МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»
Навчально-науковий інститут прикладного системного аналізу
Кафедра системного проектування

Курсова робота

з дисципліни

Паралельні обчислення

Тема: Розробка веб-сервісу для пошуку інформації за ключовими словами

Виконала: студентка IV курсу
групи ДА-21
Петрик Владислава
Перевірив: асистент Яременко В. С.

Київ – 2025

ЗМІСТ

ВСТУП

В наш час обсяги текстової інформації зростають дуже стрімко, тому все більш актуальним стає питання швидкого та ефективного пошуку даних. Звичайні методи послідовного перебору файлів вже не можуть задовольнити потреби користувачів, так як вони працюють дуже повільно, якщо обсяги даних великі. Тому враховуючи сучасні умови розвитку комп'ютерної техніки коли майже кожен процесор є багатоядерним, логічним рішенням є використання паралельних обчислень. Це дозволяє розподілити навантаження між потоками та значно пришвидшити виконання більш складних задач.

Головною метою даної курсової роботи є розробити веб-сервіс для пошуку інформації у текстових файлах за ключовими словами. Основою системи є серверна частина мовою програмування C++, яка використовує власноруч реалізований пул потоків для паралельної побудови інвертованого індексу та одночасної обробки запитів від багатьох користувачів. Для того, щоб забезпечити коректну роботу з даними у багатопотоковому середовищі, потрібно рішити проблеми синхронізації та уникнути стану гонки. Також одним із завдань роботи є організація взаємодії між сервером та клієнтом через мережеві сокети.

Робота складається зі вступу, чотирьох розділів, висновку, списку використаних джерел та додатків.

РОЗДІЛ 1. ТЕОРЕТИЧНІ ВІДОМОСТІ

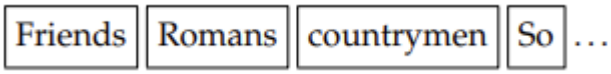
1.1. Інвертований індекс

Інвертований індекс (inverted index) – це структура даних у галузі інформаційного пошуку, яка зв’язує ключові слова з документами, у яких вони зустрічаються, що дозволяє виконувати дуже швидкий текстовий пошук. На відміну від прямого індексу, в якому для кожного документа зберігається список слів, які він містить, в інвертованому індексу все навпаки, тобто для кожного унікального слова формується список документів, в яких дане слово зустрічається.

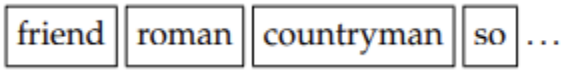
Процес побудови інвертованого індексу включає наступні етапи [1]:

1. Збір документів, які потрібно проіндексувати:

2. Токенізація тексту шляхом розбиття тексту на токени, зазвичай це слова:



3. Лінгвістична обробка, під час якої слова приводяться до нижнього регістру, видаляються розділові знаки тощо:



4. Індексація документів у яких зустрічається кожен термін через створення інвертованого індексу, що складається зі словника та списків входжень.

В даній курсовій роботі списки входжень містять не тільки ідентифікатори документів, а й позиції слів у тексті, що потрібно для реалізації пошуку фраз та відображення уривків зі знайденою фразою в результатах пошуку.

Основною перевагою інвертованого індексу, очевидно, є швидкість. Пошук слова зводиться до його знаходження в словнику та зчитування готового списку документів. В свою чергу, часова складність такого пошуку залежить від кількості знайдених документів, а не від загального обсягу даних.

1.2. Пул потоків

Пул потоків – це патерн, який створює певну кількість потоків і використовує їх для виконання задач зі спільної черги виконання. Основна перевага цього механізму полягає в тому, що потоки створюються лише один раз, що підвищує продуктивність програми, так як створення потоку вважається доволі затратною за часом та ресурсомісткою операцією.

Роботу цього патерну можна порівняти з корпоративним автопарком, оскільки для кожного співробітника було б задорого і непрактично мати власний службовий автомобіль для рідкісних поїздок, тому компанії пропонують спільний автопарк. Пул потоків реалізує схожу ідею поміщаючи завдання, що можуть виконуватися паралельно, в спільну чергу, звідки їх забирають робочі потоки. Після завершення виконання потік не знищується, а повертається до черги за новим завданням [2].

Безумовно, використання одного потоку на одного клієнта при зростанні кількості користувачів призведе до значних затрат системних ресурсів. І, як вже було сказано, створення та знищення потоку є доволі дорогою операцією для операційної системи, оскільки вимагає виділення пам'яті під стек та структур даних ядра. Більше того, занадто велика кількість одночасно працюючих потоків призводить до перемикання контексту, що може знизити загальну продуктивність сервера.

Використання пулу потоків якраз допомагає вирішити ці проблеми. Перш за все, потоки створюються один раз при старті програми і

використовуються багаторазово для виконання різних завдань. По-друге, кількість робочих потоків обмежена і відповідає кількості ядер процесора, що запобігає перевантаженню системи. І насамкінець у випадку, коли всі потоки зайняті, нові запити від клієнтів потрапляють у чергу очікування і обробляються як тільки звільняється один з потоків.

1.3. Проблеми синхронізації даних

Однією з найголовніших проблем в багатопотокових системах є забезпечення цілісності даних при одночасному доступі до них з різних потоків. Якщо два або більше потоки намагаються виконати запис одночасно в одну й ту ж саму область пам’яті, або з одного потоку дані читаються, а в цю ж саму область пам’яті паралельно записуються з іншого, виникає проблема, відома як стан гонки (race condition). Це може призвести до непередбачуваної поведінки програми, пошкодження структур даних або отримання некоректних результатів.

Для запобігання таким ситуаціям використовуються примітиви синхронізації. До основних з них відносять:

- М’ютекси – примітив синхронізації, що обмежує доступ до певної ділянки коду, дозволяючи лише одному потоку одночасно працювати зі спільним ресурсом.
- Семафор – примітив синхронізації, що контролює кількість потоків, які можуть одночасно отримувати доступ до ресурсу, використовуючи для цього лічильник дозволених входів.
- Умовна змінна – примітив синхронізації, який дозволяє потокам чекати, доки інший потік не виконає певну умову та не надішле сигнал, який розбудить очікуючі потоки.

Основною проблемою зазначених примітивів синхронізації є ризик виникнення deadlock (взаємного блокування) у випадку неправильного використання блокувань, а також ситуації, коли певні потоки можуть тривалий час не отримувати доступ до ресурсів (starvation).

Існують також нестандартні або додаткові примітиви. До них належить спін-лок (spin lock), рекурсивний лок (recursive lock) та рід-райт лок (read-write lock).

Враховуючи завдання курсової роботи, найбільш доцільним буде застосувати рід-райт лок. Оскільки операції читання відбуваються набагато частіше, чим операції запису, використання звичайного м'ютекса, який блокує доступ для всіх, є неоптимальним. Саме з цих причин краще використати рід-райт лок (в мові C++ це std::shared_mutex), який дозволяє декільком потокам одночасно читати дані, але надає доступ лише одному потоку для запису.

1.4. Організація мережевої взаємодії

Для забезпечення комунікації між клієнтом і сервером використовується механізм сокетів. Сокет – це програмний інтерфейс між прикладним процесом і транспортним протоколом в операційній системі. Образно його можна порівняти з дверима, коли процес надсилає повідомлення в сокет, після чого мережа доставляє їх до дверей отримувача на іншому комп'ютері [3].

В цій курсовій роботі для передачі даних обрано протокол TCP. Це протокол зі встановленням з'єднання, який гарантує надійну доставку потоку байтів від відправника до отримувача без втрат та дублювання, а також забезпечує перевірку цілісності і контроль порядку повідомлень. Для організації обміну даними обрано архітектуру “клієнт-сервер”, яка передбачає наявність двох типів процесів. Першим процесом є сервер, який запускається першим і переходить у стан очікування вхідних запитів. Він створює сокет для

прослуховувань і очікує на підключення. Другим процесом є клієнт, який ініціює зв'язок, створюючи власний сокет і надсилаючи запит на встановлення з'єднання з сервером за відомою IP-адресою та портом. Як тільки з'єднання встановлено, обидві сторони можуть обмінюватися даними через операції запису та читання з сокета.

РОЗДІЛ 2. ПРОЕКТУВАННЯ СИСТЕМИ

2.1. Use case діаграма

Для того, щоб визначити, як користувачі взаємодіють із системою, які дії можуть в ній виконувати, які ролі існують та які завдання вирішує програмний продукт, використовується use case діаграма. Вона є важливим етапом проектування, оскільки без чіткого розуміння сценаріїв використання доволі складно створити ефективну систему. Use case діаграма дозволяє визначити, хто саме користуватиметься системою, які функції будуть доступні кожному типу користувачів та як відбувається взаємодія із сервером. Дану діаграму зображено на рисунку нижче (рис. 2.1.1):

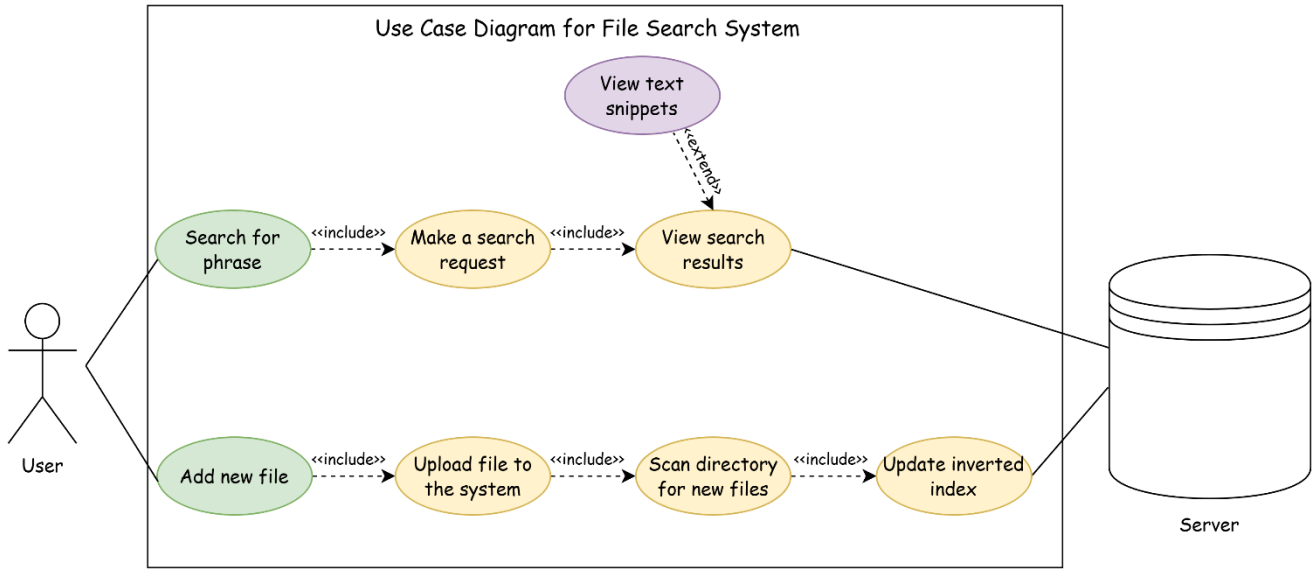


Рисунок 2.1.1 – Use case діаграма

Як видно з діаграми, в системі є актор користувач, який взаємодіє з серверною частиною. Для користувача основним сценарієм є виконання пошукових запитів. Тобто користувач вводить слово або фразу, яку потрібно знайти, після чого запит надсилається на сервер. Сервер в свою чергу обробляє запит і повертає список знайдених файлів, відсортованих за кількістю входжень шуканої фрази у кожному файлі.

Після отримання результатів користувач може ввести індекс файлу зі списку і надіслати запит на отримання уривків тексту зі знайденою фразою. Сервер знаходить всі входження фрази у файлі та формує уривки, додаючи по 40 символів до і після знайденої фрази. Щоб було зручніше прочитати і зробити вивід чистішим в цих фрагментах видаляються переноси рядків та табуляції.

Варто додати, що сервер може обробляти лише фіксовану кількість одночасних клієнтів. І якщо ця кількість досягла максимуму, нові користувачі не відхиляються, а додаються в чергу очікування. В такому разі клієнт отримує повідомлення про те, що сервер зайнятий, і просто чекає, доки звільниться місце. Як тільки потік звільняється, сервер одразу бере наступного клієнта з черги в роботу.

Також система підтримує динамічне оновлення даних. Користувач може додавати нові текстові файли у директорію з даними, а сервер раз на хвилину перевіряє наявність цих нових файлів. Слід зазначити, що оновлення індексу реалізовано так, що система не перебудовує весь індекс з нуля, а додає до бази тільки ті файли, які з'явились з моменту останньої перевірки, використовуючи для цього лічильник вже проіндексованих документів. Це дозволяє уникнути значних витрат ресурсів і часу.

2.2. Діаграма класів

Діаграма класів використовується для проектування внутрішньої структури системи та визначення взаємозв’язків між її компонентами. Вона слугує основою для подальшої реалізації, оскільки на ній наочно видно, які класи будуть створені, які атрибути та методи вони будуть містити, а також як вони взагалі будуть взаємодіяти між собою. Діаграма класів наведена нижче рис. 2.2.1:

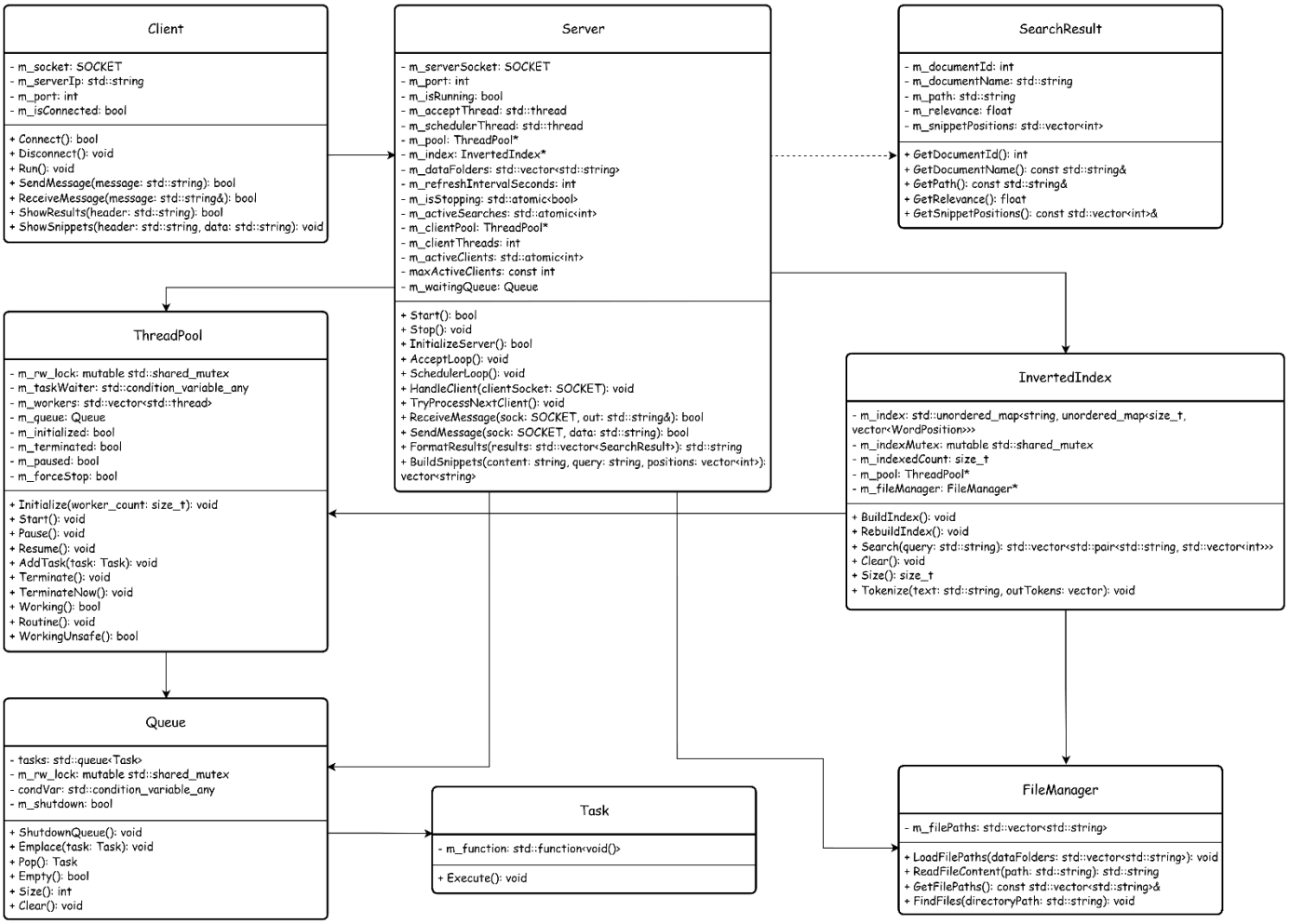


Рисунок 2.2.1 – Діаграма класів

Далі подано опис кожного класу з поясненням його призначення, атрибутів та методів:

1) Server:

Відповідає за запуск сервера, прийом з'єднань, взаємодію з індексом, файловим менеджером та пулом потоків.

- *Attributes:*

  - m_serverSocket – дескриптор слухаючого сокета сервера.

  - m_port – порт для з'єднання з клієнтами.

  - m_isRunning – флаг що показує чи працює сервер.

  - m_acceptThread – потік для прийому нових клієнтів.

- m_schedulerThread – потік для динамічного оновлення індексу.

- m_pool, m_index, m_fileManager – вказівники на класи для обробки даних і потоків.

- m_dataFolders – список директорій в датасеті, які скануються на наявність файлів.

- m_refreshIntervalSecond – проміжок часу між оновленням індексу.

- m_waitingQueue – черга очікування для клієнтів при перевантаженні.

- m_clientPool – вказівник пул потоків для обслуговування клієнтів.

- m_clientThreads –кількість потоків виділених для обробки клієнтів.

- m_activeClients – лічильник активних підключень.

- maxActiveClients – максимальна кількість одночасних підключень.

- m_isStopping – атомарна змінна для зупинки фонових потоків.

- *Methods:*

- Start() – запускає сервер, ініціалізує сокет і приймає з’єднання.

- Stop() – зупиняє сервер і звільняє ресурси.

- InitializeServer() – налаштовує сокет та мережеву адресу.

- HandleClient(SOCKET) – обробляє пошукові запити клієнта.

- AcceptLoop() – цикл для прийому нових з'єднань.

- SchedulerLoop() – для динамічного оновлення індексу.

- TryProcessNextClient() – перевіряє чергу очікування та запускає обробку наступного клієнта, якщо місце звільнилось.

- ReceiveMessage(), SendMessage() – методи для обміну даними через сокет.

- BuildSnippets() – формує фрагменти тексту разом із знайденою фразою.

- FormatResults() – форматує результати пошуку в рядок для відправки.

2) Client:

Встановлює з’єднання із сервером, надсилає пошукові запити та отримує результати.

- *Attributes:*

  - m_socket – сокет підключення.

  - m_serverIp – IP-адреса сервера.

  - m_port – порт сервера.

  - m_isConnected – стан підключення клієнта до сервера.

- *Methods:*

  - Connect(string host, int port) – встановлює з’єднання з сервером.

  - Disconnect() – розриває з'єднання та звільняє ресурси сокета.

  - Run() – головний цикл роботи клієнта з вводом команд і обробкою відповідей.

  - SendMessage(message) – відправляє текстове повідомлення на сервер.

  - ReceiveMessage(message) – отримує відповідь від сервера.

  - ShowResults(header) – виводить список знайдених файлів.

  - ShowSnippets(header, data) – форматує та відображає фрагменти тексту.

3) FileManager:

Керує скануванням файлової системи та зчитуванням вмісту файлів.

- *Attributes:*

  - m_filePaths – вектор шляхів до знайдених файлів.

- *Methods:*

  - LoadFilePaths() – сканує директорії на наявність текстових файлів.

  - ReadFileContent() – зчитує повний текст файлу за вказаним шляхом.

  - GetFilePaths() – повертає список шляхів до файлів.

  - FindFiles(directoryPath) – рекурсивний метод, що обходить директорії та шукає текстові файли.

4) InvertedIndex:

Створює та оновлює інвертований індекс для зручного і ефективного пошуку по наявних в системі документах.

- *Attributes:*

  - index – хеш-таблиця, яка зберігає слова та в яких документах і на яких місцях вони знаходяться.

  - m_indexMutex – мютекс для потокобезпечного читання та запису.

  - m_indexedCount – лічильник кількості вже проіндексованих файлів.

  - m_pool – вказівник на пул потоків.

  - m_fileManager – вказівник на файловий менеджер для доступу до файлів.

- *Methods:*

  - BuildIndex() – будує індекс для файлів, використовуючи багатопоточність.

  - RebuildIndex() – повністю очищає та перебудовує індекс.

  - Search(query) – виконує пошук за фразою та повертає відсортовані результати.

  - Clear() – очищає структуру даних індексу.

  - Size() – повертає кількість унікальних слів у індексі.

  - Tokenize(text, outTokens) – метод для розбиття тексту на токени (слова) та визначення їх позицій.

5) SearchResult:

Клас для зберігання та передачі результатів пошуку по конкретному документу.

- *Attributes:*

  - m_documentId – унікальний ідентифікатор документа.

  - m_documentName – назва файлу.

- m_path – шлях до файлу в системі.

- m_relevance – кількість знайдених збігів фрази.

- m_snippetPositions – вектор позицій слів у тексті для генерації уривків.

- *Methods:*

- GetDocumentId() – повертає ідентифікатор документа.

- GetDocumentName() – повертає назву файлу.

- GetPath() – повертає шлях до файлу.

- GetRelevance() – повертає кількість знайдених збігів шуканої фрази в документі.

- GetSnippetPositions() – повертає позиції в тексті, де зустрічається шукана фраза.

6) ThreadPool:

Розподіляє завдання між робочими потоками для забезпечення паралелізму.

- *Attributes:*

- m_workers – робочі потоки.

- m_queue – містить завдання для виконання.

- m_rw_lock – мютекс для синхронізації доступу до стану пулу.

- isPaused – стан зупинки роботи пулу потоків.

- m_taskWaiter – умовна змінна для очікування нових завдань або зміни стану.

- m_initialized, m_terminated, m_paused, m_forceStop – прапорці стану пулу.

- *Methods:*

- Initialize(worker_count) – створює пул із заданою кількістю потоків.

- Start(), Pause(), Resume() – методи для запуску пулу, тимчасової зупинки виконання задач та відновлення роботи потоків відповідно.

- AddTask(task) – додає нове завдання у чергу виконання.

- Terminate() – метод для повної зупинки роботи потоків.

- TerminateNow() – метод для примусової зупинки роботи потоків.

- Routine() – функція, яку виконує кожен робочий потік, а саме очікування задачі та її виконання.

- WorkingUnsafe() – метод перевірки стану без блокування.

7) Queue:

Блокуюча черга задач, яка використовується пулом потоків.

- *Attributes:*

  - tasks – черга для зберігання задач.

  - m_rw_lock – мютекс для захисту операцій додавання та видалення елементів.

  - condVar – умовна змінна яка повідомляє потоки коли появляються нові завдання.

  - m_shutdown – прапорець для завершення роботи черги.

- *Methods:*

  - Emplace(task) – додає нове завдання в кінець черги.

  - Pop() – видаляє завдання з черги і блокує потік, якщо черга пуста.

  - ShutdownQueue() – зупиняє чергу та розблоковує очікуючі потоки.

  - Empty() – перевіряє, чи є в черзі задачі.

  - Size() – повертає кількість задач.

  - Clear() – очищає чергу від усіх завдань.

8) Task:

Окрема задача пошуку, яку виконує потік.

- *Attributes:*

  - m_function – зберігає саму функцію, яку потрібно виконати.

- *Methods:*

  - Execute() – запускає виконання цього завдання.

2.3. Діаграма послідовностей

Наступним етапом проектування є побудова діаграми послідовностей, яка показує, як саме відбувається обмін повідомленнями між клієнтом і сервером під час обробки пошукового запиту. Вона відображає порядок надсилання запитів, обробку повідомлень різними потоками на сервері та повернення результатів користувачу. Діаграма послідовностей зображена на рисунку 2.3.1:

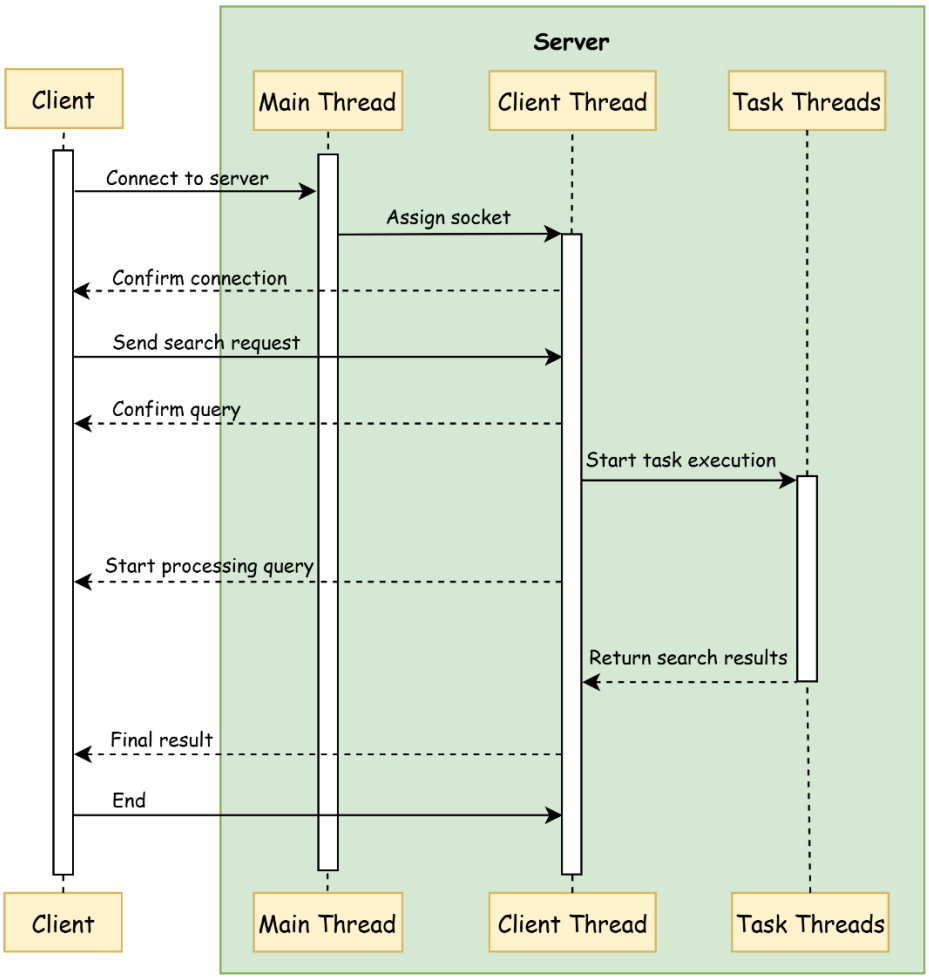


Рисунок 2.3.1 – Діаграма послідовностей

Розглянемо детальніше послідовність обміну повідомленнями, відображену на діаграмі. Спочатку клієнт встановлює з'єднання з сервером шляхом надсилання запиту на підключення. Головний потік сервера приймає цей запит, призначає сокет і передає керування клієнтському потоку, який пітверджує встановлення з'єднання. Далі клієнт надсилає пошуковий запит, який оброблюється в цьому ж клієнтському потоці, що отримав сокет для взаємодії. Отримавши команду, клієнтський потік передає задачу на виконання до пулу потоків. Один із вільних робочих потоків бере цю задачу і виконує пошук через інвертований індекс. Коли пошук завершено, робочий потік повертає знайдене слово або фразу назад до клієнтського потоку, а той надсилає фінальний результат клієнту, після чого взаємодія завершується.

2.4. Deployment діаграма

Deployment діаграма потрібна для того, щоб показати, на яких фізичних вузлах розгорнуті компоненти системи та як вони взаємодіють між собою під час роботи. На ній можна побачити, де саме знаходяться основні модулі і який тип зв'язку між ними використовується. Така діаграма є важливою саме при впровадженні системи, так як допомагає побачити її структуру з боку інфраструктури. Deployment діаграма наведена на рисунку нижче (рис. 2.4.1):

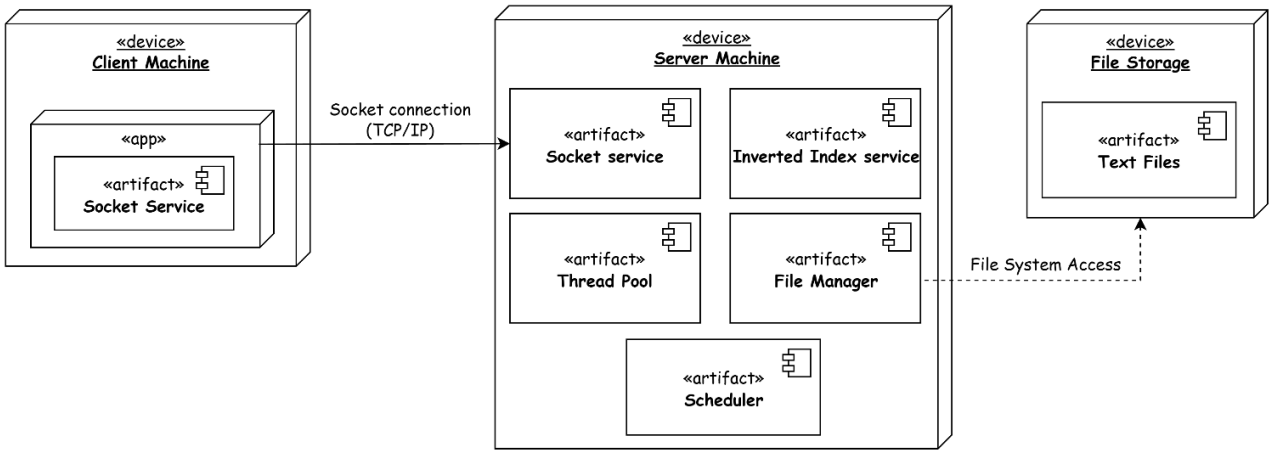


Рисунок 2.4.1 – Deployment діаграма

Як бачимо з діаграми, система складається з кількох вузлів і кожен з них виконує свою роль.

Client Machine це робоча станція користувача, яка містить клієнтський застосунок. Він в свою чергу використовує артефакт Socket Service для того, щоб встановити з’єднання з сервером через мережевий протокол TCP/IP, а також надсилати пошукові запити і отримувати результати у відповідь.

Server Machine є центральним вузлом обробки даних, що включає в себе Socket Service, який відповідає за прийом вхідних з’єднань від клієнтів, Thread Pool для багатопотокової обробки задач та Inverted Index Service, який зберігає структуру індексу в оперативній пам'яті та виконує пошук за ключовими словами. Також сюди входять File Manager, який потрібен для роботи з файловою системою, та Scheduler, який відповідає за динамічне оновлення індексу.

Взаємодія між цими артефактами відбувається всередині самого сервера. Scheduler кожну хвилину перевіряє оновлення і звертається для цього до File Manager, а той в свою чергу зчитує дані та передає їх для оновлення в Inverted Index Service, який використовує ресурси Thread Pool.

Окремим вузлом виділено File Storage, який використовується як фізичне сховище текстових файлів. Сервер звертається до нього за допомогою системних викликів файлового вводу-виводу для читання файлів під час побудови або оновлення індексу.

РОЗДІЛ 3. ОПИС ПРОГРАМНОЇ РЕАЛІЗАЦІЇ

3.1. Структура проекту і засоби розробки

Програму було реалізовано мовою програмування C++. Для забезпечення мережевої взаємодії використано бібліотеку Winsock2, з допомогою неї можна створити надійне з'єднання між клієнтом і сервером. Для самої збірки та керування залежностями проекту використано систему CMake. Конфігураційний файл розташовано в кореневій директорії проекту, в ньому знаходяться інструкції для компіляції вихідного коду, підключення необхідних бібліотек для сокетів та визначення шляхів до заголовочних файлів і ресурсів. Структура проекту наведена на рисунку 3.1.1:

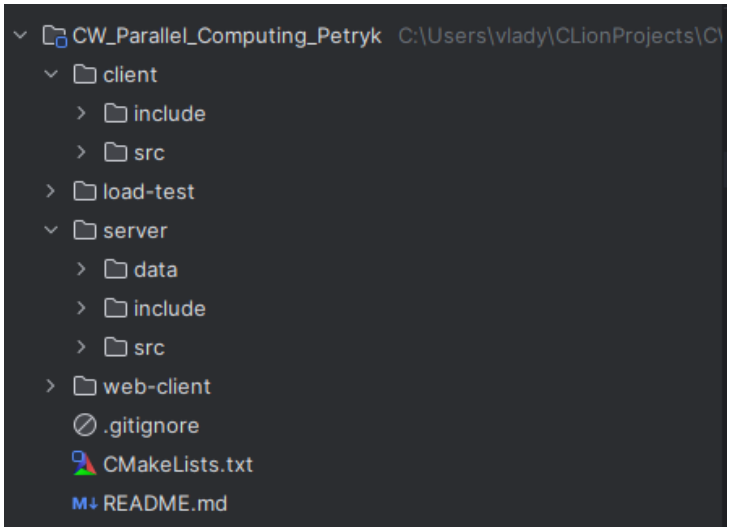


Рисунок 3.1.1 – Загальна структура проекту

Як в директорії сервера, так і в директорії консольного клієнта є папки include/ та src/. У першій знаходяться заголовочні файли “.h” для опису класів, а в другій файли “.cpp” з самою реалізацією. Розділяти файли таким чином вважається стандартною практикою, оскільки це спрощує навігацію по коду та його підтримку. Вміст цих папок для сервера і клієнта зображено на рисунках 3.1.2 та 3.1.3 відповідно:

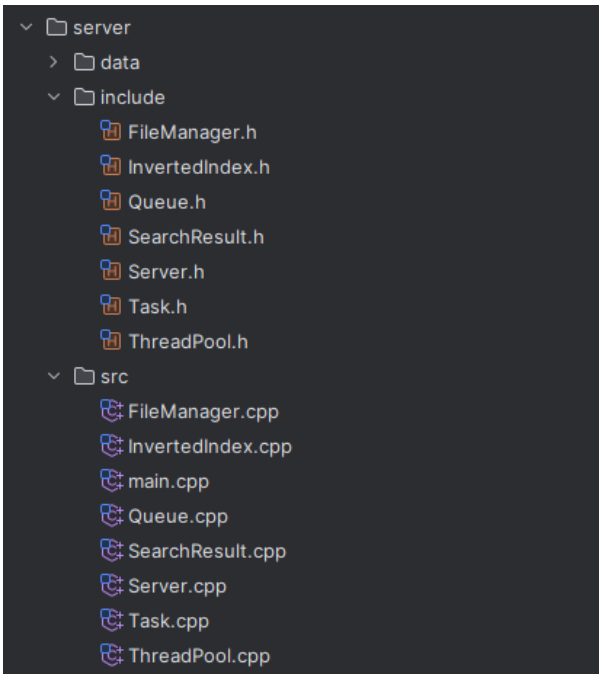


Рисунок 3.1.2 – Вміст папки сервера

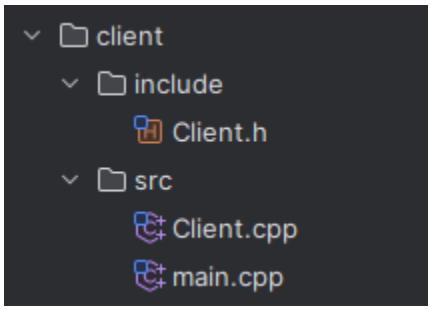


Рисунок 3.1.3 – Вміст папки клієнта

У папці web-client/ міститься код для веб-версії клієнта. Тут знаходяться файли для реалізації інтерфейсу в браузері, а також скрипт проксі-сервера на node.js, який відповідає за трансляцію HTTP-запитів у TCP-сокети для спілкування з сервером (рис. 3.1.4):

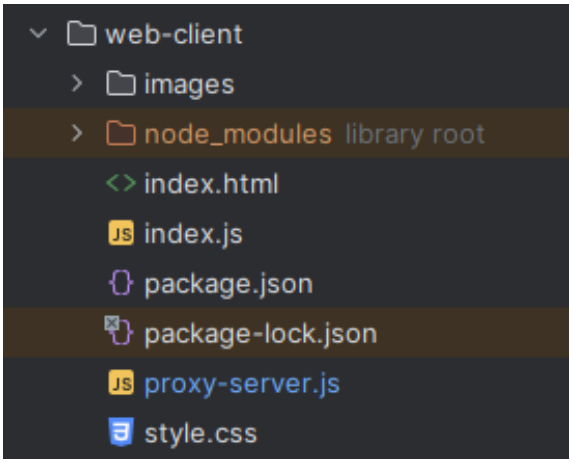


Рисунок 3.1.4 – Вміст папки веб-клієнта

Весь датасет для індексації знаходиться у папці data/, яка розташована всередині директорії сервера. Звідси програма зчитує файли для побудови інвертованого індексу (рис. 3.1.5). В свою чергу, тести для перевірки навантаження та стабільності роботи системи написані мовою python з використанням бібліотеки locust, і розташовані в директорії load-test/ (рис. 3.1.6):

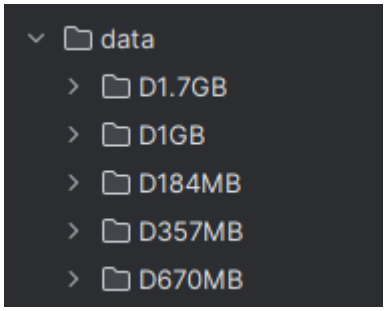


Рисунок 3.1.5 – Директорія з даними

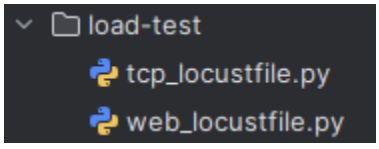


Рисунок 3.1.6 – Директорія з тестами

В результаті збірки проекту з допомогою CMake створюються готові до запуску файли сервера та консольного клієнта, а окремо через node.js запускається веб-клієнт.

В якості середовища розробки використовувалось CLion, а для реалізації веб-клієнта застосовано мову JavaScript та вище згадану платформу node.js.

3.2. Реалізація пулу потоків

В даній курсовій роботі пул потоків реалізований з допомогою класу ThreadPool. До його завдань входить керування робочими потоками, розподіл завдань з черги та управління станом потоків. Для забезпечення потокобезпечного доступу до спільних ресурсів та синхронізації використано примітив рід-райт лок та умовну змінну.

Одним із основних методів класу є Initialize(), який створює та запускає фіксовану кількість робочих потоків (воркерів). Оптимальною кількістю потоків вважається та, що відповідає кількості ядер процесора. Кожен потік одразу переходить до виконання методу Routine(). Наступний метод AddTask() відповідає за додавання нових завдань. Він перевіряє, чи активний пул, додає задачу в чергу через метод Emplace і сповіщає один із очікуючих потоків.

В свою чергу, методи Pause() та Resume() дозволяють тимчасово призупинити або відновити обробку черги. Воркери перевіряють флаг паузи на кожній ітерації циклу і якщо він встановлений переходять в режим очікування.

Для завершення роботи потоків у пулі реалізовано методи Terminate() і TerminateNow(). Метод Terminate() викликає ShutdownQueue(), чекає виконання всіх наявних у черзі завдань і лише тоді приєднує потоки. Натомість TerminateNow() моментально припиняє роботу пулу потоків, встановлюючи флаг завершення і негайної зупинки, що власне й дозволяє миттєво зупинити виконання активних завдань.

Взагалом, механізм роботи самого полу полягає в тому, що робочі потоки постійно знаходяться в безкінечному циклі, і коли черга пуста, вони не тратять ресурси процесора, а переходять в режим сну і чекають на сигнал. Як тільки

надходить нове завдання вільний потік прокидається, забирає його з черги і одразу переходить до виконання.

Черга для пулу реалізована за принципом FIFO (first in, first out). Для захисту від стану гонки при додаванні та видаленні елементів використовується блокування на запис, щоб забезпечити цілісність даних при доступі з багатьох потоків одночасно.

Реалізація власного пулу потоків дозволяє вирішити проблему повязану зі значними витратами на створення та знищення потоків для кожного окремого завдання, а обмеження кількості одночасно працюючих потоків запобігає перевантаженню операційної системи.

3.3. Реалізація інвертованого індексу

Інвертований індекс реалізовано через клас InvertedIndex, який відповідає за побудову бази даних слів та їх розташування у документах, а також за виконання пошукових запитів. Процес побудови індексу реалізований у методі BuildIndex(), і в ньому враховано те, що система повинна динамічно оновлюватись. Для цього в методі є спеціальний лічильник вже проіндексованих файлів, тому кожен раз при запуску методу будуть оброблятись тільки нові документи, що були додані в систему, не витрачаючи ресурси на повторну обробку раніше доданих до індексу файлів. Для синхронізації завершення роботи всіх потоків використовується атомарна змінна, яка підраховує кількість активних завдань. Поки ця змінна більша за нуль, головний потік переходить у режим очікування, таким чином не створюється зайве навантаження на процесор.

Сама структура індексу зберігає інформацію не лише про те, в якому документі зустрічається слово, але й де саме воно знаходиться. Для цього використовується структура WordPosition, яка зберігає індекс слова та

зміщення символу. І якщо для пошуку слова потрібен його індекс, то для того, щоб показати уривок із знайденою фразою нам потрібне зміщення. Тобто серверу не треба заново сканувати файл, щоб знайти потрібне слово, і він зразу переходить до збереженої позиції зміщення і вирізає необхідний фрагмент тексту.

Під час токенізації текст розбивається на слова, очищується від зайвих символів та приводиться до нижнього регістру. Щоб мінімізувати час простою потоків через очікування доступу до спільного ресурсу, кожен потік спочатку формує свій локальний індекс для конкретного файлу без блокування, і вже коли треба зливати дані з глобальним індексом потік на короткий час захоплює м’ютекс на запис. Такий підхід підвищує швидкість при багатопотоковій обробці.

Оскільки обсяг даних доволі великий, в алгоритмі пошуку замість послідовного перебору всіх файлів використано пошук за найрідкіснішим словом. Спочатку серед слів пошукового запиту користувача визначається те слово, яке зустрічається в найменшій кількості документів. Тоді пошук вже виконується тільки по цьому обмеженому набору файлів, що значно зменшує час виконання запиту. Це особливо ефективно для фраз, які містять часто вживані слова, оскільки вони ігноруються якщо у фразі є більш унікальні терміни.

І насамкінець для перевірки того, чи слова в знайдених документах дійсно йдуть підряд і утворюють шукану фразу, їх позиції потрібно проаналізувати. Оскільки позиції слів в індексі зберігаються у відсортованому вигляді, замість лінійного проходу для перевірки наявності наступного слова на потрібній позиції використано алгоритм бінарного пошуку. З допомогою даного алгоритму можна знайти потрібну позицію або пересвідчитись в її

відсутності, що буде працювати значно швидше навіть тоді, коли одне слово зустрічається в документі багато разів.

3.4. Реалізація клієнт-серверної взаємодії

За взаємодію в системі відповідає клас Server, який керує всіма процесами. Після запуску він ініціалізує сокет і переходить в режим прослуховування порту. Для роботи з клієнтами використовується пул потоків, і кожен клієнт обслуговується потоком з цього пулу, що дозволяє не створювати новий потік для кожного нового запиту.

Особливістю серверної реалізації є наявність механізму керування навантаженням. Перед створенням сесії для нового користувача система перевіряє лічильник активних клієнтів, тобто тих, які вже взаємодіють з сервером. Якщо ліміт не перевищено, завдання на обробку клієнта додається до клієнтського пулу потоків. Якщо сервер перевантажений (кількість активних клієнтів досягла максимуму), то запит не відхиляється, а формується у вигляді завдання і додається до черги очікування. Клієнт при цьому отримує статус SERVER_BUSY, щоб він розумів причину очікування. Як тільки один із воркерів пулу звільняється, сервер одразу витягує завдання з черги очікування та передає його на виконання в пул.

Обмін даними реалізовано через потоковий протокол TCP. Щоб повідомлення від клієнта не зливалися в один потік, читання даних з сокета робиться побайтово до символу переносу рядка, таким чином кожне повідомлення коректно розпізнається окремо.

В свою чергу, клієнтська частина системи реалізована у двох варіантах:

- Консольний клієнт, який реалізований з використанням мови програмування C++ та системної бібліотеки Winsock2. Він працює у

режимі командного рядка, дозволяє відправляти пошукові запити та отримувати текстові відповіді.

- Веб-клієнт, який реалізований з використанням HTML, JavaScript та середовища виконання node.js. Через те, що в браузері не підтримуються TCP-сокети, було реалізовано проміжний проксі сервер. Він приймає HTTP-запити від веб-сторінки, транслює їх у TCP-пакети для сервера, отримує відповідь і повертає її браузеру у форматі JSON.

РОЗДІЛ 4. РЕЗУЛЬТАТИ ВИКОНАННЯ ПРОГРАМИ

4.1. Робота консольного клієнта

Розглянемо детальніше роботу клієнт-серверного застосунку. Для цього спочатку запускаємо сервер. Для наочності в консоль виводиться інформація про ініціалізацію пулу потоків для обробки даних, а файловий менеджер зчитує всі наявні в датасеті файли, після чого розпочинається процес побудови інвертованого індексу.

Варто зазначити, що підключення клієнтів дозволяється тільки після завершення індексації. Хоча звичайно можна дозволити доступ раніше, але це призведе до того, що користувач при повторному пошуку тої самої фрази отримував би різну кількість результатів через те, що індекс ще заповнюється. Це не дуже критично, але все ж для коректних результатів було вирішено зробити, щоб сервер переходив у режим очікування з’єднань тільки після повної побудови індексу.

Як тільки побудова індексу завершується, сервер виводить повідомлення про затрачений на побудову час та готовність приймати зєднання (рис. 4.1.1):

```
PS C:\Users\vlady> cd C:\Users\vlady\Coursework_Petryk_PO\cmake-build-release
PS C:\Users\vlady\Coursework_Petryk_PO\cmake-build-release> .\Server.exe
ThreadPool: Initialized with 4 threads.
FileManager: Found 7625 new files. Total: 7625
Building inverted index...
InvertedIndex: Index built. Indexed 9762288 words in 120633 ms.
Server started listening on port 8080

Press Enter to stop the server.
```

Рисунок 4.1.1 – Запуск сервера та побудова індексу

Після цього запускаємо консольного клієнта та виконуємо пошуковий запит (рис. 4.1.2):

```
PS C:\Users\vlady> cd C:\Users\vlady\Coursework_Petryk_PO\cmake-build-release
PS C:\Users\vlady\Coursework_Petryk_PO\cmake-build-release> .\Client.exe
Connected to Search Server successfully.

Enter query (or 'quit' to exit): one of the
```

Рисунок 4.1.2 – Виконання пошукового запиту

В результаті отримуємо список проіндексованих файлів, відсортованих за кількістю збігів шуканої фрази (рис. 4.1.3):

```
Enter query (or 'quit' to exit): one of the
Found 6672 results:
  [0] C:/Users/vlady/Coursework_Petryk_PO/server/data\D1.7GB\200.txt | matches=595
  [1] C:/Users/vlady/Coursework_Petryk_PO/server/data\D184MB\200.txt | matches=595
  [2] C:/Users/vlady/Coursework_Petryk_PO/server/data\D1GB\200.txt | matches=595
  [3] C:/Users/vlady/Coursework_Petryk_PO/server/data\D357MB\200.txt | matches=595
  [4] C:/Users/vlady/Coursework_Petryk_PO/server/data\D670MB\200.txt | matches=595
  [5] C:/Users/vlady/Coursework_Petryk_PO/server/data\D1.7GB\3252.txt | matches=562
  [6] C:/Users/vlady/Coursework_Petryk_PO/server/data\D1.7GB\3400.txt | matches=502
  [7] C:/Users/vlady/Coursework_Petryk_PO/server/data\D1.7GB\3136.txt | matches=499
  [8] C:/Users/vlady/Coursework_Petryk_PO/server/data\D1.7GB\2981.txt | matches=269
  [9] C:/Users/vlady/Coursework_Petryk_PO/server/data\D1.7GB\2760.txt | matches=252
  [10] C:/Users/vlady/Coursework_Petryk_PO/server/data\D1.7GB\1340.txt | matches=237
  [11] C:/Users/vlady/Coursework_Petryk_PO/server/data\D1GB\1340.txt | matches=237
  [12] C:/Users/vlady/Coursework_Petryk_PO/server/data\D670MB\1340.txt | matches=237
  [13] C:/Users/vlady/Coursework_Petryk_PO/server/data\D1.7GB\3350.txt | matches=235
  [14] C:/Users/vlady/Coursework_Petryk_PO/server/data\D1.7GB\2045.txt | matches=234
  [15] C:/Users/vlady/Coursework_Petryk_PO/server/data\D1GB\2045.txt | matches=234
  [16] C:/Users/vlady/Coursework_Petryk_PO/server/data\D1.7GB\665.txt | matches=233
  [17] C:/Users/vlady/Coursework_Petryk_PO/server/data\D1GB\665.txt | matches=233
  [18] C:/Users/vlady/Coursework_Petryk_PO/server/data\D357MB\665.txt | matches=233
  [19] C:/Users/vlady/Coursework_Petryk_PO/server/data\D670MB\665.txt | matches=233
  [20] C:/Users/vlady/Coursework_Petryk_PO/server/data\D1.7GB\505.txt | matches=231
  [21] C:/Users/vlady/Coursework_Petryk_PO/server/data\D1GB\505.txt | matches=231
  [22] C:/Users/vlady/Coursework_Petryk_PO/server/data\D357MB\505.txt | matches=231
  [23] C:/Users/vlady/Coursework_Petryk_PO/server/data\D670MB\505.txt | matches=231
  [24] C:/Users/vlady/Coursework_Petryk_PO/server/data\D1.7GB\1370.txt | matches=207
  [25] C:/Users/vlady/Coursework_Petryk_PO/server/data\D1GB\1370.txt | matches=207
  [26] C:/Users/vlady/Coursework_Petryk_PO/server/data\D670MB\1370.txt | matches=207
  [27] C:/Users/vlady/Coursework_Petryk_PO/server/data\D1.7GB\820.txt | matches=206
  [28] C:/Users/vlady/Coursework_Petryk_PO/server/data\D1GB\820.txt | matches=206
  [29] C:/Users/vlady/Coursework_Petryk_PO/server/data\D357MB\820.txt | matches=206
  [30] C:/Users/vlady/Coursework_Petryk_PO/server/data\D670MB\820.txt | matches=206
  [31] C:/Users/vlady/Coursework_Petryk_PO/server/data\D1.7GB\3331.txt | matches=167
```

Рисунок 4.1.3 – Результати пошуку

Тим часом на стороні сервера виводиться інформація про нове підключення з IP-адресою та портом клієнта, а також час, витрачений на виконання пошуку (рис. 4.1.4):

```
Client connected: IP - 127.0.0.1, port - 49310 (Active: 1)
Search: "one of the" took 5504 ms
```

Рисунок 4.1.4 – Час витрачений на пошук фрази

Щоб перевірити коректність роботи, оберемо один із знайдених файлів за його індексом та переглянемо уривки тексту з шуканою фразою (рис. 4.1.5):

```
Enter a file index to view snippets, or 'q' to go back: 4231

--- Snippets ---
the subject disgusted me. It has proved one of the greatest evils in my life that I was no
---------------
d and thought that I had killed a bird, one of the two acted as if loading his gun, and cr
---------------
fterwards I received the proceedings of one of the meetings, in which it seemed that the s
---------------
the subject of a public discussion, and one of the speakers declared that I had the bump o
---------------
n, it was necessary that I should go to one of the English universities and take a degree
---------------
nicate my discoveries. Dr. Whewell was one of the older and distinguished men who sometim
---------------
about Teneriffe, and read them aloud on one of the above-mentioned excursions, to (I think
---------------
My father always maintained that he was one of the most sensible men in the world, and he
---------------
 His character was in several respects one of the most noble which I have ever known. Th
---------------
ent a little into society, and acted as one of the honorary secretaries of the Geological
---------------
```

Рисунок 4.1.5 – Перегляд уривків тексту із знайденого файлу

З отриманих результатів видно, що фраза дійсно присутня у текстовому файлі, а отже алгоритм пошуку та побудови фрагментів тексту працює правильно.

Тепер розглянемо ситуацію, коли кількість активних клієнтів досягла максимуму. В цьому випадку сервер не розриває з'єднання, а додає нового користувача до черги очікування. Як бачимо з рисунку 4.1.6, клієнт отримує

відповідне повідомлення від сервера та переходить у режим очікування вільного слота:

```
PS C:\Users\vlady> cd C:\Users\vlady\Coursework_Petryk_PO\cmake-build-release
PS C:\Users\vlady\Coursework_Petryk_PO\cmake-build-release> .\Client.exe
You have been added to the queue. Please wait...
```

Рисунок 4.1.6 – Повідомлення про очікування в черзі

В свою чергу, на стороні сервера можна простежувати активність, а саме поточну кількість підключених користувачів, які взаємодіють з системою, а також розмір черги очікування (рис. 4.1.7):

```
Client connected: IP - 127.0.0.1, port - 59933 (Active: 1)
Client connected: IP - 127.0.0.1, port - 59934 (Active: 2)
Client connected: IP - 127.0.0.1, port - 51847 (Active: 3)
Client connected: IP - 127.0.0.1, port - 51849 (Active: 4)
Client added to queue (Queue size: 1)
```

Рис. 4.1.7 – Відображення статусу черги на сервері

Як тільки один з клієнтів завершує сеанс, сервер одразу повідомляє про це першого користувача з черги і врешті надає йому доступ до виконання пошукових запитів (рис. 4.1.8):

```
PS C:\Users\vlady> cd C:\Users\vlady\Coursework_Petryk_PO\cmake-build-release
PS C:\Users\vlady\Coursework_Petryk_PO\cmake-build-release> .\Client.exe
You have been added to the queue. Please wait...

A slot has become available. You are now connected.

Enter query (or 'quit' to exit):
```

Рисунок 4.1.8 – Надання доступу до сервера після очікування

Насамкінець перевіримо динамічне оновлення індексу, яке виконується кожну хвилину. Для цього перед запуском з датасету було видалено 1000 файлів, а вже після запуску сервера ці файли були повернуті назад в директорію під час роботи програми (рис. 4.1.9):

```
PS C:\Users\vlady> cd C:\Users\vlady\Coursework_Petryk_PO\cmake-build-release
PS C:\Users\vlady\Coursework_Petryk_PO\cmake-build-release> .\Server.exe
ThreadPool: Initialized with 4 threads.
FileManager: Found 6625 new files. Total: 6625
Building inverted index...
InvertedIndex: Index built. Indexed 9762288 words in 96819 ms.
Server started listening on port 8080

Press Enter to stop the server.


Scheduler: Started updating index.
FileManager: Found 1000 new files. Total: 7625
Scheduler: Index update complete. 7625 files indexed.
```

Рисунок 4.1.9 – Динамічне оновлення індексу

Як бачимо з отриманих результатів, все спрацювало коректно. Важливо зазначити, що система обробила тільки додані 1000 файлів і не витрачала ресурси на переіндексацію вже існуючих даних, якраз через лічильник в методі BulidIndex().

4.2. Робота веб-клієнта

Тепер розглянемо як працює веб-клієнт. Як було згадано раніше, браузер не може напряму підключатися до C++ сервера через TCP-сокети, тому використовується проміжний проксі-сервер.

Для початку роботи потрібно запустити C++ сервер, після чого в терміналі запускаємо вже проксі-сервер командою npm run proxy (рис. 4.2.1):

```
PS C:\Users\vlady>  cd C:\Users\vlady\Coursework_Petryk_PO\web-client
PS C:\Users\vlady\Coursework_Petryk_PO\web-client> npm run proxy

> web-client@1.0.0 proxy
> node proxy-server.js

Proxy listening on http://localhost:3000
Server: Connected to 127.0.0.1:8080
Server: Welcome to Search Server!
```

Рисунок 4.2.1 – Запуск проксі-сервера

В іншому терміналі запускаємо веб-інтерфейс командою npm start, що активує локальний сервер і щоб сторінка відкрилась в браузері (рис. 4.2.2):

```
PS C:\Users\vlady>  cd C:\Users\vlady\Coursework_Petryk_PO\web-client
PS C:\Users\vlady\Coursework_Petryk_PO\web-client> npm start

> web-client@1.0.0 start
> parcel index.html

Server running at http://localhost:1234
✨ Built in 6ms
```

Рисунок 4.2.2 – Запуск веб-інтерфейсу

Після цього відкриваємо головну саму сторінку веб-інтерфейсу, яка виглядає наступним чином (рис. 4.2.3):

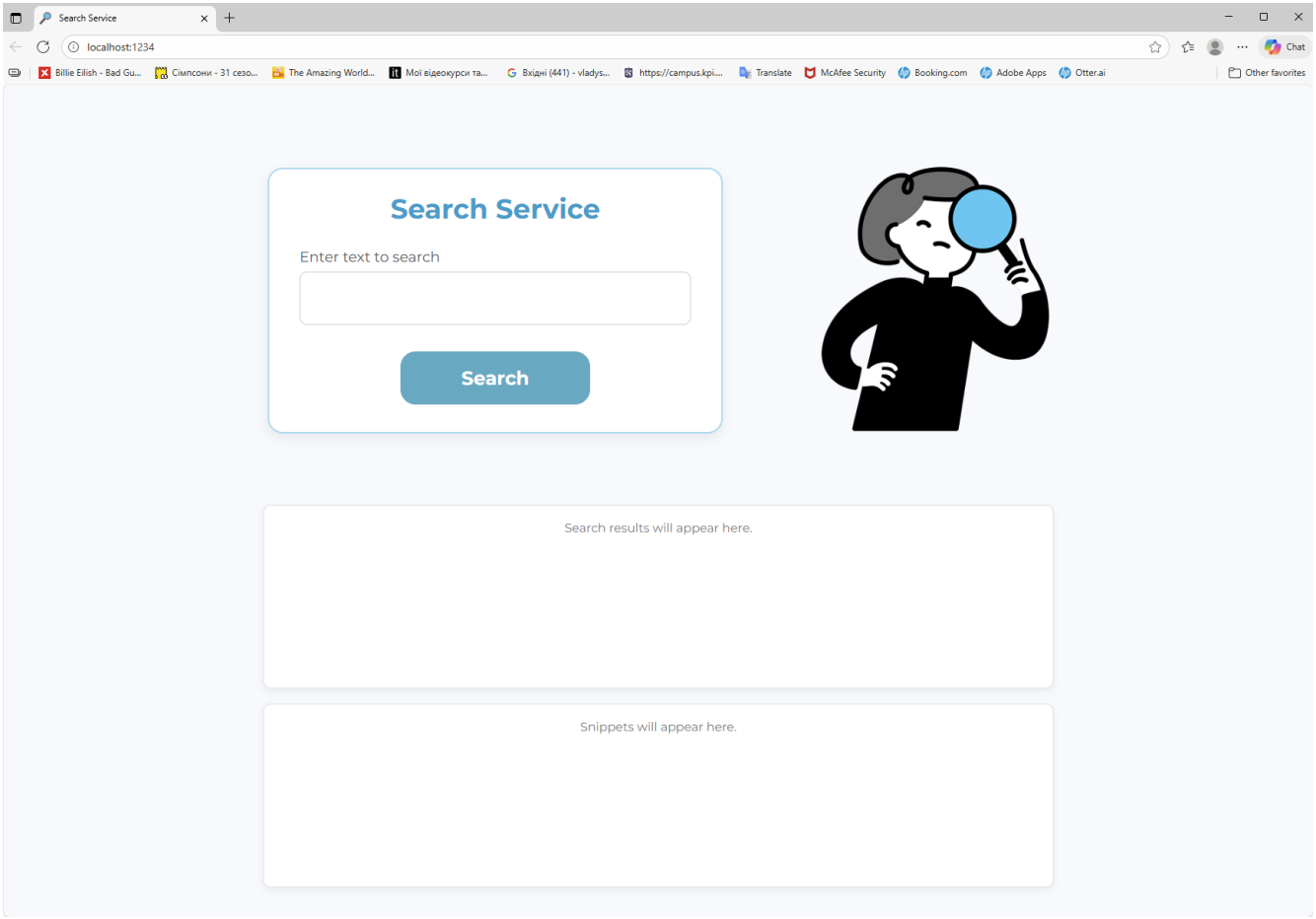


Рисунок 4.2.3 – Головна сторінка веб-клієнта

Спробуємо виконати пошуковий запит. Після введення фрази та натискання відповідної кнопки для початку пошуку, запит передається через проксі на сервер, там обробляється, і користувач отримує список знайдених документів у відповідному блоці для результатів (рис. 4.2.4):

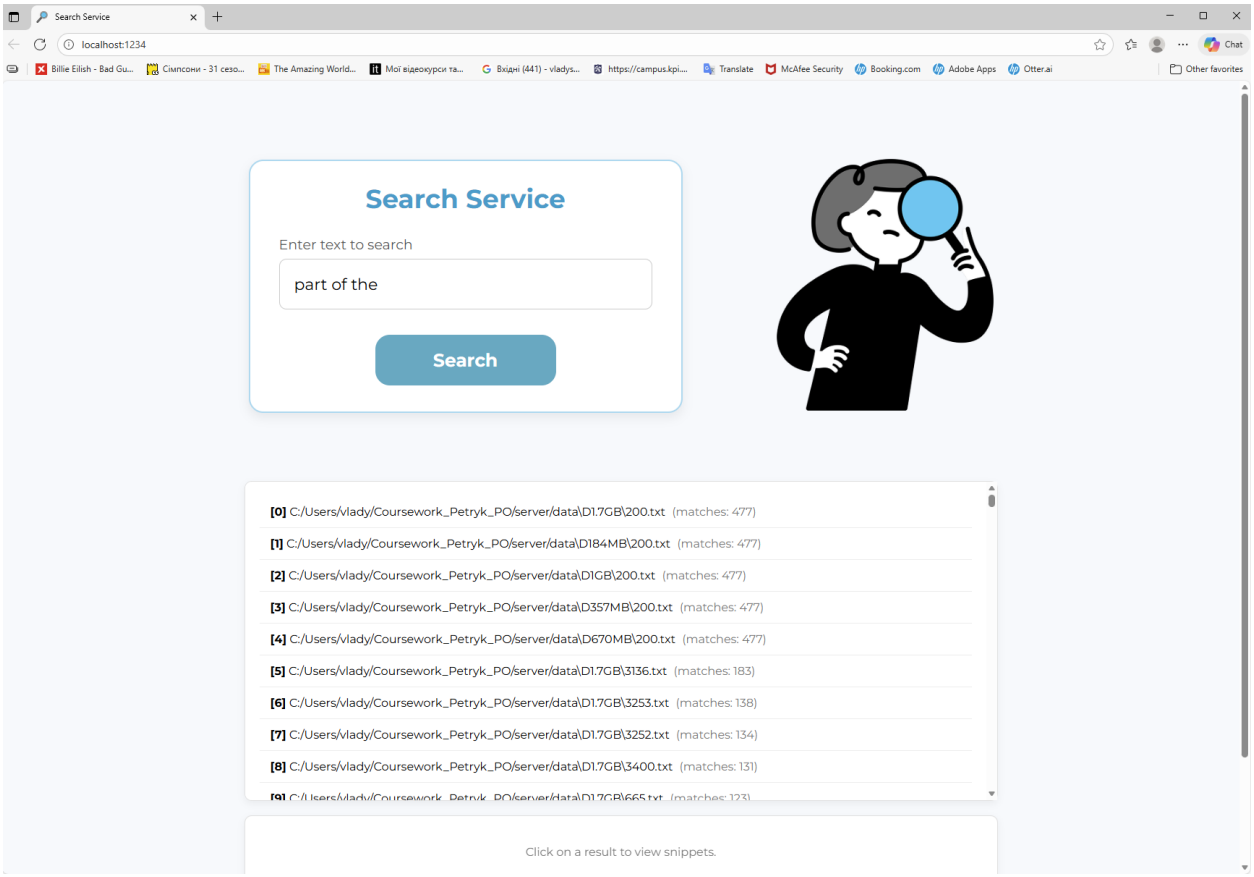


Рисунок 4.2.4 – Відображення результатів пошуку

Якщо натиснути на один із знайдених файлів зі списку, користувач побачить у блоці нижче фрагменти тексту з всіма входженнями шуканої фрази (рис. 4.2.5):

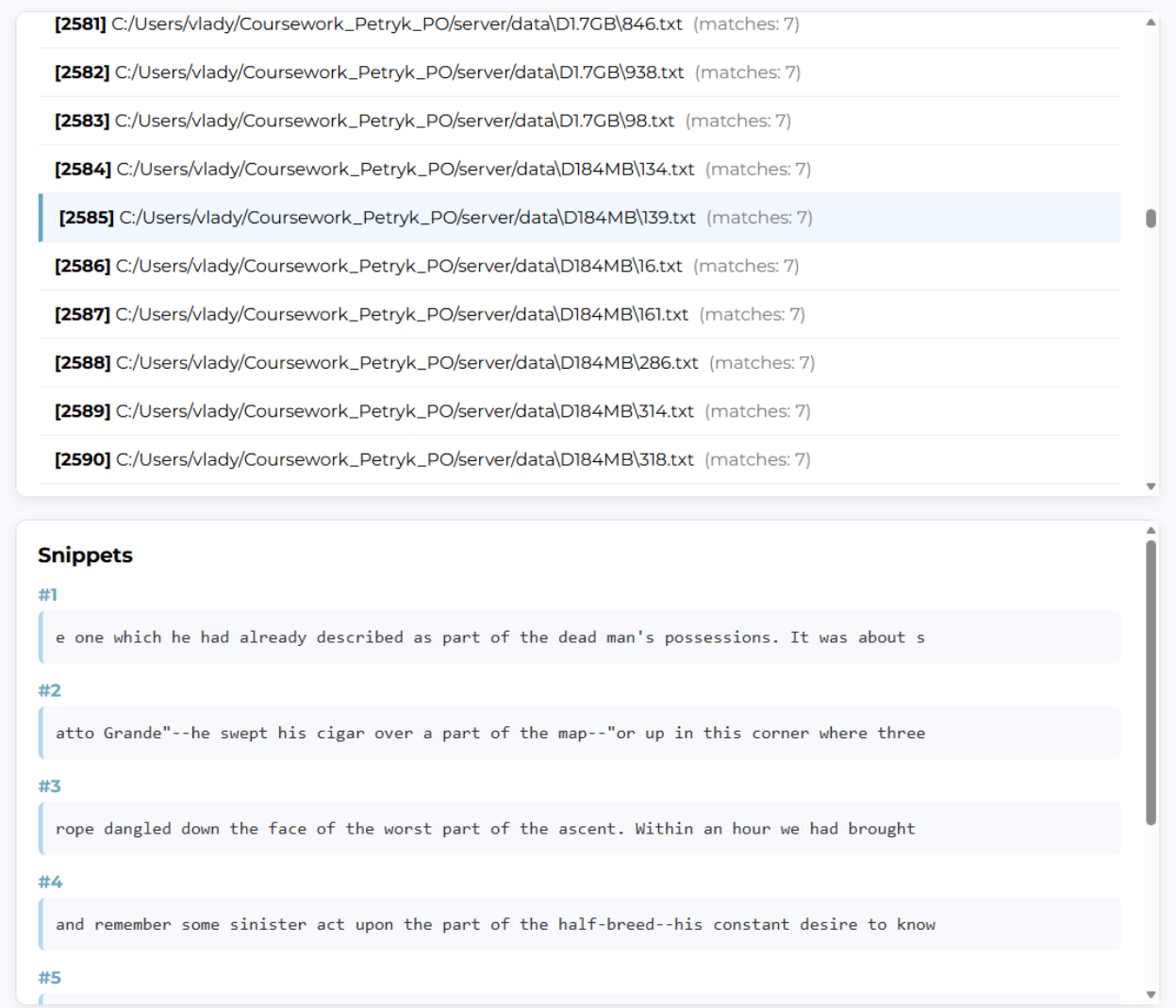


Рисунок 4.2.5 – Відображення уривків тексту з шуканою фразою

Аналогічно, якщо досягнуто ліміт підключень на сервері, для веб-клієнта так само як і для консольного відображається повідомлення, що потрібно почекати на вільний слот (рис. 4.2.6). Як тільки місце звільняється, інтерфейс автоматично оновлюється і користувач може виконати пошук фрази.

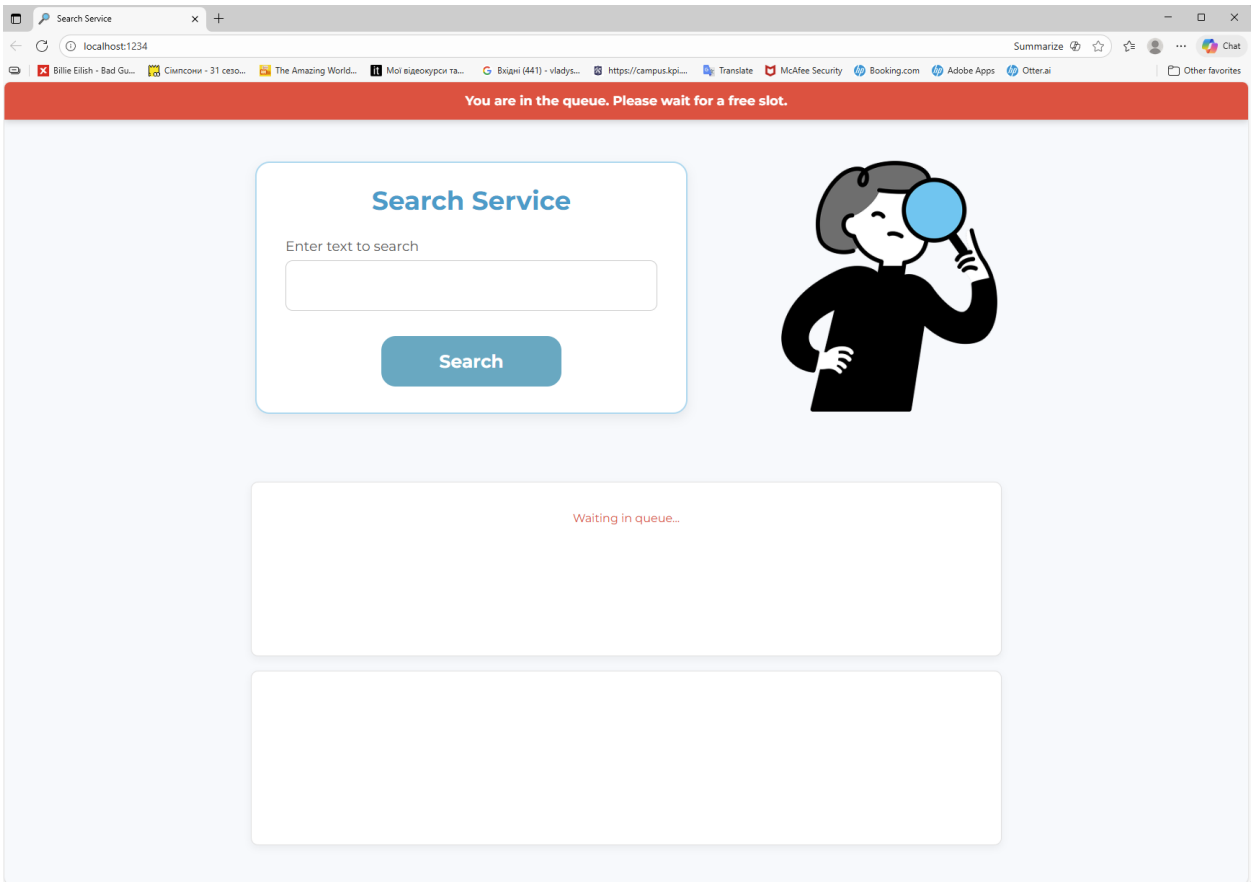


Рисунок 4.2.6 – Повідомлення про очікування вільного слота у веб-інтерфейсі

РОЗДІЛ 5. ТЕСТУВАННЯ

5.1. Навантажувальне тестування

Проведемо навантажувальне тестування, використавши для цього фреймворк Locust, у якому легко налаштовувати сценарії користувача. Для тестування консольного клієнта було реалізовано клас TcpClient мовою python, який імітує реальну поведінку користувача, а саме встановлює з'єднання, відправляє команду для пошуку фрази або отримання уривків у текстовому форматі та очікує відповідь від сервера.

Для початку спробуємо поставити ліміт активних клієнтів на сервері 200. Саме тестування проведемо на 200 користувачах з швидкістю додавання 10 юзерів на секунду (рис. 5.1.1):

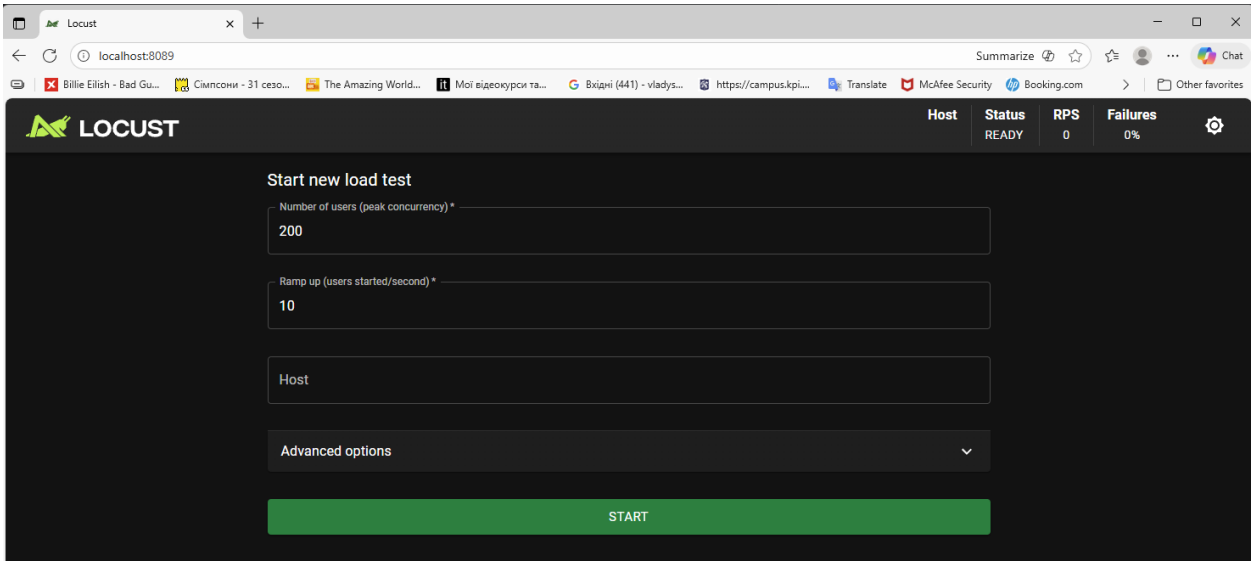


Рисунок 5.1.1 – Параметри навантаження

Отримуємо наступні результати (рис. 5.1.2), і так як кількість користувачів не перевищує ліміт на сервері, черга не утворюється.

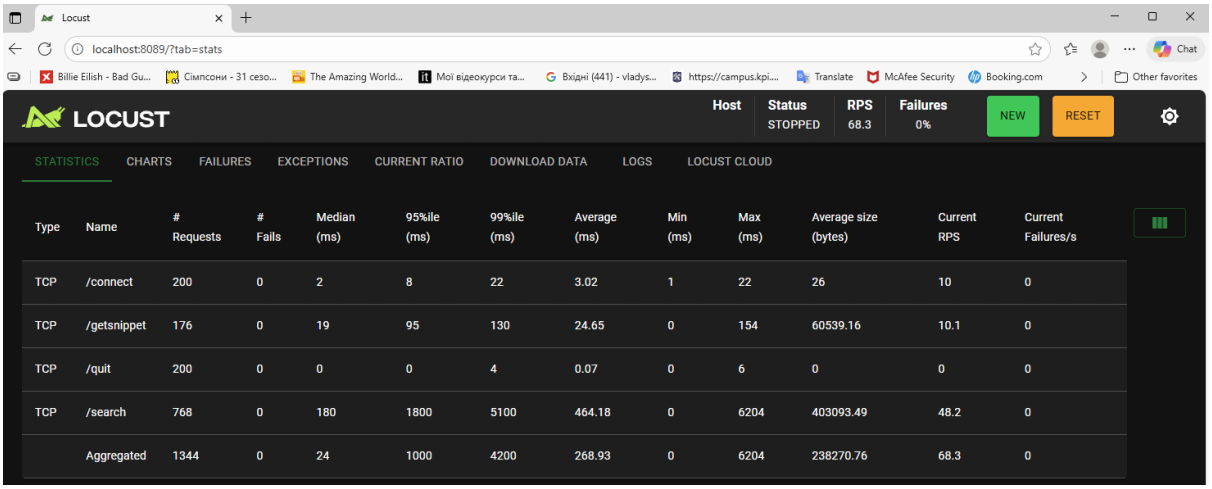


Рисунок 5.1.2 – Результати запитів при тестуванні сервера на 200 користувачах

Нескладно помітити, що сервер працює стабільно. Середня кількість запитів на секунду становить 68.3, тоді як середній час відповіді 24 мс, а отже запити обробляються майже миттєво і без затримок при такому помірному навантаженні.

Далі проведемо тестування на 500 користувачах з 20 новими юзерами на секунду, при цьому кількість активних клієнтів на сервері збільшимо до 250. У цьому випадку частина користувачів буде чекати вільного слота (рис. 5.1.3):

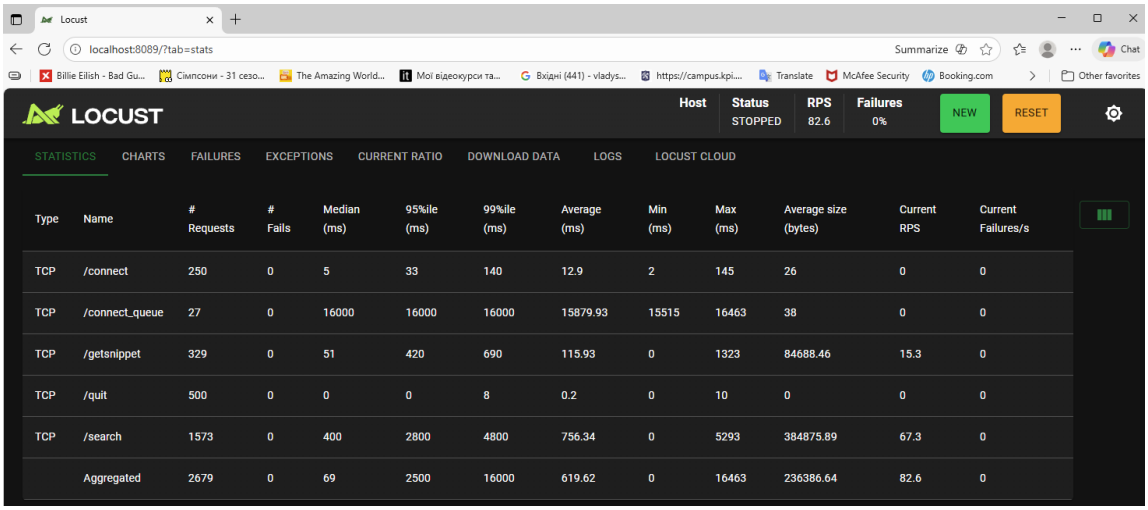
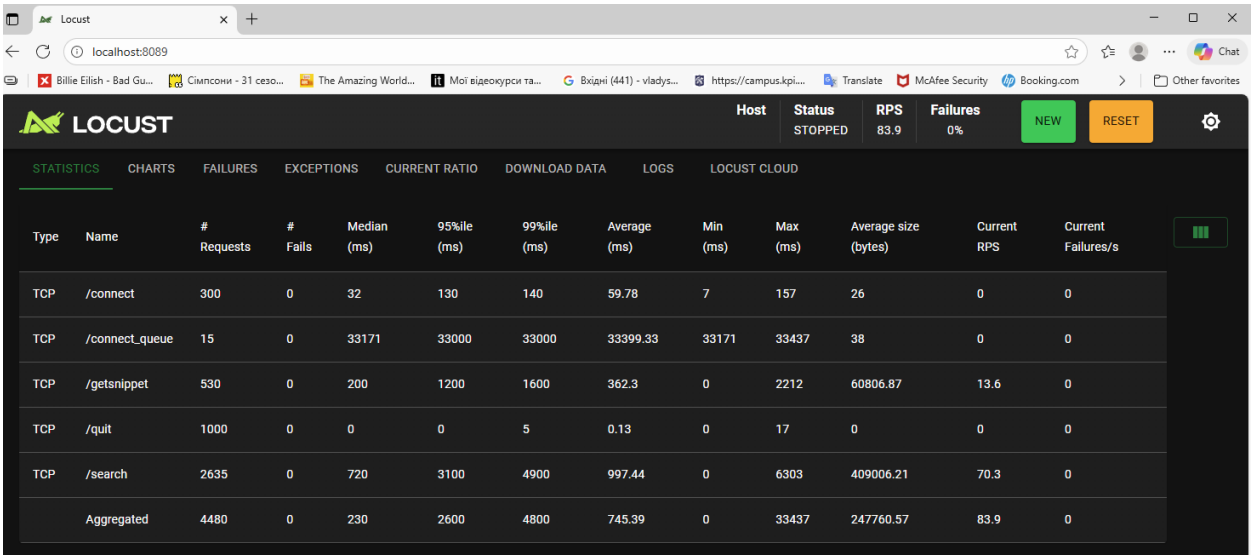


Рисунок 5.1.3 – Результати запитів при тестуванні сервера на 500 користувачах

Як бачимо, хоч при 500 користувачах деякі з клієнтів потрапляють в чергу, сервер все одно працює стабільно. Середня кількість запитів на секунду зросла лише до 82.6, а медіанний час відповіді складає 69 мс, що можна вважати непоганим показником. Найдовшим серед всіх запитів вийшов час обробки приєднання клієнта до черги. Можна пояснити це тим, що користувач, який не попадає в пул, чекає на вільний слот, а сервер тим часом надсилає йому повідомлення про статус в черзі. Тому фактично тривалість цього запиту якраз показує час очікування клієнта в черзі.

Насамкінець збільшимо кількість користувачів до 1000 з швидкістю додавання 50 нових за секунду, а ліміт користувачів поставимо 300 (рис. 5.1.4):



Type	Name	# Requests	# Fails	Median (ms)	95%ile (ms)	99%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS	Current Failures/s
TCP	/connect	300	0	32	130	140	59.78	7	157	26	0	0
TCP	/connect_queue	15	0	33171	33000	33000	33399.33	33171	33437	38	0	0
TCP	/getsnippet	530	0	200	1200	1600	362.3	0	2212	60806.87	13.6	0
TCP	/quit	1000	0	0	0	5	0.13	0	17	0	0	0
TCP	/search	2635	0	720	3100	4900	997.44	0	6303	409006.21	70.3	0
	Aggregated	4480	0	230	2600	4800	745.39	0	33437	247760.57	83.9	0

Рисунок 5.1.4 – Результати запитів при тестуванні сервера на 1000 користувачах

При такому навантаженні сервер все одно лишається стійким. Середня кількість запитів на секунду практично не зросла в порівнянні з попереднім результатом, а середній час відповіді становить 230 мс, що є хорошим результатом для великої кількості одночасних з’єднань і при цьому не виникає жодної помилки чи розриву з’єднання.

Для тестування веб-клієнта було реалізовано сценарій WebClientUser. Кожну секунду користувач надсилає пошуковий запит з рандомною фразою зі списку, отримує список результатів, і якщо відповідь успішна викликає отримання уривків тексту для першого знайденого файлу.

Протестуємо роботу системи на 200 користувачах зі швидкістю додавання 5 юзерів за секунду. Результат наведено на рис. 5.1.5:

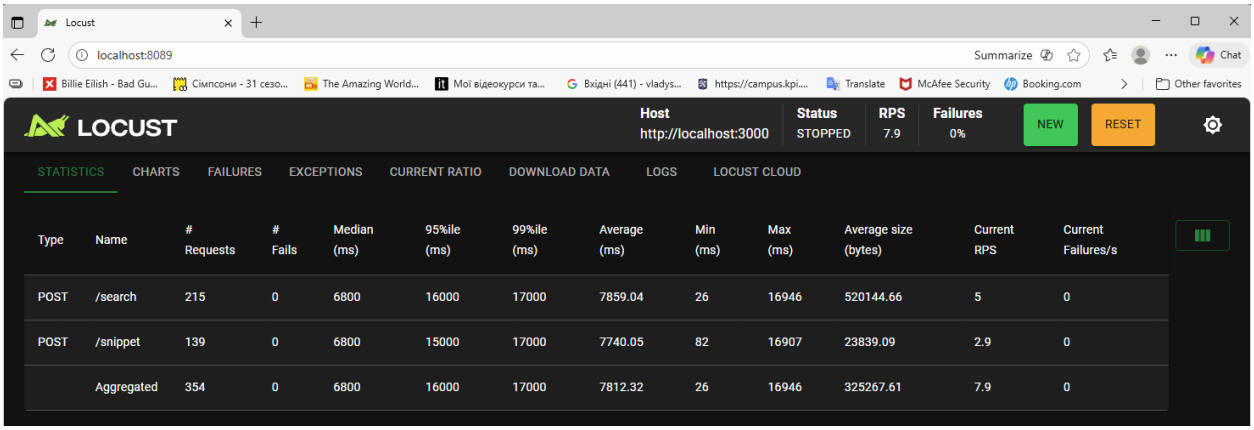


Рисунок 5.1.5 – Результати статистики навантажувального тестування веб-інтерфейсу для 200 користувачів

Зі статистики бачимо, що обидва ендпоінти обробляються без помилок. Для запиту на пошук середній час відповіді складає 6800 мс, а мінімальний та максимальний час 26 мс і 16,9 с відповідно. Схожі показники було отримано і для запиту на отримання уривків тексту із знайденою фразою. В свою чергу, середня кількість запитів на секунду при такому навантаженні становить близько 7,9. В загальному веб-інтерфейс коректно витримує задане навантаження без збоїв, а затримки у відповіді для запиту на пошук є очікуваними через накладні витрати на передачу великих обсягів текстових даних через проксі-сервер.

5.2. Аналіз залежності часу виконання від кількості потоків

Проаналізуємо, як кількість потоків впливає на швидкість побудови інвертованого індексу, результати наведено на наступних графіках (рис. 5.2.1 – 5.2.2):

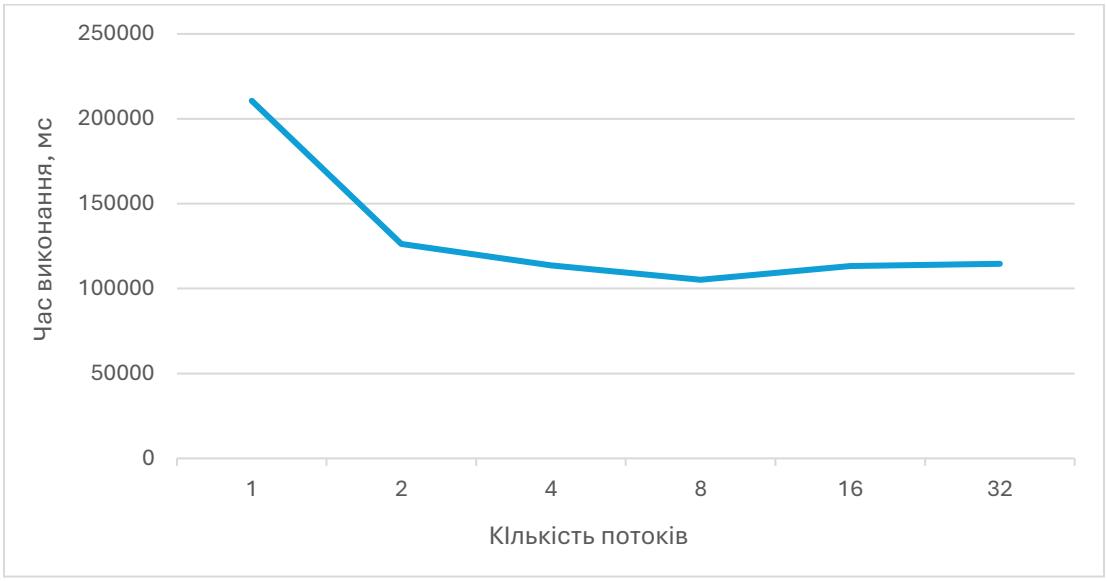


Рисунок 5.2.1 – Графік залежності часу побудови індексу від кількості потоків (1, 2, 4, 8, 16 та 32)

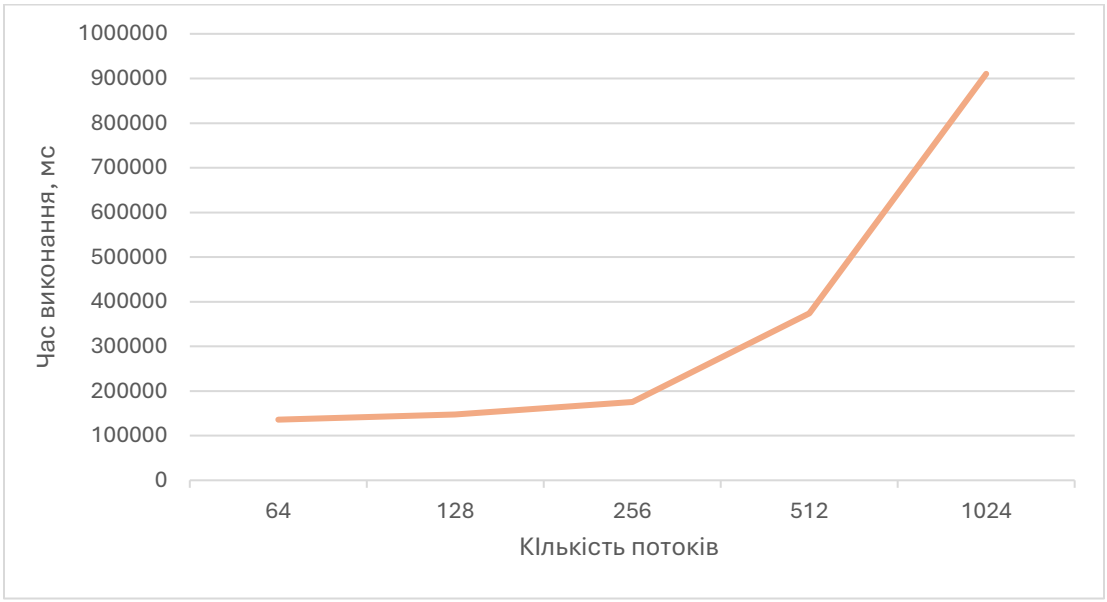


Рисунок 5.2.2 – Графік залежності часу побудови індексу від кількості потоків (64, 128, 256, 512 та 1024)

Як бачимо, при послідовній обробці час виконання є найдовшим і це логічно, оскільки з використанням одного потоку всі операції виконуються синхронно, та й ресурси багатоядерного процесора при цьому неефективно використовуються. В свою чергу, найкращий результат досягнуто при використанні 8 потоків, оскільки це відповідає кількості фізичних ядер процесора мого ПК. В такому випадку кожне ядро виконує свою задачу йому не треба ділити час з іншими потоками. При цьому 4 та 16 потоків для будування індексу також показали нормальну ефективність.

Проте вже на другому графіку помітно, що подальше збільшення кількості потоків не призводить до покращення результатів. При 1024 потоках час виконання різко зріс, так як надмірна кількість потоків створює значні накладні витрати на перемикання контексту операційною системою. Відповідно, процесор по більшості просто витрачає час на перемикання між тисячею потоків, що і призводить до падіння продуктивності.

Також проаналізуємо залежність часу побудови індексу від обсягу вхідних даних при різній кількості потоків. Результати зображено на графіку нижче (рис. 5.2.3):

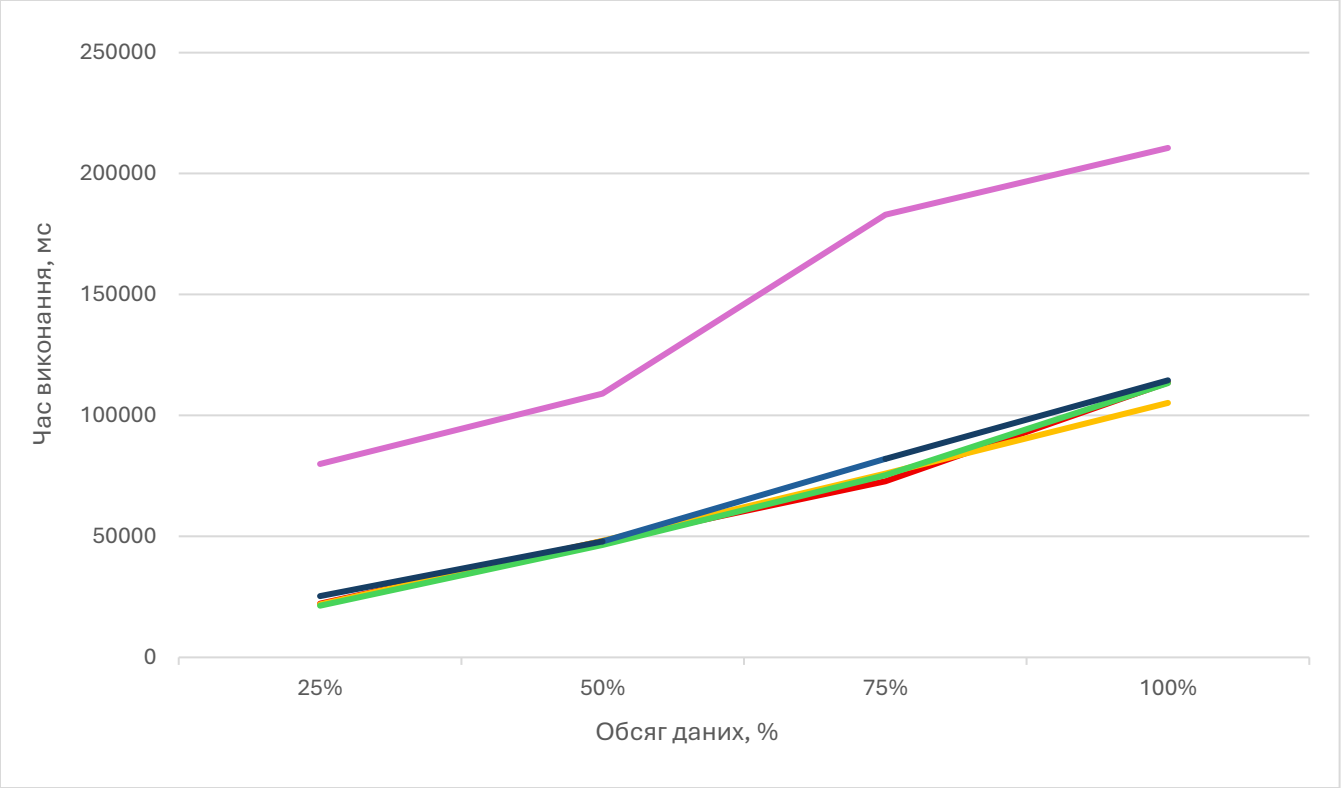


Рисунок 5.2.3 – Графік залежності часу побудови індексу від обсягу даних (рожева лінія – 1 потік, червона – 4 потоки, жовта – 8 потоків, зелена – 16 потоків, синя – 32 потоки)

З наведеного графіку видно, що час виконання збільшується разом із зростанням обсягу даних для всіх потоків. При цьому рожева лінія, яка відповідає одному потоку, знаходиться вище за інші, що знову доказує неефективність послідовної обробки великих масивів даних. В той же час лінії для решти потоків розташовані близько одна до одної. Пояснюється це тим, що після 4 – 8 потоків продуктивність залежить вже від швидкості читання файлів, а не від потужності процесора, тому подальше збільшення кількості потоків не дасть суттєвого приросту швидкості.

ВИСНОВОК

В результаті виконання курсової роботи було розроблено веб-сервіс для пошуку інформації у великих обсягах текстових файлів. Головним завданням було створити систему, яка зможе ефективно обробляти запити користувачів, використовуючи можливості сучасних багатоядерних процесорів. Для цього було обрано мову програмування C++.

Основу розробленої системи складає інвертований індекс. Це структура даних, яка дозволяє знаходити необхідні документи дуже швидко, не перебираючи їх вміст кожного разу заново. Процес побудови цього індексу був реалізований із використанням власного пулу потоків. Це означає, що файли зчитуються та обробляються паралельно кількома потоками, що значно пришвидшує час індексації порівняно зі звичайним послідовним підходом. Окрім цього, для уникнення конфліктів при записі даних у спільну пам'ять було застосовано примітиви синхронізації, зокрема рід-райт лок, що дозволяє багатьом потокам одночасно читати дані, але блокує доступ на момент запису.

Архітектура системи побудована за класичною моделлю “клієнт-сервер” з використанням мережевих сокетів. Це дало змогу реалізувати консольного клієнта а також веб-інтерфейс, який працює через проміжний проксі-сервер.

Було приділено окрему увагу стабільності роботи під навантаженням. Якщо кількість активних користувачів перевищує заданий ліміт, нові запити не відхиляються, а стають у чергу очікування. Як тільки звільняється місце, сервер автоматично бере в роботу наступного клієнта. Також реалізовано механізм динамічного оновлення даних, де окремий потік періодично перевіряє наявність нових файлів і додає їх до індексу без необхідності перезапуску всієї системи.

Насамкінець було проведено аналіз залежності часу побудови інвертованого індексу від кількості потоків, а також навантажувальне

тестування. Результати показали, що при використанні 8 потоків, що відповідає кількості ядер мого компютера, побудова інвертованого індексу займає найменше часу. В свою чергу, навантажувальні тести за допомогою платформи Locust показали, що сервер здатен стабільно обробляти велику кількість одночасних підключень, коректно обслуговує чергу клієнтів та забезпечує швидкий час відгуку.

Отже, було створено надійний, швидкий та масштабований сервіс пошуку, який ефективно використовує ресурси комп'ютера та забезпечує зручну взаємодію з користувачем.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Manning C. D., Raghavan P., Schütze H. Introduction to Information Retrieval. Cambridge : Cambridge University Press, 2008. 506 p. URL: http://diglib.globalcollege.edu.et:8080/xmlui/bitstream/handle/123456789/1096/Manning_introduction_to_information_retrieval.pdf?sequence=1&isAllowed=y.
2. Williams A. C++ Concurrency in Action. Practical Multithreading. Shelter Island : Manning Publications, 2012. 528 p. URL: https://www.bogotobogo.com/cplusplus/files/CplusplusConcurrencyInAction_PracticalMultithreading.pdf.
3. Kurose J. F., Ross K. W. Computer Networking: A Top-Down Approach. 8th ed. Harlow : Pearson, 2021. 794 p.
4. Fowler M. UML Distilled: A Brief Guide to the Standard Object Modeling Language. 3rd ed. Boston : Addison-Wesley, 2004. 208 p.
5. Parcel Documentation. URL: https://parceljs.org/docs/.
6. Gamma E., Helm R., Johnson R., Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software. Reading : Addison-Wesley, 1994. 395 p.
7. Locust Documentation. URL: https://docs.locust.io/en/stable/.
8. Tanenbaum A. S., Bos H. Modern Operating Systems. 4th ed. Upper Saddle River : Pearson, 2015. 1136 p.

ДОДАТОК А

Лістинг коду:

Посилання на репозиторій у Github.