НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ АТОМНОЇ ТА ТЕПЛОВОЇ ЕНЕРГЕТИКИ
кафедра ЦИФРОВИХ ТЕХНОЛОГІЙ В ЕНЕРГЕТИЦІ

“До захисту допущено”
Завідувач кафедри ЦТЕ
_____________ Наталія АУШЕВА

“___”____________2024 р.

Дипломна робота

на здобуття ступеня бакалавра
за освітньо-професійною програмою
“Цифрові технології в енергетиці”
зі спеціальності *122 “Комп’ютерні науки”*

на тему: Методи пришвидшення завантаження веб-сторінок на основі кешування

Виконав:
студент IV курсу, групи ТР-01

ОЛІЙНИК Максим Сергійович ________
(прізвище, ім’я, по батькові) (підпис)

Керівник:
доцент, к.т.н., Кузьменко Ігор Миколайович ________
(посада, науковий ступінь, вчене звання, прізвище, ім’я, по батькові) (підпис)

Рецензент:
________ ________
(посада, науковий ступінь, вчене звання, науковий ступінь, прізвище, ім’я, по батькові) (підпис)

Нормоконтролер: *старший викладач Беспала Ольга Миколаївна* ________
(підпис)

Засвідчую, що у цій дипломній роботі немає запозичень з праць інших авторів без відповідних посилань.
Студент _____________
(підпис)

Київ – 2024

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ АТОМНОЇ ТА ТЕПЛОВОЇ ЕНЕРГЕТИКИ

Кафедра ЦИФРОВИХ ТЕХНОЛОГІЙ В ЕНЕРГЕТИЦІ

Рівень вищої освіти – перший (бакалаврський)
спеціальність 122 “Комп’ютерні науки”
Освітньо-професійна програма “Цифрові технології в енергетиці”

ЗАТВЕРДЖУЮ
Завідувач кафедри ЦТЕ
_________ Наталія АУШЕВА

«__»__________2024 р.

ЗАВДАННЯ
на дипломну роботу студенту

ОЛІЙНИКУ Максиму Сергійовичу
(прізвище, ім’я, по батькові)

1. Тема роботи Методи пришвидшення завантаження веб-сторінок на основі кешування

керівник роботи ***Кузьменко І.М, доцент, к.т.н.***
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від «___»_______ 2024 р. №________

2. Термін подання роботи студентом **07.06.2024 року**

3. Вихідні дані до роботи мова програмування Python, Varnich Cache,, середовище розробки IntelliJ Idea, інструмент для тестування Unit Tests____________________

4. Перелік питань, які потрібно розробити:
1) провести аналіз предметної області_______________________________________
2) дослідити засоби та технології, здійснити їх вибір_____________________________
3) розробити архітектуру програмного забезпечення_____________________________
4) реалізувати про провести тестування програмного забезпечення_______________

5. Орієнтований перелік ілюстративного матеріалу: UML діаграми, що демонструють роботу алгоритмів, архітектуру системи, взаємодію користувачів із

системою; приклади роботи з програмним продуктом.

6. Дата видачі завдання «17» червня 2023 р.

КАЛЕНДАРНИЙ ПЛАН

№з/п	Назва етапів виконання дипломної роботи	Термін виконання етапів роботи	Примітка
1.	Затвердження теми роботи		
2.	Вивчення та аналіз задачі	17-20.04.24	
3.	Розробка архітектури та загальної структури системи	21-25.04.24	
4.	Розробка частин окремих підсистем	25.04-02.05.24	
5.	Програмна реалізація системи	03-07.05.24	
6.	Оформлення пояснювальної записки	08-12.05.24	
7.	Захист програмного продукту	13-17.05.23	
8.	Передзахист	03-06.06.23	
9.	Подання готової роботи на кафедру	07.06.2024	
10.	Захист	17-21.06.2024	

Студент ______________ **Максим ОЛІЙНИК**
(підпис) (ім'я, ПРІЗВИЩЕ)

Керівник роботи ______________ **Ігор КУЗЬМЕНКО**
(підпис) (ім'я, ПРІЗВИЩЕ)

ЗМІСТ

ВСТУП 6
1 ОГЛЯД СУЧАСНИХ МЕТОДІВ КЕШУВАННЯ ДЛЯ ПОКРАЩЕННЯ ЗАВАНТАЖЕННЯ ВЕБ-СТОРІНОК 7
1.1 Поняття кешування 7
1.2 Опис принципів кешування веб-сторінок 10
1.3 Огляд існуючих методів оптимізації завантаження веб-сторінок 12
1.4 Аналіз впливу кешування на швидкість завантаження сторінок 16
1.5 Постановка мети та задачі 18
2 ВИБІР ТЕХНОЛОГІЙ КЕШУВАННЯ ТА ФОРМУЛЮВАННЯ ВИМОГ ДО СИСТЕМИ 20
2.1 Вибір технологічного стеку 20
2.2 Опис інструментів для реалізації системи кешування 21
2.3 Вимоги до програмного забезпечення 23
2.3.1 Функціональні вимоги 23
2.3.2 Нефункціональні вимоги 24
3 ПРОЕКТУВАННЯ АРХІТЕКТУРИ СИСТЕМИ КЕШУВАННЯ ТА ЇЇ РЕАЛІЗАЦІЯ 26
3.1 Проектування архітектури системи кешування 26
3.2 Детальний аналіз та проектування 28
3.2.1 Діаграма послідовності 28
3.2.2 Діаграма активності 33
3.2.3 Діаграма використання 35
4 РЕАЛІЗАЦІЯ ТА ПРОГРАМУВАННЯ СИСТЕМИ КЕШУВАННЯ 37
4.1. Опис структури даних кешування 37
4.2. Вибір методу отримання даних з кешу 39
4.3 Опис алгоритму кешування 41
4.4 Тестування 43
ВИСНОВКИ 48
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 50
ДОДАТОК 51

АНОТАЦІЯ

Дипломна робота виконана на 50 сторінках, містить 7 ілюстрацій, 1 таблицю, 1 додаток, 12 джерел в переліку посилань.

Мета роботи – створення програмного забезпечення для пришвидшення завантаження веб сторінок на основі кешування.

Методи та засоби: алгоритм Varnish LRU, MFU, мова програмування Python.

Результат – програмний інструментарій пришвидшення завантаження веб-сторінок на основі кешування.

Ключові слова: КЕШ, ВЕБ СТОРІНКИ, ПРИШВИДШЕННЯ, ЕФЕКТИВНІСТЬ, VARNISH, MFU, LRU.

ABSTRACT

The thesis consists of 50 pages, 7 illustrations, 1 table, 1 appendix, 12 sources in the list of references

Purpose – to create software to speed up the loading of web pages based on caching.

Methods and tools: Varnish LRU algorithm, MFU, Python programming language.

Result – software tools for speeding up the loading of web pages based on cashing.

Keywords: CACHE, WEB PAGES, ACCELERATION, EFFICIENCY, VARNISH, MFU, LRU.

ВСТУП

Актуальність. У сучасному цифровому світі швидкість інтернету та завантаження веб-сторінок стає критичним фактором для користувацького досвіду. Із зростанням обсягів інформації в інтернеті та збільшенням вимог до швидкості обробки даних, питання оптимізації завантаження веб-ресурсів набуває все більшої актуальності. В контексті безперервного росту кількості веб-сайтів та користувачів інтернету, забезпечення швидкості та ефективності доступу до веб-контенту є ключовим фактором успіху веб-розробки та управління веб-ресурсами.

Завантаження веб-сторінок без затримок є важливим не тільки для забезпечення зручності користувачів але й для підвищення рівня задоволеності користувачів, що, в свою чергу, впливає на позицію веб-сайтів у пошукових системах. Оптимізація завантаження веб-сторінок через механізми кешування дозволяє значно покращити швидкість доступу до інформації та ефективність використання мережевих ресурсів. В цьому контексті, кешування виступає як ефективний інструмент для зменшення навантаження на сервери, зниження трафіку та забезпечення миттєвого доступу до веб-сторінок.

Метою даного дипломного дослідження є розробка та аналіз методів пришвидшення завантаження веб-сторінок на основі кешування.

Предметом дослідження є методи кешування в контексті оптимізації завантаження веб-сторінок.

Об’єктом дослідження виступає процес завантаження веб-сторінок в інтернеті та вплив кешування на цей процес.

Для досягнення мети було поставлено наступні завдання:

- Аналіз підходів, методів та алгоритмів розв’язання пришвидшення завантаження веб-сторінок на основі кешування.
- Аналіз програмних засобів для реалізації програмної системи пришвидшення завантаження веб-сторінок на основі кешування.
- Розробка програмного забезпечення пришвидшення завантаження веб-сторінок на основі кешування.
- Тестування розробленого програмного забезпечення.

1 ОГЛЯД СУЧАСНИХ МЕТОДІВ КЕШУВАННЯ ДЛЯ ПОКРАЩЕННЯ ЗАВАНТАЖЕННЯ ВЕБ-СТОРІНОК

У сучасному світі високопродуктивні обчислювальні системи є ключовим елементом для успішного виконання різноманітних завдань, від наукових досліджень до забезпечення швидкодії веб-додатків. Однією з фундаментальних технологій, що сприяють оптимізації продуктивності таких систем, є кешування. Цей механізм дозволяє значно зменшити час доступу до часто використовуваних даних, знижуючи навантаження на основну пам'ять та підвищуючи загальну ефективність обчислювальних процесів. У даному розділі розглядаються основні принципи кешування, його застосування у веб-технологіях, а також існуючі методи оптимізації завантаження веб-сторінок, що базуються на цій технології.

1.1 Поняття кешування

Кешування, у контексті обчислювальних систем, представляє собою стратегічний механізм оптимізації доступу до даних, спрямований на підвищення продуктивності та зменшення витрат ресурсів. Основною метою кешування є забезпечення швидкодії доступу до часто використовуваних даних шляхом їх передбачуваного збереження в швидкодіємній пам'яті, відомій як кеш[1].

Центральною концепцією в кешуванні є використання ієрархії пам'яті, де кеш, зазвичай, знаходиться на більш швидкодієвому рівні, ніж основна пам'ять. Ця ієрархія може включати кеш першого рівня (L1), кеш другого рівня (L2), кеш третього рівня (L3) тощо, причому швидкодія знижується з кожним наступним рівнем, але збільшується обсяг пам'яті[1].

Основною ідеєю кешування є використання локальності даних та інструкцій у програмах, що виконуються. Ця локальність може бути як в часовому, так і в просторовому вимірах. Часова локальність передбачає, що дані, які були доступні

нещодавно, мають велику ймовірність бути запитаними знову[1]. Просторова локальність передбачає, що доступ до сусідніх даних також може бути потрібним у майбутньому. Кешування ефективно використовує ці властивості, забезпечуючи швидкий доступ до даних, що мають високу ймовірність використання.

З механічної точки зору, процес кешування полягає у збереженні копій часто використовуваних даних в кеш-пам'яті, де вони можуть бути швидко доступні для майбутнього використання. Коли дані запитуються, спершу виконується перевірка у кеші. Якщо дані знайдені в кеші (попадається кешоване попадання), вони повертаються без необхідності звертатися до більш повільної основної пам'яті. Якщо ж дані відсутні в кеші (відсутність кешованого попадання), вони завантажуються з основної пам'яті, а також кешовані для майбутнього використання[1].

У контексті обчислювальної технології кешування є фундаментальним механізмом, спрямованим на оптимізацію та підвищення продуктивності обчислювальних процесів. Його значення визначається не лише можливістю зменшення часу доступу до даних, але й економією ресурсів, в тому числі енергії та обчислювальної потужності.

Передова технологія сучасних обчислювальних систем вимагає надзвичайної швидкості та ефективності у роботі з великими обсягами даних. Кешування в цьому контексті виконує критичну роль, сприяючи зниженню часу доступу до часто використовуваних даних, які зазвичай зберігаються на швидкодіємних пристроях зберігання даних, таких як кеш-пам'ять[1]. Це дозволяє обчислювальним системам оперативно виконувати операції, що вимагають інтенсивної обробки даних, та мінімізує затримки, пов'язані з віддаленим доступом до даних у повільній основній пам'яті.

Крім того, кешування зменшує навантаження на більш повільні системи зберігання даних, такі як твердотільні диски або мережеві сховища, оскільки часто використовувані дані зберігаються на швидкодіємних пристроях зберігання. Це призводить до зменшення витрат енергії та підвищення загальної ефективності обчислювальних систем.

Таким чином, кешування в контексті обчислювальної технології є ключовим елементом для досягнення високої продуктивності, оптимізації ресурсів та енергоефективності в сучасних обчислювальних системах. Воно забезпечує швидкий доступ до даних та оптимальне використання обчислювальних ресурсів, сприяючи подальшому розвитку та інноваціям у цій галузі[1].

Узагальнюючи, кешування є ключовим механізмом для підвищення ефективності та продуктивності обчислювальних систем, що базується на використанні локальності даних та інструкцій у програмах, забезпечуючи швидкий доступ до часто використовуваних ресурсів через їх передбачуване збереження в швидкодієемній кеш-пам'яті.

Проблема зі швидкодією доступу до даних стоїть перед обчислювальними системами як одна з ключових викликів, що ускладнює ефективне виконання різноманітних завдань. Завдання обробки та аналізу великого обсягу даних стає складним, оскільки основна пам'ять, що забезпечує доступ до інформації, має обмежену швидкість доступу та обмежений обсяг[2]. Відповідно, затримки при доступі до даних можуть стати суттєвими перешкодами у виконанні обчислювальних операцій.

Швидкодійність доступу до даних є важливою для забезпечення різноманітних сфер діяльності, включаючи наукові дослідження, фінансовий аналіз, медичну діагностику, веб-сервіси та багато інших. Наприклад, у веб-додатках затримки при завантаженні веб-сторінок можуть призвести до погіршення користувацького досвіду та зменшення продуктивності[2]. У сфері фінансів, мілісекундні затримки можуть мати величезне значення при виконанні операцій з великими фінансовими потоками.

Кешування виникає як відповідь на цю проблему, надаючи механізм оптимізації доступу до даних шляхом збереження копій часто використовуваних даних у швидкодіймній кеш-пам'яті. Це дозволяє зменшити час доступу до цих даних та знизити навантаження на повільну основну пам'ять[2]. Отже, кешування є необхідним елементом для забезпечення оптимальної продуктивності та ефективності обчислювальних систем у сучасному інформаційному середовищі.

Кешування допомагає вирішити проблему зі швидкодією доступу до даних шляхом ефективного управління доступом до інформації, що часто використовується. Основний механізм полягає в збереженні копій цих часто використовуваних даних у швидкодіймній кеш-пам'яті, ближче до процесора, що дозволяє швидко отримувати доступ до них[2].

Під час виконання обчислювальних операцій, програми часто отримують доступ до тих самих даних або виконують одні й ті самі операції з інформацією, що зберігається в пам'яті. Завдяки локальності даних, велика частина цих запитів спрямована на обробку тієї ж самої групи даних. Кешування використовує цю особливість, зберігаючи копії цих часто використовуваних даних у кеш-пам'яті[2].

Коли програма запитує дані, спершу виконується перевірка у кеші. Якщо дані знайдені в кеші, вони негайно повертаються без необхідності звертатися до повільнішої основної пам'яті. Цей процес значно прискорює доступ до даних, оскільки час, необхідний для звернення до кеш-пам'яті, значно менший порівняно з часом доступу до основної пам'яті[2].

Як наслідок, використання кешування дозволяє значно зменшити затримки при доступі до даних, що робить обчислювальні операції більш ефективними та продуктивними. Крім того, кешування допомагає зменшити навантаження на основну пам'ять, оскільки часто використовувані дані зберігаються в кеші, звільняючи ресурси для інших завдань. Таким чином, кешування грає важливу роль у поліпшенні швидкодії та ефективності обчислювальних систем.

1.2 Опис принципів кешування веб-сторінок

Кешування, у контексті обчислювальних систем, є механізмом, який забезпечує оптимізацію доступу до даних шляхом збереження копій часто використовуваних даних у швидкодіймній кеш-пам'яті.

Процес роботи кешування можна описати наступним чином[3]:

Етап 1. Запит даних

Коли програма або процесор потребує доступу до даних, він видає запит на основну пам'ять. Цей запит може включати як читання, так і запис даних.

Етап 2. Перевірка кешу

Перш ніж звертатися до основної пам'яті, система перевіряє, чи містить кеш-пам'ять копію запитаних даних. Ця перевірка виконується на основі адреси, яка вказує на місцезнаходження даних у кеші.

Кешоване попадання (Cache Hit): якщо копія даних виявлена у кеші, це означає успішне кешоване попадання. Система повертає дані з кешу, уникнувши звернення до повільнішої основної пам'яті.

Етап 3. Відсутність кешованого попадання

У випадку, якщо копія даних відсутня у кеші, це означає відсутність кешованого попадання. Система тоді звертається до основної пам'яті для отримання необхідних даних.

Етап 4. Оновлення кешу

Після отримання даних з основної пам'яті, вони можуть бути збережені у кеші для подальших використань. Цей процес, відомий як кешування, дозволяє покращити швидкодію доступу до даних у майбутньому.

Цей цикл запиту, перевірки та оновлення кешу повторюється при кожному доступі до даних. Кешування допомагає значно знизити час доступу до даних та підвищити ефективність обчислювальних систем, зменшуючи навантаження на основну пам'ять та забезпечуючи швидкий доступ до часто використовуваних ресурсів.

Принцип кешування веб-сторінок базується на збереженні копій веб-сторінок на клієнтському пристрої або на проміжному сервері для подальшого використання. Основною метою є зменшення часу завантаження сторінок для користувачів та оптимізація використання мережевих ресурсів.

Принцип кешування веб-сторінок[3]:

Отримання запиту на веб-сторінку, коли користувач відкриває веб-сторінку у своєму браузері, він надсилає запит на сервер, що містить цю сторінку.

Перевірка кешу на клієнтському пристрої або на проміжному сервері. Сервер або клієнтський браузер перевіряє, чи має він збережену копію цієї веб-сторінки у своєму кеші. Якщо така копія існує, це означає кешоване попадання. Кешоване попадання (Cache Hit), якщо копія сторінки знаходиться у кеші, сервер або браузер повертає цю копію користувачеві. Цей процес значно прискорює завантаження сторінки, оскільки вона не потребує повного завантаження з сервера.

Відсутність кешованого попадання (Cache Miss), якщо копія сторінки відсутня у кеші, це означає відсутність кешованого попадання. У цьому випадку сервер повинен звернутися до свого джерела, щоб отримати оригінальну версію сторінки та відправити її користувачеві.

Оновлення кешу, після отримання оновленої версії сторінки від сервера, клієнтський браузер або проміжний сервер може оновити свій кеш, зберігаючи нову копію сторінки для майбутніх запитів.

Цей процес дозволяє ефективно використовувати мережеві ресурси та зменшує час завантаження веб-сторінок для користувачів шляхом збереження копій часто використовуваних сторінок у кеші. Таким чином, принцип кешування веб-сторінок допомагає поліпшити продуктивність та швидкість роботи веб-систем.

1.3 Огляд існуючих методів оптимізації завантаження веб-сторінок

Існує ряд методів оптимізації завантаження веб-сторінок, які можна використовувати для забезпечення швидкодії та ефективності сайту. Кешування є однією з ключових стратегій оптимізації завантаження веб-сторінок, спрямованою на збереження копій ресурсів (таких як зображення, стилі, скрипти тощо) для подальшого використання, замість повторного завантаження їх з сервера при кожному запиті.

- Браузерне кешування

Браузерне кешування полягає в збереженні копій ресурсів на стороні клієнта (веб-браузера) після їх першого завантаження. При подальших запитах до того ж ресурсу браузер може використати локальну копію замість звернення до сервера.

Переваги:

- зменшення часу завантаження сторінок для користувачів, оскільки ресурси завантажуються з локального сховища;
- зменшення навантаження на сервер, оскільки кількість запитів до нього зменшується.

Недоліки:

- ризик застаріння кешованих ресурсів, якщо вони були змінені на сервері;
- потреба у правильному керуванні кешуванням для запобігання проблемам з оновленнями.

- Серверне кешування:

Серверне кешування передбачає збереження копій ресурсів на стороні сервера. Сервер може кешувати цілі сторінки або їх частини, щоб пришвидшити обробку запитів від клієнтів[4].

Переваги*:*

- ефективне зменшення часу відповіді сервера, оскільки він не повинен генерувати сторінку заново при кожному запиті;
- зменшення навантаження на сервер і покращення масштабованості.

Недоліки:

- потреба у відповідному налаштуванні сервера та програмного забезпечення кешування;
- ризик використання застарілих даних у випадку неправильного кешування.
- HTTP-кешування

HTTP-кешування використовує заголовки HTTP для контролю кешування ресурсів. Вони вказують браузерам і проксі-серверам, як тривалий час може зберігатися кешована копія ресурсу[4].

Переваги*:*

- простота в реалізації, оскільки використовуються стандартні заголовки HTTP;
- контроль за часом життя кешованих ресурсів.

Недоліки:

- обмежена гнучкість управління кешуванням, порівняно з іншими методами;
- ризик несправності кешування через неправильну конфігурацію сервера або проксі.

- CDN (методи і переваги)

CDN (мережа доставки контенту) - це розподілені системи серверів, розташовані по всьому світу, які забезпечують швидке і надійне доставляння вмісту до кінцевих користувачів[4].

Переваги:

- підвищення швидкості завантаження сторінок за рахунок розміщення кешованих копій вмісту на серверах, що ближчі до користувачів;
- зменшення навантаження на основний сервер, оскільки частина трафіку обробляється CDN.

Недоліки:

- витрати на підключення до CDN можуть бути високими;
- ризик збоїв у роботі CDN, що може призвести до недоступності контенту.

Ці методи кешування відображають сучасні стратегії оптимізації завантаження веб-сторінок, спрямовані на покращення швидкості завантаження і зменшення навантаження на сервер. Однак вони потребують правильного налаштування та управління для досягнення максимальної ефективності і уникнення проблем.

Наступним важливим методом є стиснення. Стиснення є важливим етапом оптимізації завантаження веб-сторінок, спрямованим на зменшення обсягу передаваних даних між сервером і клієнтом шляхом упакування їх у більш компактний формат.

- Методи стиснення

gzip (GNU zip) є одним із найпоширеніших методів стиснення для передачі веб-ресурсів. Він використовує алгоритм DEFLATE для стиснення текстових файлів, таких як HTML, CSS та JavaScript[5].

Переваги:

- ефективне стиснення даних, що дозволяє зменшити обсяг передаваних даних і, відповідно, час завантаження сторінок.
- підтримується більшістю веб-серверів і браузерів.

Недоліки:

- не так ефективний для стиснення вже стиснених або неструктурованих даних, таких як вже стиснені зображення або великі файли.

Brotli - це новіший алгоритм стиснення, розроблений Google, який виявляє вищу ступінь стиснення, особливо для текстових даних[5].

Переваги:

- забезпечує кращу стискаючу ефективність порівняно з gzip, що призводить до ще більшого зменшення обсягу передаваних даних і швидкості завантаження;
- підтримується більшістю сучасних браузерів і веб-серверів.

Недоліки:

- потребує більшої обчислювальної потужності для стискання та розпакування даних порівняно з gzip.
- Відповідність стиснення сторінок стандартам:

Стиснення веб-сторінок повинне відповідати стандартам HTTP і заголовкам відповідей сервера. У заголовку HTTP має бути вказано, яким методом стиснення використовується, а також характеристики стиснення (наприклад, рівень стиснення)[5].

Переваги:

- забезпечення сумісності з браузерами і проксі-серверами, що дозволяє коректно розпаковувати стиснені дані.

Недоліки:

- потреба у правильному налаштуванні сервера та програмного забезпечення для генерації відповідних заголовків HTTP.

Стиснення є важливим етапом оптимізації завантаження веб-сторінок, яке дозволяє зменшити час завантаження і обсяг передаваних даних. Обидва методи, gzip і Brotli, мають свої переваги і недоліки, і вибір між ними залежить від потреб конкретного проекту та його аудиторії.

1.4 Аналіз впливу кешування на швидкість завантаження сторінок

Зменшення часу завантаження веб-сторінок для користувачів через використання локальних копій ресурсів, як результат кешування, є ключовою складовою оптимізації веб-процесів та поліпшення користувацького досвіду. Даний явище базується на принципах ефективного управління доступом до інформації та мінімізації часу, необхідного для передачі даних між клієнтським пристроєм та сервером[6].

Використання локальних копій ресурсів, отриманих через кешування, дозволяє уникнути повторного завантаження і пересилання даних, що вже були отримані раніше. Замість цього, клієнтський пристрій може використати локальні копії, що значно скорочує час, необхідний для отримання контенту веб-сторінок.

Цей позитивний вплив кешування на швидкість завантаження сторінок базується на декількох ключових механізмах. По-перше, кешування дозволяє зменшити кількість запитів до сервера, оскільки частину контенту можна отримати з локальних джерел. По-друге, відсутність необхідності повторного завантаження ресурсів зменшує час передачі даних та час обробки запитів, що призводить до загального скорочення часу завантаження сторінок для користувачів[6].

Таким чином, використання локальних копій ресурсів, отриманих через кешування, сприяє поліпшенню швидкості завантаження веб-сторінок, забезпечуючи більш ефективне використання ресурсів мережі та зменшення часу очікування для кінцевих користувачів.

Зменшення навантаження на сервер внаслідок зменшення кількості запитів до нього, спричинене механізмами кешування, відображає принцип оптимізації

веб-систем за допомогою ефективного управління трафіком та ресурсами сервера. Цей феномен базується на розподілі навантаження та оптимізації обробки запитів з метою забезпечення високої доступності та продуктивності веб-сервісів.

Зменшення кількості запитів до сервера досягається завдяки можливості використання локальних копій ресурсів, збережених на клієнтських пристроях або на проміжних серверах кешування. Замість того, щоб кожен раз звертатися до сервера для отримання ресурсів, клієнтські пристрої можуть використовувати збережені локальні копії, що зменшує потребу в частіших запитах до сервера.

Цей процес призводить до ряду переваг, зокрема:

- зменшення витрат на обробку запитів на сервері, оскільки частина контенту може бути отримана безпосередньо з локальних джерел;
- зменшення часу відповіді сервера, оскільки він не потребує обробки кожного окремого запиту на ресурси, які вже збережені на клієнтських пристроях.

Отже, зменшення кількості запитів до сервера в результаті використання кешування сприяє зниженню навантаження на сервер та покращує продуктивність веб-системи в цілому, забезпечуючи користувачам швидший та більш ефективний доступ до необхідного контенту.

Фактори, що впливають на ефективність кешування, є важливими аспектами оптимізації веб-систем та визначають успішність використання механізму кешування для забезпечення швидкості завантаження та ефективного використання ресурсів мережі. Враховуючи детальне вивчення цих факторів, можна досягти оптимального рівня продуктивності веб-системи та задоволення потреб користувачів[6].

- Час життя кешованих ресурсів:

Час життя кешованих ресурсів визначає період, протягом якого кешовані копії ресурсів залишаються актуальними та можуть використовуватися для відповіді на запити користувачів. Вибір оптимального часу життя кешованих ресурсів впливає на збереження актуальності контенту та ефективне використання кешування[7].

- Оновлення кешованих даних:

Процес оновлення кешованих даних визначає, якість та актуальність інформації, яка доступна користувачам через механізм кешування. Регулярність оновлення кешованих даних впливає на забезпечення коректності та актуальності контенту та підтримує високу якість веб-системи.

- Місце зберігання кешованих копій:

Місце зберігання кешованих копій ресурсів, таке як клієнтський пристрій, сервер або CDN, визначає доступність та швидкість отримання кешованих даних для користувачів. Вибір оптимального місця зберігання кешованих копій впливає на загальну продуктивність веб-системи та забезпечує ефективне використання ресурсів мережі[7].

Розуміння та належне управління цими факторами є важливими для досягнення високої ефективності кешування веб-систем та забезпечення задоволення потреб користувачів у швидкому та ефективному доступі до веб-контенту.

1.5 Постановка мети та задачі

Метою даного дипломного дослідження є розробка та аналіз методів пришвидшення завантаження веб-сторінок на основі кешування. Основною метою є вивчення та оцінка різних підходів до кешування веб-ресурсів з метою забезпечення високої швидкості завантаження сторінок для користувачів.

Для досягнення мети було поставлено наступні завдання:

- Аналіз підходів, методів та алгоритмів розв’язання пришвидшення завантаження веб-сторінок на основі кешування.
- Аналіз програмних засобів для реалізації програмної системи пришвидшення завантаження веб-сторінок на основі кешування.
- Розробка програмного забезпечення пришвидшення завантаження веб-сторінок на основі кешування.
- Тестування розробленого програмного забезпечення.

Ці задачі спрямовані на досягнення мети дослідження та розробки практичних рекомендацій щодо використання кешування для покращення швидкості завантаження веб-сторінок.

2 ВИБІР ТЕХНОЛОГІЙ КЕШУВАННЯ ТА ФОРМУЛЮВАННЯ ВИМОГ ДО СИСТЕМИ

У цьому розділі детально розглядається процес вибору технологічного стеку для реалізації системи кешування та формулювання вимог до програмного забезпечення. Обгрунтовується вибір конкретних інструментів та технологій для оптимальної реалізації поставлених завдань. Крім того, розглядаються функціональні та нефункціональні вимоги до програмного забезпечення, що визначають ключові аспекти системи кешування ресурсів веб-сторінок. Ретельний аналіз цих вимог допоможе забезпечити успішну реалізацію системи кешування та досягнення встановлених цілей щодо її функціональності та якості.

2.1 Вибір технологічного стеку

Для написання програмного забезпечення було обрано мову програмування Python. Python має велику кількість бібліотек та фреймворків для розробки веб-додатків та аналізу даних. Наприклад, Django та Flask для бекенду, або бібліотеки, такі як BeautifulSoup для парсингу веб-сторінок. У Python є підтримка асинхронного програмування, що може бути корисним для оптимізації процесів завантаження веб-сторінок та взаємодії з сервером.

Python має потужні бібліотеки для аналізу даних, такі як pandas, NumPy, та Matplotlib, що дозволяє проводити детальний аналіз та візуалізацію даних про швидкість завантаження веб-сторінок.

Отже, вибір Python для розробки ПЗ для аналізу методів пришвидшення завантаження веб-сторінок на основі кешування є обгрунтованим з точки зору швидкості розробки, доступності різноманітних інструментів та бібліотек, а також можливостей для ефективного аналізу та оптимізації процесів.

2.2 Опис інструментів для реалізації системи кешування

Varnish Cache - це програмне забезпечення для кешування HTTP-запитів, яке дозволяє створювати швидкодіючий кеш для веб-сайтів та застосунків. В основі Varnish лежить проксі-сервер, який використовується для перехоплення HTTP-запитів та відповідей між клієнтами та серверами.

Основні характеристики та можливості Varnish Cache:

HTTP-кешування:

Varnish Cache використовує HTTP-кешування для збереження копій ресурсів, таких як HTML-сторінки, CSS, JavaScript, зображення, аудіо та відео файли тощо. Це дозволяє швидко відповідати на HTTP-запити, використовуючи кешовані версії ресурсів.

Адаптивне кешування:

Varnish Cache може автоматично адаптувати стратегії кешування в залежності від характеристик запитів та серверної динаміки. Він може налаштовуватися для кешування статичних ресурсів або динамічних сторінок, в залежності від конкретних потреб веб-сайту.

Шарована архітектура:

Varnish Cache побудований на шарованій архітектурі, що дозволяє легко розширювати його можливості та підтримувати високу продуктивність при великих навантаженнях. Він може бути розгорнутий як одиночний проксі-сервер або як частина кластеру серверів.

Підтримка конфігурації через VCL (Varnish Configuration Language):

Конфігурація Varnish Cache здійснюється через мову програмування VCL, що дозволяє точно налаштувати різні аспекти кешування, такі як правила кешування, зберігання кешованих об'єктів, обробка запитів тощо.

Моніторинг та статистика:

Varnish Cache надає різноманітні засоби моніторингу та збору статистики про роботу кешу, що дозволяє аналізувати продуктивність, виявляти проблеми та вносити відповідні корективи.

Загалом, Varnish Cache є потужним і ефективним інструментом для реалізації кешування веб-сайтів та застосунків, який допомагає покращити швидкодію та ефективність роботи веб-серверів.

Принцип роботи Varnish Cache базується на ідеї швидкодіючого HTTP-кешування, яке дозволяє зберігати копії ресурсів на проміжному сервері (проксі-сервері) для швидкого доступу до них безпосередньо з клієнтських пристроїв.
Основні принципи роботи Varnish Cache:

- Перехоплення HTTP-запитів:

Varnish Cache розташовується між клієнтом (браузером) та основним веб-сервером. Коли клієнт відправляє HTTP-запит на сервер для отримання певного ресурсу (наприклад, HTML-сторінки, CSS-файлу або зображення), Varnish перехоплює цей запит.

- Перевірка наявності кешованого ресурсу

Перш ніж передати запит до основного веб-сервера, Varnish перевіряє, чи міститься копія запитуваного ресурсу в його кеші. Якщо така копія вже існує та вона не прострочена, Varnish негайно повертає цей ресурс клієнту.

- Запит до основного сервера

Якщо копії ресурсу в кеші немає або вона застаріла, Varnish пересилає запит до основного веб-сервера для отримання оригінального ресурсу.

- Кешування відповіді

Після отримання оригінальної відповіді від основного сервера, Varnish кешує цей ресурс. Це означає, що копія ресурсу буде збережена в пам'яті або на диску для подальшого використання.

- Відправка відповіді клієнту

Після кешування відповіді, Varnish передає цей ресурс клієнту. Тепер цей ресурс також буде доступний у кеші Varnish для майбутніх запитів.

- Оновлення кешу

В разі, якщо дані на сервері змінюються (наприклад, відбувається оновлення сторінки або змінюється зображення), Varnish може автоматично оновлювати свій

кеш. Це дозволяє підтримувати актуальність даних у кеші та уникнути доставки застарілих версій ресурсів.

Цей принцип дії дозволяє Varnish Cache забезпечувати швидке і ефективне кешування веб-ресурсів, що значно поліпшує швидкодію та продуктивність веб-сайтів для користувачів.

Flask - це легкий та гнучкий мікрофреймворк для створення веб-додатків на мові програмування Python. Він має простий та інтуїтивно зрозумілий синтаксис, що дозволяє швидко розробляти веб-додатки з мінімальним обсягом коду.

Flask є легким та ефективним фреймворком, що дозволяє швидко обробляти запити та реагувати на них, що особливо важливо для системи кешування, де швидкість генерації відповідей має велике значення. Flask має велику кількість розширень та сторонніх бібліотек, які можна використовувати для реалізації різноманітних функцій, включаючи систему кешування. Flask має вбудовану підтримку для тестування, що дозволяє швидко виконувати тестування коду, включаючи систему кешування.

Flask може легко інтегруватися з іншими інструментами та сервісами, що робить його відмінним вибором для розробки комплексних систем кешування. Ці переваги роблять Flask привабливим вибором для реалізації системи кешування для оптимізації завантаження веб-сторінок.

2.3 Вимоги до програмного забезпечення

2.3.1 Функціональні вимоги

Функціональні вимоги - це вимоги, що описують функціональність системи, тобто те, що система повинна робити. Ці вимоги описують конкретні функції, операції або послуги, які система має надавати для задоволення потреб користувачів або вирішення певних бізнес-задач. Наприклад, це може бути можливість реєстрації користувача, можливість створення замовлення або можливість відправлення повідомлення.

Для розробки програми було розроблено наступні функціональні вимоги:

- Кешування ресурсів – система повинна забезпечувати можливість кешування різних типів ресурсів, таких як HTML-сторінки, CSS, JavaScript, зображення тощо.
- Керування часом життя кешованих ресурсів – користувач повинен мати можливість налаштовувати час життя кешованих ресурсів для оптимізації ефективності кешування.
- Оновлення кешу – система повинна автоматично оновлювати кеш при оновленні вихідних даних на сервері.
- Адміністрування кешу – система повинна надавати простий інтерфейс адміністратора для налаштування та моніторингу кешування, включаючи можливість очищення кешу та перевірку статусу кешованих об'єктів.

2.3.2 Нефункціональні вимоги

Нефункціональні вимоги - це вимоги, що описують якісні характеристики системи, такі як продуктивність, надійність, безпека та інші аспекти, які не пов'язані безпосередньо з функціональністю системи, але впливають на її ефективність та якість. Ці вимоги описують умови експлуатації системи та критерії, за якими буде оцінюватися її успішність. Наприклад, це може бути швидкодія системи, час доступності, рівень безпеки тощо.

Для розробки програми було розроблено наступні нефункціональні вимоги:

- Продуктивність – система повинна забезпечувати швидке та ефективне кешування ресурсів для покращення швидкості завантаження веб-сторінок.
- Масштабованість – система повинна бути здатна обробляти великі обсяги трафіку та кількість запитів для ефективного використання у різних масштабних середовищах.
- Надійність – система повинна бути стабільною та надійною, здатною працювати без збоїв та втрати даних навіть при великому навантаженні.
- Безпека – система повинна забезпечувати захист кешованих даних від несанкціонованого доступу та зловживання.
- Сумісність – система повинна бути сумісною з різними типами веб-серверів та фреймворків, а також з різними типами контенту (статичний, динамічний тощо).

- Простота використання – інтерфейс користувача повинен бути інтуїтивно зрозумілим та легким у використанні для адміністраторів системи.

Отже, вимоги до програмного забезпечення, що включають функціональні та нефункціональні вимоги, визначають ключові аспекти системи кешування ресурсів веб-сторінок. Функціональні вимоги визначають функціональність системи та її можливості, такі як кешування ресурсів, керування часом життя кешу, оновлення кешу, підтримка CDN та адміністрування кешу. Нефункціональні вимоги, з іншого боку, визначають якісні характеристики системи, такі як продуктивність, масштабованість, надійність, безпека, сумісність та простота використання.

Забезпечення виконання цих вимог є важливим для створення ефективної та надійної системи кешування ресурсів, яка забезпечить оптимізацію завантаження веб-сторінок та покращення користувацького досвіду. Ретельне розуміння та врахування цих вимог на кожному етапі розробки допоможе забезпечити успішну реалізацію системи кешування ресурсів і досягнення поставлених цілей щодо її функціональності та якості.

3 ПРОЕКТУВАННЯ АРХІТЕКТУРИ СИСТЕМИ КЕШУВАННЯ ТА ЇЇ РЕАЛІЗАЦІЯ

Проектування архітектури системи кешування є ключовим етапом у покращенні продуктивності, швидкості відгуку та доступності веб-додатків. У цьому контексті розглядаються проблеми та вимоги, що стимулюють використання кешування, а також вибір даних та операцій для кешування з метою максимізації його ефективності. Після аналізу різних стратегій та механізмів зберігання кешованих даних, увага приділяється детальному аналізу та проектуванню, щоб визначити оптимальні алгоритми та процеси взаємодії між різними компонентами системи кешування.

3.1 Проектування архітектури системи кешування

Використання кешування в архітектурі веб-систем значно покращує їх продуктивність, швидкість відгуку та доступність. Однак, перед впровадженям кешування, важливо розуміти конкретні проблеми або вимоги, які стимулюють його застосування, а також визначити, які дані або операції будуть кешуватися.
Проблеми або вимоги, що стимулюють використання кешування

- Зниження часу відгуку системи:

Веб-системи часто стикаються з проблемою високих затримок при обробці запитів. Кешування дозволяє зберігати результати частих запитів або тяжких обчислювальних операцій, зменшуючи час на їх повторне виконання.

- Зменшення навантаження на сервер:

Під час піків навантаження сервери можуть перевантажуватись, що призводить до збоїв або зниження продуктивності. Кешування ефективно розвантажує сервер, зменшуючи кількість обчислень та запитів до бази даних.

- Оптимізація доступності:

Висока доступність є критичною вимогою для веб-систем. Кешування допомагає забезпечити стабільну роботу системи навіть у випадках часткових збоїв, зберігаючи кешовані копії даних доступними для користувачів.

Дані та операції, які будуть кешуватися

Статичний контент - це включає файли зображень, CSS-стилі, JavaScript-скрипти тощо. Статичний контент ідеально підходить для кешування, оскільки він не змінюється з кожним запитом.

Результати запитів до бази даних - часті запити, які вимагають значних обчислювальних ресурсів для виконання, є ідеальними кандидатами для кешування. Кешування результатів таких запитів зменшує навантаження на базу даних та покращує швидкість відгуку системи.

Проектування ефективної системи кешування вимагає глибокого розуміння типів даних та операцій, які є критичними для продуктивності, швидкості відгуку та доступності веб-системи. Вибір даних або операцій для кешування має базуватися на аналізі частоти запитів, обсягу даних, динамічності контенту та вимог до свіжості даних. Кешування, застосоване з розумінням і обережністю, може значно покращити загальну продуктивність і користувацький досвід веб-системи.

Локальне кешування

Локальне кешування передбачає зберігання кешованих даних на тому ж сервері, що і веб-додаток. Це може бути ефективним для додатків із меншим обсягом трафіку або коли вся система розгортана на одному сервері.

Механізми зберігання

Varnish Cache зберігає кешовані дані у пам'яті (RAM) для швидкого доступу, хоча може використовувати і дисковий простір для зберігання більш об'ємних даних. Зберігання в пам'яті забезпечує низьку затримку і високу швидкість доступу до кешованих ресурсів, але обмежене загальним обсягом доступної пам'яті.

Стратегії валідності даних

Для управління актуальністю даних у кеші використовуються різні стратегії:

- Тайм-ту-лів (TTL)- кожен кешований елемент зберігається протягом певного періоду часу. Після закінчення цього періоду дані вважаються застарілими і при наступному запиті оновлюються.
- Залежність від тегів - дані в кеші можуть інвалідуватися або оновлюватися на основі змін у пов'язаних з ними даних або тегах, що дозволяє динамічно управляти валідністю кешованих даних.

Стратегії відсутності даних

Управління ситуаціями, коли запитувані дані відсутні у кеші, також є важливим аспектом архітектури кешування:

- Lazy Loading - дані завантажуються і додаються до кешу лише при їх першому запиті. Цей підхід зменшує початкове навантаження на систему, але може призводити до затримок при першому запиті до нових даних.
- Warm-up - система активно заповнює кеш даними, які, як очікується, будуть запитані, щоб уникнути затримок при першому доступі до них. Це може включати попереднє завантаження популярного контенту або даних, які часто запитуються.

Архітектура кешування з Varnish Cache вимагає ретельного планування та налаштування, щоб оптимізувати швидкість відгуку системи, зменшити навантаження на сервери та забезпечити високу доступність даних. Вибір між локальним та розподіленим кешуванням, налаштування механізмів зберігання, стратегії валідності та управління відсутніми даними повинні базуватися на конкретних потребах та обсязі веб-додатку.

3.2 Детальний аналіз та проектування

3.2.1 Діаграма послідовності

Діаграма послідовності (Sequence Diagram) - це тип діаграми в моделюванні систем, що показує взаємодії між об'єктами в рамках певного процесу чи

функціональності[10]. Вона використовується для візуалізації того, як об'єкти комунікують між собою впродовж часу.

У діаграмі послідовності об'єкти представлені у вигляді вертикальних ліній, а повідомлення між ними - у вигляді стрілок. Вона дозволяє моделювати послідовність викликів та відповідей між об'єктами, що відображає реакцію системи на певні події або запити.

Діаграми послідовності часто використовуються для аналізу взаємодій між об'єктами в рамках виконання певного сценарію, для специфікації логіки роботи програмних систем, для тестування, а також для візуалізації та комунікації між розробниками та стейкхолдерами.

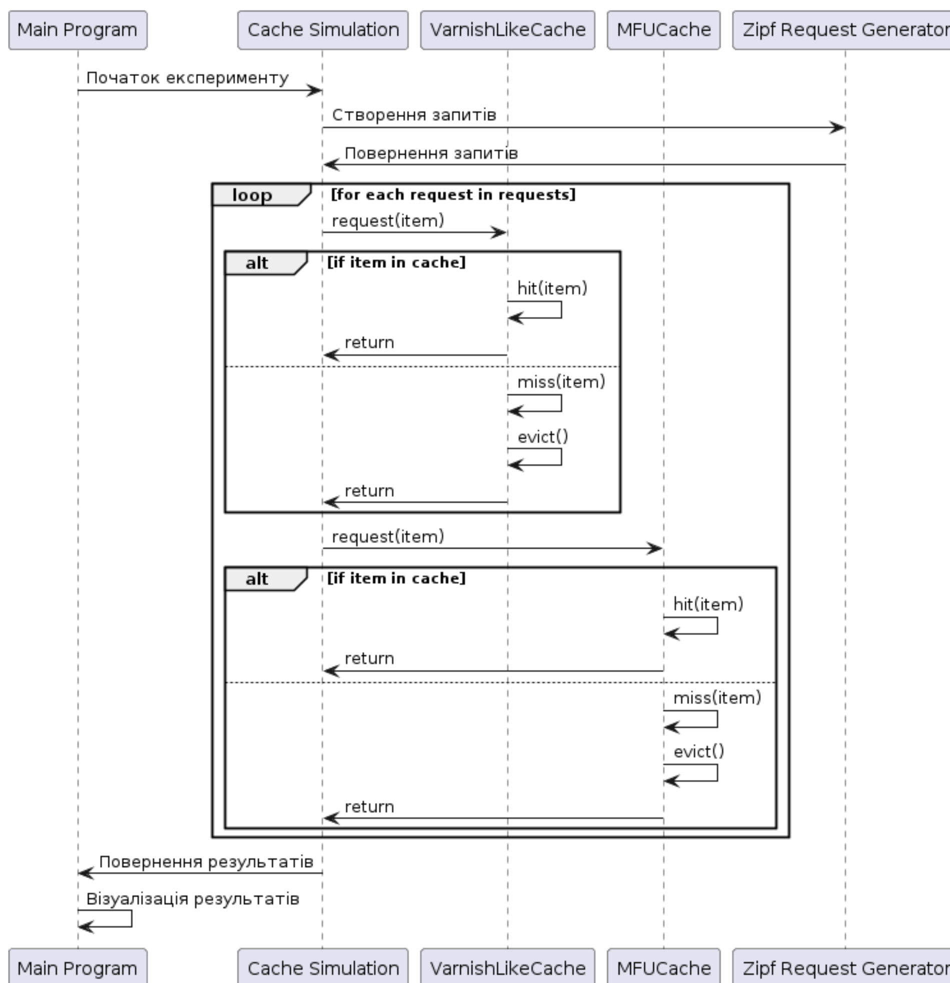

Рисунок 3.1 - Діаграма послідовності

Дана діаграма послідовності надає високорівневе уявлення про те, як працює програма:

- Основна програма: ініціює симуляцію, викликаючи run_simulation функцію з різними розмірами кешу та кількістю елементів.
- Моделювання кешу: керує потоком моделювання, спочатку генеруючи набір запитів на основі закону Зіпфа.
- Генератор запитів Zipf: генерує та повертає послідовність запитів елементів.
- Кеші (LRU та MFU): для кожного запиту елемента кеші обробляють запит як збіг або промах. Якщо це помилка, викликаються процедури виселення (специфічні для LRU або MFU).
- Результати: після обробки всіх запитів результати (як-от середній час завантаження, частота попадань і кількість промахів) повертаються до основної програми.
- Побудова графіків: нарешті, основна програма будує графіки цих результатів для аналізу.

На рисунку 3.2 відображено діаграму послідовності алгоритму Varnish.

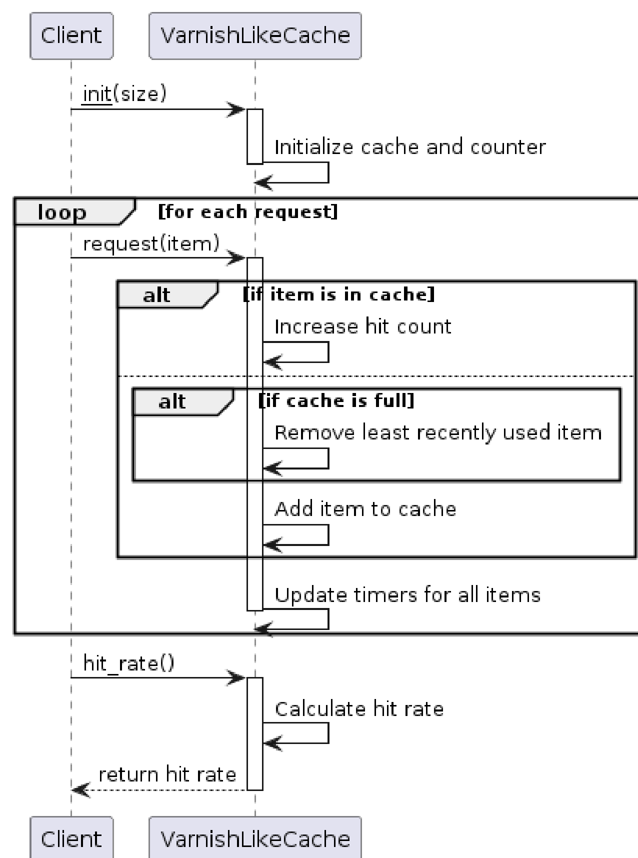

Рисунок 3.2 - Діаграма послідовності Varnish

Діаграма послідовності для VarnishLikeCache відображає взаємодію між клієнтом та кешем, який використовує алгоритм Least Recently Used (LRU). Процес можна розділити на такі етапи:

- Ініціалізація: клієнт створює об'єкт VarnishLikeCache, передаючи розмір кешу. кеш ініціалізується з певним розміром та лічильником попадань (hits).
- Запит на об'єкт: для кожного запиту клієнта, кеш перевіряє, чи існує об'єкт у кеші:
  - Якщо об'єкт існує, лічильник попадань збільшується.
  - Якщо об'єкт відсутній, перевіряється чи кеш повний.
  - Якщо так, видаляється найменш нещодавно використаний об'єкт.
  - Об'єкт запиту додається до кешу.
- Оновлення таймерів: після кожного запиту таймери всіх об'єктів в кеші оновлюються для відстеження їхнього використання.
- Розрахунок hit rate: на завершення, клієнт може запитати hit rate, який розраховується як відношення попадань до загальної кількості запитів.

На рисунку 3.3 відображено діаграму послідовності алгоритму MFU.

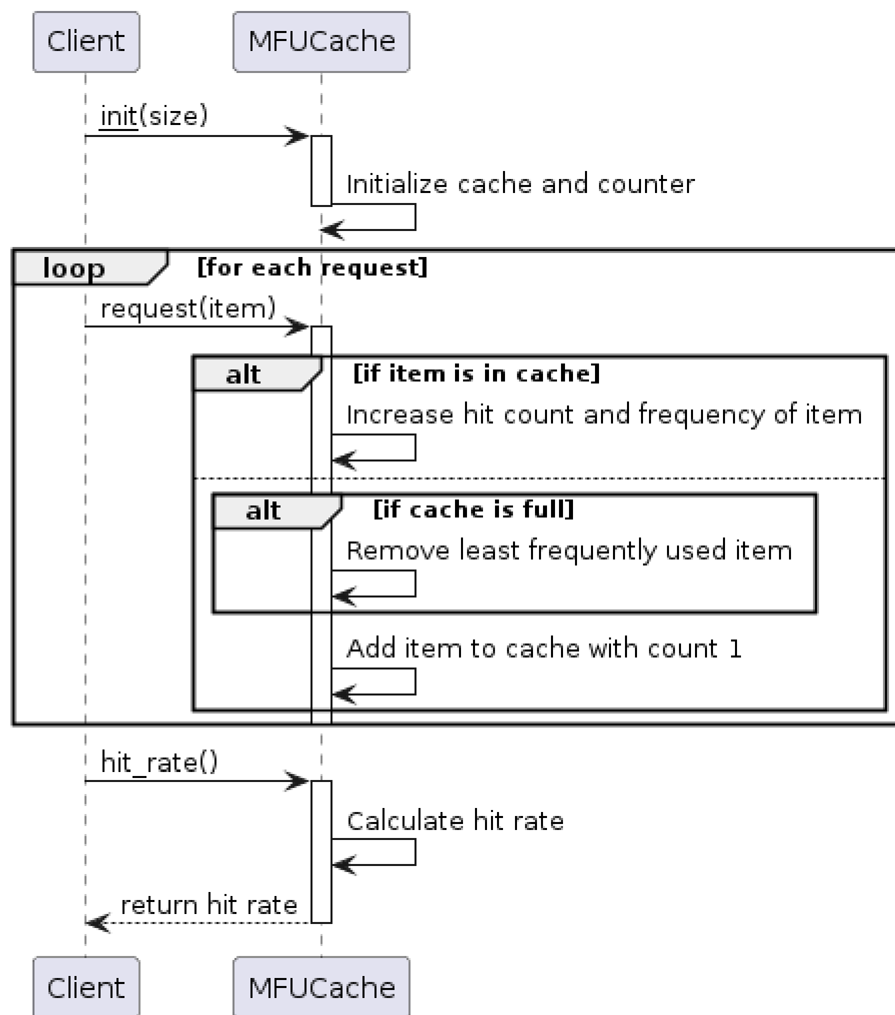

Рисунок 3.3 - Діаграма послідовності MFU

Діаграма послідовності для MFUCache відображає взаємодію з кешем, який використовує алгоритм Most Frequently Used (MFU). Її процес:

- Ініціалізація: аналогічно, клієнт створює об'єкт MFUCache із вказаним розміром кешу.
- Запит на об'єкт: при обробці запиту:
    - Якщо об'єкт вже є в кеші, його частота використання збільшується, як і лічильник попадань.
    - Якщо об'єкту немає, перевіряється чи кеш заповнений.
    - Якщо так, видаляється об'єкт з найменшою частотою використання, після чого новий об'єкт додається до кешу.

- Розрахунок hit rate: клієнт може запросити hit rate, який розраховується так само, як і в VarnishLikeCache.

Основні відмінності між кешами

Метод видалення: VarnishLikeCache видаляє найменш нещодавно використаний об'єкт, в той час як MFUCache видаляє об'єкт з найнижчою частотою використання.

Логіка оновлення: VarnishLikeCache оновлює таймери для всіх об'єктів після кожного запиту, що допомагає визначити "найменш нещодавно використані" об'єкти. MFUCache збільшує лічильник частоти тільки для тих об'єктів, які знаходяться в кеші і запитуються.

Оптимізація: LRU оптимізується для випадків, де нещодавно використані об'єкти ймовірно будуть запитані знову. MFU припускає, що об'єкти, які часто запитуються, продовжуватимуть бути популярними.

Для порівняльного аналізу двох методів кешування — Varnish (LRC) та найчастіше використовуваних (MFU) — створимо експеримент, який оцінює їхню ефективність за допомогою швидкості обробки веб-запитів.

3.2.2 Діаграма активності

Діаграма активності - це графічне представлення послідовності дій, які виконуються в системі або компоненті. Вона зображує послідовність дій, процесів або операцій, що відбуваються в системі від початку до кінця[10].

Використовується для моделювання бізнес-процесів, алгоритмів, поведінки системи, способів взаємодії між об'єктами та рішення в рамках програмного забезпечення.

Елементами діаграми активності є дії (actions), рішення (decisions), вилучення (forks), злиття (joins) та інші конструкції, що дозволяють виразно моделювати логіку процесу.

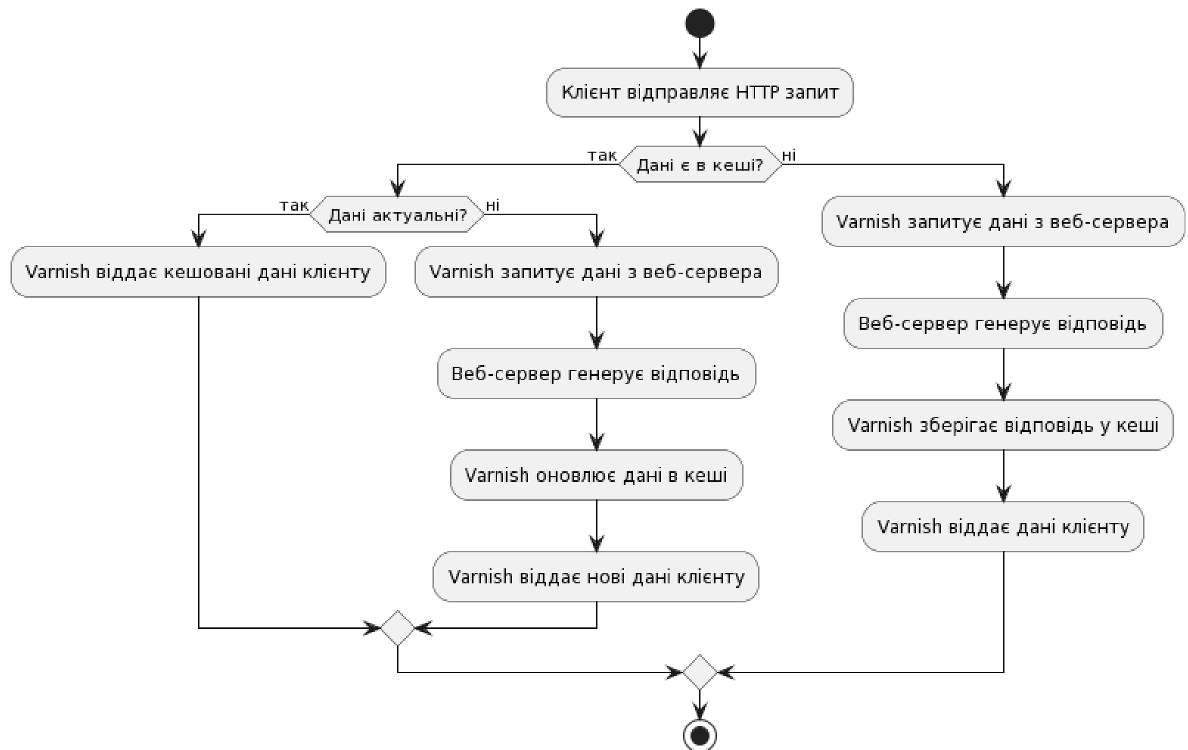

Рисунок 3.4 - Діаграма активності

- Початок: процес розпочинається, коли клієнт відправляє HTTP запит.
- Перевірка наявності даних у кеші: система перевіряє, чи існують запитувані дані в кеші Varnish.
  - Якщо дані відсутні в кеші або неактуальні: процес переходить до запиту даних з веб-сервера.
  - Якщо дані присутні і актуальні: дані негайно віддаються клієнту з кешу.
- Запит даних з веб-сервера: якщо дані в кеші відсутні або потребують оновлення, Varnish запитує свіжі дані від веб-сервера.
- Веб-сервер генерує відповідь: веб-сервер обробляє запит і генерує відповідь для Varnish.
- Оновлення кешу (якщо потрібно): Varnish оновлює дані в своєму кеші новою відповіддю від веб-сервера.
- Віддача даних клієнту: Varnish відсилає відповідь клієнту, завершуючи процес.

- Кінець процесу: процес завершується після віддачі даних клієнту.

Ця діаграма активності демонструє, як Varnish Cache оптимізує віддачу веб-контенту за допомогою кешування, зменшуючи навантаження на веб-сервери та скорочуючи час відгуку для кінцевих користувачів. Важливою частиною цього процесу є рішення про оновлення кешу: якщо дані в кеші застарілі, система автоматично звертається за свіжими даними до веб-сервера, оновлює кеш і забезпечує актуальність відповідей, відданих користувачам.

3.2.3 Діаграма використання

Діаграма використання - це графічне зображення функціональності системи та способів взаємодії між її складовими частинами (акторами) та використовуваними сценаріями (використаннями)[10].

Використовується для ідентифікації, аналізу та опису вимог до системи через розкриття взаємодії між акторами та використаннями (use cases).

Актори - це будь-які зовнішні сутності (користувачі, інші системи тощо), які взаємодіють з системою. Використання - це конкретні сценарії використання системи, що можуть бути запрошені акторами.

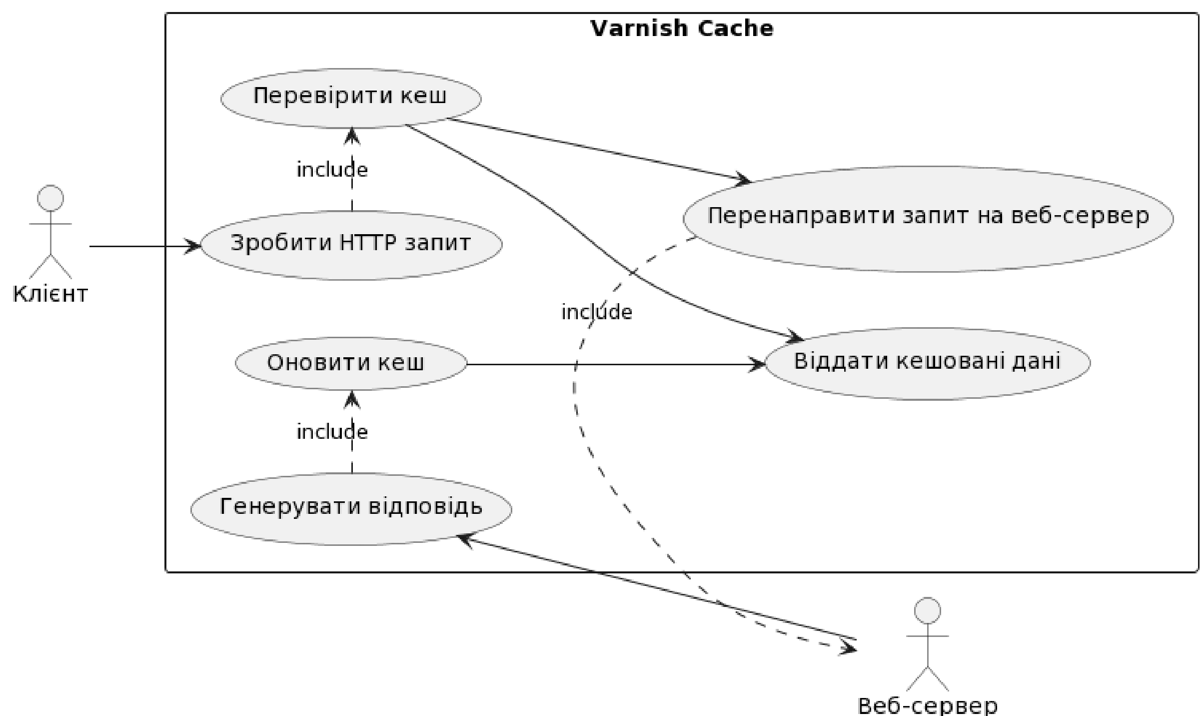

Рисунок 3.5 - Діаграма використання

На рисунку 3.3 представлено наступні взаємодії:

- Зробити HTTP запит:

Клієнт ініціює запит до системи через Varnish Cache.

- Перевірити кеш:

Varnish Cache перевіряє наявність запитуваних даних у кеші.

- Віддати кешовані дані:

Якщо дані доступні в кеші, вони негайно віддаються клієнту.

- Перенаправити запит на веб-сервер:

Якщо дані не знайдені в кеші, запит перенаправляється на веб-сервер.

- Генерувати відповідь:

Веб-сервер обробляє запит і генерує відповідь.

- Оновити кеш:

Varnish Cache оновлює свій кеш з новою відповіддю від веб-сервера.

- Віддати кешовані дані:

Оновлені дані віддаються клієнту.

Ця діаграма використання чітко показує, які дії може ініціювати клієнт та як веб-сервер взаємодіє з Varnish Cache, щоб забезпечити ефективне кешування і віддачу даних.

4 РЕАЛІЗАЦІЯ ТА ПРОГРАМУВАННЯ СИСТЕМИ КЕШУВАННЯ

У цьому розділі досліджуються аспекти реалізації та програмування системи кешування, яка є ключовою складовою сучасних високопродуктивних веб-систем. Кешування дозволяє значно зменшити час відгуку системи та розподілити навантаження на основні обчислювальні ресурси. Один з важливих аспектів ефективного кешування - це структура даних, яка дозволяє швидко зберігати, шукати та інвалідувати кешовані елементи.

Детально розглядається структура даних кешування, яка має відповідати вимогам швидкого доступу до даних, автоматичного управління розміром та високої адаптивності. Для Varnish Cache, як високопродуктивного проксі-сервера та кешувальника веб-запитів, оптимальною є комбінація хеш-таблиці та двозв'язного списку, яка використовується для реалізації алгоритму заміни сторінок "найменш нещодавно використаного" (LRU).

Далі детально розглядаються методи отримання даних з кешу, зокрема хешування запитів, алгоритм LRU, Grace Mode та Saint Mode, а також тегування кешу. Проводиться аналіз найкращого варіанту отримання даних з кешу для Varnish Cache та обгрунтування його переваг.

У заключній частині описується алгоритм кешування, розглядаються два підходи: Varnish-like Cache (LRC - Least Recently Used) та Most Frequently Used (MFU), які демонструють ефективність кешування веб-сторінок в Python.

4.1. Опис структури даних кешування

Кешування є критичною компонентою в архітектурі сучасних високопродуктивних веб-систем. Ефективне кешування забезпечує значне зниження часу відгуку системи та зменшує навантаження на основні обчислювальні ресурси. Структура даних кешування відіграє ключову роль у

реалізації цих переваг, дозволяючи швидко зберігати, пошуковувати, та інвалідувати кешовані елементи.

Структура даних для кешування має задовольняти наступні вимоги:

- Швидкий доступ до даних: операції пошуку, вставки, та видалення повинні виконуватися за мінімально можливий час, ідеально в середньому за час O(1).
- Автоматичне управління розміром: кеш має підтримувати механізми обмеження свого розміру, такі як видалення найменш нещодавно використаних (LRU) або найрідше використовуваних (LFU) елементів.
- Висока адаптивність: структура повинна ефективно адаптуватися до різних моделей доступу до даних і підтримувати різні стратегії кешування.

Для Varnish Cache, який є високопродуктивним проксі-сервером та кешувальником веб-запитів, оптимальною структурою даних для забезпечення ефективного кешування є комбінація хеш-таблиці та двозв'язного списку. Ця структура даних відома як алгоритм заміни сторінок найменш нещодавно використаного (Least Recently Used, LRU) кешу. Така комбінація дозволяє Varnish Cache швидко знаходити та обслуговувати кешовані об'єкти, а також ефективно управляти обмеженим простором кешу, видаляючи найменш нещодавно використані об'єкти.

Хеш-таблиця використовується для швидкого мапінгу ключів запитів до їх відповідних об'єктів у кеші. Ключем може бути URI запиту або будь-яка інша унікальна ідентифікація об'єкта. Хеш-таблиця дозволяє Varnish Cache забезпечувати доступ до кешованих об'єктів за амортизованим часом O(1), що є критично важливим для підтримки високої продуктивності при обслуговуванні великої кількості запитів.

Двозв'язний список використовується разом з хеш-таблицею для реалізації LRU-алгоритму кешування. Коли об'єкт додається до кешу або доступ до існуючого об'єкта здійснюється, він переміщується на передню частину списку. Це означає, що найбільш нещодавно використані об'єкти знаходяться ближче до голови списку, тоді як найменш використані - ближче до хвоста. Коли розмір кешу

досягає свого максимального значення, об'єкти з кінця списку (тобто найменш нещодавно використані) видаляються для звільнення місця для нових об'єктів.

Переваги комбінованої структури

Ця комбінована структура даних дозволяє Varnish Cache ефективно управляти кешем, забезпечуючи швидкий доступ до кешованих даних та оптимізуючи використання обмежених ресурсів пам'яті шляхом видалення найменш нещодавно використаних об'єктів. Вона ідеально підходить для сценаріїв використання Varnish, де важливо мінімізувати затримку відгуку та оптимізувати продуктивність при великому обсязі веб-трафіку.

4.2. Вибір методу отримання даних з кешу

Для ефективного отримання даних з кешу в Varnish Cache, система використовує декілька ключових методів та підходів, які дозволяють оптимізувати процес відновлення кешованих об'єктів[9]. Вибір методу отримання даних базується на потребах високої швидкодії, ефективного використання пам'яті та здатності швидко реагувати на зміни у даних. Нижче представлено детальний опис ключових аспектів цього процесу.

- Хешування запитів

Основою механізму отримання даних з кешу в Varnish є використання хеш-функцій для генерації унікальних ідентифікаторів (ключів) для кешованих об'єктів на основі характеристик запитів, таких як URI, параметри запиту та заголовки. Цей підхід забезпечує миттєвий доступ до об'єктів у кеші та дозволяє ефективно управляти кешем[9].

- Алгоритм LRU (Least Recently Used)

Varnish використовує алгоритм LRU для визначення, які об'єкти слід видалити з кешу, коли необхідно звільнити місце для нових об'єктів. Цей алгоритм дозволяє зберігати в кеші найбільш часто та недавно запитувані об'єкти, тим самим оптимізуючи швидкість відгуку системи[11].

- Grace Mode та Saint Mode

Для оптимізації обробки запитів до тимчасово недоступних або повільних бекендів, Varnish використовує механізми Grace Mode та Saint Mode. Grace Mode дозволяє Varnish віддавати застарілі об'єкти з кешу на деякий час після їхнього офіційного "спливання", тоді як Saint Mode дозволяє обходити бекенди, які повертають помилки, та спробувати інший бекенд або віддати стару копію з кешу.

- Тегування кешу

Varnish дозволяє використовувати теги кешу для гнучкого управління кешованими об'єктами, дозволяючи інвалідувати всі об'єкти, пов'язані з певним тегом. Це особливо корисно в динамічних веб-додатках, де контент може швидко змінюватися, і потрібно ефективно управляти кешем без видалення всієї його вмісту[9].

Вибір методу отримання даних з кешу в Varnish Cache залежить від багатьох факторів, включаючи тип запиту, актуальність даних, та стан бекендів. Через використання хешування, алгоритму LRU, Grace та Saint Modes, а також тегування кешу, Varnish забезпечує високу продуктивність та ефективність управління кешованими даними, підтримуючи при цьому гнучкість у відповідях на динамічно змінюваний веб-контент.

Для Varnish Cache найкращий варіант отримання даних з кешу — це використання комбінованого підходу, який базується на хеш-таблицях для швидкого мапінгу запитів до кешованих об'єктів, разом із застосуванням алгоритму LRU (Least Recently Used) для управління видаленням даних з кешу[12].

Чому цей підхід є найкращим?

Швидкість та ефективність: використання хеш-таблиць забезпечує миттєвий доступ до кешованих об'єктів, оскільки час доступу в ідеальному випадку становить O(1). Це критично важливо для систем, які обробляють велику кількість запитів.

Автоматичне управління ресурсами: алгоритм LRU дозволяє автоматично управляти обмеженими ресурсами пам'яті, видаляючи найменш нещодавно

використані об'єкти для звільнення місця для нових. Це спрощує управління пам'яттю та забезпечує ефективне використання ресурсів.

Простота реалізації: комбінація хеш-таблиць та алгоритму LRU є досить простою для реалізації та інтеграції в існуючу архітектуру Varnish Cache, не вимагаючи складних алгоритмічних або структурних змін.

Саме поєднання простоти реалізації, високої ефективності та широкої підтримки робить цей підхід ідеальним варіантом для системи кешування Varnish Cache.

4.3 Опис алгоритму кешування

Для моделювання ефективності використання кешування веб-сторінок в Python, ми розглянемо два підходи: застосування системи кешування, схожої на Varnish (популярний засіб кешування HTTP-запитів), та простіший метод кешування, наприклад, заснований на найрідше використовуваних елементах (MFU).

- Varnish-like Cache (LRC - Least Recently Used)

Цей метод кешування вибирає для вилучення ті сторінки, які найменше нещодавно використовувались. Це допомагає підтримувати у кеші лише ті елементи, які регулярно запитуються[9;12].

Алгоритм:

- Перевірити, чи є запитуваний елемент у кеші.
- Якщо елемент присутній у кеші (hit):
  - Позначити цей доступ як недавній.
- Якщо елемент відсутній у кеші (miss):
  - Якщо кеш заповнений, вилучити елемент, який мав найменший час доступу.
  - Додати новий елемент до кешу.
- Most Frequently Used (MFU)

Цей метод кешування зосереджений на частоті запитів до сторінок. Він припускає, що сторінки, які часто запитуються, мають більшу ймовірність бути запитаними знову[11].

Алгоритм:

- Перевірити, чи є запитуваний елемент у кеші.
- Якщо елемент присутній у кеші (hit):
    - Збільшити лічильник частоти для цього елемента.
- Якщо елемент відсутній у кеші (miss):
    - Якщо кеш заповнений, вилучити елемент із найнижчою частотою використання.
    - Додати новий елемент до кешу з початковою частотою.

В таблиці 4.1 описано функції та їх призначення.

Таблиця 4.1 - Методи та їх призначення

Функція	**Varnish-like Cache (LRC)**	**Most Frequently Used (MFU)**
Пошук елемента	Перевірка на наявність елемента в кеші.	Перевірка на наявність елемента в кеші.
Обробка hit	Оновлення часу доступу до поточного моменту.	Збільшення лічильника частоти запитів до цього елемента.
Обробка miss	Видалення елемента з найменшим часом доступу, додавання нового.	Видалення елемента з найнижчою частотою, додавання нового.
Політика заміни	Видалення елементів, які не використовувались найдовше.	Видалення елементів із найнижчою частотою запитів.

Логіка експерименту

- Визначення метрик для оцінки:

    - Частота попадань (hit rate): Відсоток запитів, які були успішно оброблені кешем (тобто елемент вже був у кеші).
    - Час обробки запитів: Середній час, необхідний для обробки одного запиту, включаючи час на пошук у кеші та додавання елементів у кеш.
- Створення вхідних даних:

Генерування великої кількості випадкових веб-запитів до різноманітних ресурсів. Запити можуть бути смодельовані на основі ймовірності, що відображає реальну поведінку користувачів, де деякі сторінки запитуються частіше за інші.

- Проведення експерименту:
    - Запустити кеш із зазначеним розміром для кожного методу.
    - Послідовно відправляти запити до обох кешів та записувати результати кожного запиту, включаючи чи було попадання в кеш, та час обробки.
- Аналіз результатів:
    - Обчислити загальну частоту попадань та середній час обробки для кожного методу.
    - Порівняти результати, щоб визначити, який кеш ефективніше справляється з навантаженням і швидше обробляє запити.
- Візуалізація результатів:

Підготувати графіки, що показують частоту попадань та середній час обробки для кожного методу кешування. Це може включати стовпчасті діаграми для частоти попадань та лінійні графіки для часу обробки.

4.4 Тестування

Для визначення ефективного методу пришвидшення сторінок, запустимо програму та проаналізуємо результати. На рисунку 4.1 відображено залежність середнього часу завантаження від кількості унікальних елементів.

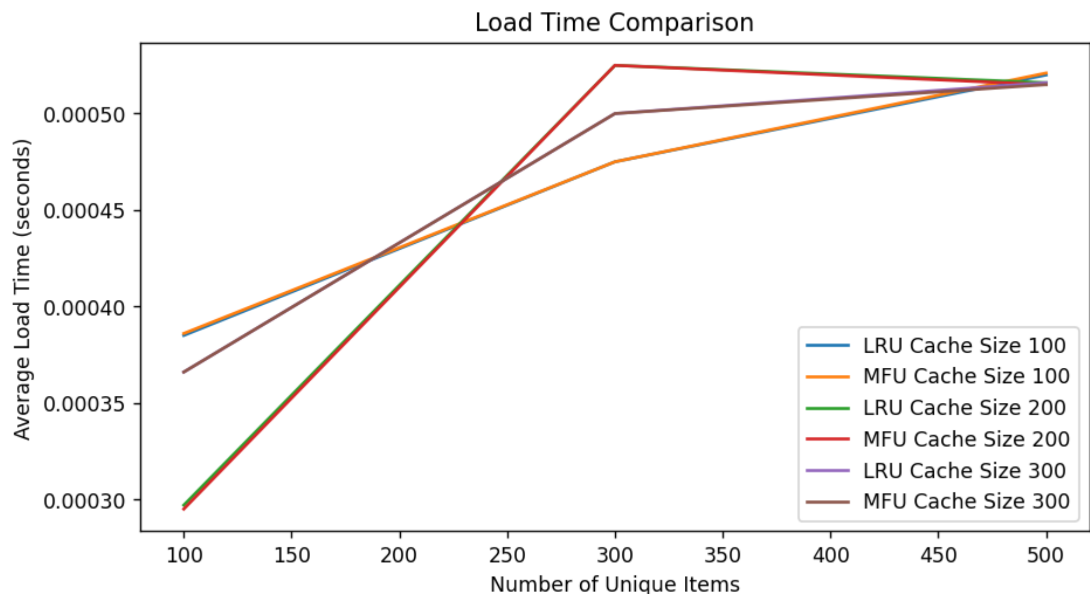

Рисунок 4.1 - Графік залежності середнього часу завантаження від кількості унікальних елементів

Рисунок 4.1 ілюструє, як середній час завантаження сторінки залежить від кількості унікальних елементів у запитах. Це допоможе визначити, наскільки добре кожен метод справляється зі збільшенням різноманітності запитів. Час завантаження зазвичай збільшується зі збільшенням кількості унікальних елементів. Це очікувано, оскільки більше унікальних елементів може спричинити більше промахів кешу, що призведе до довшого часу отримання з основної пам'яті чи сховища.

Аномалії продуктивності: є помітні аномалії приблизно від 300 до 350 унікальних елементів, де MFU з розміром кешу 200 тимчасово покращує (зменшує) свій час завантаження, перш ніж продовжувати збільшуватися. Це може свідчити про певну поведінку алгоритму MFU за певних умов або це може бути викидом через конкретні умови експерименту.

Подібно до графіка коефіцієнта пропусків, більші кеші призводять до кращого (нижчого) часу завантаження, що вказує на менше пропусків кешу. LRU та MFU працюють однаково з варіаціями в певних точках, зокрема, коли MFU з розміром кешу 200 показує падіння часу завантаження, що не відображається в даних LRU.

Зі збільшенням кількості унікальних елементів час завантаження зростає для обох методів. Це може вказувати на те, що кеш частіше стикається з ситуаціями, коли необхідно завантажувати дані з основного джерела, замість того щоб отримувати їх з кешу. Різниця у часі завантаження між LRU і MFU не є значною, що може вказувати на схожу загальну ефективність цих методів у цих умовах.

Рисунок 4.2 показує, як частота попадань у кеш (hit rate) змінюється зі збільшенням розміру кешу для обох методів. Це допомагає зрозуміти, наскільки ефективно кожен метод використовує доступний простір. Цей графік ілюструє порівняння частоти звернень для різних розмірів кешу (100, 200, 300) за стратегіями кешування «Нещодавно використовувані» (LRU) і «Найчастіше використовувані» (MFU), оскільки кількість унікальних елементів коливається від 100 до 500.

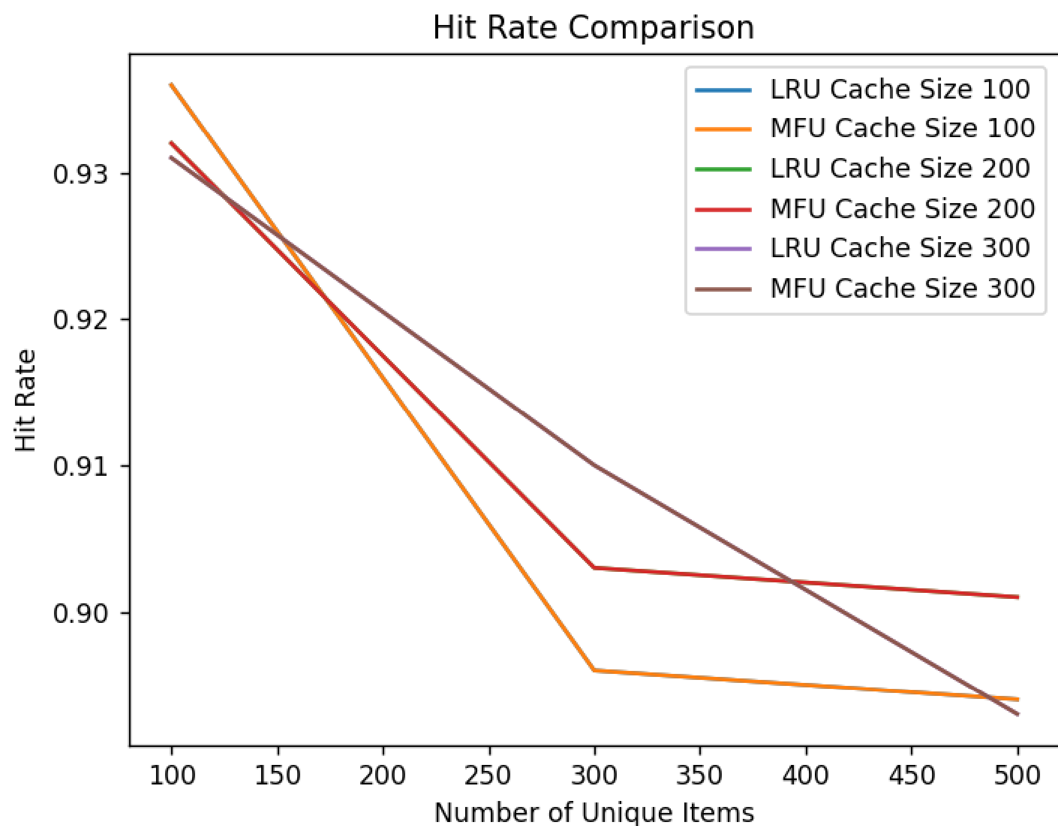

Рисунок 4.2 - Графік залежності хітрейту від розміру кешу

У всіх сценаріях зі збільшенням кількості унікальних елементів відсоток попадань зменшується. Це очікувана поведінка в системах кешування, оскільки

збільшення різноманітності елементів зазвичай призводить до більшої кількості промахів кешу, особливо коли кеш не може містити всі різні елементи, до яких часто звертаються.

Більші розміри кешу зазвичай підтримують вищі показники звернень. Наприклад, рядки для розміру кешу 300 постійно залишаються вищими за рядки для 200 і 100. Це тому, що більший кеш може зберігати більше елементів, що потенційно зменшує кількість промахів.

І LRU, і MFU демонструють подібну схему зниження показника звернень зі збільшенням кількості унікальних елементів. Однак стратегія MFU, здається, має дещо нижчий коефіцієнт звернень, ніж LRU, за тих самих розмірів кешу. Ця різниця особливо помітна при менших розмірах кешу (100 і 200), але менш помітна при розмірі кешу 300. Це може свідчити про те, що MFU може бути не таким ефективним, як LRU, у сценаріях, коли різноманітність запитуваних елементів висока.

Зменшення кількості звернень є відносно лінійним для всіх рядків, що вказує на постійне зниження продуктивності зі збільшенням складності робочого навантаження (кількості унікальних елементів).

Підсумовуючи, результати з рисунку 4.4 підкріплює фундаментальний принцип кешування, що більші кеші працюють краще з точки зору частоти звернень. Він також підкреслює деякі тонкі відмінності між стратегіями LRU і MFU, причому LRU, як правило, забезпечує кращі показники влучень у протестованих конфігураціях. Цей аналіз допомагає зрозуміти, як ці стратегії кешування можуть працювати за різних умов, і може керувати рішеннями щодо конфігурації на основі конкретних потреб програми.

Рисунок 4.3 відображає, як частота промахів (miss rate) змінюється з розміром кешу. Це важливо для розуміння, наскільки ефективно кеш зменшує навантаження на основне сховище даних. Аналіз порівняльного графіка оцінки пропуску.

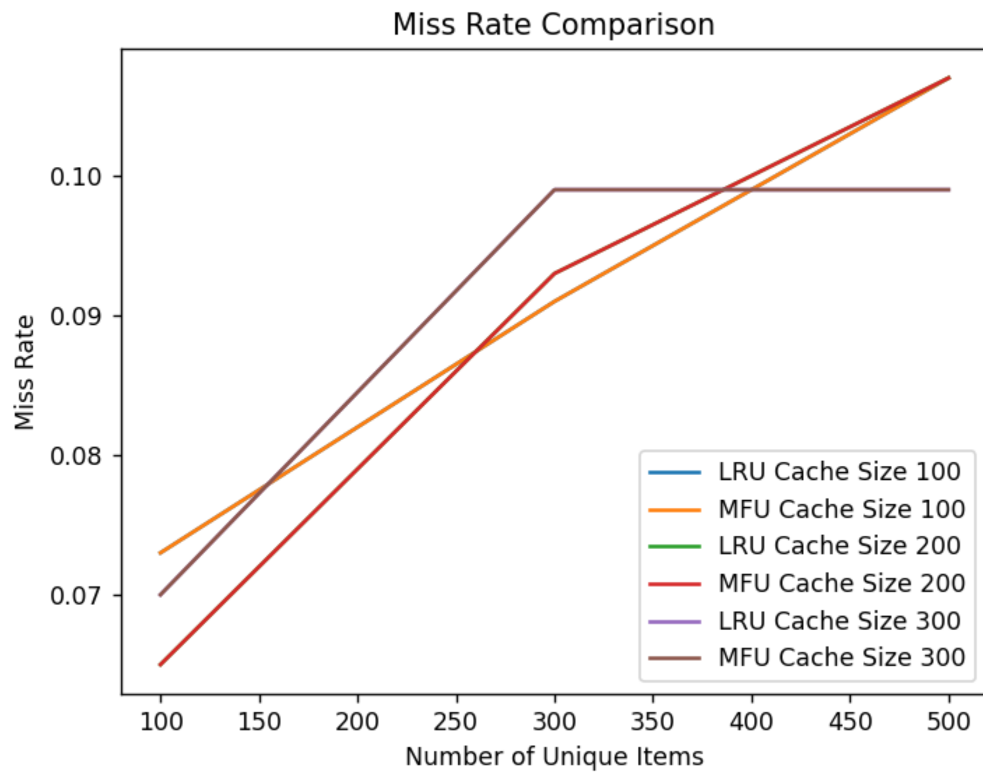

Рисунок 4.3 - Графік залежності між розміром кешу та частотою промахів

Спостереження за тенденцією: зі збільшенням кількості унікальних елементів відсоток промахів зростає для всіх сценаріїв. Це свідчить про те, що зі збільшенням різноманітності елементів у кеші LRU і MFU менш ефективні у зберіганні найнеобхідніших елементів у кеші. Вплив розміру кешу: більші розміри кешу мають стабільно нижчий рівень промахів в обох стратегіях. Це демонструє основний принцип кешу, коли більші кеші можуть утримувати більше елементів і, таким чином, зменшити ймовірність промаху кешу. Лінії для стратегій кешу LRU та MFU дуже близькі одна до одної за продуктивністю, хоча з невеликими варіаціями в різних точках. MFU, здається, працює дещо краще (нижчий коефіцієнт помилок), коли розмір кешу становить 100, але ця різниця зменшується зі збільшенням розміру кешу.

ВИСНОВКИ

У ході дослідження було проведено глибокий аналіз предметної області, що включав вивчення різних стратегій кешування, таких як Least Recently Used (LRU) і Most Frequently Used (MFU). Особливу увагу було приділено аналізу їхньої ефективності у різних сценаріях використання, що дозволило глибше зрозуміти ключові аспекти і потенційні переваги кожного методу. На основі проведеного аналізу було обрано використання LRU (зокрема, реалізація Varnish) як основного методу кешування через його вищу стабільність та ефективність у широкому спектрі умов. Вибір технології також заснований на здатності LRU швидко адаптуватися до змін в патернах доступу до даних, що є критично важливим для динамічних додатків. Було розроблено детальний опис архітектури програмного забезпечення, який включає компоненти кешування, механізми управління пам'яттю та інтерфейси взаємодії з додатком. Архітектура спроектована таким чином, що підтримує легке масштабування та впровадження додаткових оптимізацій.

На основі аналізу результатів тестування, що порівнює стратегії кешування Least Recently Used (LRU, використовується в Varnish) та Most Frequently Used (MFU), можна зробити наступні висновки щодо їхньої ефективності та придатності для різних сценаріїв використання. LRU показала себе трохи краще за MFU, особливо на більших об'ємах кешу. Це може бути пов'язано з тим, що LRU ефективно обробляє сценарії з великою кількістю унікальних запитів, швидко адаптуючись до змін у використанні даних.

У графіках показників влучань (hit rate) та невлучань (miss rate) LRU майже завжди випереджає MFU, що свідчить про вищу ефективність цього методу у різноманітних умовах. LRU виявляється більш ефективним зі збільшенням розміру кешу, що робить його відмінним вибором для систем з великою кількістю пам'яті або для систем, де можливе динамічне збільшення кешу. MFU може виявитися кращим вибором у випадках, коли відомо, що певні дані будуть запитуватися значно частіше інших. Однак, в загальних сценаріях, де розподіл доступу до даних менш передбачуваний, MFU може не давати таких же високих показників, як LRU.

Хоча MFU показує конкурентоспроможні результати на маленьких кешах, його продуктивність може страждати при збільшенні числа унікальних елементів, як видно з графіків.

LRU (Varnish), завдяки його здатності до швидкої адаптації до найновіших запитів та ефективності в умовах великих кешів, є трохи кращим та ефективнішим вибором для загального використання в системах кешування. Ця стратегія виявляється особливо корисною в динамічних умовах, де частота доступу до елементів може швидко змінюватися, що робить її ідеальним варіантом для веб-серверів та інтерактивних застосунків, які мають високі вимоги до швидкодії та чутливості до затримок.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Луотонен, Арі (1997). Web Proxy Servers. Prentice Hall. ISBN 0-13-680612-0.
2. Весселс, Дуане (2001). Web Caching. O'Reilly and Associates. ISBN 1-56592-536-X.
3. Рабинович, Майкл; Спатщак, Олівер (2001). Web Caching and Replication. Addison Wesley. ISBN 0-201-61570-3.
4. "Top Varnish performance metrics". Top Varnish performance metrics. Jul 28, 2015. Retrieved Sep 4, 2020.
5. Varnish cache. URL: https://www.varnish-software.com/products/varnish-cache/
6. How to Use Varnish to Speed up my Website. URL: https://symfony.com/doc/current/http_cache/varnish.html
7. Douglass, Bruce (2002). Real-Time Design Patterns. Addison-Wesley Professional. ISBN 978-0201699562.
8. Douglass, Bruce (2009). Real-Time Agility. Addison-Wesley Professional. ISBN 978-0321545497.
9. Douglass, Bruce (2010). Design Patterns for Embedded Systems in C. Newnes. ISBN 978-1856177078.
10. Fowler, Martin (2004). UML Distilled: A Brief Guide to the Standard Object Modeling Language (3rd ed.). Addison-Wesley. ISBN 0-321-19368-7.
11. Most Frequently Used (MFU) Algorithm in Operating System. URL: https://www.geeksforgeeks.org/most-frequently-used-mfu-algorithm-in-operating-system/
12. LRU Cache.URL: https://leetcode.com/problems/lru-cache/

ДОДАТОК А

ТЕКСТ ПРОГРАМНОГО МОДУЛЯ

«МЕТОДИ ПРИШВИДШЕННЯ ЗАВАНТАЖЕННЯ ВЕБ-СТОРІНОК НА ОСНОВІ КЕШУВАННЯ»

УКР.НТУУ"КПІ ім. Ігоря Сікорського"_ІАТЕ_ЦТЕ_ТР-01_24Б

Аркушів 5

Київ-2024

```
import random
import time
import matplotlib.pyplot as plt
import numpy as np

NUM_REQUESTS = 1000
NUM_ITEMS = 500

def     generate_zipf_requests(num_items,     num_requests,
alpha=1.5):
    weights = np.array([1.0 / (i ** alpha) for i in range(1,
num_items + 1)])
    weights /= weights.sum()
    requests  =  np.random.choice(range(1,  num_items  +  1),
size=num_requests, p=weights)
    return requests

class Cache:
    def __init__(self, size):
        self.size = size
        self.cache = {}
        self.hits = 0
        self.load_time = 0
        self.miss_count = 0  # Track misses for analysis

    def request(self, item):
        start_time = time.time()
        if item in self.cache:
            self.hits += 1
            self.cache[item]  =  time.time()   #  Update  last
accessed time
            self.load_time += (time.time() - start_time)
        else:
            if len(self.cache) >= self.size:
                self.evict()
            self.cache[item] = time.time()
            self.load_time  +=  (time.time()  -  start_time)  +
0.005  # Simulate network delay
            self.miss_count += 1

    def hit_rate(self):
        return self.hits / NUM_REQUESTS

    def average_load_time(self):
        return self.load_time / NUM_REQUESTS
```

```
    def miss_rate(self):
        return self.miss_count / NUM_REQUESTS

    def evict(self):
        pass  # Override in subclasses

class VarnishLikeCache(Cache):
    def evict(self):
        # LRU eviction
        least_recent = min(self.cache.keys(), key=lambda x: 
self.cache[x])
        del self.cache[least_recent]

class MFUCache(Cache):
    def evict(self):
        # MFU eviction
        least_frequent          =          min(self.cache, 
key=self.cache.get)
        del self.cache[least_frequent]

def run_simulation(cache_size, num_items):
    requests      =      generate_zipf_requests(num_items, 
NUM_REQUESTS)
    lru_cache = VarnishLikeCache(cache_size)
    mfu_cache = MFUCache(cache_size)
    for item in requests:
        lru_cache.request(item)
        mfu_cache.request(item)
    return                  lru_cache.average_load_time(), 
mfu_cache.average_load_time(),        lru_cache.hit_rate(), 
mfu_cache.hit_rate(),                lru_cache.miss_rate(), 
mfu_cache.miss_rate()

cache_sizes = [100, 200, 300]
num_items = [100, 300, 500]

results = {}
for size in cache_sizes:
    results[size] = {}
    for items in num_items:
        lru_time,  mfu_time,  lru_hit,  mfu_hit,  lru_miss, 
mfu_miss = run_simulation(size, items)
        results[size][items] = (lru_time, mfu_time, lru_hit, 
mfu_hit, lru_miss, mfu_miss)
```

```
# Plotting Load Time Comparison
fig, ax = plt.subplots()
for size, data in results.items():
    lru_times = [data[items][0] for items in num_items]
    mfu_times = [data[items][1] for items in num_items]
    ax.plot(num_items, lru_times, label=f'LRU Cache Size
{size}')
    ax.plot(num_items, mfu_times, label=f'MFU Cache Size
{size}')
ax.set_xlabel('Number of Unique Items')
ax.set_ylabel('Average Load Time (seconds)')
ax.legend()
ax.set_title('Load Time Comparison')
plt.show()

# Plotting Hit Rate Comparison
fig, ax = plt.subplots()
for size, data in results.items():
    lru_hits = [data[items][2] for items in num_items]
    mfu_hits = [data[items][3] for items in num_items]
    ax.plot(num_items, lru_hits, label=f'LRU Cache Size
{size}')
    ax.plot(num_items, mfu_hits, label=f'MFU Cache Size
{size}')
ax.set_xlabel('Number of Unique Items')
ax.set_ylabel('Hit Rate')
ax.legend()
ax.set_title('Hit Rate Comparison')
plt.show()

# Plotting Miss Rate Comparison
fig, ax = plt.subplots()
for size, data in results.items():
    lru_misses = [data[items][4] for items in num_items]
    mfu_misses = [data[items][5] for items in num_items]
    ax.plot(num_items, lru_misses, label=f'LRU Cache Size
{size}')
    ax.plot(num_items, mfu_misses, label=f'MFU Cache Size
{size}')
ax.set_xlabel('Number of Unique Items')
ax.set_ylabel('Miss Rate')
ax.legend()
ax.set_title('Miss Rate Comparison')
plt.show()
```