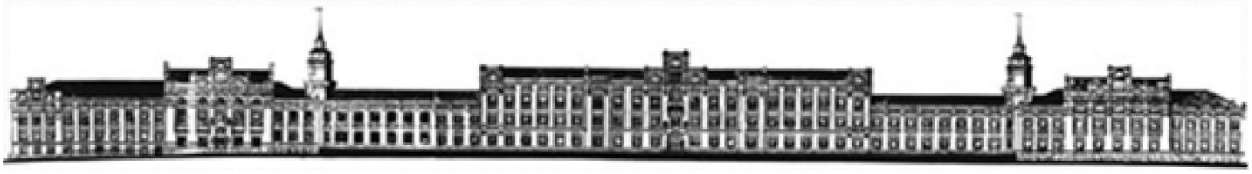

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«Київський політехнічний інститут ім. І. Сікорського»
Інститут Прикладного Системного Аналізу
Кафедра Системного Проектування

ПОЯСНЮВАЛЬНА ЗАПИСКА

про виконання курсової роботи

з дисципліни «Паралельні обчислення»

Виконав:
студент IV курсу, ДА-21
Терещенко Олексій Ігорович
Прийняв:
асистент Яременко В. С.

Київ – 2025

ЗМІСТ

1. ВСТУП

У межах курсової роботи реалізовано клієнт-серверну систему для побудови та використання інвертованого індексу для пошуку в текстових файлах за ключовими словами. Інвертований індекс дозволяє швидко знаходити документи, що містять задані токени, та є типовою основою для пошукових систем.

Актуальність роботи зумовлена необхідністю обробки великих наборів даних із використанням паралельних обчислень, забезпеченням коректної синхронізації спільних ресурсів та дослідженням впливу кількості потоків на продуктивність.

Мета роботи: розробити багатопоточний веб-сервер для побудови інвертованого індексу з текстових файлів із підтримкою пошуку, регулярного (динамічного) оновлення індексу та навантажувального тестування.

2. ПРОЕКТУВАННЯ СИСТЕМИ

2.1. Use Case діаграма

Опишемо use case діаграмою основні потреби клієнта, що треба буде врахувати:

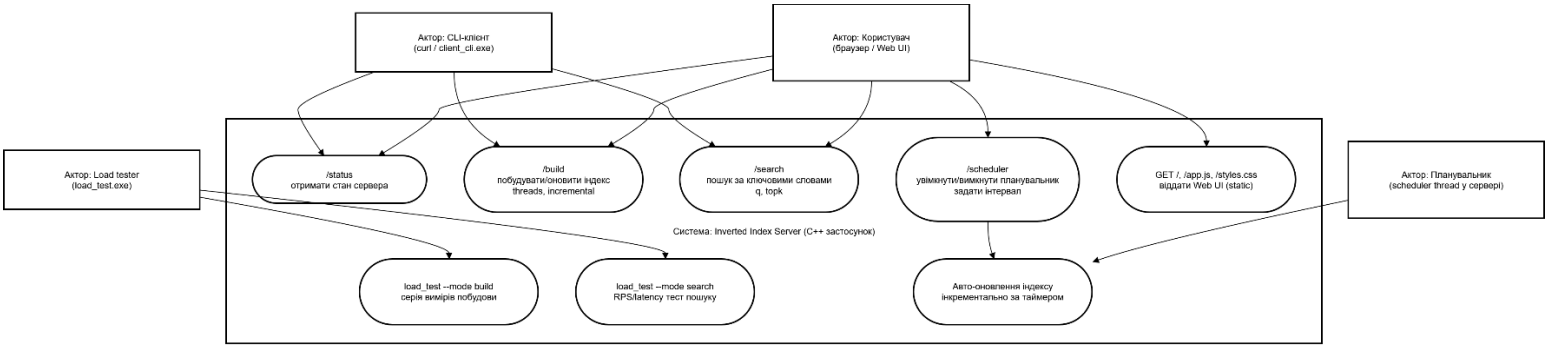

Рисунок 1 — Use Case діаграма системи.

2.2. Class діаграма

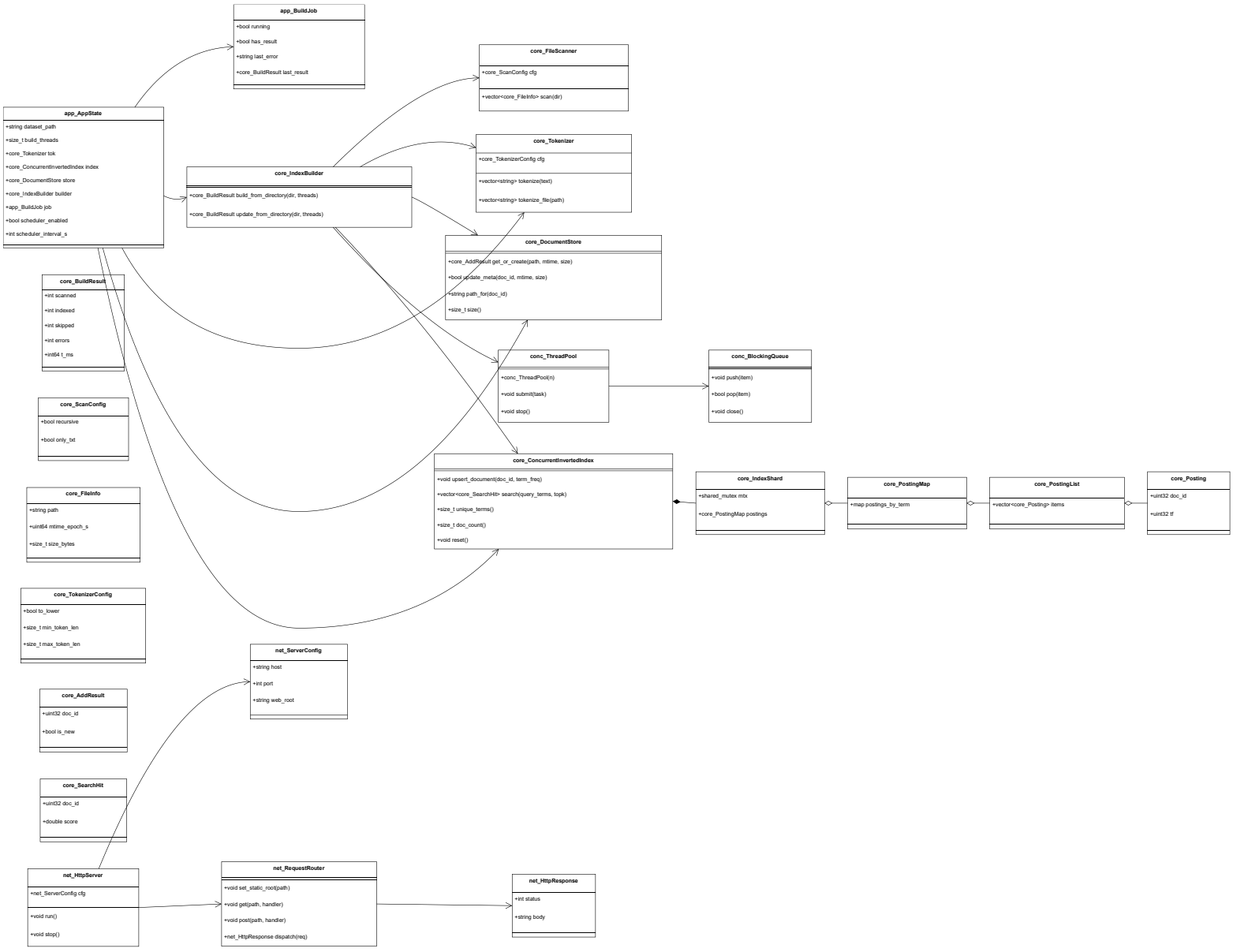

Рисунок 2 — Class діаграма (основні класи).

2.3 Sequence діаграми

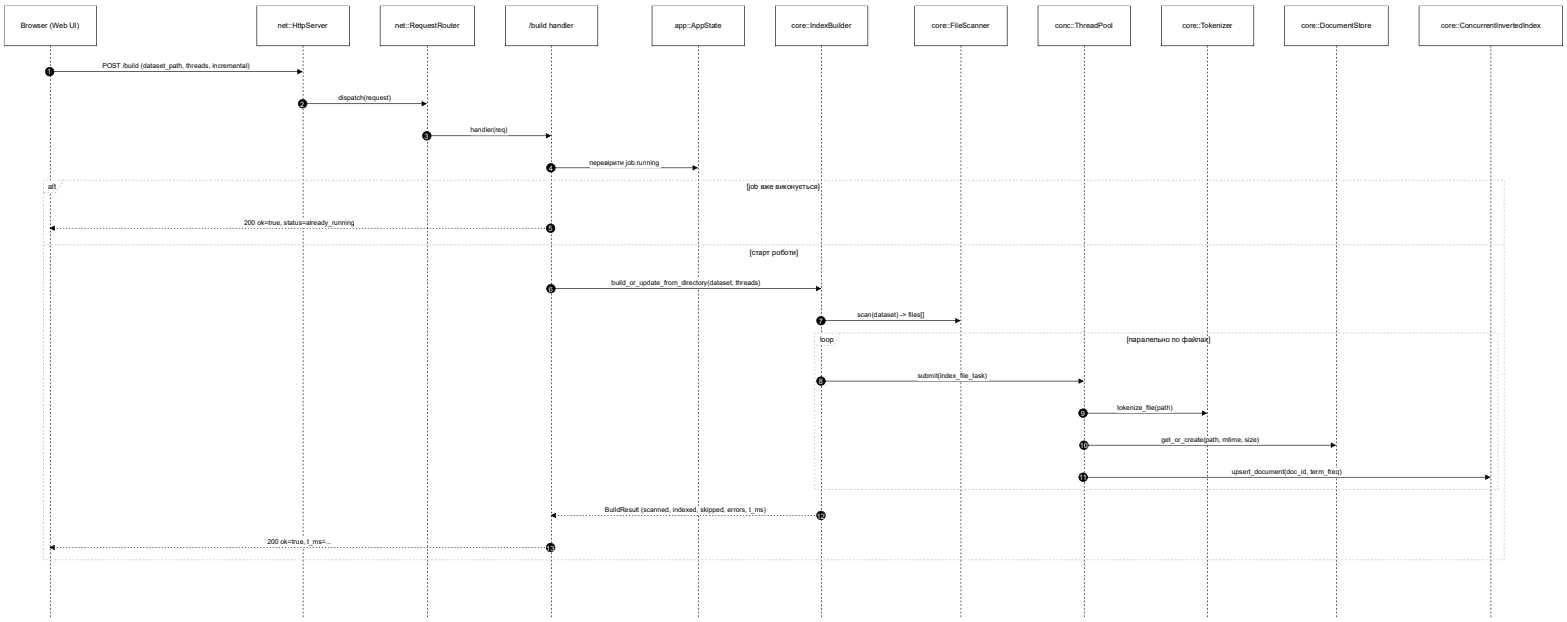

Рисунок 3 — Sequence діаграма Build/Update Index.

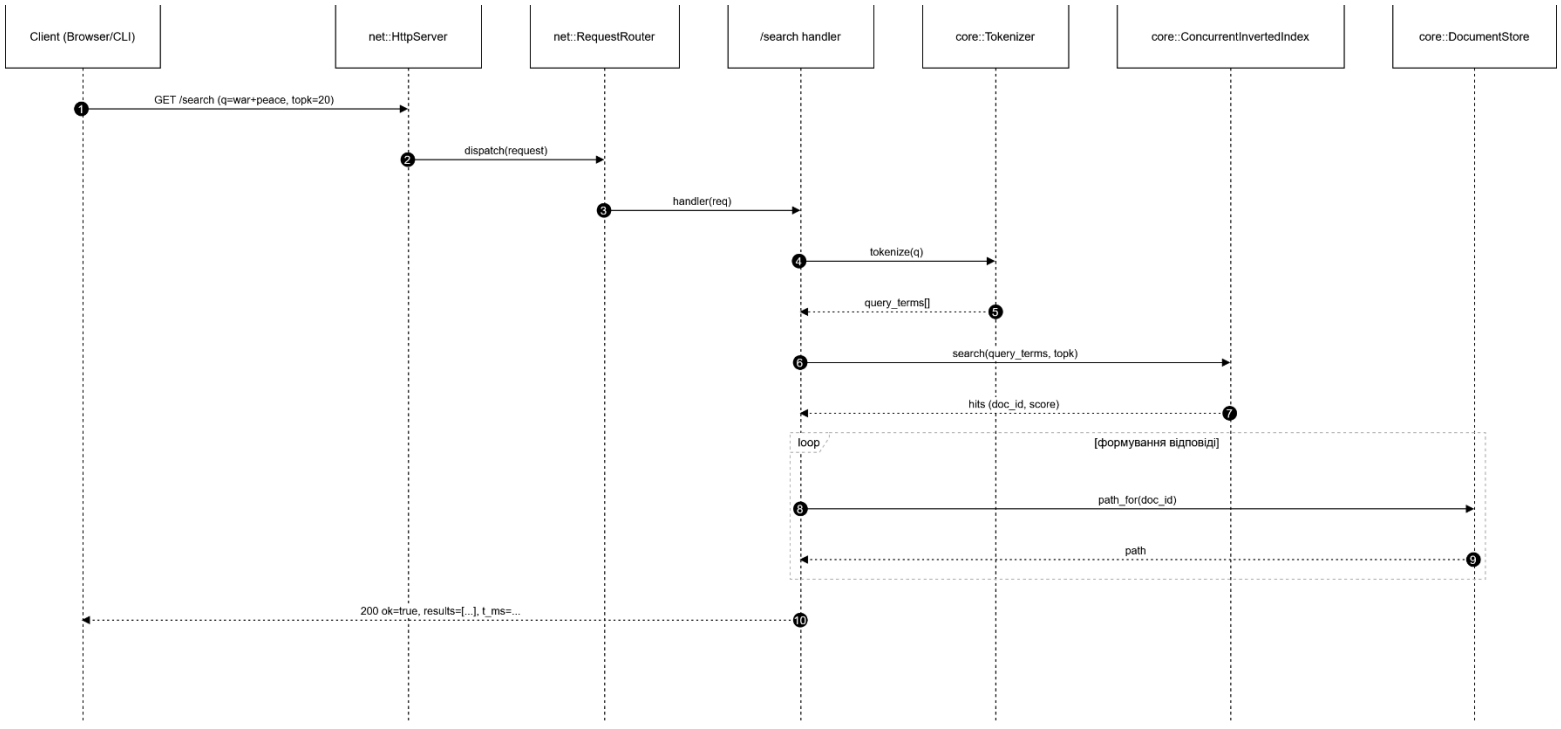

Рисунок 4 — Sequence діаграма Search.

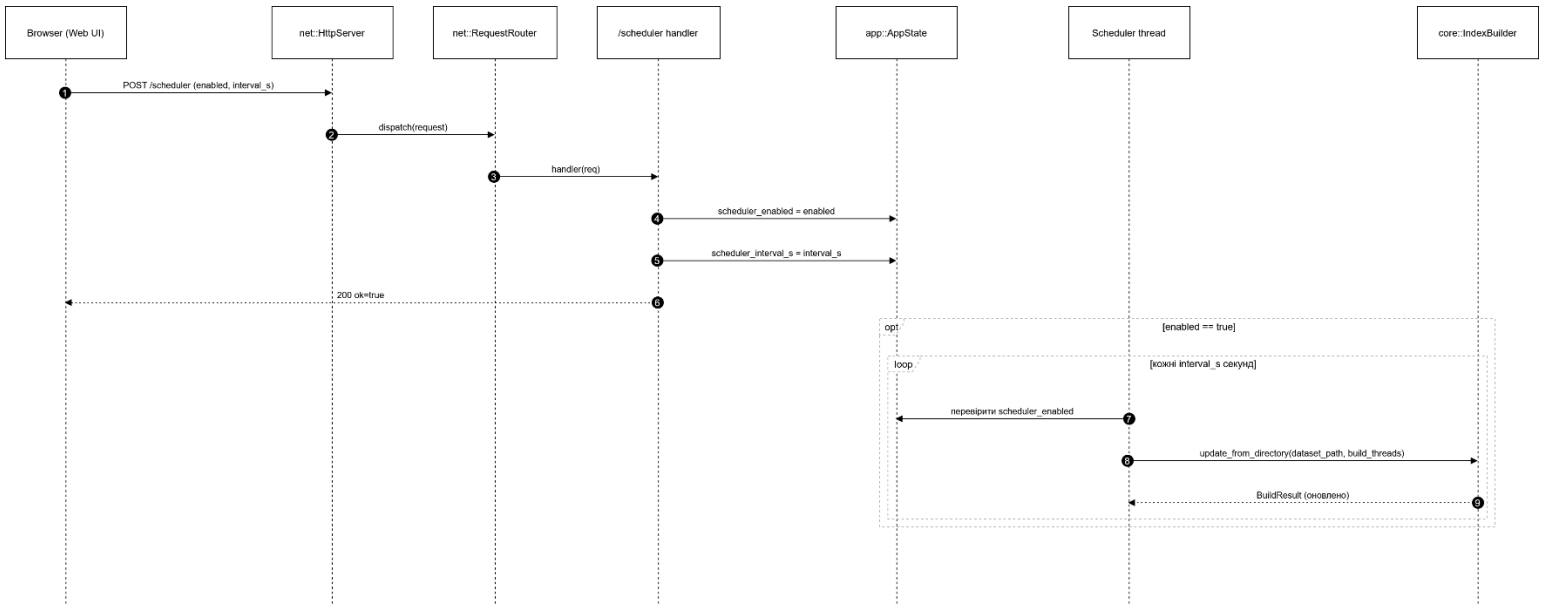

Рисунок 5 — Sequence діаграма scheduler update.

2.4 Deployment діаграма

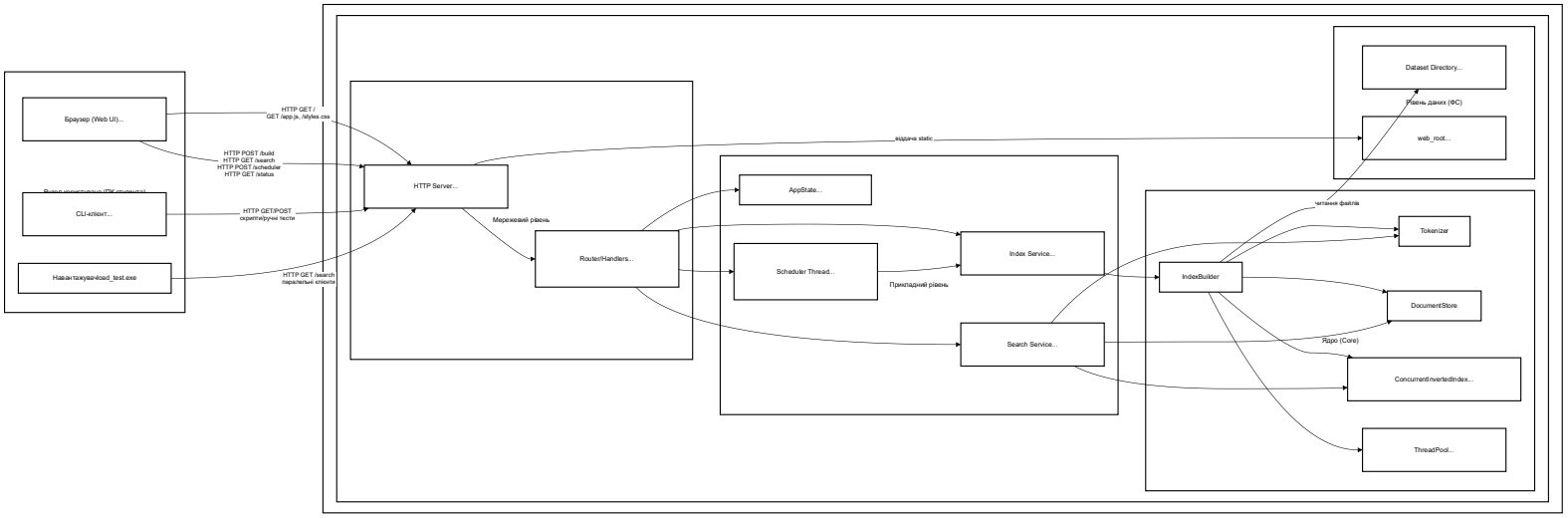

Рисунок 6 — Deployment діаграма.

3. РЕАЛІЗАЦІЯ ПРОГРАМНОГО ПРОДУКТУ

3.1 Архітектура та склад програмних модулів

Програмний продукт реалізовано мовою C++ у вигляді набору виконуваних застосунків та бібліотек (модулів), які взаємодіють між собою через чітко визначені інтерфейси.

Основні виконувані компоненти:

1. server.exe — серверний застосунок (C++), що виконує такі функції:
   - побудова повного інвертованого індексу за текстовою колекцією;
   - інкрементальне оновлення індексу (оновлення за розкладом або вручну);
   - обробка пошукових HTTP-запитів клієнтів;
   - надання статичних веб-ресурсів web UI (наприклад: index.html, app.js, styles.css) через вбудований HTTP-сервер;
   - підтримка режиму scheduler (оновлення індексу з заданим інтервалом без зупинки сервера).
   - обробка HTTP-запитів через пул потоків (черга з'єднань + робочі потоки)
2. client_cli.exe — консольний клієнт (C++), виконує HTTP-запити до сервера: status/search/build/scheduler.
3. load_test.exe — утиліта тестування продуктивності (C++), яка має два режими:
   - build — вимірювання часу побудови індексу залежно від кількості потоків (--threads_list) з автоматичним записом результатів у CSV;
   - search — навантажувальне тестування пошукового API (RPS, latency percentiles p50/p95/p99) залежно від кількості паралельних клієнтів.
   - verify — перевірка коректності: порівняння результатів послідовної (1 потік) та паралельної побудови індексу.

Бібліотеки/модулі проєкту:

- core — реалізація інвертованого індексу, токенізації та побудови/оновлення;
- net — HTTP-сервер і роутер (прийом з'єднань, парсинг запитів, формування відповідей).
- utils — конфігурація, логування, таймінг/затримки.

Таким чином, система відповідає архітектурі клієнт–сервер, де клієнтами можуть бути як web-інтерфейс у браузері, так і консольні утиліти, що виконують роль тестерів і клієнтів.

3.2 Реалізація інвертованого індексу та забезпечення паралельності

Інвертований індекс у системі використовується для швидкого пошуку за ключовими словами у колекції текстових документів. Під час побудови індексу система:

1. перебирає файли датасету;
2. читає текст і виконує токенізацію (нормалізація/фільтрація);
3. додає знайдені токени до структури індексу, фіксуючи відповідність "термін -> список документів та частота".

Побудова індексу є обчислювально затратною операцією, тому реалізовано паралельну обробку документів:

- кожен файл розглядається як незалежна задача "обробити документ";
- задачі розподіляються між потоками через thread pool (черга задач + робочі потоки);
- паралельний запис у спільний індекс виконується коректно завдяки синхронізації та шардінгу індексу.

Індекс реалізований як ConcurrentInvertedIndex і розбитий на shards (за хешем терміна), де кожен shard має власний shared_mutex. Це зменшує contention:

- для read-mostly доступу використовується shared_lock (пошук);

- для оновлення використовується unique_lock (індексація).

Операція пошуку працює як read-mostly:

- запит GET /search не змінює індекс;
- тому декілька запитів пошуку можуть виконуватись паралельно.

Для коректної взаємодії пошуку та оновлення індексу використовується контроль доступу, а оновлення виконується in-place: спочатку видалення старих postings, потім вставка нових.

Це дозволяє виконувати динамічне оновлення без падіння сервера та без некоректних результатів.

Окремо, для обробки клієнтських з'єднань на сервері використовується thread pool: кожне з'єднання ставиться у чергу як задача.

3.3 Протокол взаємодії клієнт–сервер

Взаємодія компонентів реалізована на прикладному рівні через HTTP-запити, що спрощує підключення різних клієнтів і тестерів. Сервер приймає запити на заданому host:port.

Підтримувані ендпоїнти:

- GET /status

  Повертає стан сервера та індексу (наприклад: чи є активний індекс, який датасет використовується, чи ввімкнений scheduler тощо). Також повертає статистику індексу (documents/terms/postings) і дані про останній build (scanned/indexed/skipped/errors).
- POST /build

  Запускає побудову/оновлення індексу. Передаються параметри:
  - dataset_path — шлях до директорії з файлами;
  - threads — кількість потоків для побудови;
  - incremental — режим (повний build або update).

- GET /search?q=...&topk=...

  Виконує пошук за ключовою фразою q. Повертає topk найрелевантніших документів (сортування за спаданням суми частот термінів у документі (простий baseline)).
- POST /scheduler

  Керує планувальником оновлень:
  - enabled — увімкнути/вимкнути;
  - interval_s — інтервал оновлення у секундах.

Пояснення вибору протоколу.

HTTP дозволяє:

- використовувати браузерний клієнт (Web UI) без складних залежностей;
- використовувати CLI (curl або власний C++ клієнт);
- легко проводити навантажувальні тести (через load_test.exe), що прямо відповідає вимогам силабусу.

3.4 Динамічне оновлення індексу (scheduler)

Для виконання вимоги щодо постійного оновлення індексу реалізовано режим автоматичного оновлення:

- у сервері працює окремий фоновий потік/модуль scheduler;
- через інтервал interval_s scheduler запускає інкрементальну індексацію;
- інкрементальний режим спрямований на обробку нових/змінених файлів без повної перебудови, що економить ресурси та зменшує час оновлення.

Важливо, що scheduler працює без зупинки сервера, тобто сервер паралельно:

- приймає запити клієнтів;
- надає результати пошуку;
- періодично актуалізує дані індексу.

3.5 Клієнтські компоненти та перевірка роботи системи

У межах проєкту передбачені різні типи клієнтів/інструментів:

- Web UI (браузер) — користувач взаємодіє через веб-сторінку:
  - задає шлях до датасету та кількість потоків для build/update;
  - запускає побудову/оновлення;
  - виконує пошук за фразою;
  - переглядає статус сервера.
- Консольні клієнти/тестери (C++):
  - load_test.exe — автоматизоване тестування продуктивності (search) і локальна перевірка коректності build (verify).
  - client_cli.exe або використання curl для ручних запитів.

3.6 Файлова структура

Маємо 5 основних папок – папки з данними, результатами load тестингу, скриптами для побудови графіків, основним C++ кодом та веб сторінкою:

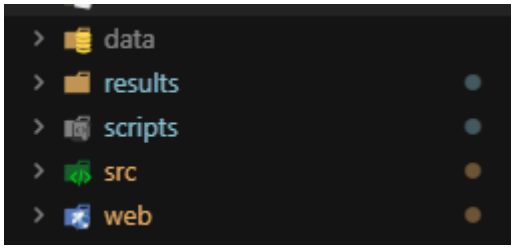

Рисунок 7 — Корінь проекту.

Для даних маємо 1.7 гігабайтів даних Gutenberg:

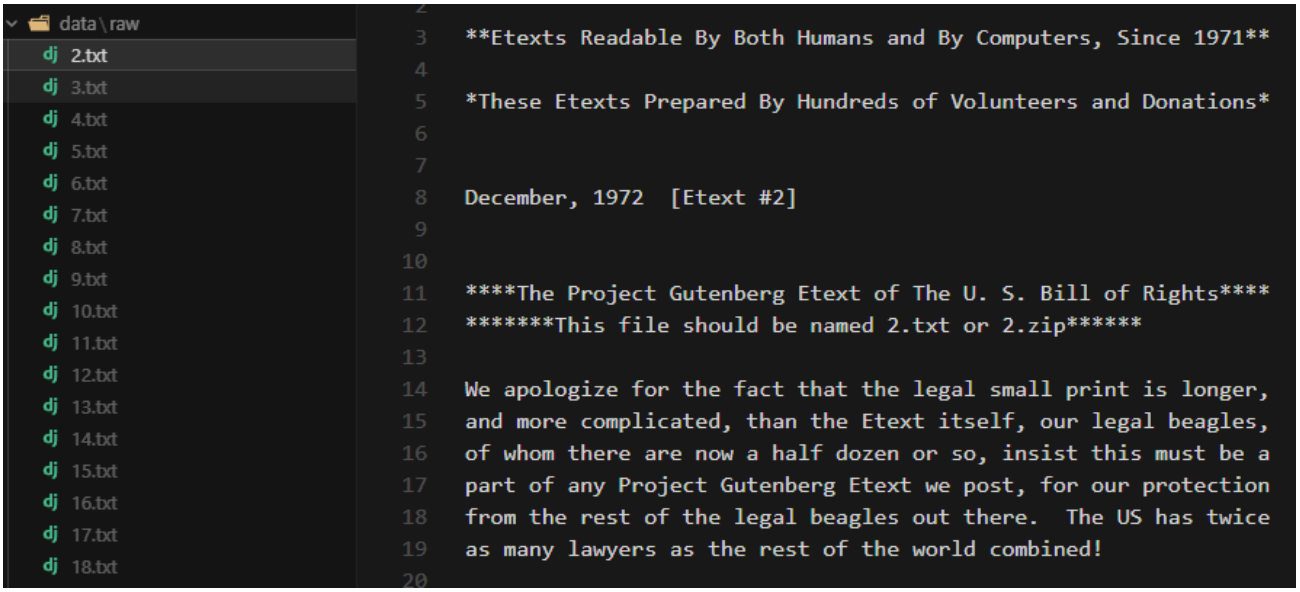

Рисунок 8 — Дані.

У папці results маємо результати навантажувального тестування.

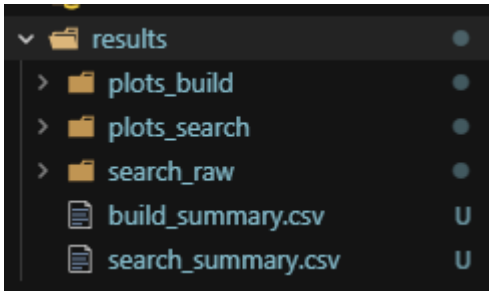

Рисунок 9 — Тестування.

У основній частині проекту стоїть C++ сервер та клієнт, а також програма для навантажувального тестування:

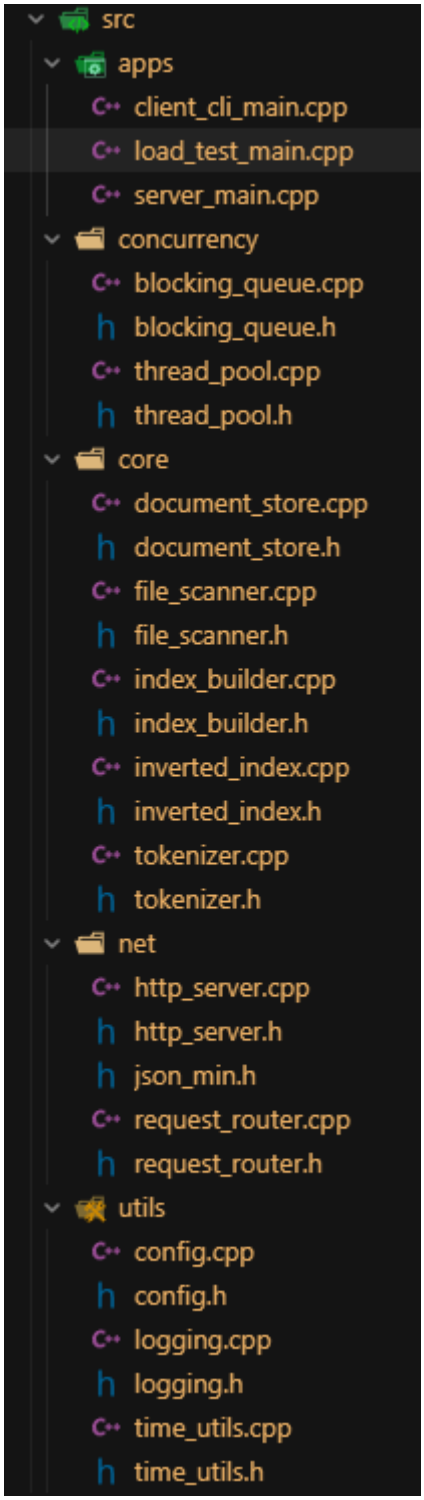

Рисунок 10 — Папка з файлами C++ сервера та клієнта.

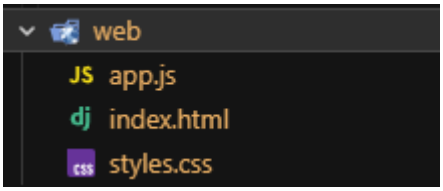

Рисунок 11 — Папка з файлами веб-клієнта.

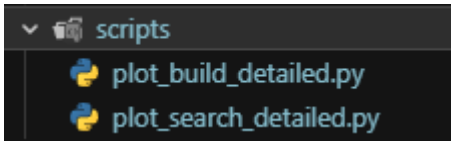

Рисунок 12 — Папка з скриптами.

4. РЕЗУЛЬТАТИ

В проекті є README файл, який допомагає встановити застосунок на пристрій. Запуск програми проводитиметься чітко за його інструкцією.

Припустимо, що всі необхідні залежності (CMake, C++17 компілятор, Python 3.10+ (для графіків)) вже встановлено.

Заходимо в будь-яку папку на комп'ютері, через термінал клонуємо туди проект, збираємо проект за допомогою команд у README, і можна запускати.

Зібрані файли знаходяться в папці build/Release

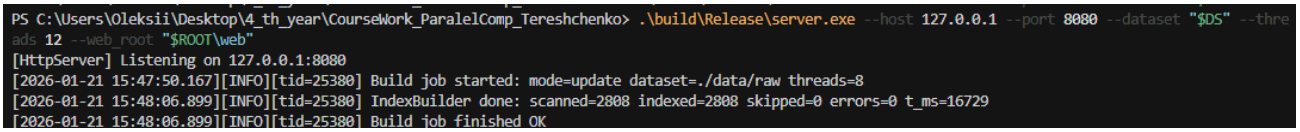

Рисунок 13 — Запуск C++ сервера.

Рисунок 14 — Запуск C++ клієнта, створення індексу, перевірка статусу та пошук слова «print».

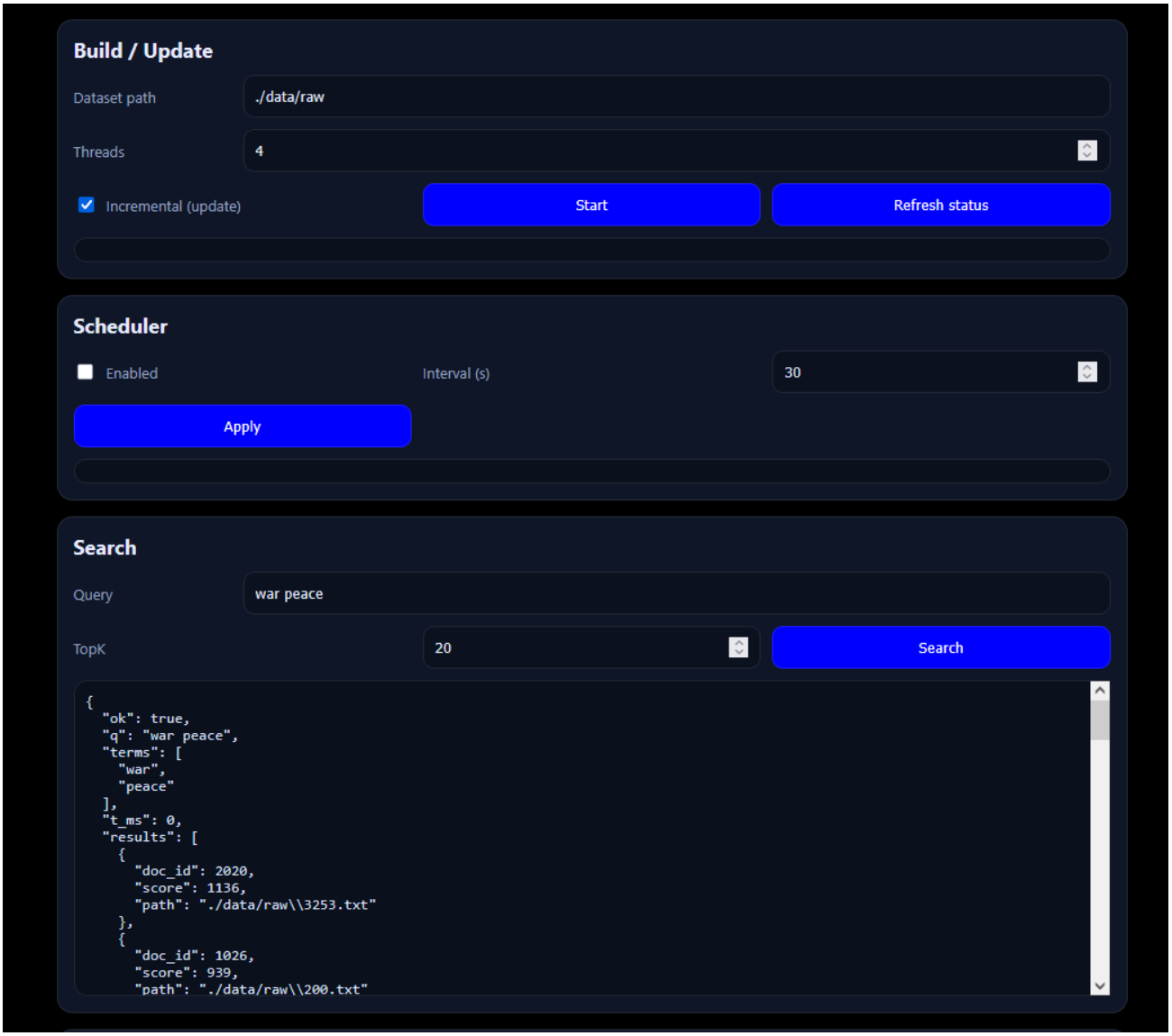

Рисунок 15 — Веб-клієнт у браузері.

5. ТЕСТУВАННЯ

Build-бенчмарк проводився на одному й тому самому наборі даних (data\raw, 2808 файлів). Для кожного значення кількості потоків виконано 5 повторів. Для обробки результатів обчислено:

- середній час mean_ms;
- стандартне відхилення std_ms;
- мінімум/максимум;
- прискорення speedup = T1 / Tn;
- ефективність efficiency = speedup / n.

threads	mean_ms	std_ms	min_ms	max_ms	runs	speedup	efficiency
1	85098.8	4717.76	80867	90338	5	1.000	1.000
2	60761.8	376.48	60326	61327	5	1.401	0.700
3	37258.4	1328.93	36435	39623	5	2.284	0.761
4	32157.8	1343.57	30814	33983	5	2.646	0.662
6	22835.8	149.95	22636	23044	5	3.727	0.621
8	18178.4	85.28	18095	18275	5	4.681	0.585
12	14864.4	164.69	14745	15148	5	5.725	0.477

Таблиця 1 — Агреговані результати побудови індексу (build).

Отримані результати демонструють суттєве прискорення при збільшенні кількості потоків до 6–8, після чого спостерігається ефект спадної віддачі. Максимальне прискорення на 12 потоках становить ≈5.73×, однак ефективність зменшується до ≈0.48, що пояснюється обмеженнями підсистеми введення-виведення (читання файлів), конкуренцією за спільні ресурси (к'онтеншн при записі в індекс) та накладними витратами синхронізації.

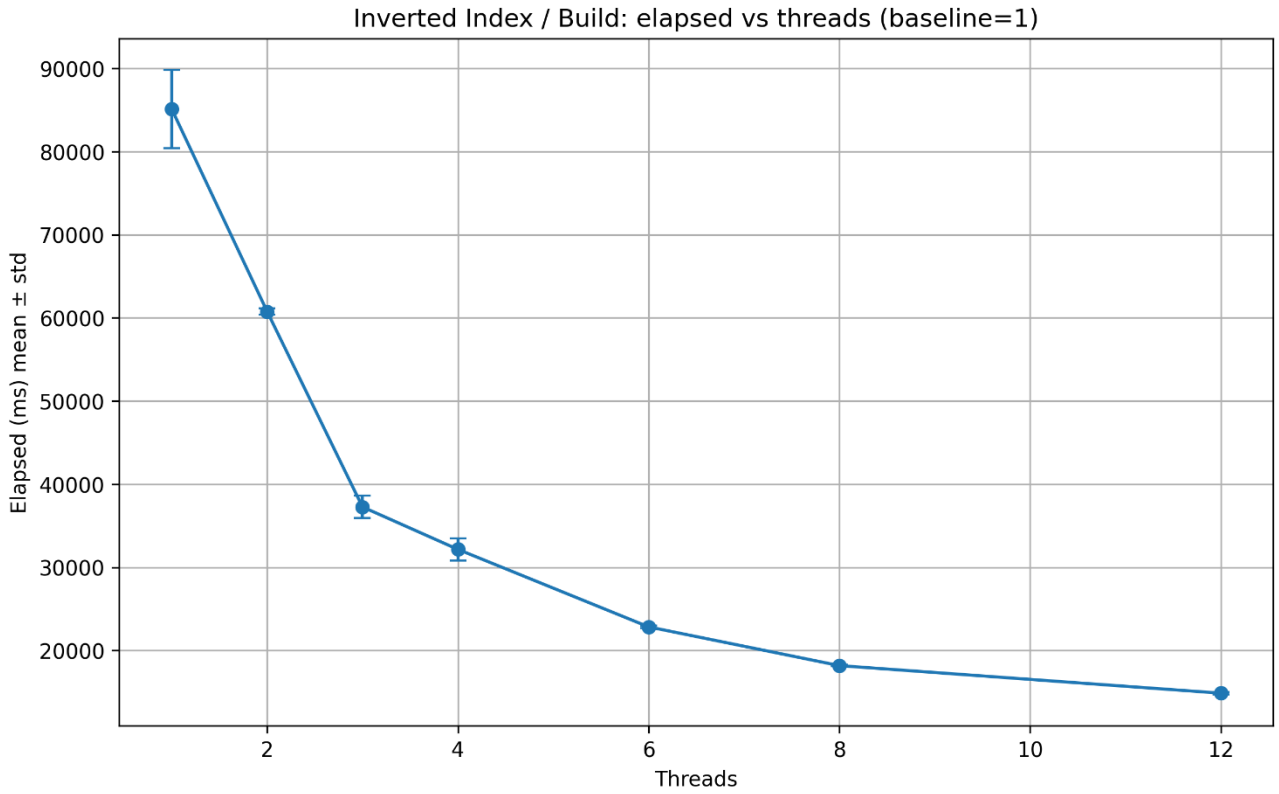

Рисунок 7 — Залежність середнього часу побудови індексу від кількості потоків (mean elapsed time vs threads).

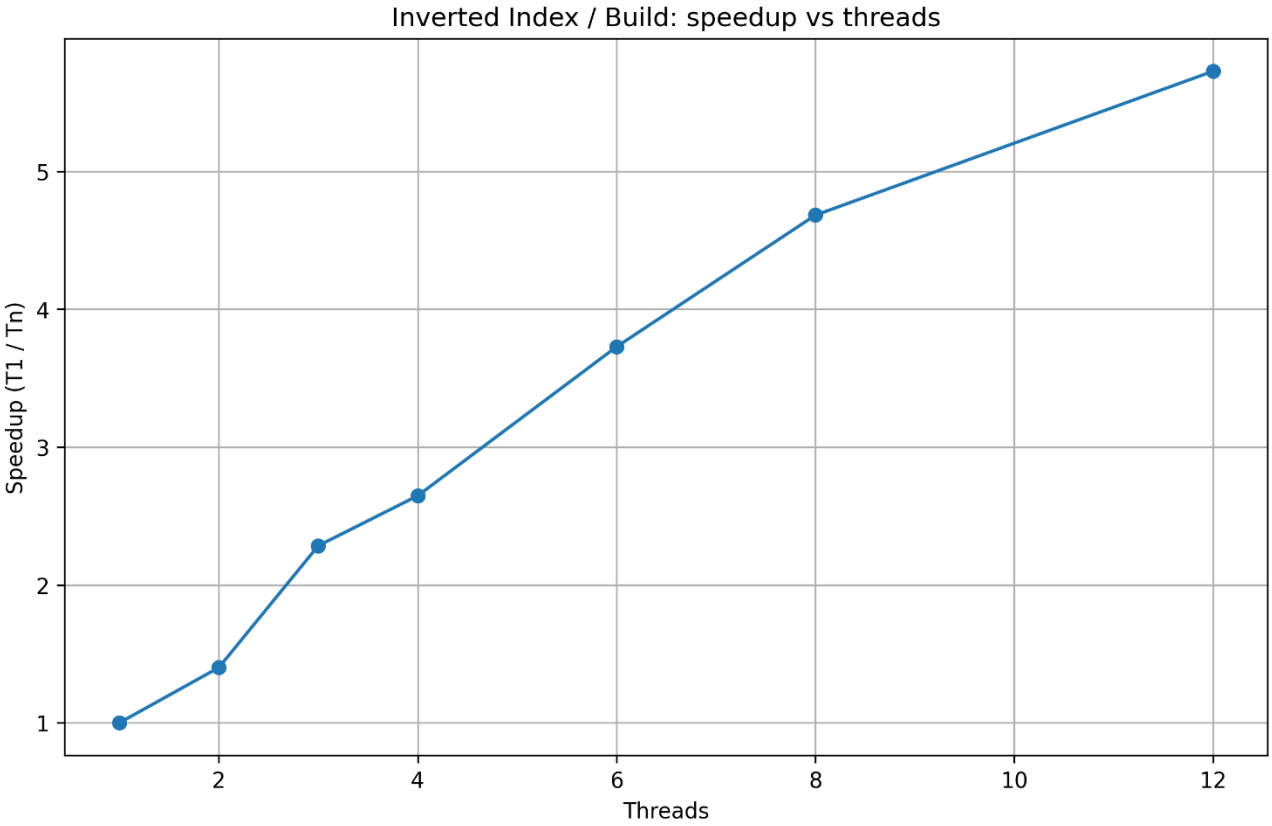

Рисунок 8 — Прискорення побудови (speedup = T1/Tn) залежно від кількості потоків.

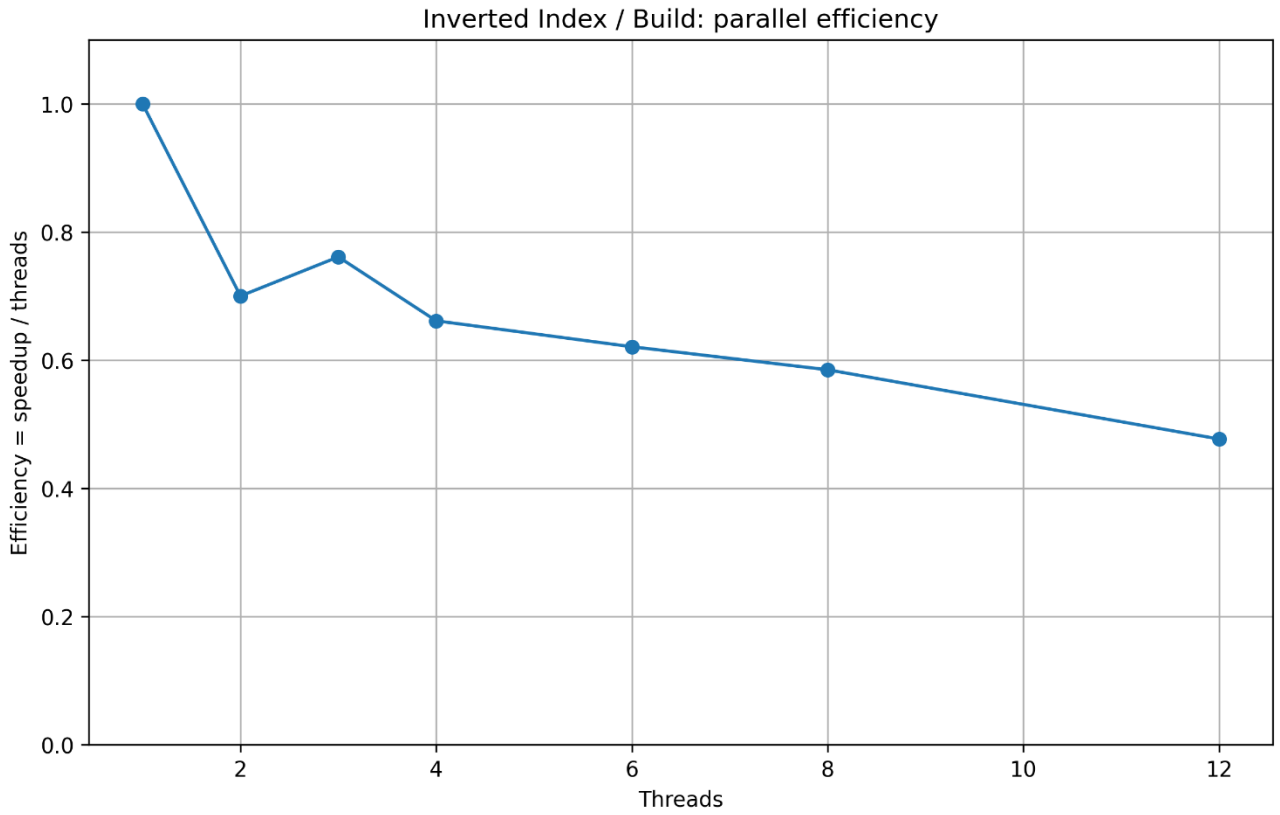

Рисунок 9 — Ефективність використання потоків (efficiency = speedup/n).

З графіків видно, що найоптимальніша кількість потоків — 8-10 на моїй системі (12 ядер, 24 потоки).

Далі ішло навантажувальне тестування:

1. Сервер запускався з попередньо побудованим індексом на наборі data\raw.
2. Утилітою load_test.exe моделювалися паралельні клієнти, які безперервно виконували GET /search?q=...&topk=... протягом фіксованого часу duration_s.
3. Для кожного значення кількості клієнтів виконано 7 повторів.
4. Фіксувалися:

- rps — запити/сек;
- латентність p50/p95/p99 (мс);
- кількість успішних/неуспішних відповідей.

clients	runs	mean_rps	std_rps	p50_ms	p95_ms	p99_ms	fails
1	7	3608.65	95.11	0.00	0.00	0.00	0
2	7	7041.20	259.61	0.00	0.00	0.00	0
5	7	13915.97	263.14	0.00	0.00	0.00	0
10	7	17698.91	187.61	0.00	0.00	0.00	0
20	7	18994.57	96.01	1.00	1.00	1.00	0
30	7	17654.14	491.92	1.00	2.00	3.14	0
40	7	17621.93	522.81	2.00	2.29	3.43	0
50	7	17952.10	581.95	2.00	3.14	4.29	0
75	7	18743.01	352.07	3.29	4.14	4.86	0
100	7	18405.94	496.84	5.00	5.71	7.43	0
150	7	17328.99	138.97	6.00	6.71	7.71	0
200	7	17279.09	46.87	6.00	7.00	93.29	1

Таблиця 2 — Агреговані результати навантажувального тестування пошуку (search).

За результатами тестування видно, що пропускна здатність (RPS) зростає зі збільшенням кількості клієнтів до певного рівня, після чого система виходить на плато. Латентність також зростає, особливо у високих перцентилях (p99), що відображає рідкісні “довгі” запити при піковому навантаженні. Зафіксовано поодинокий збій на 200 клієнтах, що може свідчити про досягнення межі ресурсів або тимчасові мережеві/планувальні затримки ОС.

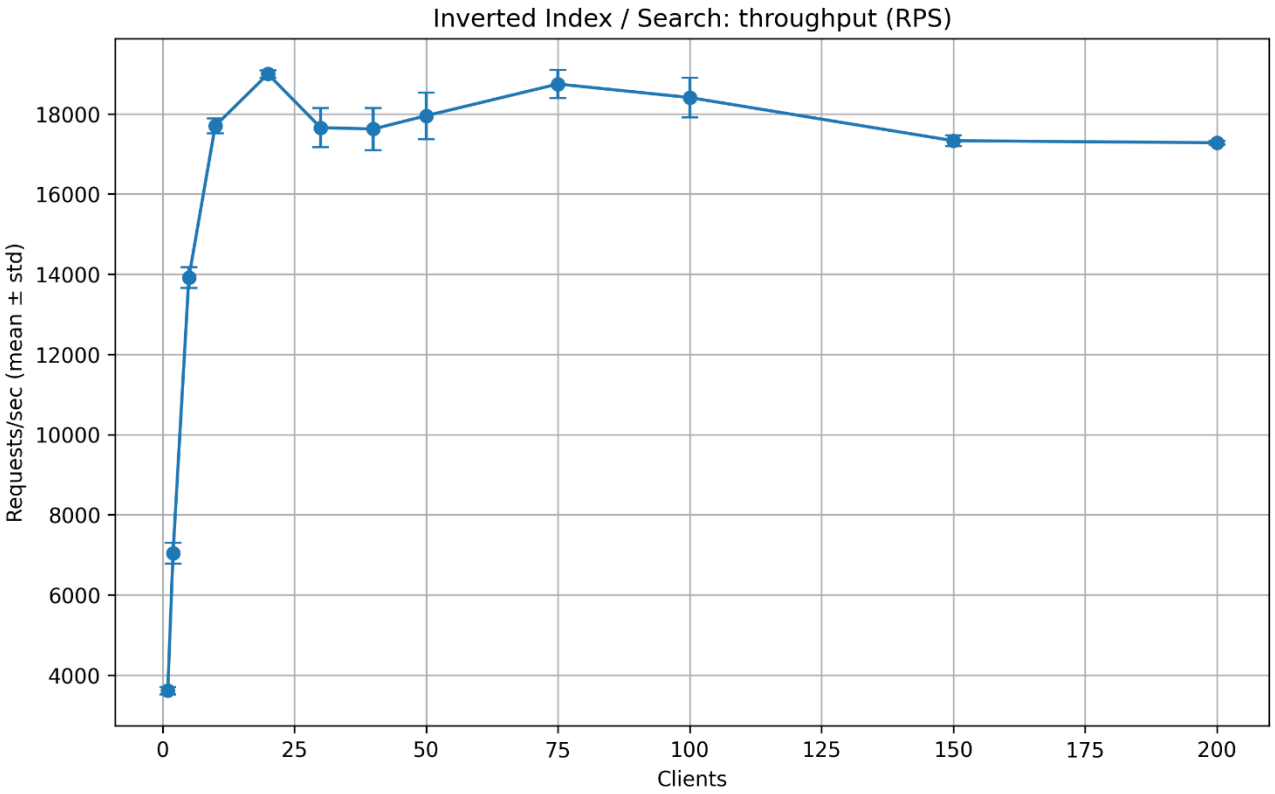

Рисунок 11 — Залежність пропускної здатності (RPS) від кількості паралельних клієнтів.

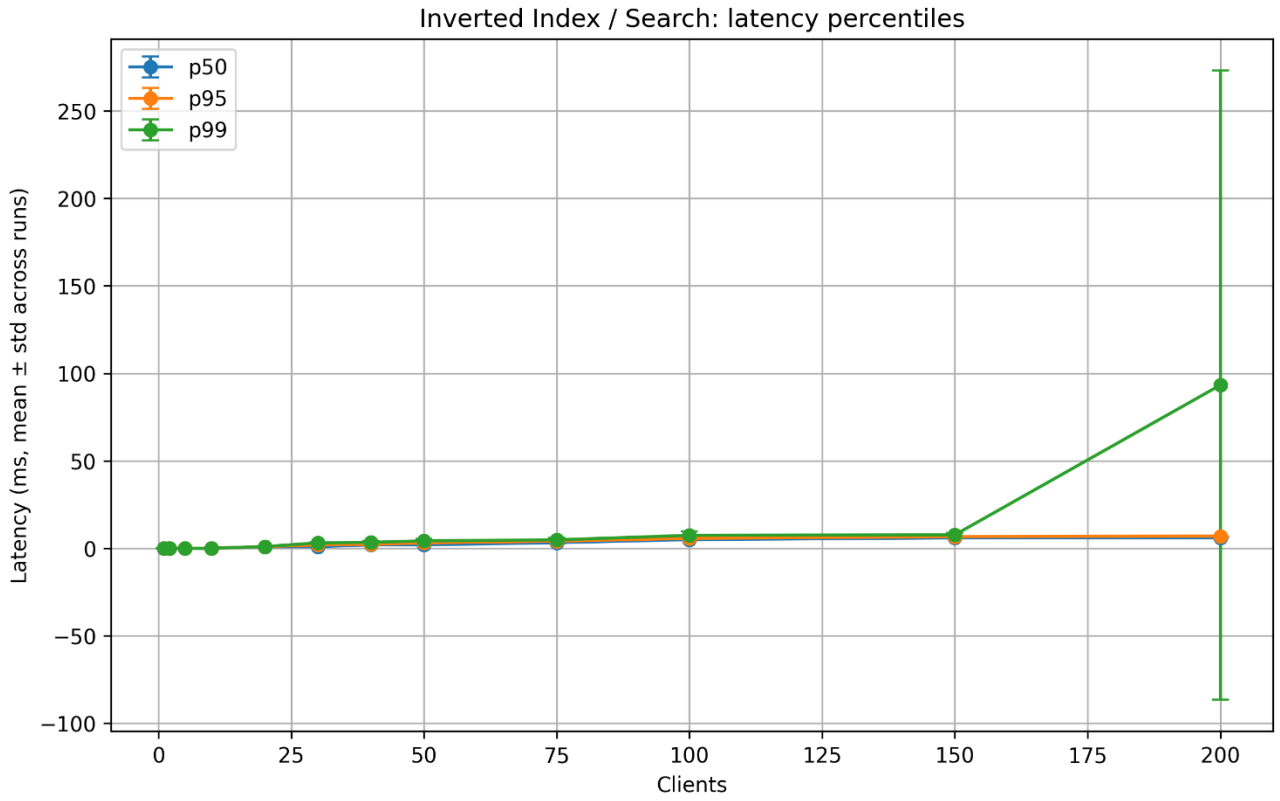

Рисунок 12 — Залежність затримок (latency p50/p95/p99) від кількості клієнтів.

Окремо виконано перевірку коректності послідовної та паралельної побудови індексу: режим verify у load_test порівнює сигнатури індексу для 1 потоку та для N потоків. Отримані результати збіглися.

```
PS C:\Users\Oleksii\Desktop\4_th_year\CourseWork_ParalelComp_Tereshchenko> .\build\Release\load_test.exe --mode verify --dataset "./data/raw" --threads_list "2,4,8"
[2026-01-21 16:45:12.928][INFO][tid=4064] IndexBuilder done: scanned=2808 indexed=2808 skipped=0 errors=0 t_ms=87629
[2026-01-21 16:46:24.364][INFO][tid=4064] IndexBuilder done: scanned=2808 indexed=2808 skipped=0 errors=0 t_ms=56710
verify threads=2 ok
[2026-01-21 16:47:10.411][INFO][tid=4064] IndexBuilder done: scanned=2808 indexed=2808 skipped=0 errors=0 t_ms=28126
verify threads=4 ok
[2026-01-21 16:47:55.057][INFO][tid=4064] IndexBuilder done: scanned=2808 indexed=2808 skipped=0 errors=0 t_ms=23682
verify threads=8 ok
PS C:\Users\Oleksii\Desktop\4_th_year\CourseWork_ParalelComp_Tereshchenko>
```

Рисунок 13 — Залежність затримок (latency p50/p95/p99) від кількості клієнтів.

ВИСНОВКИ

У курсовій роботі реалізовано клієнт-серверний веб-сервіс для пошуку в текстових даних на основі інвертованого індексу з підтримкою паралельної побудови та динамічного оновлення індексу. Забезпечено мережеву взаємодію через сокети, реалізовано два клієнти (веб-інтерфейс та консольні утиліти для тестів), а також проведено експериментальне дослідження продуктивності.

Експерименти build показали прискорення до ≈5.73× на 12 потоках при зниженні ефективності до ≈0.48, що пояснюється накладними витратами синхронізації та обмеженнями I/O. Навантажувальне тестування search підтвердило наявність плато пропускної здатності та зростання латентності у високих перцентилях під великим навантаженням, що дозволяє визначити практичні режими експлуатації та наближення до точки відмови.

Github: https://github.com/Fr0ndeur/CourseWork_ParalelComp_Tereshchenko

ДЖЕРЕЛА

1. Manning C., Raghavan P., Schütze H. Introduction to Information Retrieval. Cambridge University Press, 2008.
2. Williams A. C++ Concurrency in Action (Second Edition). Manning Publications, 2019.
3. Goetz B., Peierls T., Bloch J., Bowbeer J., Holmes D., Lea D. Java Concurrency in Practice. Addison-Wesley (Pearson), 2006.
4. RFC Editor. RFC 9110: HTTP Semantics (IETF), 2022.