масштабування (горизонтальне масштабування з боку клієнта) |
| Реплікація даних | Підтримує реплікацію даних | Не підтримує |

На основі порівняльного аналізу, проведеного в таблиці 2.1, можна зробити наступні висновки:

1. Memcached – це просте сховище ключів і значень, яке підтримує невеликі довільні типи даних, такі як рядки та об’єкти. Він ідеально підходить для програм, які потребують простих функцій кешування та швидкого пошуку на основі ключів. Як наслідок, він не є виключно придатним для більш складних випадків використання, які вимагають передових методів обробки даних. Водночас Redis підтримує більший спектр структур даних, включаючи рядки, хеші, списки, набори та геопросторові дані. Ця гнучкість дозволяє застосовувати більш складні випадки використання, наприклад реалізацію аналітики в реальному часі або отримання даних на основі місцезнаходження.

2. Memcached надає пріоритет досягненню високої продуктивності та надзвичайно швидкому відгуку. На додаток до зосередженості на швидкості, Memcached також пропонує вертикальну масштабованість, що означає, що можна додавати більше серверів до пулу кешування, таким чином враховуючи збільшений трафік і оброблюючи більші обсяги даних. Redis забезпечує подібні можливості продуктивності для простих завдань кешування, а також обслуговує більш складні випадки використання, пропонуючи ряд додаткових функцій. Однією з помітних особливостей Redis є його здатність виконувати асинхронні та неблокуючі операції вводу/виводу. Це означає, що програма може ефективно обробляти кілька одночасних завдань, що призводить до покращення продуктивності, особливо під час з великого навантаження. Redis підтримує кластери, забезпечуючи горизонтальне масштабування [16].

3. Memcached працює лише в пам’яті, тобто зберігає всі дані в оперативній пам’яті та не має вбудованого збереження. Такий підхід забезпечує максимальну продуктивність і низьку затримку доступу до даних. Однак Memcached не зберігає дані автоматично у разі збою системи. Новіші версії підтримують відновлення даних після перезапуску та постійної пам’яті через монтування файлової системи DAX. Redis пропонує необов’язкове збереження даних за допомогою двох різних методів: знімка та файлу лише

для додавання (AOF). Знімок передбачає створення знімків даних у кеші та збереження даних на диску через певний час. AOF є більш надійним методом, який складається з додавання команд до AOF для зміни даних.

4. Redis використовує одне ядро та демонструє кращу продуктивність, ніж Memcached, у зберіганні невеликих наборів даних, якщо виміряти кількість ядер. Memcached реалізує багатопотокову архітектуру, використовуючи кілька ядер. Тому для зберігання більших наборів даних Memcached може працювати краще, ніж Redis. Ще однією перевагою багатопоточної архітектури Memcached є її висока масштабованість, яка досягається за рахунок використання кількох обчислювальних ресурсів.

5. Оскільки обсяг пам’яті на серверах обмежений, застарілі дані в кеші повинні бути автоматично видалені з пам’яті, щоб звільнити місце для нових даних. І Redis, і Memcached мають способи видалення застарілих даних, однак вони не завжди однакові. Поширений спосіб вилучення даних називається Least Recently Used – видалення даних, які нещодавно не використовувалися. Якщо дані збігаються, це зазначається, що означає, що вони не будуть кандидатом на виселення. Це єдиний спосіб роботи Memcached [17].

Однак Redis пропонує низку інших способів боротьби з видаленням даних, наприклад No Eviction, коли Redis просто дозволяє пам’яті заповнитися, а потім більше не бере жодних ключів, і Volatile TTL (Time to Live), де Redis намагатиметься видалити спочатку ключі з установленим TTL, намагаючись зберегти дані, не анотовані TTL, які мали зберігатися довше.

6. Memcached фокусується на простому, зрозумілому підході до кешування. Незважаючи на те, що він чудовий у цьому, його обмежена підтримка типів даних обмежує його здатність обробляти більш складні вимоги моделювання даних або виконувати спеціалізовані операції. Такі обмеження обмежують розширюваність Memcached у сценаріях, які вимагають передових структур даних і методів маніпулювання даними. Тим часом, Redis пропонує численні функції на додаток до базового кешування, включаючи транзакції та повідомлення про публікацію/підписку (pub-sub).

## 2.4 Огляд існуючих технологій кешування в Cloud-середовищах

У сучасному світі цифрових технологій, де обсяги даних зростають з кожним днем, важливість ефективного кешування у Cloud-середовищах стає надзвичайно актуальною. Кешування в хмарі дозволяє покращити продуктивність додатків шляхом зменшення затримок при доступі до даних та зниження навантаження на основні бази даних. Використання кешування в Cloud-середовищах також сприяє оптимізації витрат, оскільки зменшує необхідність використання дорожчих ресурсів обчислення та зберігання. Нижче будуть розглянуті такі технології як AeroSpike, Apache Ignite, Hazelcast.

### 2.4.1 AeroSpike

Aerospike служить високоефективною системою керування базами даних NoSQL. Він може похвалитися чудовим часом доступу до даних, що робить його особливо придатним для обробки даних у реальному часі. За словами його розробників [16], він досягає виняткової продуктивності: 99% запитів виконуються менш ніж за 1 мілісекунду, а 99,9% запитів виконуються протягом максимум 5 мілісекунд. Використовуючи Aerospike, користувачі можуть створювати конфігурації кластерів, які полегшують реплікацію даних на кількох вузлах, що призводить до майже лінійного підвищення продуктивності при додаванні нових вузлів. Обсяг пам’яті Aerospike може досягати десятків терабайт і вміщати більше ста мільярдів об’єктів. Система підтримує різні схеми реплікації з метою забезпечення відмовостійкості, оптимізації продуктивності та забезпечення географічного розподілу бази даних між різними центрами обробки даних. Впроваджуючи блокування на рівні рядків і миттєву фіксацію транзакцій, Aerospike надійно захищає від втрати даних через збої. Коли використовується синхронна реплікація, кластер зберігання даних Aerospike відповідає суворим вимогам ACID, гарантуючи атомарність, послідовність, ізоляцію та надійність. У разі збою вузла Aerospike

автоматично ініціює перебалансування даних, дозволяючи продовжувати роботу без збоїв. Оновлення та резервне копіювання можна виконувати без проблем, не впливаючи на продуктивність системи.

Aerospike зберігає дані в записах, структурованих як пари ключ/значення, які можна згрупувати в набори та таблиці в окремих просторах імен. Він також підтримує обробку даних на стороні сервера за допомогою визначених користувачем функцій (UDF), написаних на мові програмування Lua. Платформа дозволяє виконувати складні аналітичні запити з використанням методів скорочення карт і полегшує маніпулювання великими типами даних. Крім того, виконання запиту можна розпаралелювати між кількома вузлами, причому кожен сприяє робочому навантаженню обробки.

Aerospike був спеціально розроблений для підтримки великих обсягів даних та великої кількості операцій вводу-виводу за секунду (IOPS). Його архітектура дозволяє легко масштабуватися горизонтально, що робить його ідеальним для додатків з великим обсягом даних [18].

На відміну від Redis і Memcached, які в основному є системами зберігання в пам'яті, Aerospike ефективно працює з даними, збереженими на SSD, забезпечуючи високу продуктивність навіть при роботі з дисками.

### 2.4.2 Apache Ignite

Apache Ignite служить розподіленою системою керування базами даних, яка вирізняється високопродуктивними обчисленнями. З точки зору рівнів зберігання та обробки, Apache Ignite переважно покладається на оперативну пам’ять, що робить її частиною категорії обчислювальних платформ в пам’яті. Хоча дисковий рівень не є обов’язковим, його можна ввімкнути, щоб містити повний набір даних, тоді як рівень пам’яті функціонує як кеш, здатний вмістити весь або частковий набір даних залежно від його ємності. Дані в Apache Ignite структуровані як пари ключ-значення, а компонент бази даних розумно розподіляє ці пари по кластеру. Отже, кожен вузол у кластері володіє

частиною загального набору даних. Автоматичне відновлення балансу відбувається кожного разу, коли вузол додається або видаляється з кластера, забезпечуючи оптимальний розподіл даних. Для ефективної організації база даних Ignite зберігає дані в розподілених «кешах». Хоча ця термінологія походить від початкової підтримки рівня пам'яті, вона залишається у використанні з історичних причин. По суті, кожен кеш відповідає певному типу сутності, наприклад, працівнику або організації. Крім того, кожен кеш ділиться на фіксовану кількість «розділів», які рівномірно розподіляються між вузлами кластера за допомогою алгоритму хешування рандеву. Кожен розділ складається з однієї основної копії та потенційно кількох резервних копій, коефіцієнт реплікації яких можна налаштувати. Якщо ввімкнено режим повної реплікації, кожен вузол кластера зберігає повну копію кожного розділу [19].

На рисунку 2.11 представлена архітектура транзакцій Apache Ignite.

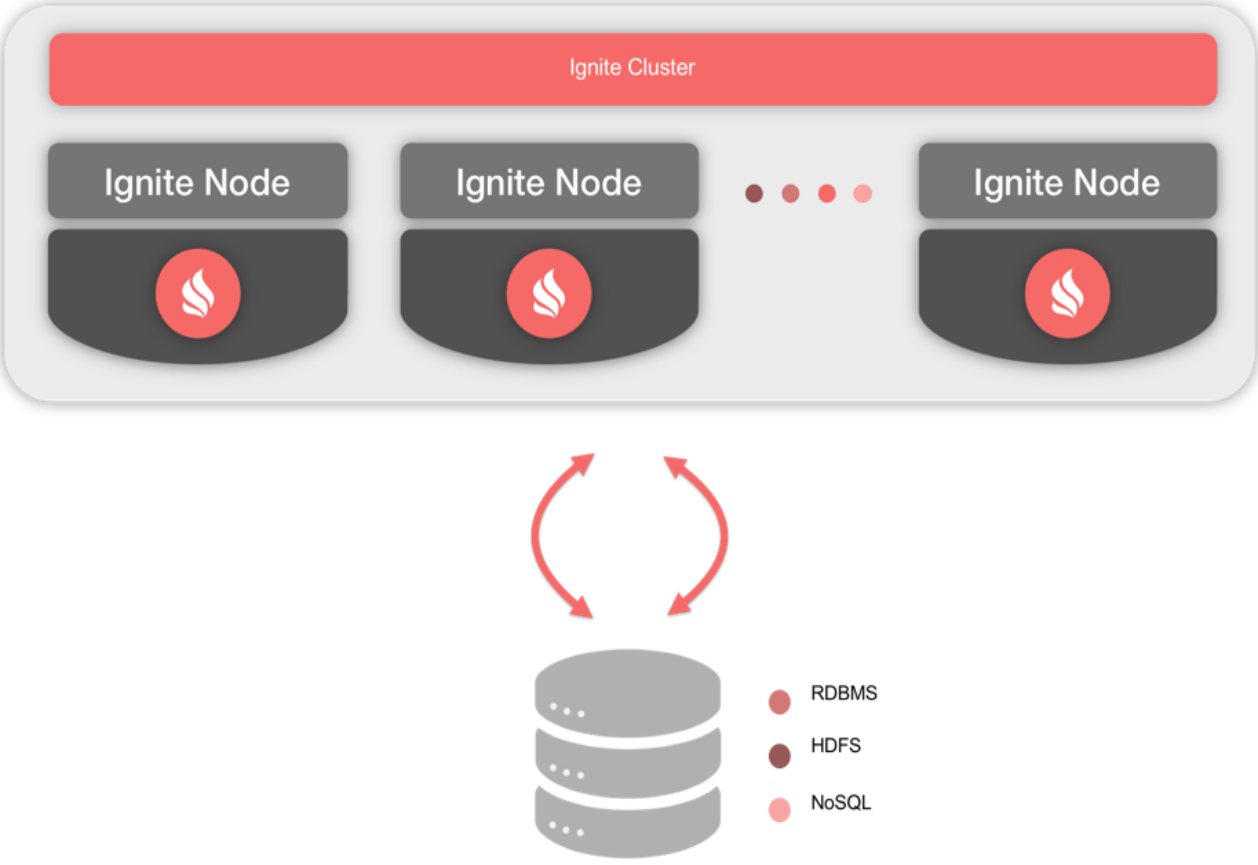



Рисунок 2.11 – Архітектура виконання транзакцій в системі Apache Ignite

### 2.4.3 Hazelcast

Hazelcast – це уніфікована платформа даних реального часу на основі Java, яка поєднує швидке сховище даних із потоковою обробкою. Це також назва компанії, що розробляє продукт. У сітці (гріді) Hazelcast дані рівномірно розподіляються між вузлами комп'ютерного кластера, що дозволяє горизонтально масштабувати обробку та доступну пам'ять. Резервні копії також розподіляються між вузлами для захисту від збою будь-якого окремого вузла. Hazelcast забезпечує централізоване передбачуване масштабування додатків за допомогою доступу в пам'яті до часто використовуваних даних і через еластично масштабовану мережу даних. Ці методи зменшують навантаження запитів на бази даних і покращують швидкість. Hazelcast може працювати локально, у хмарі (Amazon Web Services, Microsoft Azure, Cloud Foundry, OpenShift), віртуально (VMware) і в контейнерах Docker. Hazelcast пропонує технологічну інтеграцію для багатьох технологій конфігурації та розгортання хмари, зокрема Apache jclouds, Consul тощо, Eureka, Kubernetes і Zookeeper. Hazelcast Cloud Discovery Service Provider Interface (SPI) дозволяє хмарним або локальним вузлам автоматично виявляти один одного.

Далі, проведемо порівняльний аналіз основних функціональних можливостей Hazelcast з Redis [20]:

– Будучи багатопоточним, Hazelcast може перевершити Redis у деяких випадках використання з точки зору продуктивності.

– Hazelcast підтримує автоматичне виявлення вузла, просто підтверджуючи IP-адреси, і піклується про перебалансування даних. У Redis є хороші ручні кроки для додавання нового вузла, оскільки перебалансування кластера не виконується з коробки. Те саме стосується видалення вузлів, у Redis для видалення вузлів потрібно збалансувати порти та перенести дані.

– Хоча обидва підтримують зберігання ключових значень, але Hazelcast може зберігати складні об'єкти даних. Hazelcast підтримує створення SQL-

запитів, за допомогою яких можна запитувати дані за допомогою певних полів у значенні. У Redis можна запитувати дані лише за допомогою ключів.

– Hazelcast надає вбудовані можливості кластеризації з автоматичним виявленням і керуванням вузлами, тоді як Redis вимагає сторонніх рішень кластеризації для високої доступності.

– Також, Hazelcast забезпечує інтеграцію з такими популярними фреймворками Java, як Spring і Hibernate, а Redis має ширший спектр клієнтських бібліотек для багатьох мов програмування.

– Hazelcast – це розподілена сітка даних у пам’яті (IMDG), тоді як Redis – це сховище ключів і значень у пам’яті. Hazelcast розподіляє дані між кількома вузлами в кластері, тоді як Redis працює на одному вузлі.

– Redis завжди працює як автономний і не має жодної інтеграції з жодною БД. У той час як Hazelcast забезпечує шаблон читання і запису, де його можна використовувати як кеш для бази даних. На рисунку 2.12 представлений приклад використання Cache-Aside та Read-Through патернів.

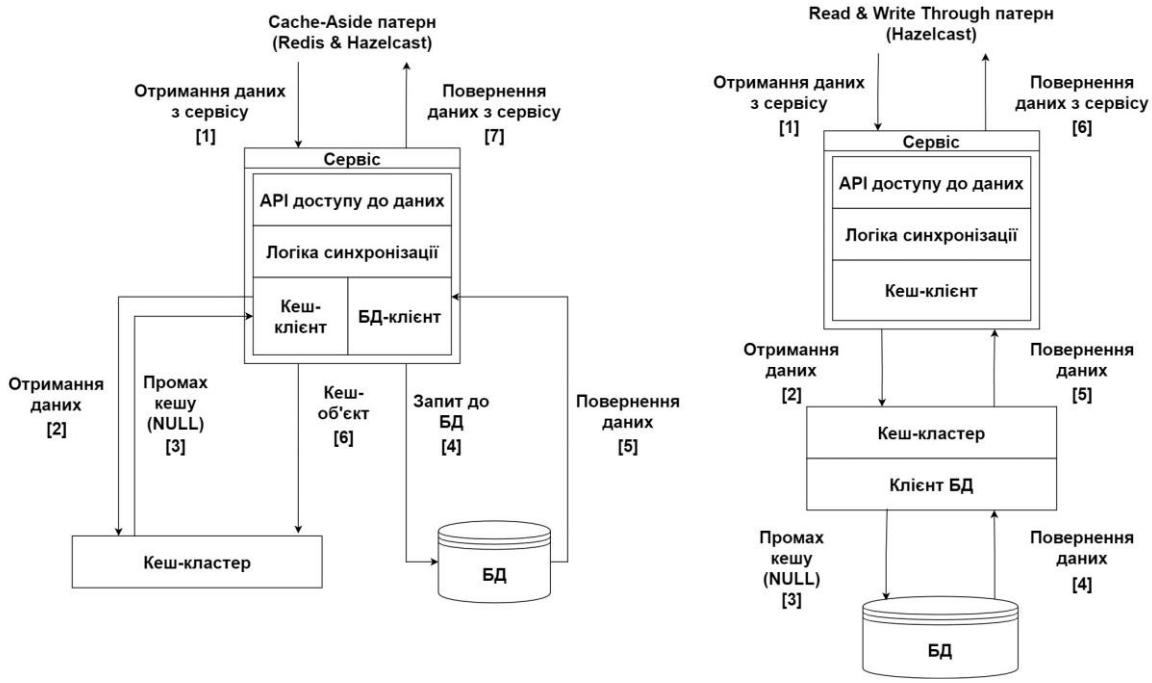



Рисунок 2.12 – Використання Cache-Aside та Read-Through патернів на прикладі технологій Redis та Hazelcast

# 3 ПРОЕКТУВАННЯ ТА РОЗРОБКА ПРОГРАМНОЇ СИСТЕМИ З ВИКОРИСТАННЯМ PYTHON

## 3.1 Огляд технологій для розробки програмної системи

Для розробки високонавантаженої інформаційної системи були використані наступні технології: мова програмування Python, інструмент контейнеризації застосунків Docker, бібліотека FastAPI, система керування базами даних PostgreSQL, а також системи кешування Redis та Memcached.

### 3.1.1 Мова програмування Python

Python – це універсальна та широко використовувана мова програмування, яка відома своїм акцентом на читабельності коду. Вона вважається мовою загального призначення високого рівня, яка підтримує різні парадигми програмування, включаючи структуроване, об’єктно-орієнтоване та функціональне програмування. Одним з унікальних аспектів Python є його синтаксис на основі відступів, який покращує читабельність коду та є відмінною рисою мови. Python має динамічну типізацію, що означає, що типи змінних не потребують явного оголошення, що забезпечує більшу гнучкість програмування. Крім того, Python включає автоматичне збирання сміття (garbage collection), яке керує пам’яттю, що полегшує зосередження на написанні коду, а не на управлінні розподілом пам’яті. Одна з ключових сильних сторін Python полягає в широкій підтримці бібліотек, якI дозволяють реалізувати такі методи, як контрактне та логічне програмування. З точки зору архітектури, Python пропонує динамічну типізацію, яка забезпечує гнучкість у роботі з різними типами даних. Він також забезпечує можливість інтроспекції, дозволяючи перевіряти властивості та методи об’єктів під час виконання.

Python також пропонує ряд інструментів і підходів для високонавантажених систем. Щоб підвищити продуктивність системи,

бібліотека asyncio дає змогу асинхронного програмування, дозволяючи виконувати кілька операцій одночасно, не викликаючи блокування. У контексті кешування для високонавантажених систем Python забезпечує виняткову підтримку зовнішніх систем кешування, таких як Redis або Memcached. Використання цих систем дозволяє швидко отримувати дані, до яких часто звертаються, що зберігаються в оперативній пам’яті, зменшуючи навантаження на базу даних. При роботі з базами даних бібліотека SQLAlchemy пропонує зручну абстракцію для роботи з різними базами даних та полегшує оптимізацію запитів, щоб мінімізувати навантаження на сервер.

### 3.1.2 Інструмент контейнеризації застосунків Docker

Docker є універсальним інструментом, призначеним для керування ізольованими контейнерами Linux, який доповнює набір інструментів LXC, надаючи API вищого рівня для керування контейнерами на рівні процесу.

За допомогою Docker можна без особливих зусиль виконувати різноманітні процеси в режимі ізоляції, знімаючи занепокоєння щодо вмісту контейнера. Крім того, Docker забезпечує плавну передачу та клонування створених контейнерів на інші сервери, виконуючи всі необхідні завдання, пов’язані зі створенням, обслуговуванням і підтримкою контейнерів.

Docker оптимально працює на немодифікованих сучасних ядрах Linux і в стандартних середовищах основних дистрибутивів Linux, таких як Fedora, RHEL, Ubuntu, Debian, SUSE, Gentoo та Arch. Використовуючи легкі контейнери, Docker ефективно ізолює процеси як один від одного, так і від хост-системи. Оскільки кожен контейнер використовує власну автономну файлову систему, їх середовище та розташування стають неважливими. Ізоляція файлової системи досягається шляхом виконання кожного процесу в окремій кореневій файловій системі, що призводить до повного розділення.

На рисунку 3.1 представлена візуалізація роботи з інструментом Docker.

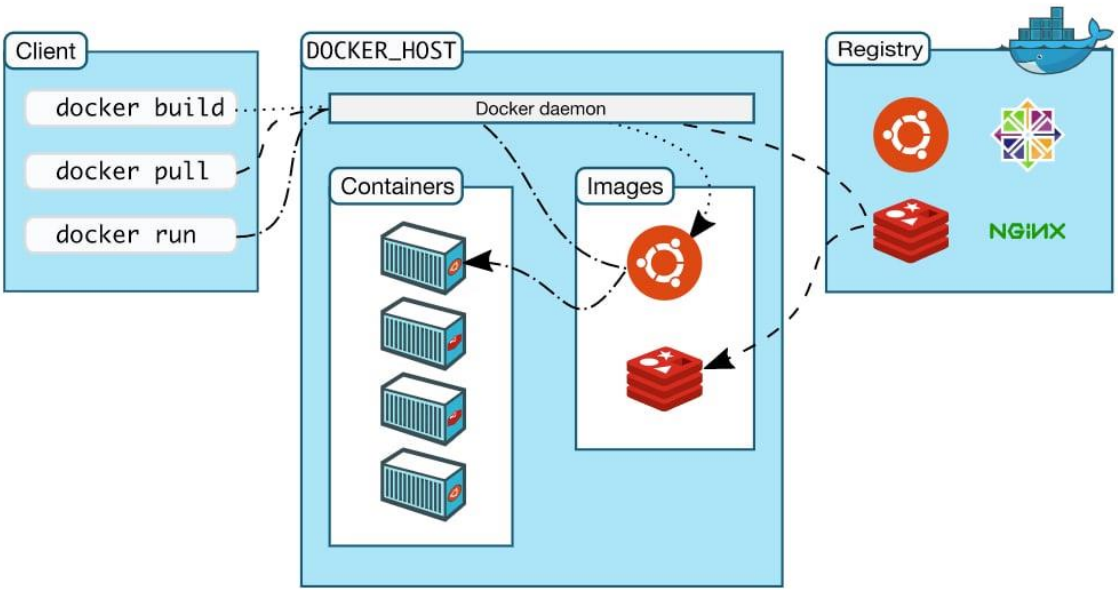



Рисунок 3.1 – Архітектура використання контейнеризації з Docker [12]

Docker полегшує ізоляцію ресурсів, уможливлюючи незалежні обмеження системних ресурсів, таких як споживання пам’яті та навантаження ЦП для кожного окремого контейнера, завдяки контрольним групам.

На мережевому рівні Docker надає ізольованим процесам доступ виключно до мережевого простору імен, пов’язаного з їхнім відповідним контейнером, надаючи контроль над інтерфейсом віртуальної мережі та пов’язаними з ним IP-адресами. Крім того, Docker ефективно створює кореневу файлову систему для контейнерів за допомогою механізму копіювання під час запису, фактично зберігаючи окремо змінені та нові дані.

Однією з ключових переваг Docker є його здатність використовувати змінену файлову систему існуючого контейнера як основу для створення нових базових образів і створення інших контейнерів. Це позбавляє від необхідності розробляти шаблони або вручну налаштовувати композиції зображень. Крім того, Docker полегшує створення контейнерів, які охоплюють складні стеки програмного забезпечення, інтелектуально зв’язуючи разом уже існуючі контейнери, які містять необхідні компоненти стеку. Це зв’язування

не передбачає об’єднання вмісту, а натомість забезпечує безперебійну взаємодію між контейнерами через створення мережевого тунелю. Окрім цього, Docker допомагає в горизонтальному масштабуванні застосунку. Замість того, щоб розширювати ресурси одного сервера, можна легко створювати додаткові екземпляри застосунку в нових контейнерах для розподілу навантаження. Застосування таких оркестраторів контейнерів, як Kubernetes, може автоматизувати процес масштабування, розгортання та відновлення контейнерів відповідно до потреб. Також, Docker сприяє ізоляції ресурсів і сервісів. Це означає, що кожен компонент високонавантаженого застосунку може бути розгорнутий у власному контейнері, ізольованому від інших, що дозволяє оптимізувати ресурси та уникати потенційних конфліктів.

### 3.1.3 Бібліотека FastAPI

FastAPI – це сучасний та швидкий веб-фреймворк для побудови RESTful API на Python. Ця бібліотека набула популярності завдяки своїй продуктивності, стандартізації та легкості використання.

Основні особливості FastAPI включають:

– Автоматична генерація документації (за допомогою Swagger UI та ReDoc), що спрощує тестування та інтеграцію API.

– Використання сучасних стандартів (наприклад, OpenAPI).

– Підтримка асинхронного програмування, що дозволяє виконувати одночасно велику кількість запитів без необхідності великої кількості потоків.

– Засноване на типах валідування та серіалізація даних, що підвищує швидкість розробки та зменшує кількість помилок.

Для високонавантаженого застосунку FastAPI може стати ідеальним вибором завдяки декільком ключовим перевагам:

– FastAPI дозволяє інтегрувати асинхронний код, що оптимізує час відгуку та дозволяє системі обробляти велику кількість одночасних запитів.

– Завдяки Starlette (лежить в основі FastAPI) та Pydantic для валідування даних, FastAPI часто виходить на однаковому рівні з продуктивністю такої мови програмування як Go, особливо при використанні асинхронних запитів.

– FastAPI має вбудовані інструменти для захисту від різноманітних атак (напр. SQL Injection), та підтримує системи аутентифікації та авторизації.

### 3.1.4 База даних PostgreSQL

PostgreSQL – це реляційна база даних з відкритим вихідним кодом, яка пропонує підтримку запитів як SQL, так і JSON. Завдяки більш ніж 20-річному досвіду розвитку спільноти, PostgreSQL зарекомендувала себе як надійна та надійна система керування базами даних. Цей спільний підхід значною мірою сприяв його стійкості, цілісності та точності. PostgreSQL широко використовується як основне сховище даних або сховище даних у різноманітних додатках, таких як Інтернет, мобільні, геопросторові та аналітичні програми. PostgreSQL може похвалитися широким набором потужних функцій, включаючи табличні простори, асинхронну реплікацію, вкладені транзакції, резервне копіювання та вдосконалений планувальник/оптимізатор запитів. Ці функції надають користувачам повний набір інструментів для ефективного керування своїми даними та маніпулювання ними. Одним із важливих аспектів PostgreSQL є його відповідність вимогам ACID, що гарантує надійне виконання транзакцій бази даних. Відповідність ACID відноситься до чотирьох основних характеристик транзакцій бази даних: атомарності, узгодженості, ізоляції та довговічності.

Ці якості гарантують точність і надійність збережених даних, гарантуючи, що неповні зміни ніколи не зберігаються. Щоб досягти відповідності вимогам ACID, PostgreSQL включає різні механізми, такі як ведення журналу з попереднім записом, багатоверсійний контроль паралелізму (MVCC) і відновлення на певний момент часу. Ці функції

дозволяють PostgreSQL підтримувати цілісність даних, ізолювати транзакції та забезпечувати довговічність. Postgres має підтримку даних форматів JSON і JSONB. Хоча він може ефективно зберігати та запитувати дані JSON, формат JSONB дозволяє ще швидше виконувати операції запитів, зберігаючи дані у двійковому форматі. Це надзвичайно корисно для додатків, що мають справу зі структурованими форматами даних. Що стосується просторових даних, розширення PostGIS підносить PostgreSQL до ліги передових просторових баз даних, підтримуючи широкий спектр запитів на основі розташування. Іншим інтригуючим аспектом Postgres є його здатність об'єднувати зовнішні дані через зовнішні оболонки даних (FDW). За допомогою FDW різні джерела даних можуть безперешкодно інтегруватися та запитуватися так, ніби це рідні таблиці PostgreSQL. Цілісність даних, яка є першорядною проблемою для баз даних, керується за допомогою запису журналу (WAL). Реєструючи зміни перед тим, як внести їх у базу даних, WAL гарантує, що дані залишаються послідовними та такими, що їх можна відновити.

PostgreSQL пропонує асинхронну систему обміну повідомленнями, доступ до якої можна отримати за допомогою таких команд, як NOTIFY, LISTEN і UNLISTEN. За допомогою цієї системи сеанс може ініціювати команду NOTIFY разом із налаштованим каналом і додатковим корисним навантаженням для вказівки на виникнення певної події. Згодом інші сеанси можуть виявити ці події, виконавши команду LISTEN, за допомогою якої вони можуть налаштуватися на певний канал. Ця потужна функціональність служить широкому спектру цілей. Наприклад, це дозволяє кільком сеансам знати про оновлення, внесені до таблиці, або дозволяє окремим програмам виявляти, коли мала місце певна дія. Завдяки використанню цієї системи безперервне опитування додатків для перевірки змін стає непотрібним, що призводить до зменшення непотрібних витрат на обробку. Варто зазначити, що сповіщення є повністю транзакційними, тобто повідомлення надсилаються лише після завершення транзакції, з якої вони походять. Це гарантує, що

повідомлення не надсилатимуться передчасно для дій, які пізніше відкочуються, усуваючи будь-які проблеми з надійністю повідомлень системи.

### 3.2 Опис реалізації програмної системи

Розглянемо детальніше реалізацію основних алгоритмів програмної системи. Для конфігурації середовища розробки розроблено файл з налаштуваннями config.py, який є модулем конфігурації, призначеним для завантаження налаштувань з оточення та їх представлення у вигляді структурованого об'єкта. На рисунку 3.2 представлена реалізація класу GlobalConfig, який в подальшому буде використаний для модулів системи.

```python
import os

from dotenv import load_dotenv
from pydantic import BaseConfig

load_dotenv()


class GlobalConfig(BaseConfig):
    title: str = os.environ.get("TITLE")
    version: str = "1.0.0"
    description: str = os.environ.get("DESCRIPTION")
    docs_url: str = "/docs"
    redoc_url: str = "/redoc"
    openapi_url: str = "/openapi.json"
    api_prefix: str = "/api"
    debug: bool = os.environ.get("DEBUG")

    postgres_user: str = os.environ.get("POSTGRES_USER")
    postgres_password: str = os.environ.get("POSTGRES_PASSWORD")
    postgres_server: str = os.environ.get("POSTGRES_SERVER")
    postgres_port: int = int(os.environ.get("POSTGRES_PORT"))
    postgres_db: str = os.environ.get("POSTGRES_DB")
    postgres_db_tests: str = os.environ.get("POSTGRES_DB_TESTS")
    db_echo_log: bool = True if os.environ.get("DEBUG") == "True" else False

    redis_server: str = os.environ.get("REDIS_SERVER")
    redis_port: int = int(os.environ.get("REDIS_PORT"))

    @property
    def sync_database_url(self) -> str:
        return f"postgresql://{self.postgres_user}:{self.postgres_password}@{self.postgres_server}:{self.postgres_port}/{self.postgres_db}"

    @property
    def async_database_url(self) -> str:
        return f"postgresql+asyncpg://{self.postgres_user}:{self.postgres_password}@{self.postgres_server}:{self.postgres_port}/{self.postgres_db}"


settings = GlobalConfig()
```

Рисунок 3.2 – Реалізація класу-конфігуратора GlobalConfig

Для реалізації класу-конфігуратора GlobalConfig були використані наступні бібліотеки:

– Бібліотека os, яка використовується для роботи з операційною системою, зокрема для доступу до оточення з використанням os.environ.get().

– Бібліотека pydantic, а саме базовий клас BaseConfig, який дозволяє створювати класи налаштувань системи з валідацією даних та типізацією.

В процесі розробки програмної системи, було визначено модель для збереження данів про користувачів системи. Ця модель описує структуру профілю користувача в базі даних за допомогою класів, заснованих на бібліотеці sqlmodel. Вона має два класи: ProfileBase та Profile. На рисунку 3.3 представлена реалізація моделі Profile з використанням бібліотеки sqlmodel.

```python
from datetime import date
from typing import List
from uuid import UUID, uuid4

from sqlmodel import JSON, Column, Field, SQLModel


class ProfileBase(SQLModel):
    username: str = Field(default=None, nullable=True)
    mail: str = Field(default=None, nullable=True)
    name: str = Field(default=None, nullable=True)
    ssn: str = Field(default=None, nullable=True)
    sex: str = Field(default=None, nullable=True)
    birthdate: date = Field(default=None, nullable=True)
    blood_group: str = Field(default=None, nullable=True)
    address: str = Field(default=None, nullable=True)
    residence: str = Field(default=None, nullable=True)
    website: List[str] = Field(default=None, nullable=True, sa_column=Column(JSON))
    current_location: List[str] = Field(
        default=None, nullable=True, sa_column=Column(JSON)
    )
    job: str = Field(default=None, nullable=True)
    company: str = Field(default=None, nullable=True)


class Profile(ProfileBase, table=True):
    id: UUID = Field(
        default_factory=uuid4,
        primary_key=True,
        index=True,
        nullable=False,
    )

    __tablename__ = "profiles"
```

Рисунок 3.3 – Лістинг реалізації моделі Profile

Модель Porifle включає наступні поля:

1. username: Ім’я користувача в системі.

2. mail: Електронна адреса користувача в системі.

3. name: Реальне ім'я користувача.

4. ssn: Соціальний номер користувача.

5. sex: Стать користувача, яка може бути необов'язковою.

6. birthdate: Дата народження користувача.

7. blood_group: Група крові користувача.

8. address: Адреса користувача.

9. residence: Місце проживання користувача.

10. website: Список веб-сайтів, асоційованих з користувачем, який зберігається в форматі JSON.

11. current_location: Поточне місцезнаходження користувача, також зберігається в форматі JSON.

12. job: Посада або рід занять користувача.

13. company: Компанія, де працює користувач.

Клас Profile наслідує від ProfileBase та додає додатковий атрибут для ідентифікації запису в базі даних.

1. id: унікальний ідентифікатор для профілю користувача, який генерується автоматично за допомогою uuid4 та є ключем для таблиці.

2. tablename: визначає назву таблиці в базі даних, де будуть зберігатися профілі. Ця модель служить для представлення даних користувача в структурі бази даних, забезпечуючи механізм для взаємодії з БД через об'єкти Python.

Далі, були визначені інструменти для налаштування бази даних і створення випадкових профілів користувачів за допомогою бібліотеки Faker.

Для роботи створюються два двигуни для бази даних: звичайний двигун engine для синхронних операцій та асинхронний async_engine для асинхронних операцій. Обидва двигуни конфігуруються за допомогою значень з глобальної конфігурації.

Для створення асинхронних сесій до бази даних визначається фабрика сесій async_session, прив'язана до асинхронного двигуна.

Функція create_db_and_tables служить для створення бази даних та таблиць. Ця функція спершу видаляє існуючі таблиці, а потім створює їх.

Функція bulk_create_profiles призначена для створення багатьох профілів користувачів в базі даних, яка використовує Faker для генерації даних про користувачів. Профілі додаються до асинхронної сесії та зберігаються в базі даних. Реалізація вищеописаних елементів представлена на рисунку 3.4.

```python
from faker import Faker
from sqlalchemy.ext.asyncio import create_async_engine
from sqlalchemy.orm import sessionmaker
from sqlmodel import SQLModel, create_engine
from sqlmodel.ext.asyncio.session import AsyncSession

from core.config import settings
from db.tables.profiles import Profile

engine = create_engine(
    settings.sync_database_url,
    echo=settings.db_echo_log,
)

async_engine = create_async_engine(
    settings.async_database_url,
    echo=settings.db_echo_log,
    future=True,
)

async_session = sessionmaker(
    bind=async_engine, class_=AsyncSession, expire_on_commit=False
)


async def create_db_and_tables() -> None:
    SQLModel.metadata.drop_all(engine)
    SQLModel.metadata.create_all(engine)


async def bulk_create_profiles(number: int) -> None:
    fake = Faker()
    async with AsyncSession(async_engine) as session:
        for _ in range(number):
            profile = Profile(**fake.profile())
            session.add(profile)
        await session.commit()
```

Рисунок 3.4 – Лістинг реалізації інструментів для налаштування бази даних

Для роботи з базою даних був створений репозиторій для роботи з профілями користувачів в асинхронному режимі, який використовує клас Profile з модуля db.tables.profiles для взаємодії з базою даних і визначає методи для створення, отримання, оновлення та видалення профілів (CRUD операції).

На рисунках 3.5 – 3.6 представлена реалізація репозиторію.

```python
from typing import Optional, List
from uuid import UUID

from sqlmodel import select
from sqlmodel.ext.asyncio.session import AsyncSession

from db.errors import EntityDoesNotExist
from db.tables.profiles import Profile
from schemas.profiles import ProfileCreate, ProfilePatch, ProfileRead


class ProfileRepository:
    def __init__(self, session: AsyncSession) -> None:
        self.session = session

    async def _get_instance(self, profile_id: UUID):
        statement = select(Profile).where(Profile.id == profile_id)
        results = await self.session.exec(statement)

        return results.first()

    async def create(self, profile_create: ProfileCreate) -> ProfileRead:
        db_profile = Profile.from_orm(profile_create)
        self.session.add(db_profile)
        await self.session.commit()
        await self.session.refresh(db_profile)

        return ProfileRead(**db_profile.dict())

    async def list(self, limit: int = 10, offset: int = 0) -> list[ProfileRead]:
        statement = select(Profile).offset(offset).limit(limit)
        results = await self.session.exec(statement)

        return [ProfileRead(**profile.dict()) for profile in results]

    async def list_exclude_ids(self, exclude_ids: List[UUID], limit: int) -> List[ProfileRead]:
        statement = select(Profile).where(
            Profile.id.notin_(exclude_ids)
        ).limit(limit)
        results = await self.session.exec(statement)

        return [ProfileRead(**profile.dict()) for profile in results]

    async def get(self, profile_id: UUID) -> Optional[ProfileRead]:
        db_profile = await self._get_instance(profile_id)

        if db_profile is None:
            raise EntityDoesNotExist

        return ProfileRead(**db_profile.dict())
```

Рисунок 3.5 – Лістинг реалізації методів для взаємодії з користувачами

```python
    async def patch(
        self, profile_id: UUID, profile_patch: ProfilePatch
    ) -> Optional[ProfileRead]:
        db_profile = await self._get_instance(profile_id)

        if db_profile is None:
            raise EntityDoesNotExist

        profile_data = profile_patch.dict(exclude_unset=True, exclude={"id"})
        for key, value in profile_data.items():
            setattr(db_profile, key, value)

        self.session.add(db_profile)
        await self.session.commit()
        await self.session.refresh(db_profile)

        return ProfileRead(**db_profile.dict())

    async def delete(self, profile_id: UUID) -> None:
        db_profile = await self._get_instance(profile_id)

        if db_profile is None:
            raise EntityDoesNotExist

        await self.session.delete(db_profile)
        await self.session.commit()
```

Рисунок 3.6 – Лістинг реалізації методів для взаємодії з користувачами (продовження)

Нижче представлений опис вищепредставлених методів:

1. Конструктор (__init__) приймає об'єкт AsyncSession та ініціалізує сесію для взаємодії з базою даних.

2. Асинхронний метод _get_instance: Приймає ідентифікатор профілю (profile_id) і повертає екземпляр профілю з бази даних, якщо він існує.

3. Асинхронний метод create: Створює новий профіль на основі даних profile_create. Додає профіль у сесію та зберігає його в базі даних.

4. Асинхронний метод list: Повертає список профілів з бази даних з можливістю пагінації та фільтрації (використовуючи параметри limit та offset).

5. Асинхронний метод list_exclude_ids: Повертає список профілів, виключаючи ті, що мають ідентифікатори у списку exclude_ids, з обмеженням на кількість елементів (limit).

6. Асинхронний метод get: За ідентифікатором профілю (profile_id) повертає об'єкт ProfileRead. Якщо профіль не знайдено, повертає виключення EntityDoesNotExist.

7. Асинхронний метод patch: Оновлює дані профілю за його ідентифікатором (profile_id) відповідно до переданого об'єкта profile_patch. Якщо профіль не знайдено, викидає виняток EntityDoesNotExist.

На рисунку 3.7 представлений лістинг ініціалізації Redis та Memcached.

```python
import asyncio

from aiocache import caches
from core.config import settings

caches.set_config({
    'default': {
        'cache': "aiocache.MemCache",
        'serializer': {
            'class': "aiocache.serializers.StringSerializer"
        }
    },
    'cache': {
        'cache': "aiocache.RedisCache",
        'endpoint': settings.redis_server,
        'port': settings.redis_port,
        'timeout': 1,
        'serializer': {
            'class': "aiocache.serializers.PickleSerializer"
        },
        'plugins': [
            {'class': "aiocache.plugins.HitMissRatioPlugin"},
            {'class': "aiocache.plugins.TimingPlugin"}
        ]
    }
})
```

Рисунок 3.7 – Лістинг ініціалізації Redis та Memcached клієнтів

Функція redis_cache створює та повертає клієнт Redis. Параметри підключення (host і port) беруться з конфігураційних налаштувань (settings), що може включати адресу сервера Redis та порт. Повертає об'єкт, що дозволяє взаємодіяти з Redis, наприклад, зберігати або отримувати закешовані дані.

Функція memcached_client створює та повертає клієнт Memcached. Параметри підключення (memcached_server та memcached_port) вказують на сервер Memcached, до якого необхідно підключитися. Функція повертає об'єкт клієнта, який може використовуватися для взаємодії з Memcached операціями.

Далі, розглянемо реалізацію основних елементів програмної системи, з якими можуть взаємодіяти користувачі для проведення операцій з даними.

На рисунку 3.8 представлена реалізація методу для отримання випадкового унікального ідентифікатора користувача системи.

```python
from typing import Dict, Optional, List
from uuid import UUID
from random import shuffle, sample
from fastapi import APIRouter, Body, Depends, HTTPException, Query, status

from api.dependencies.redis import cache
from api.dependencies.repositories import get_repository
from db.errors import EntityDoesNotExist
from db.repositories.profiles import ProfileRepository
from schemas.profiles import (
    ProfilePatch,
    ProfileRead,
    ProfileCreate
)

router = APIRouter()


@router.get(
    "/get_random_profile_id",
    status_code=status.HTTP_200_OK,
    name="Get random profile id",
)
async def get_random_profile_id(
    limit: int = Query(default=0, lte=100),
    repository: ProfileRepository = Depends(get_repository(ProfileRepository)),
) -> Dict:
    profiles_list = await repository.list(limit=limit)
    shuffle(profiles_list)
    return {"profile_id": f"{profiles_list[0].id}"}
```

Рисунок 3.8 – Реалізація методу отримання випадкового унікального ідентифікатора користувача системи

На рисунку 3.9 представлена реалізація методу для створення нового користувача з унікальним ідентифікатором в системі.

```python
@router.post(
    "/profiles",
    status_code=status.HTTP_201_CREATED,
    name="Create profile"
)
async def create_profile(
    profile_create: ProfileCreate = Body(...),
    redis_client: cache = Depends(cache),
    repository: ProfileRepository = Depends(get_repository(ProfileRepository)),
) -> None:
    profile = await repository.create(profile_create=profile_create)
    redis_client.hset(
        f"profile_{profile.id}", mapping={k: str(v) for k, v in dict(profile).items()}
    )
    return {
        'status': status.HTTP_201_CREATED,
        'response': f'Profile with id {profile.id} was succesfully created!'
    }
```

Рисунок 3.9 – Реалізація методу створення нового користувача з унікальним ідентифікатором в системі

Метод create_profile – це асинхронна функція у веб-додатку, яка використовує фреймворк FastAPI для створення нових профілів. Вона визначається за допомогою декоратора @router.post, який позначає, що функція обробляє POST-запити за адресою /profiles і встановлює статус-код HTTP у 201 Created при успішному виконанні. У сигнатурі функції є три параметри: profile_create, який очікує дані для створення профілю та використовує моделі Pydantic для валідації та парсингу запиту; redis_client, який вводить клієнт Redis для кешування за допомогою системи впровадження залежностей FastAPI; та repository, який надає екземпляр ProfileRepository для взаємодії з базою даних. У тілі функції спочатку виконується асинхронний виклик для створення профілю в базі даних за допомогою наданих даних, після чого відбувається оновлення кешу Redis за допомогою методу hset. Ключем у

Redis є profile_{profile.id}, а значенням є словник з атрибутами профілю. На завершення, метод повертає словник зі статус-кодом та повідомленням про успішне створення профілю, включаючи ID новоствореного профілю.

На рисунку 3.10 представлена реалізація методу для отримання профілів користувачів системи з врахуванням параметрів limit та offset.

```python
@router.get(
    "/profiles",
    response_model=List[Optional[ProfileRead]],
    status_code=status.HTTP_200_OK,
    name="Get profiles",
)
async def get_profiles(
    limit: int = Query(default=10, lte=100),
    redis_client: cache = Depends(cache),
    repository: ProfileRepository = Depends(get_repository(ProfileRepository)),
) -> list[Optional[ProfileRead]]:

    def fetch_from_cache(keys):
        profiles = []
        for key in keys:
            profile_data = redis_client.hgetall(key)
            profile_dict = {k.decode(): v.decode() for k, v in profile_data.items()}
            for field in ["website", "current_location"]:
                if (
                    isinstance(profile_dict.get(field), str)
                    and profile_dict[field].startswith('[')
                    and profile_dict[field].endswith(']')
                ):
                    profile_dict[field] = eval(profile_dict[field])
            profiles.append(ProfileRead(**profile_dict))
        return profiles

    all_profile_keys = redis_client.keys("profile_*")
    profiles_list = fetch_from_cache(
        sample(all_profile_keys, min(limit, len(all_profile_keys)))
    )
    cached_ids = [profile.id for profile in profiles_list]

    if len(profiles_list) < limit:
        additional_profiles = await repository.list_exclude_ids(
            cached_ids, limit=(limit - len(profiles_list))
        )
        for profile in additional_profiles:
            redis_client.hset(
                f"profile_{profile.id}", mapping={k: str(v) for k, v in dict(profile).items()}
            )
            profiles_list.append(profile)

    return profiles_list
```

Рисунок 3.10 – Реалізація методу отримання масиву профілів користувачів системи з врахуванням параметрів limit та offset

Метод get_profiles є ключовою частиною веб-додатку, який побудований на основі фреймворку FastAPI. Ця асинхронна функція використовується для отримання інформації про користувальницькі профілі, з використанням HTTP GET-запитів на шлях /profiles. Відповідно до налаштувань FastAPI, функція повертає відповідь зі статусом 200 OK, що вказує на успішне отримання даних.

Параметри методу: limit (за замовчуванням 10, максимум 100): Цей параметр дозволяє користувачу вказати максимальну кількість профілів, яку вони хочуть отримати. Це корисно для контролю обсягу даних, які надсилаються клієнту, особливо важливо для зменшення навантаження на мережу та сервер. redis_client: Цей клієнт Redis використовується для взаємодії з кешованими даними. Користуючись перевагами швидкості та ефективності Redis, цей параметр дозволяє методу швидко отримувати дані без потреби кожного разу звертатися до основної бази даних. Екземпляр репозиторію ProfileRepository забезпечує доступ до операцій з базою даних.

Це дозволяє функції обробляти запити, які не можуть бути виконані через кеш, наприклад, коли запитувана інформація ще не кешована. Логіка виконання: Метод спочатку намагається отримати дані з Redis. Це включає в себе вибірку ключів профілів і витяг даних для кожного профілю. Отримані дані декодуються та обробляються для перетворення у формат, сумісний з моделлю ProfileRead. Запит до бази даних за потреби: Якщо кількість кешованих профілів менша, ніж потрібно за параметром limit, функція здійснює запит до бази даних для отримання додаткових профілів.

Це забезпечує, що користувач отримує необхідну кількість профілів, навіть якщо вони не всі збережені у кеші. Додаткові профілі, отримані з бази даних, додаються до кешу Redis. Це знижує ймовірність необхідності майбутніх запитів до бази даних для цих же профілів, забезпечуючи швидший доступ до них у майбутньому. В результаті повертається список, який може включати як кешовані профілі, так і отримані з БД.

Далі, на рисунку 3.11 представлена реалізація методу для отримання профілів користувачів системи за вказаним унікальним ідентифікатором.

```python
@router.get(
    "/profiles/{profile_id}",
    response_model=ProfileRead,
    status_code=status.HTTP_200_OK,
    name="Get profile data",
)
async def get_profile(
    profile_id: UUID,
    redis_client: cache = Depends(cache),
    repository: ProfileRepository = Depends(get_repository(ProfileRepository)),
) -> ProfileRead:
    """
        Fetches profile data from cache, and if it doesn't exist, get it from the database and then set in cache.
    """

    cached_profile_data = redis_client.hgetall(f"profile_{profile_id}")

    if cached_profile_data:
        cached_data = {k.decode(): v.decode() for k, v in cached_profile_data.items()}
        for field in ["website", "current_location"]:
            if (
                isinstance(cached_data.get(field), str)
                and cached_data[field].startswith('[')
                and cached_data[field].endswith(']')
            ):
                cached_data[field] = eval(cached_data[field])
        return ProfileRead(**cached_data)

    try:
        profile = await repository.get(profile_id=profile_id)
        redis_client.hset(
            f"profile_{profile_id}", mapping={k: str(v) for k, v in dict(profile).items()}
        )
        return profile
    except EntityDoesNotExist:
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND, detail="Profile not found!"
        )
```

Рисунок 3.11 – Реалізація методу для отримання профілів користувачів системи за вказаним унікальним ідентифікатором

Метод get_profile є частиною веб-додатку, що використовує фреймворк FastAPI для отримання даних конкретного користувацького профілю. Він визначений за допомогою декоратора @router.get, що вказує на обробку GET-запитів до адреси /profiles/{profile_id}, де {profile_id} - це змінна частина шляху, що представляє унікальний ідентифікатор профілю (UUID). Метод повертає об'єкт типу ProfileRead у разі успішного виконання зі статусом HTTP 200 OK. Параметри методу: profile_id: Унікальний ідентифікатор профілю (UUID), який визначає конкретний профіль, що потрібно отримати. redis_client: Клієнт Redis, який використовується для доступу до кешованих даних. repository: Екземпляр ProfileRepository, який надає методи для взаємодії

з базою даних. Спочатку метод намагається знайти дані профілю у кеші Redis. Якщо дані існують, вони декодуються та повертаються. У цьому процесі також відбувається спеціальна обробка для певних полів, якщо вони представлені у форматі списку. Якщо профіль не знайдено у кеші, метод звертається до бази даних за допомогою repository. Це включає в себе асинхронний запит на отримання даних профілю за його ідентифікатором. Якщо дані профілю успішно отримані з бази даних, вони кешуються у Redis для швидшого доступу в майбутньому. Після отримання даних, незалежно від джерела, метод повертає об'єкт ProfileRead, заповнений відповідними даними профілю. У випадку, якщо профіль не існує у базі даних, метод генерує HTTP-виняток зі статусом 404 (Not Found), інформуючи про те, що профіль не знайдений.

На рисунку 3.12 представлена реалізація методу для оновлення даних в профілі користувача в базі даних, а також в системі кешування Redis.

```python
@router.put(
    "/profiles/{profile_id}",
    status_code=status.HTTP_200_OK,
    name="Update profile data",
)
async def update_profile(
    profile_id: UUID,
    redis_client: cache = Depends(cache),
    profile_patch: ProfilePatch = Body(...),
    repository: ProfileRepository = Depends(get_repository(ProfileRepository)),
) -> None:
    """
        Stores the given profile item in the database and cache, replacing the previous one.
    """

    try:
        updated_profile = await repository.patch(
            profile_id=profile_id, profile_patch=profile_patch
        )
        redis_client.delete(f"profile_{profile_id}")
        redis_client.hset(
            f"profile_{updated_profile.id}", mapping={k: str(v) for k, v in dict(updated_profile).items()}
        )
        return {
            'status': status.HTTP_200_OK,
            'response': f'Profile with id {profile_id} was succesfully updated!'
        }
    except EntityDoesNotExist:
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND, detail="Profile not found!"
        )
```

Рисунок 3.12 – Реалізація методу для оновлення даних в профілі користувача в базі даних та в системі кешування

На рисунку 3.13 представлена реалізація методу для видалення даних профілю користувача в базі даних, а також в системі кешування Redis.

```python
@router.delete(
    "/profiles/{profile_id}",
    status_code=status.HTTP_200_OK,
    name="Delete profile data",
)
async def delete_profile(
    profile_id: UUID,
    redis_client: cache = Depends(cache),
    repository: ProfileRepository = Depends(get_repository(ProfileRepository)),
) -> None:
    """
        Delete the profile item with the given profile unique identifier.
    """

    try:
        await repository.delete(
            profile_id=profile_id
        )
        redis_client.delete(f"profile_{profile_id}")
        return {
            'status': status.HTTP_200_OK,
            'response': f'Profile with id {profile_id} was succesfully deleted!'
        }
    except EntityDoesNotExist:
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND, detail="Profile not found!"
        )
```

Рисунок 3.13 – Реалізація методу для видалення даних профілю користувача в базі даних та в системі кешування

Метод delete_profile призначений для видалення користувацького профілю. Він використовує анотацію @router.delete, що означає, що метод обробляє HTTP DELETE запити на шлях /profiles/{profile_id}, де {profile_id} є змінною частиною URL, що представляє унікальний ідентифікатор профілю Параметри методу: profile_id: Унікальний ідентифікатор профілю (UUID), який визначає профіль, що потрібно видалити. redis_client: Клієнт Redis для взаємодії з кешованими даними. repository: Екземпляр ProfileRepository, який

надає доступ до функцій взаємодії з базою даних. Спочатку метод робить асинхронний запит до repository для видалення профілю за допомогою його ідентифікатора. Це забезпечує, що профіль буде вилучений з основної бази даних. Після видалення профілю з бази даних, відповідний запис у кеші Redis також видаляється. Це запобігає ситуації, коли застарілі або недійсні дані можуть бути отримані з кешу. У разі успішного видалення, метод повертає повідомлення зі статусом HTTP 200 OK, інформуючи, що профіль було успішно видалено. Якщо профіль з вказаним ідентифікатором не знайдено, метод генерує HTTP відповідь зі статусом 404 (Not Found).

Додатково були визначені ендпоінти, які відповідають за попередню ініціалізацію бази даних та створення профілів користувачів (рисунок 3.14).

```python
app = FastAPI(
    title=settings.title,
    version=settings.version,
    description=settings.description,
    docs_url=settings.docs_url,
    openapi_url=settings.openapi_url,
)

app.include_router(router, prefix=settings.api_prefix)


@app.get("/")
async def root():
    return {"status": "200", "service": "FastAPI Cache Comparison"}


@app.get("/create_tables")
async def create_tables():
    await create_db_and_tables()
    return {"response": "Table users successfully created in database!"}


@app.get("/create_profiles/{number}")
async def create_profiles(number: int):
    await bulk_create_profiles(number)
    return {"response": f"{number} profiles succesfully created!"}
```

Рисунок 3.14 – Реалізація ендпоінтів, які відповідають за попередню ініціалізацію бази даних та створення профілів користувачів

### 3.3 Тестування розробленої програмної системи

Для тестування розробленої програмної системи було розроблено додатковий Docker-скрипт для запуску всіх сервісів в одному контейнері.

На рисунку 3.15 представлена реалізація скрипта docker-compose.yaml.

```yaml
version: "3.9"

services:
  fastapi_service:
    build:
      context: ./app
      dockerfile: Dockerfile
    hostname: fastapi_service
    container_name: fastapi_service
    depends_on:
      - db_postgres
    ports:
      - "8000:8000"
    env_file:
      - app/.env
    volumes:
      - ./app:/home/app
    networks:
      - my-net

  cache-redis:
    image: redis:alpine
    hostname: cache-redis
    container_name: cache-redis
    restart: on-failure
    ports:
      - '6379:6379'
    expose:
      - '6379'
    command: redis-server
    volumes:
      - cache:/data
    networks:
      - my-net

  db_postgres:
    image: postgres:14.3-alpine
    hostname: db_postgres
    container_name: db_postgres
    restart: on-failure
    environment:
      - POSTGRES_USER=postgres
      - POSTGRES_PASSWORD=postgres
    ports:
      - "5432:5432"
    volumes:
      - db-postgres:/var/lib/postgresql/data
    networks:
      - my-net
```

Рисунок 3.15 – Реалізація Docker-скрипта для запуску застосунку

Для запуску застосунку за вищезазначеним скриптом необхідно запустити команду: docker-compose up. На рисунку 3.16 представлений результат запуску програмної системи з відображенням в Docker Desktop.


Рисунок 3.16 – Відображення запущених сервісів Docker контейнера

На рисунку 3.17 представлений результат перевірки логування FastAPI застосунку, де видно успішну ініціалізацію та запуск програмної системи.

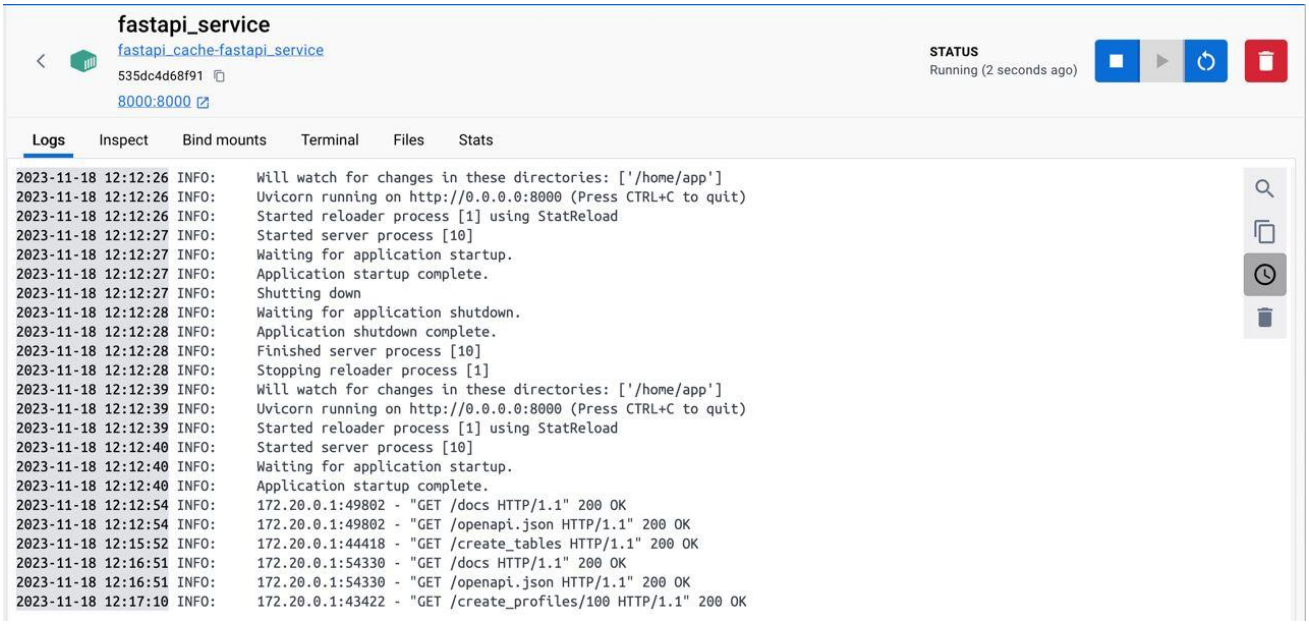



Рисунок 3.17 – Відображення результату успішної ініціалізації сервіса FastAPI

На рисунку 3.18 представлений вигляд Swagger інтерфейсу з відображенням доступних базових ендпоінтів та ендпоінтів для користувачів.

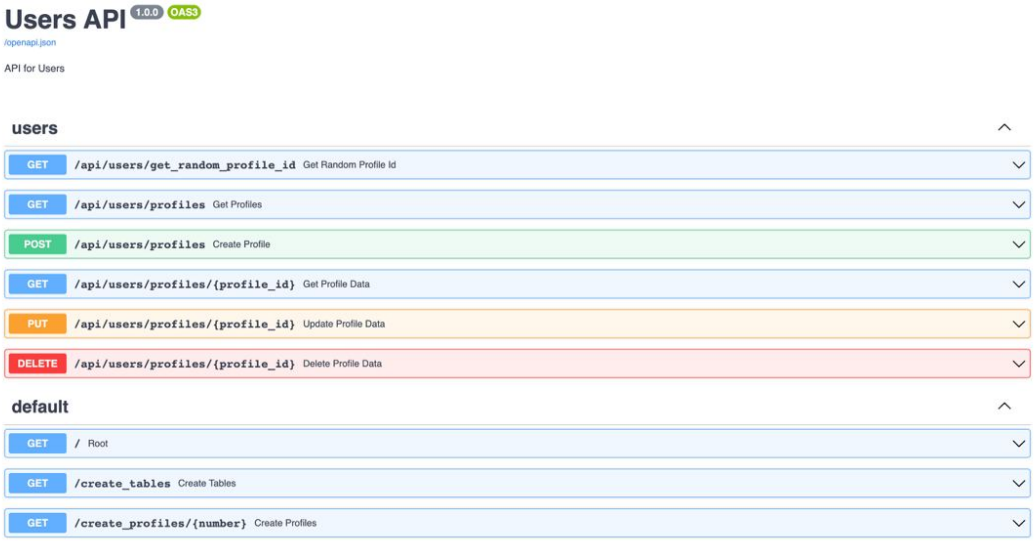



Рисунок 3.18 – Swagger інтерфейс розробленої програмної системи

Для початку роботи з профілями користувачів, необхідно створити таблицю в базі даних для їх збереження, а також самі профілі користувачів. На рисунку 3.19 представлена відповідь від сервера на запит створення таблиці.

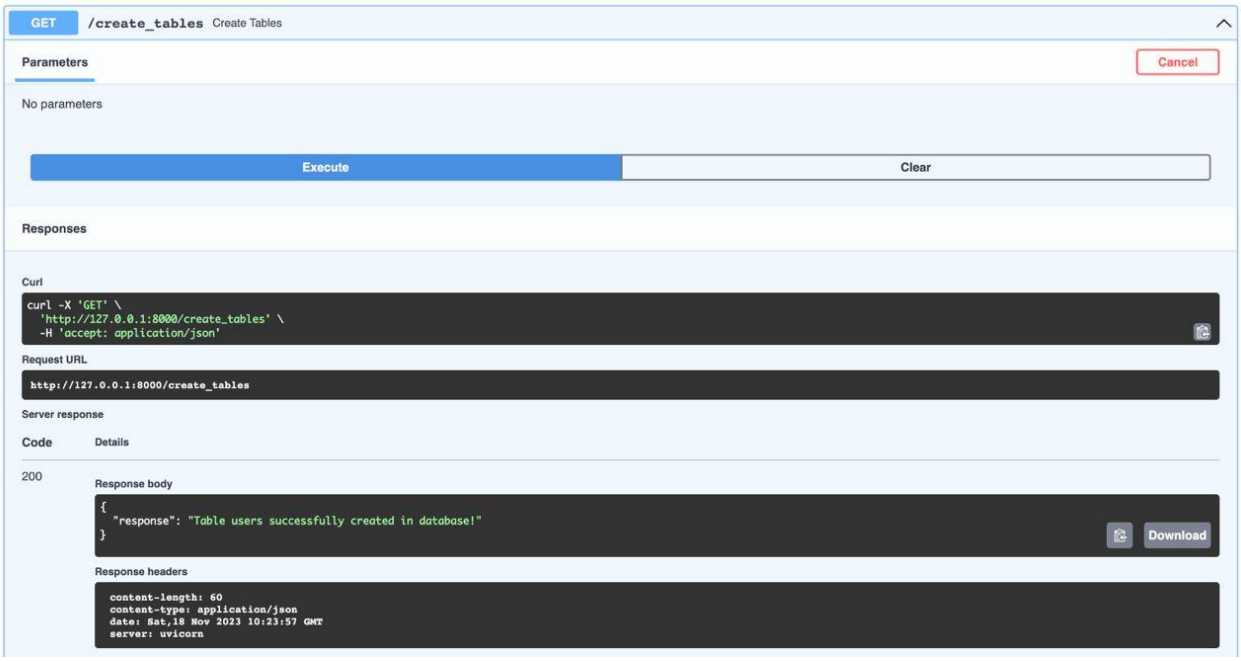



Рисунок 3.19 – Результат створення таблиці для збереження користувачів в високонавантаженій програмній системі

Після успішного створення таблиці для збереження користувачів, створимо 1000 користувачів для тестування можливостей програмної системи. На рисунку 3.20 представлений результат створення користувачів.

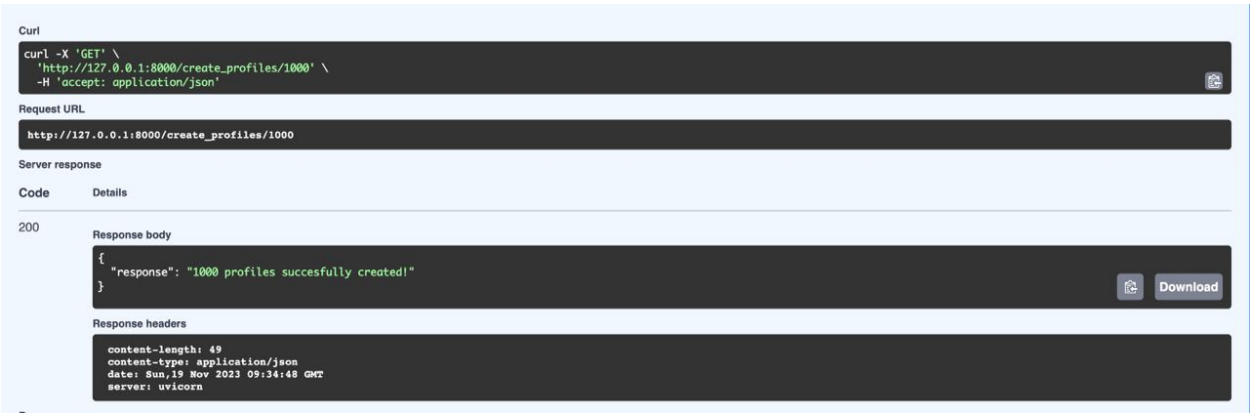



Рисунок 3.20 – Результат створення користувачів в системі

На рисунку 3.21 представлений результат отримання випадкового унікального ідентифікатора користувача. Параметр limit вказує на кількість профілів, які будуть взяті для формування випадкового ідентифікатора.

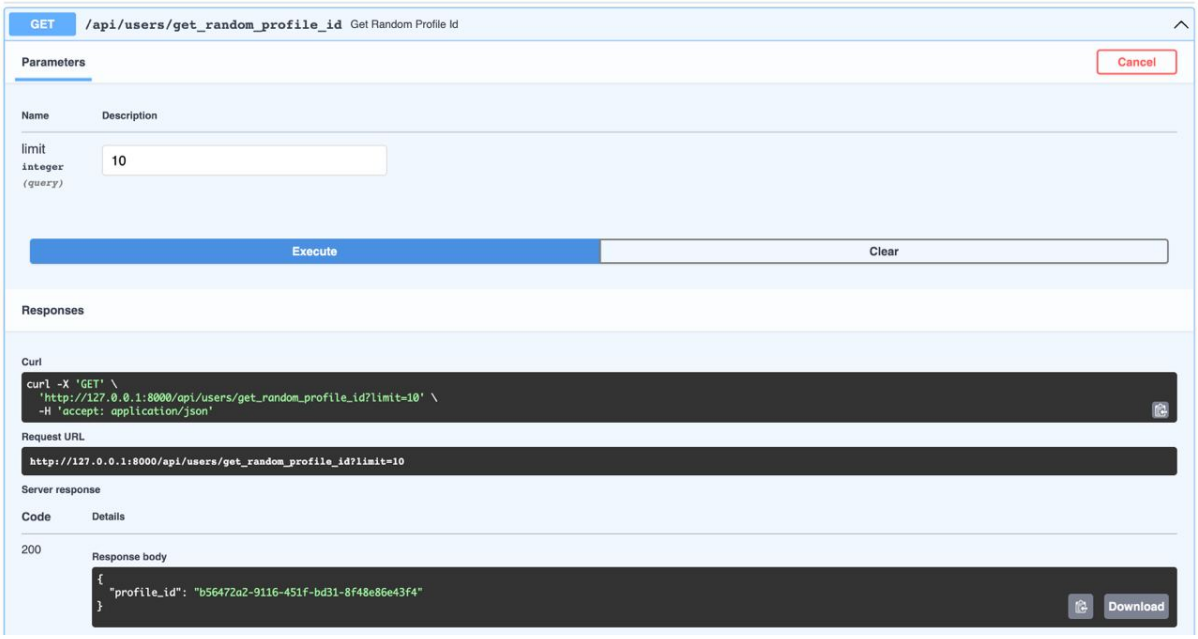



Рисунок 3.21 – Результат отримання випадкового унікального ідентифікатора

Далі, розглянемо ендпоінт для отримання інформації по користувачу за унікальним ідентифікатором. У випадку, якщо система знаходить запитуваний ідентифікатор у кеші – об’єкт повертається з кешу, якщо ні – з БД, а потім встановлюється в кеш. На рисунку 3.22 представлений вигляд параметру для формування запиту, а на рисунку 3.23 представлена відповідь від сервера.

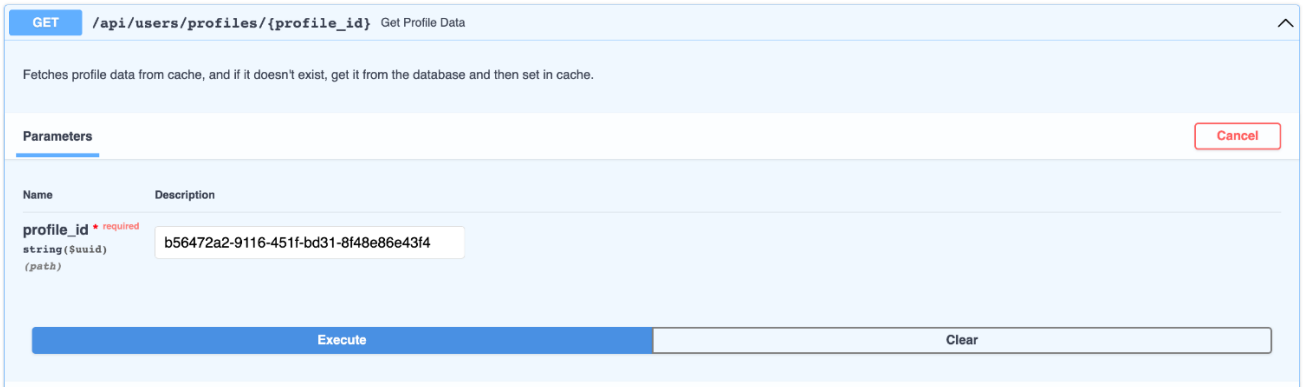



Рисунок 3.22 – Задання параметру ідентифікатора

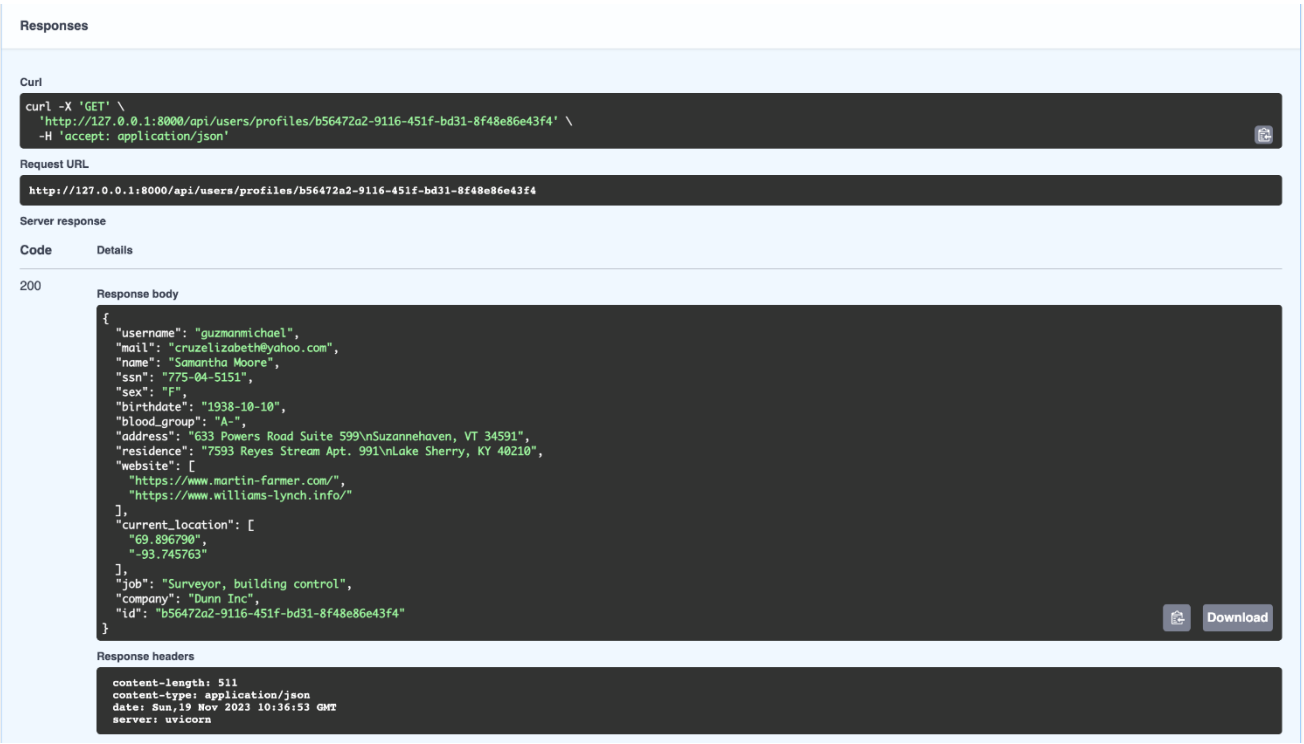



Рисунок 3.23 – Результат отримання даних по користувачу (з бази даних)

Для перевірки коректності встановлення значень по користувачу до кешу використовується клієнт RedisInsight (безкоштовний інструмент для управління та моніторингу стану кешу Redis). На рисунку 3.24 представлено відображення встановленого до кешу користувача з відповідними значеннями.

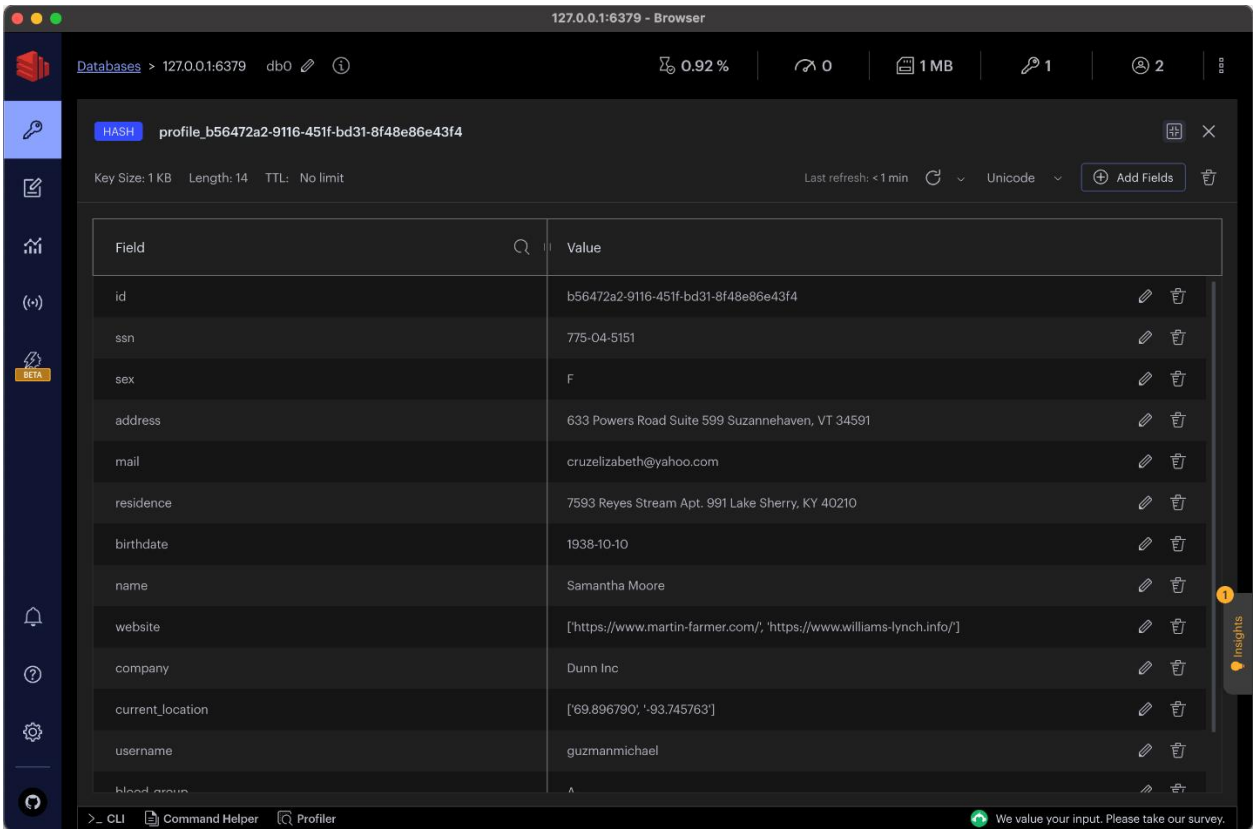



Рисунок 3.24 – Результат встановлення даних користувача до кешу Redis

На цьому етапі можна провести початкове порівняння продуктивності використання бази даних та систем кешування Redis/Memcached. Оскільки Memcached не має доступних GUI-клієнтів, було окремо протестовано встановлення даних до кешу та отримання даних з кешу.

Для порівняння продуктивності використовувалась наступна команда: curl 'http://localhost:8000/api/users/profiles/{profile_id}' -s -o /dev/null -w

"%{time_starttransfer}\n". Час time_starttransfer вимірюється в секундах від моменту початку операції до початку передачі першого байту відповіді.

Порівняльний аналіз вищезазначених систем представлено в табл 3.1.

Таблиця 3.1 – Порівняльний аналіз Redis, Memcached, PostgreSQL (ендпоінт profile/{profiles_id}

| Система / Операція | Redis | Memcached | PostgreSQL |
|---|---|---|---|
| GET | **8,1 мс** | 10,1 мс | 61,22 мс |
| SET | 12,6 мс | **11,4 мс** | Не виконується |

На основі проведеного аналізу для одного запису користувача виявлено, що Redis є продуктивнішим за Memcached в операціях GET, а при проведенні операції встановлення до кешу Memcached є трохи швидшим, але необхідно провести тестування з більшою кількістю операцій для наочності.

На рисунку 3.25 представлений результат запиту для отримання переліку записів по користувачам високонавантаженої системи.

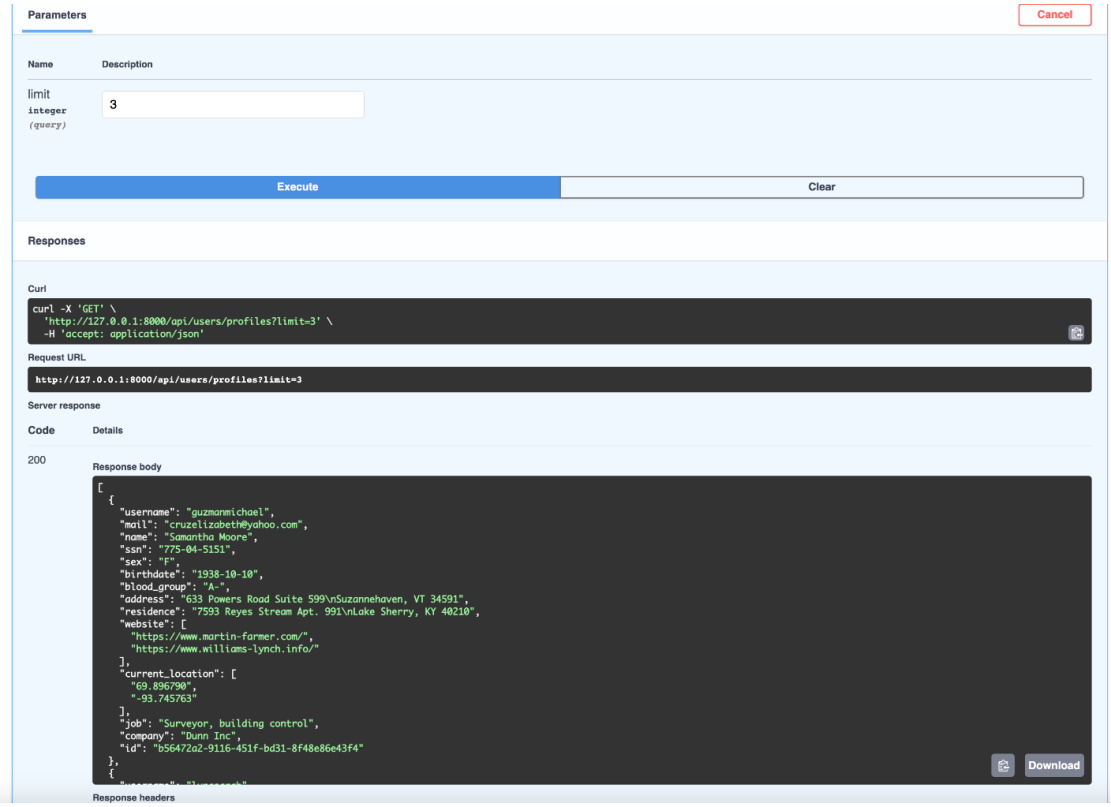



Рисунок 3.25 – Результат отримання списку користувачів системи

На рисунку 3.26 представлено результат встановлення нових даних до Redis сховища. Оскільки в кеші вже знаходився один HASH-запис, то він був використаний для формування швидшої відповіді від сервера.

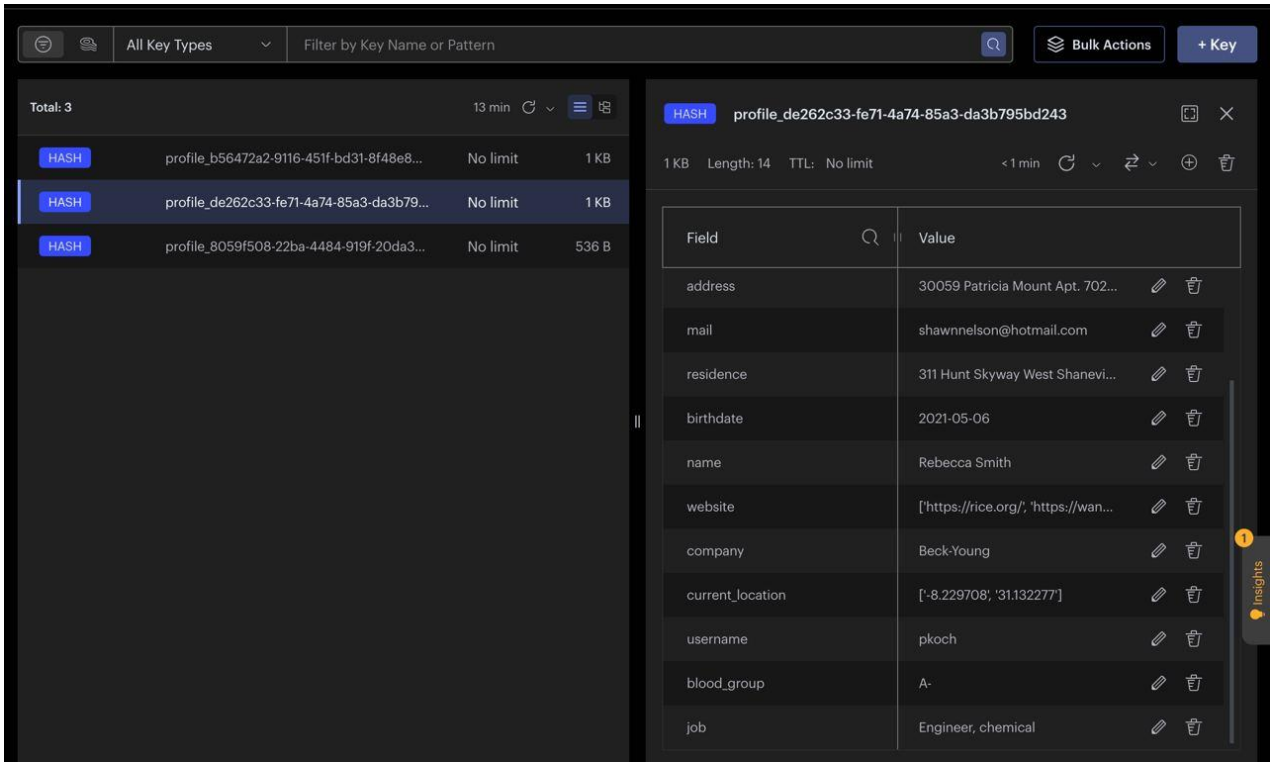



Рисунок 3.26 – Результат встановлення нових записів до сховища

На рисунку 3.27 представлена параметризація запиту для видалення даних по користувачу з бази даних та з кешу (якщо ID користувача існує).

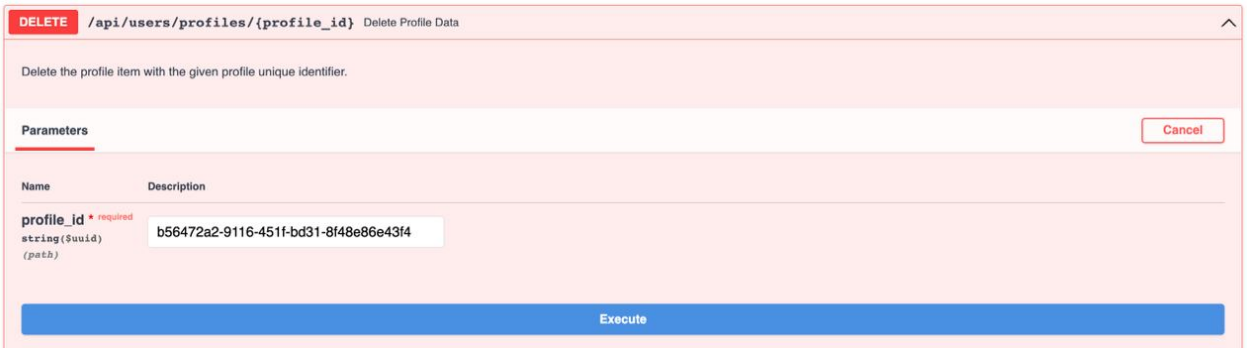



Рисунок 3.27 – Формування запиту для видалення користувача з системи

На рисунку 3.28 представлений результат видалення даних по користувачу з системи.

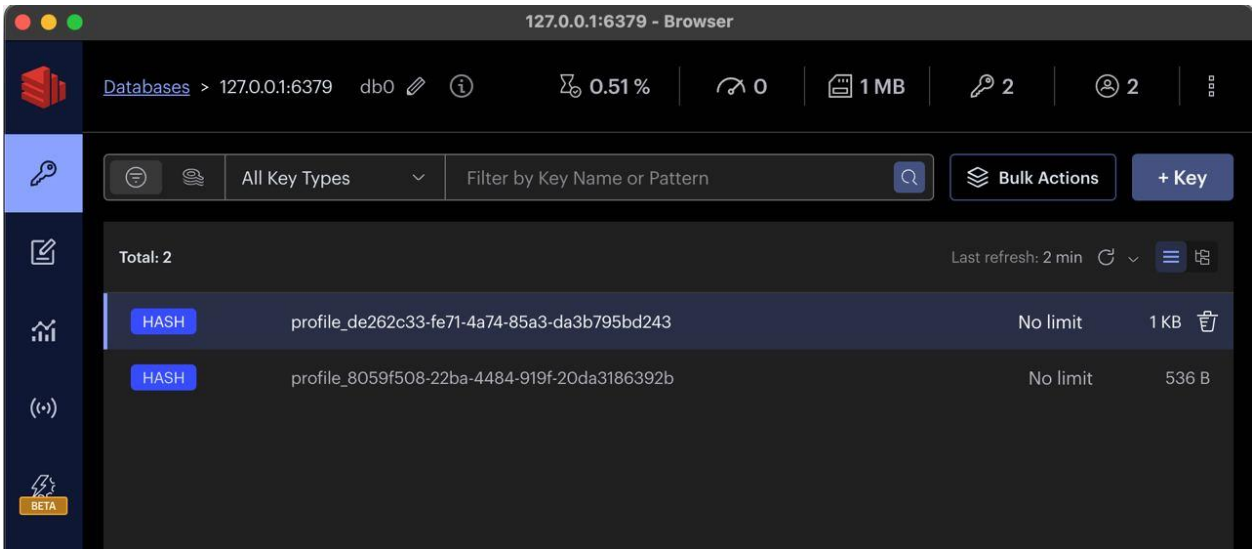



Рисунок 3.28 – Результат видалення даних по користувачу з системи

На рисунку 3.29 представлений приклад формування запиту на оновлення даних по користувачу за унікальним ідентифікатором.

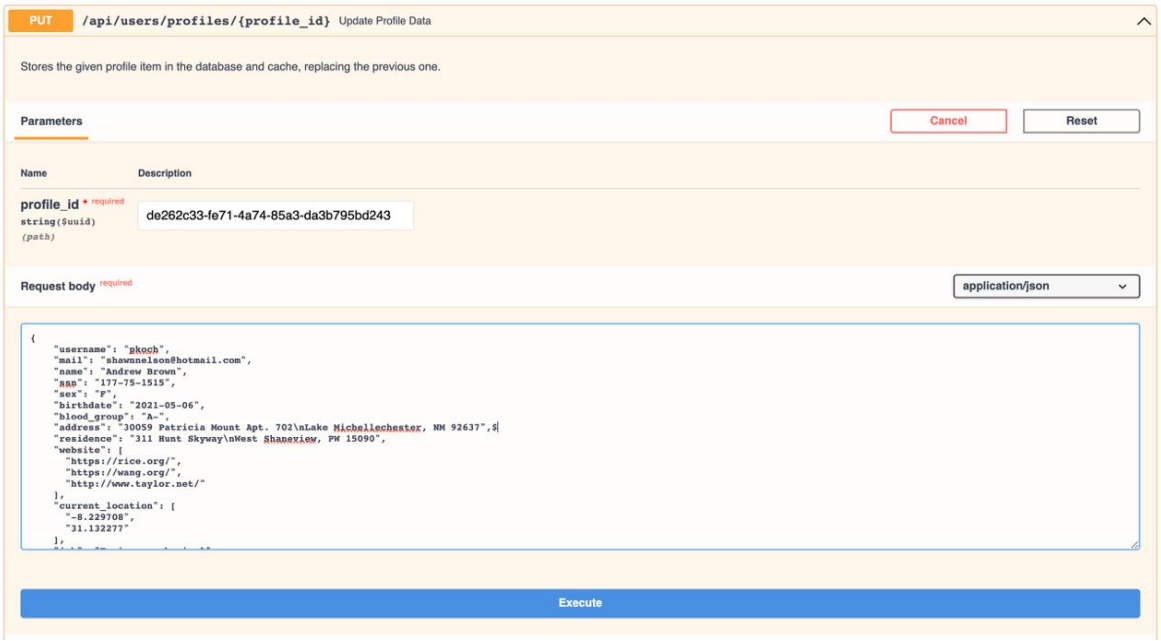



Рисунок 3.29 – Параметризація запиту на оновлення даних користувача

На рисунку 3.30 представлені дані по користувачу з кешу перед оновленням даних, а на рисунку 3.31 після відправки запита на оновлення.

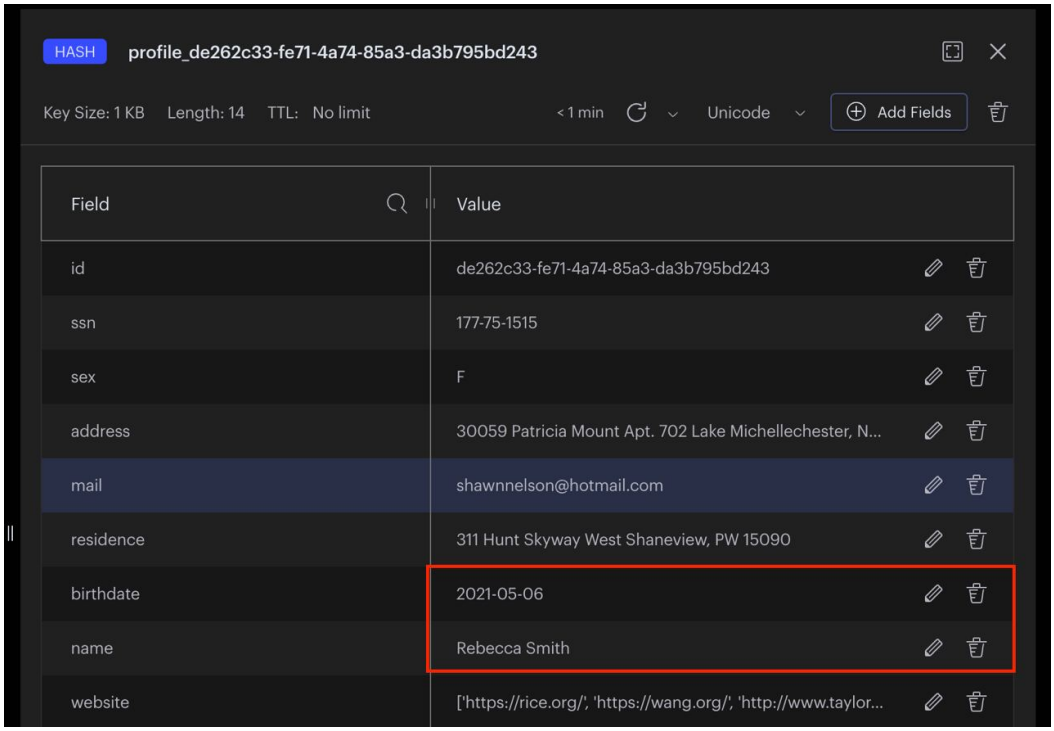



Рисунок 3.30 – Дані по користувачу до оновлення даних з API

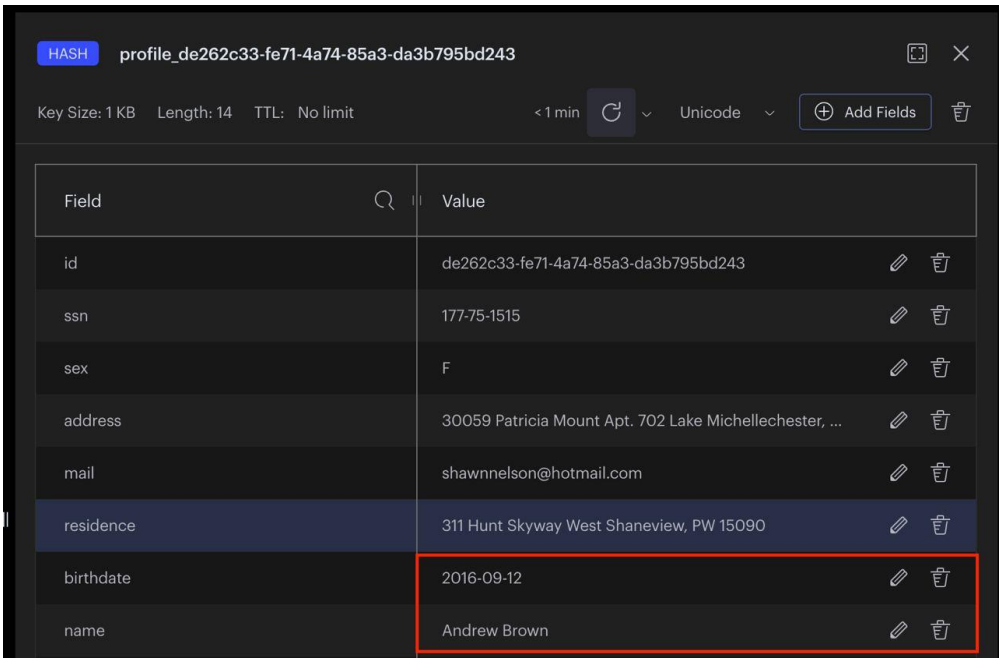



Рисунок 3.31 – Дані по користувачу після оновлення даних з API

На рисунку 3.32 представлена параметризація запиту на створення користувача в системі, а на рисунку 3.33 результат відправки запиту на сервер.

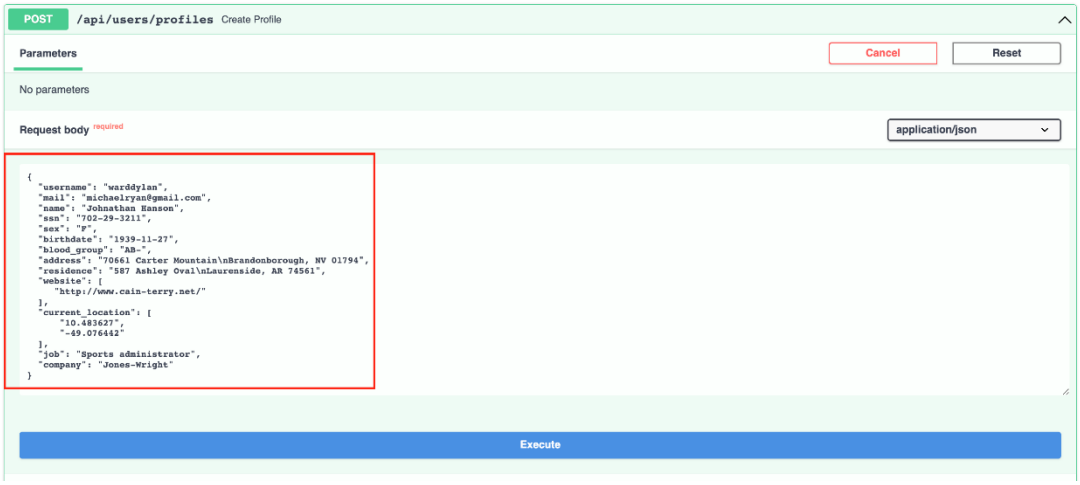



Рисунок 3.32 – Параметризація запиту на створення нового користувача

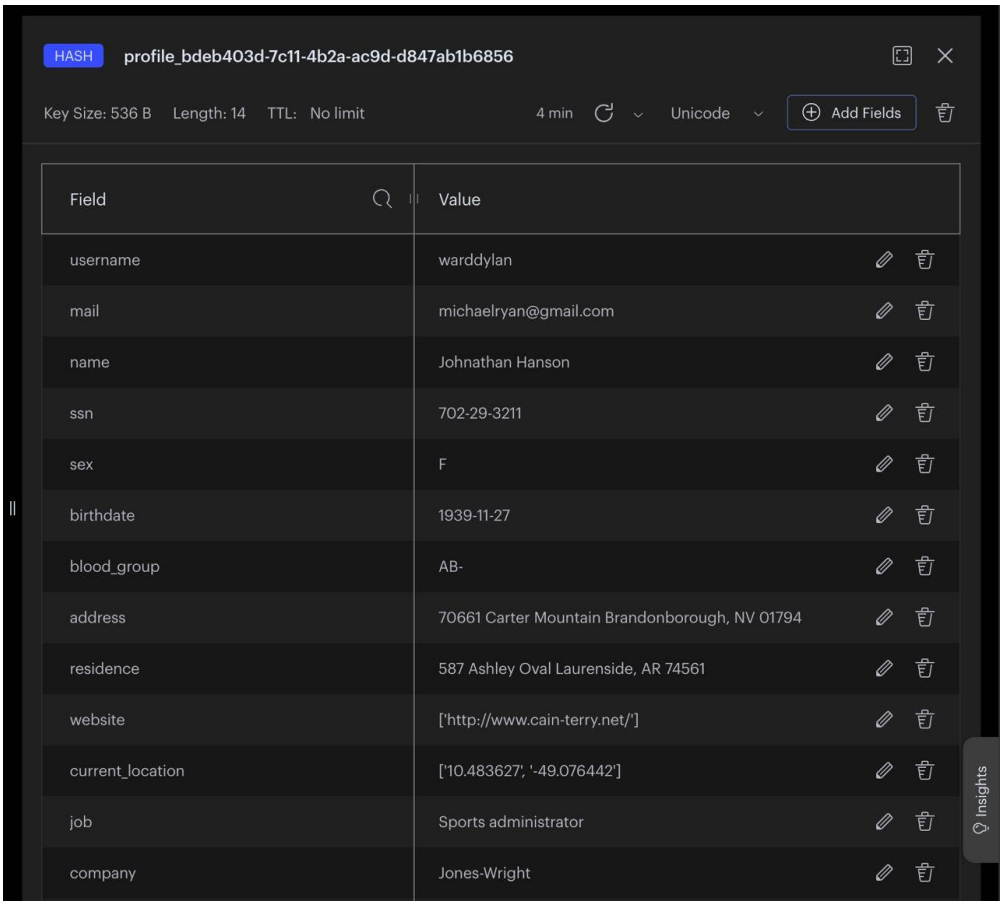



Рисунок 3.33 – Результат створення нового користувача в системі

# 4 ПОРІВНЯЛЬНИЙ АНАЛІЗ ПІДХОДІВ КЕШУВАННЯ

На основі розробленої системи необхідно провести порівняльний аналіз основних показників систем кешування, які були використані при розробці.

Для цього буде використано підхід навантаженого тестування. Навантажувальне тестування передбачає створення змодельованого середовища, де декілька користувачів одночасно отримують доступ до програмного забезпечення, таким чином імітуючи очікувані моделі використання. Ця практика спрямована на оцінку ефективності програми за реалістичних і складних умов. Піддаючи програмне забезпечення значному робочому навантаженню, це дозволяє оцінити його здатність обробляти одночасну взаємодію користувача та визначити потенційні обмеження. Більшість інструментів і фреймворків навантажувального тестування дотримуються традиційної парадигми навантажувального тестування, яка передбачає запис спілкування між клієнтами та вашим веб-сайтом за допомогою запису сценаріїв. На основі цих записів генеруються сценарії взаємодії. Під час процесу навантажувального тестування ці записані сценарії відтворюються за допомогою генератора навантаження, який за потреби можна налаштувати за допомогою інших параметрів тестування.

Для проведення тестування було вирішено використовувати фреймворк Locust. Locust – це інструмент навантажувального тестування з відкритим кодом, призначений для оцінки продуктивності та масштабованості веб-додатків. На відміну від багатьох традиційних інструментів тестування, він дозволяє писати тестові сценарії на Python, уможливлюючи моделювання складної поведінки користувача та взаємодії з програмою. Ця гнучкість є ключовою перевагою Locust, оскільки вона дозволяє моделювати широкий спектр дій користувача. Однією з визначальних особливостей Locust є його здатність виконувати розподілене тестування. Це означає, що тести можна запускати на кількох машинах, що важливо для симуляції великої кількості користувачів і розуміння того, як веб-додаток працює під значним

навантаженням. Locust також містить зручний веб-інтерфейс. За допомогою цього інтерфейсу можна розпочинати тести, відстежувати прогрес у режимі реального часу та переглядати детальну статистику продуктивності програми, таку як час відповіді, кількість запитів та кількість активних користувачів.

Іншим важливим аспектом Locust є його масштабованість. Він призначений для ефективного моделювання дуже великої кількості одночасно працюючих користувачів, від тисяч до потенційно мільйонів.

Ця масштабованість життєво важлива для тестування програм, які, як очікується, обслуговуватимуть велику базу користувачів. Крім того, Locust відомий своєю легкістю та ефективністю. Він мінімально навантажує саму машину для тестування, що дозволяє виділяти більше ресурсів для генерування трафіку та точного моделювання поведінки користувача.

На рисунку 4.1 представлений інтерфейс Locust для налаштування параметрів. Для наочності, тестування систем кешування Redis/Memcached буде запущено при відсутності початкових даних у кеші, а тільки у БД.

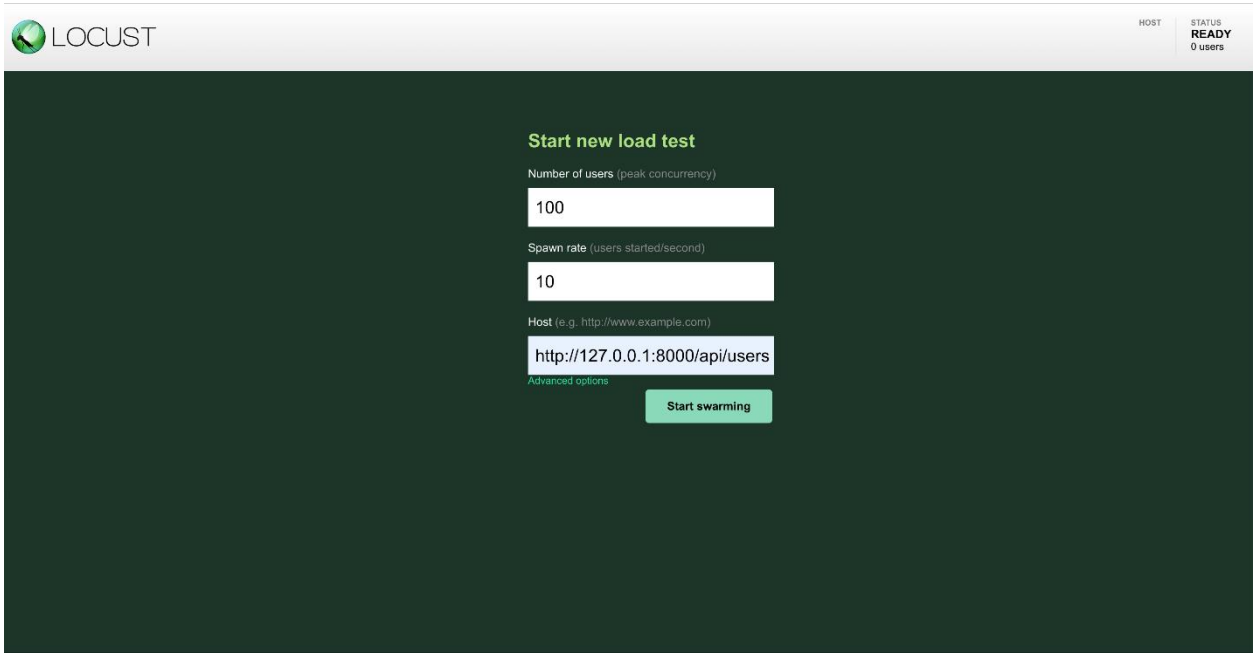



Рисунок 4.1 – Інтерфейс Locust для налаштування параметрів

На рисунках 4.2 – 4.3 представлений результат тестування розроблених ендпоінтів з використанням Memcached та Redis для операції запису даних.


Рисунок 4.2 – Результат навантажувального тестування системи з використанням Memcached (операція запису даних)

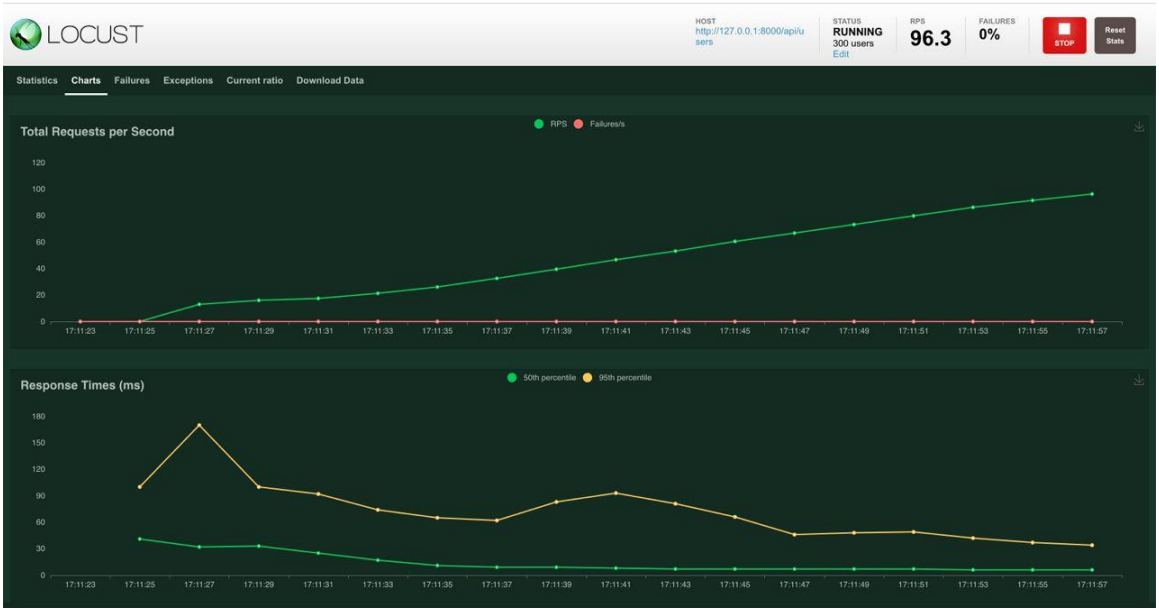


Рисунок 4.3 – Результат навантажувального тестування системи з використанням Redis (операція запису даних)

На рисунках 4.4 – 4.5 представлений результат тестування розроблених ендпоінтів з використанням Memcached та Redis для операції читання.


Рисунок 4.4 – Результат навантажувального тестування системи з використанням Memcached (операція читання даних)

Рисунок 4.5 – Результат навантажувального тестування системи з використанням Redis (операція читання даних)

На основі проведеного тестування, можна сформувати результати проведення порівняльного аналізу систем Redis та Memcached в таблиці 4.1. Наведені результати є значеннями медіани для зазначених в таблиці операцій.

Таблиця 4.1 – Порівняльний аналіз підходів кешування, мс

| Система кешування/ Кількість операцій | Redis | Memcached |
|---|---|---|
| **Читання (операція GET)** | | |
| 1000 операцій | 8 | 9 |
| 10000 операцій | 6 | 14 |
| 100000 операцій | 8 | 14 |
| **Запис (операція SET)** | | |
| 1000 операцій | 34 | 23 |
| 10000 операцій | 214 | 100 |
| 100000 операцій | 1,666 | 276 |
| **Використання пам’яті (операція SET)** | | |
| 1000 операцій | 2,5 | 5,3 |
| 1000 операцій | 3,8 | 27,2 |
| 100000 операцій | 4,3 | 211 |

На основі представлених даних в таблиці 4.1 можна зробити наступні висновки відносно вибору системи кешування:

1. Redis ефективніше використовує пам'ять при операціях запису, що забезпечує кращу продуктивність в умовах великих об'ємів даних.

2. Redis демонструє більш консистентні результати у читанні, що важливо для високонавантажених систем, де стабільність є критичною.

3. При невеликій та середній кількості операцій запису Memcached виявився ефективнішим, вимагаючи менше часу на виконання.

4. При великій кількості операцій запису, Memcached показує кращу продуктивність, вимагаючи значно менше часу порівняно з Redis.

5. Redis забезпечує більшу стабільність і передбачуваність часу відгуку, які є важливими для систем, що обробляють велику кількість запитів.

Загалом, Memcached надає пріоритет високій продуктивності та винятковому часу відгуку. Він також масштабується вертикально, дозволяючи додавати більше серверів до пулу кешування для збільшення трафіку та навантаження даних. Redis пропонує порівнянну продуктивність для простих завдань кешування та додаткові функції для більш складних випадків використання. Він забезпечує асинхронні та неблокуючі операції введення-виведення, дозволяючи програмі виконувати більше одночасних завдань.

Ця функціональність покращує продуктивність під час великих навантажень. Redis підтримує кластери для горизонтального масштабування, що дозволяє продовжувати виконання, якщо вузли виходять з ладу. Redis розроблений як постійний, масштабований і надійний. Redis підтримує розширені структури даних, такі як списки, набори, хеші та відсортовані набори, що робить його більш гнучким, ніж Memcached, але також трохи повільнішим. З точки зору тестів, він може змінюватися залежно від конкретного випадку використання та робочого навантаження.

На основі вищезазначеного можна зробити висновок, що Redis є універсальним рішенням для високонавантажених систем завдяки його ефективному використанню пам'яті, функціональності та масштабованості.

# ВИСНОВКИ

В першому розділі кваліфікаційної роботи було проведено аналіз предметної області, а саме: розглянуто теоретичні основи кешування даних, досліджено технології кешування та визначено актуальність дослідження. На основі наведеного аналізу зроблено висновок, що кешування – це важливий атрибут комп'ютеризованих систем, який допомагає підвищити продуктивність та зменшити навантаження на системи. Кешування може бути застосоване на різних рівнях інфраструктури, включаючи веб-браузери, веб-сервери, що збільшує продуктивність програм і підвищує пропускну здатність.

В другому розділі кваліфікаційної роботи було розглянуто алгоритми, методи та технології кешування даних в розподілених системах. Для проведення дослідження виділено дві основні системи: Redis та Memcached. На основі проведеного аналізу визначено, що Memcached – це просте сховище ключів і значень, яке підтримує невеликі довільні типи даних, такі як рядки та об’єкти. Він підходить для програм, які потребують простого кешування та швидкого пошуку на основі ключів. Як наслідок, він не є виключно придатним для більш складних випадків використання, які вимагають передових методів обробки даних. Водночас Redis підтримує більший спектр структур даних, включаючи рядки, хеші, списки, набори та геопросторові дані. Ця гнучкість дозволяє застосовувати складні випадки використання, наприклад реалізацію аналітики в реальному часі або отримання даних на основі місцезнаходження.

В третьому розділі кваліфікаційної роботи розглянуто проектування та розробка програмної системи з використанням Python. Проведено опис реалізації програмної системи з використанням FastAPI. В результаті, проведено мануальне тестування розроблених ендпоінтів програмної системи.

В четвертому розділі проведено навантажувальне тестування програмної системи з використанням Locust та сформовано порівняльний аналіз. В результаті проведення порівняльного аналізу визначено, що Redis ефективніше використовує пам'ять при операціях запису, що забезпечує кращу продуктивність в умовах великих об'ємів даних. Також, Redis демонструє більш консистентні результати у читанні, що важливо для високонавантажених систем, де стабільність є критичною. Додатково визначено, що Redis забезпечує більшу стабільність і передбачуваність часу відгуку, які є важливими для систем, що обробляють велику кількість запитів.

Загалом, Memcached надає пріоритет високій продуктивності та винятковому часу відгуку. Він також масштабується вертикально, дозволяючи додавати більше серверів до пулу кешування для збільшення трафіку та навантаження даних. Redis пропонує порівнянну продуктивність для простих завдань кешування та додаткові функції для більш складних випадків використання. Він забезпечує асинхронні та неблокуючі операції введення-виведення, дозволяючи програмі виконувати більше одночасних завдань.

Memcached працює лише в пам’яті, тобто зберігає всі дані в оперативній пам’яті та не має вбудованого збереження. Такий підхід забезпечує максимальну продуктивність і низьку затримку доступу до даних. Однак Memcached не зберігає дані автоматично у разі збою системи. Новіші версії підтримують відновлення даних після перезапуску та постійної пам’яті через монтування файлової системи DAX. Redis пропонує необов’язкове збереження даних за допомогою двох різних методів: знімка та файлу лише для додавання (AOF). Знімок передбачає створення знімків даних у кеші та збереження даних на диску через певний час. AOF є більш надійним методом.

Ця функціональність покращує продуктивність під час великих навантажень. Redis підтримує кластери для горизонтального масштабування, що дозволяє продовжувати виконання, якщо вузли виходять з ладу. Redis розроблений як постійний, масштабований і надійний. Redis підтримує

розширені структури даних, такі як списки, набори, хеші та відсортовані набори, що робить його більш гнучким, ніж Memcached, але повільнішим.

На основі проведеного аналізу в четвертому розділі зроблено висновок, що Redis є універсальним рішенням для високонавантажених систем завдяки ефективному використанню пам'яті, функціональності та масштабованості.

# ПЕРЕЛІК ПОСИЛАНЬ


1. Caching Overview. What is Caching? How does it work? URL: https://aws.amazon.com/caching/?nc1=h_ls (дата звернення: 16.11.2023).

2. What Is Cache (Computing): Definition, How It Works & Types. URL: https://hazelcast.com/glossary/caching/ (дата звернення: 18.11.2023).

3. HTTP caching basics. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Caching (дата звернення: 12.11.2023).

4. In-Memory Cache. URL: https://www.gridgain.com/resources/glossary/in-memory-computing-platform (дата звернення: 14.11.2023).

5. Refresh-Ahead Pattern. URL: https://medium.com/caching-in-system-design-an-in-depth-exploration-b51e2c2e4dbd (дата звернення: 20.11.2023).

6. Cache replacement policies. Wikipedia: офіційний сайт. URL: https://en.wikipedia.org/wiki/Cache_replacement (дата звернення: 20.11.2023).

7. LRU Cache Data Concept and Structure. URL: https://www.interviewcake.com/concept/lru-cache (дата звернення: 22.11.2023).

8. Cache algorithm: examples and explanations. URL: https://simple.wikipedia.org/wiki/Cache_algorithm (дата звернення: 22.11.2023).

9. LRU and MRU Cache Implementation. URL: https://wesome.org/lru-and-mru-cache-implementation (дата звернення: 22.11.2023).

10. T. Lykouris, S. Vassilvitskii. Competitive Caching with Machine Learned. 2021. URL: https://dl.acm.org/doi/10.1145/3447579 (дата звернення: 24.11.2023).

11. Pseudo-LRU Caching Algortihm. Wikipedia: офіційний сайт. URL: https://en.wikipedia.org/wiki/Pseudo-LRU (дата звернення: 12.11.2023).

12. What is Redis? URL: https://backendless.com/redis-what-it-is-what-it-does-and-why-you-should-care/ (дата звернення: 10.11.2023).

13. What is Memcached? A practical introduction with examples. URL: https://medium.com/swlh/what-is-memcached (дата звернення: 26.11.2023).

14. How does Memcached protocol work? URL: https://www.extrahop.com/resources/protocols (дата звернення: 24.11.2023).

15. Memcached – The Ultimate Beginner's Guide. URL: https://www.dragonflydb.io/guides/memcached (дата звернення: 24.11.2023).

16. Memcached vs Redis: Choose Your In-Memory Cache. URL: https://kinsta.com/blog/memcached-vs-redis (дата звернення: 24.11.2023).

17. Difference Between Redis vs Memcached. URL: https://www.educba.com/redis-vs-memcached/ (дата звернення: 28.11.2023).

18. What is Aerospike? URL: https://www.devopsschool.com/blog/what-is-aerospike/ (дата звернення: 28.11.2023).

19. What is Apache Ignite? How is Apache Ignite Used? URL: https://www.stack.technology/what-is-apache-ignite (дата звернення: 22.11.2023).

20. An Introduction to Hazelcast In-Memory Data Grid. URL: https://doc.hazelcast.com/imdg/hazelcast-overview (дата звернення: 24.11.2023).

21. Difference Between Hazelcast vs Redis. URL: https://hazelcast.org/comparing-with-redis/ (дата звернення: 24.11.2023).

22. What Is Python? (Definition, Uses, Difficulty). URL: https://builtin.com/software-engineering/python (дата звернення: 26.11.2023).

23. Python : Focus on the most popular programming language. URL: https://datascientest.com/en/python-language (дата звернення: 18.11.2023).

24. Docker: Accelerated Container Application Development. URL: https://aws.amazon.com/docker/overview (дата звернення: 18.11.2023).

25. Using FastAPI to Build Python Web APIs. URL: https://realpython.com/fastapi-python-web-apis/ (дата звернення: 18.11.2023).

26. FastAPI Tutorial: An Introduction to Using FastAPI. URL: https://www.datacamp.com/tutorial/fastapi-tutorial (дата звернення: 18.11.2023).

27. What Is PostgreSQL? URL: https://www.red-gate.com/blog/database-development/what-is-postgresql (дата звернення: 18.11.2023).

28. Load Testing Tutorial: What is? How to? (Examples). URL: https://www.opentext.com/what-is/load-testing (дата звернення: 18.11.2023).

29. What is Locust Load Testing? URL: https://www.blazemeter.com/blog/locust-load-tests (дата звернення: 18.11.2023).

# ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

**Державний університет інформаційно-комунікаційних технологій**

**Кафедра Інженерії програмного забезпечення автоматизованих систем**

## ПРЕЗЕНТАЦІЯ ДО КВАЛІФІКАЦІЙНОЇ РОБОТИ
## на тему:
## «Аналіз підходів кешування даних на основі високонавантаженого Python веб-додатку»

**на здобуття освітнього ступеня магістра**
**зі спеціальності 126 Інформаційні системи та технології**
**освітньо-професійної програми Інформаційні системи та технології**

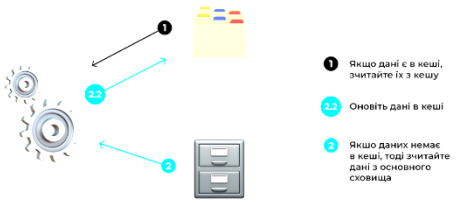
**Виконав: здобувач вищої освіти гр. ІСДМ-61**
**Антон Ритов**
**Керівник: Доцент кафедри ПУАД**
**Вадим Власенко**

**Київ - 2023**

# АКТУАЛЬНІСТЬ, МЕТА, ПРЕДМЕТ ТА ОБ'ЄКТ РОБОТИ

- **Актуальність теми:** У сучасну епоху інформаційного суспільства значний акцент приділяється проведенню численних досліджень, спрямованих на підвищення якості обслуговування користувачів інформаційних технологій. Високонавантажені веб-додатки є необхідною складовою багатьох сфер життя, починаючи від електронної комерції і закінчуючи соціальними мережами та фінансовими сервісами. Однією з ключових вимог до таких додатків є висока швидкодія та надійність, яка вимагає ефективного керування обробкою даних. В цьому контексті кешування даних стає важливим інструментом для оптимізації роботи веб-додатків, що підтверджує актуальність дослідження.
- **Об`єкт роботи:** високонавантажені Python веб-додатки.
- **Предмет роботи:** методи та підходи кешування даних у контексті оптимізації продуктивності високонавантажених Python веб-додатків.
- **Мета роботи:** аналіз та оцінка різних підходів до кешування даних у високонавантажених веб-додатках та ідентифікації оптимальних стратегій кешування для забезпечення швидкої та надійної роботи.

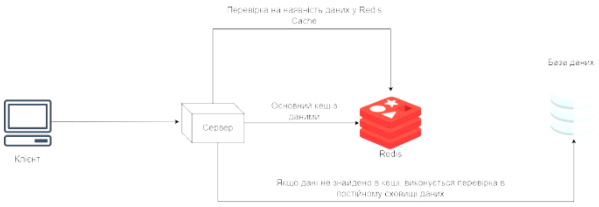
## ШАБЛОНУ ДОСТУПУ ДО ДАНИХ ПРИ КЕШУВАННІ




Cache-Aside


Read-Through

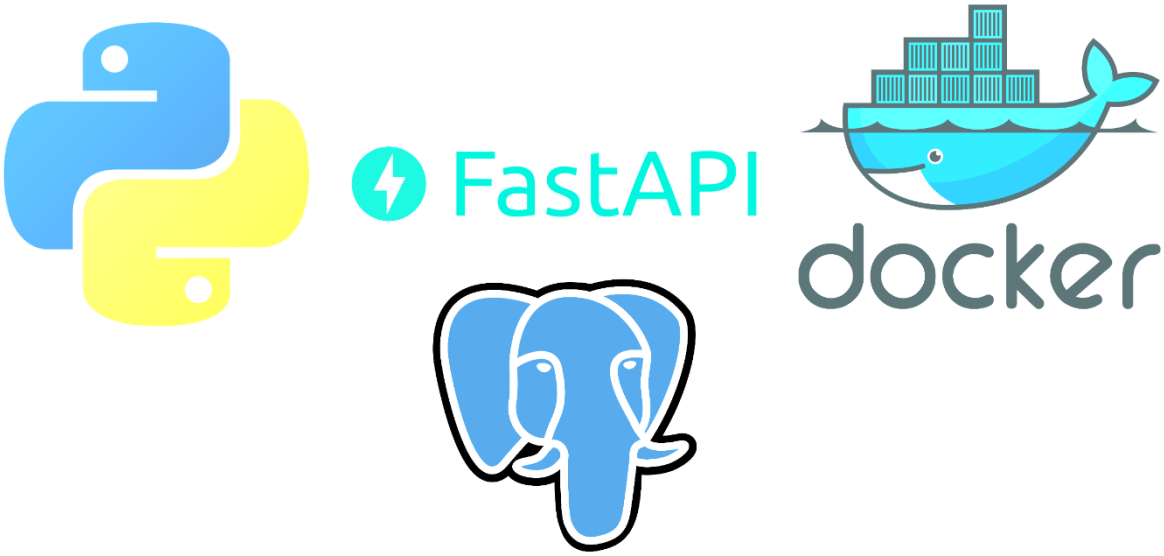
## ВИКОРИСТАННЯ REDIS ТА MEMCACHED




Алгоритм роботи застосунку з використанням Redis


Алгоритм роботи застосунку з використанням Memcached

## ВИКОРИСТАНІ ТЕХНОЛОГІЇ ПРИ РОЗРОБЦІ




## РЕАЛІЗАЦІЯ ЛОГІКИ РОБОТИ СИСТЕМИ

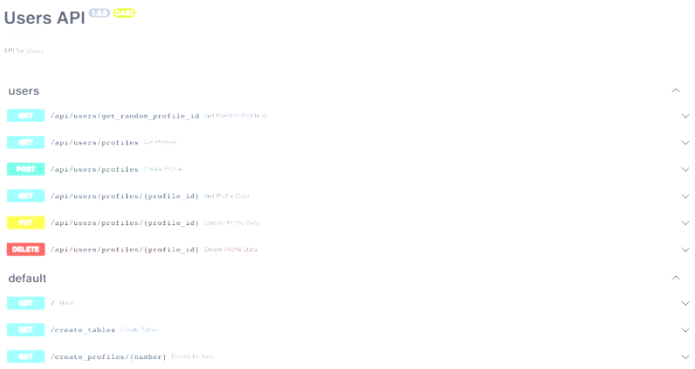
```python
import asyncio

from aiocache import caches
from core.config import settings

caches.set_config({
    'default': {
        'cache': "aiocache.MemCache",
        'serializer': {
            'class': "aiocache.serializers.StringSerializer"
        }
    },
    'cache': {
        'cache': "aiocache.RedisCache",
        'endpoint': settings.redis_server,
        'port': settings.redis_port,
        'timeout': 1,
        'serializer': {
            'class': "aiocache.serializers.PickleSerializer"
        },
        'plugins': [
            {'class': "aiocache.plugins.HitMissRatioPlugin"},
            {'class': "aiocache.plugins.TimingPlugin"}
        ]
    }
})
```

Ініціалізація кешів

```python
@router.get(
    "/profiles",
    response_model=List[Optional[ProfileRead]],
    status_code=status.HTTP_200_OK,
    name="Get profiles",
)
async def get_profiles(
    limit: int = Query(default=10, lte=100),
    redis_client: cache = Depends(cache),
    repository: ProfileRepository = Depends(get_repository(ProfileRepository)),
) -> list[Optional[ProfileRead]]:

    def fetch_from_cache(keys):
        profiles = []
        for key in keys:
            profile_data = redis_client.hgetall(key)
            profile_dict = {k.decode(): v.decode() for k, v in profile_data.items()}
            for field in ["website", "current_location"]:
                if (
                    isinstance(profile_dict.get(field), str)
                    and profile_dict[field].startswith('[')
                    and profile_dict[field].endswith(']')
                ):
                    profile_dict[field] = eval(profile_dict[field])
            profiles.append(ProfileRead(**profile_dict))
        return profiles

    all_profile_keys = redis_client.keys("profile_*")
    profiles_list = fetch_from_cache(
        sample(all_profile_keys, min(limit, len(all_profile_keys)))
    )
    cached_ids = [profile.id for profile in profiles_list]

    if len(profiles_list) < limit:
        additional_profiles = await repository.list_exclude_ids(
            cached_ids, limit=(limit - len(profiles_list))
        )
        for profile in additional_profiles:
            redis_client.hset(
                f"profile_{profile.id}", mapping={k: str(v) for k, v in dict(profile).items()}
            )
            profiles_list.append(profile)

    return profiles_list
```

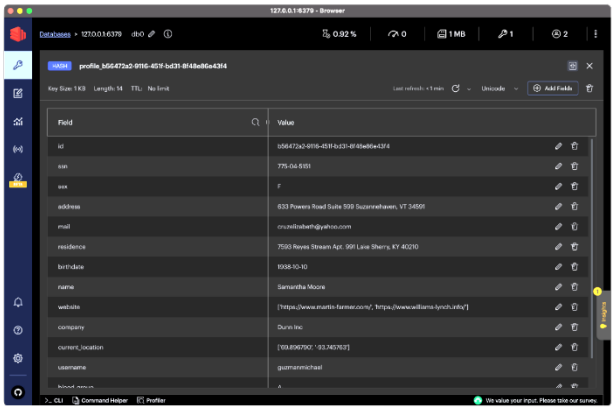
Реалізація логіки для отримання інформації про користувачів

## API РОЗРОБЛЕНОЇ СИСТЕМИ



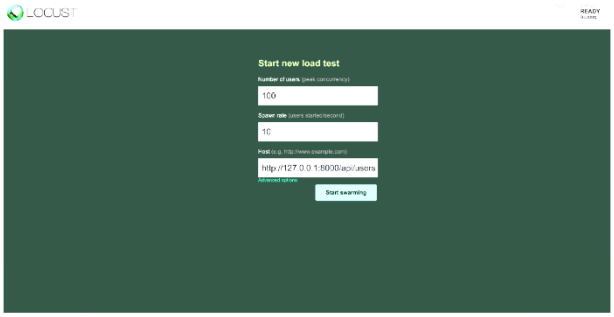

## ПРИКЛАД ЗБЕРЕЖЕННЯ ДАНИХ В КЕШІ

Отримання даних з БД




Результат встановлення даних до кешу

## ІНІЦІАЛІЗАЦІЯ СИСТЕМИ ДЛЯ ТЕСТУВАННЯ




Інтерфейс Locust для проведення навантажувального тестування

## ТЕСТУВАННЯ СИСТЕМИ З ВИКОРИСТАННЯМ MEMCACHED


Результат навантажувального тестування системи з використанням Memcached (операція WRITE)


Результат навантажувального тестування системи з використанням Memcached (операція READ)

## ТЕСТУВАННЯ СИСТЕМИ З ВИКОРИСТАННЯМ REDIS


Результат навантажувального тестування системи з використанням Redis (операція WRITE)


Результат навантажувального тестування системи з використанням Redis (операція READ)

## РЕЗУЛЬТАТИ ПРОВЕДЕННЯ ТЕСТУВАНЬ СИСТЕМИ

| Система кешування/ Кількість операцій | Redis | Memcached |
|---|---|---|
| **Читання (операція GET)** | | |
| 1000 операцій | 8 | 9 |
| 10000 операцій | 6 | 14 |
| 100000 операцій | 8 | 14 |
| **Запис (операція SET)** | | |
| 1000 операцій | 34 | 23 |
| 10000 операцій | 214 | 100 |
| 100000 операцій | 1,666 | 276 |
| **Використання пам'яті (операція SET)** | | |
| 1000 операцій | 2,5 | 5,3 |
| 1000 операцій | 3,8 | 27,2 |
| 100000 операцій | 4,3 | 211 |

# ВИСНОВКИ

- У кваліфікаційній магістерській роботі розглянуто теоретичні основи кешування даних, досліджено технології кешування та визначено актуальність дослідження. На основі наведеного аналізу зроблено висновок, що кешування – це важливий атрибут комп'ютеризованих систем, який допомагає підвищити продуктивність та зменшити навантаження на системи.
- Розглянуто алгоритми, методи та технології кешування даних в розподілених системах. Для проведення дослідження виділено дві основні системи: Redis та Memcached.
- Спроєктовано та розроблено високонавантажену систему з використанням Python та FastAPI.
- Проведено навантажувальне тестування програмної системи з використанням Locust та сформовано порівняльний аналіз. В результаті проведення порівняльного аналізу визначено, що Redis ефективніше використовує пам'ять при операціях запису, що забезпечує кращу продуктивність в умовах великих об'ємів даних. Також, Redis демонструє більш консистентні результати у читанні, що важливо для високонавантажених систем, де стабільність є критичною. Додатково визначено, що Redis забезпечує більшу передбачуваність часу відгуку, які є важливими для систем, що є високонавантаженими.

# АПРОБАЦІЯ

- Ритов А.А. «АНАЛІЗ ПІДХОДІВ КЕШУВАННЯ ДАНИХ НА ОСНОВІ ВИСОКОНАВАНТАЖУВАНОГО PYTHON WEB-ДОДАТКУ».
  Тези доповіді на ВСЕУКРАЇНСЬКА НАУКОВО-ТЕХНІЧНА КОНФЕРЕНЦІЯ «ТЕХНОЛОГІЧНІ ГОРИЗОНТИ: ДОСЛІДЖЕННЯ ТА ЗАСТОСУВАННЯ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ ДЛЯ ТЕХНОЛОГІЧНОГО ПРОГРЕСУ УКРАЇНИ І СВІТУ» 28 листопада 2023 р. ст. 5-9
- Ритов А.А. «АЛГОРИТМИ ТА ПОЛІТИКИ КЕШУВАННЯ ДАНИХ» .
  Тези доповіді на ВСЕУКРАЇНСЬКА НАУКОВО-ТЕХНІЧНА КОНФЕРЕНЦІЯ «ТЕХНОЛОГІЧНІ ГОРИЗОНТИ: ДОСЛІДЖЕННЯ ТА ЗАСТОСУВАННЯ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ ДЛЯ ТЕХНОЛОГІЧНОГО ПРОГРЕСУ УКРАЇНИ І СВІТУ» 28 листопада 2023 р. ст. 9-13
- Ритов А.А. «ОГЛЯД ТЕХНОЛОГІЙ КЕШУВАННЯ». ВСЕУКРАЇНСЬКА НАУКОВО-ТЕХНІЧНА КОНФЕРЕНЦІЯ «ТЕХНОЛОГІЧНІ ГОРИЗОНТИ: ДОСЛІДЖЕННЯ ТА ЗАСТОСУВАННЯ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ ДЛЯ ТЕХНОЛОГІЧНОГО ПРОГРЕСУ УКРАЇНИ І СВІТУ» 28 листопада 2023 р. ст. 13-16

ДЯКУЮ ЗА УВАГУ!