Звіт по мультипоточності

Маємо з лаби2 застосунок, який за допомогою HTTP-запиту до Google Books API (стандартного, за URL форматом: [https://www.googleapis.com/books/v1/volumes?q={keyword}&maxResults](https://www.googleapis.com/books/v1/volumes?q=%7bkeyword%7d&maxResults)= ) для різних ключових слів із параметром maxResults= виконує пошук книжок.

*maxResults - кількість відображених результатів для 1 запиту.*

Застосунок іноді “тормозить” через те, що пошукові запити виконуються послідовно, і це займає багато часу. Для підвищення продуктивності запропоновано використовувати мультипоточність, що дозволяє виконувати запити паралельно. Це має зменшити загальний час очікування та покращити швидкодію застосунку. У звіті буде проведено порівняння послідовного та паралельного виконання запитів, щоб оцінити реальний ефект від впровадження мультипоточності.

**Спершу вимірюємо час виконання запитів БЕЗ використання мультипоточності.**

Для оцінки продуктивності пошуку книг у системі було виміряно час виконання HTTP-запиту від початку запиту до виведення результатів.

Масив слів складається з **11 слів**:

* Технічних термінів (*python, java, c++, javascript*)
* Загальних термінів (*animal, history, nature*)
* Специфічних термінів (*machine learning, quantum physics*)
* Часто використовуваних слів (*love*)
* Рідкісних слів (*zymurgy*)

Запити для кожного з цих слів по черзі виконуються для різної кількості виведених результатів (*maxResults=5, 10, 20, 40*).

Рахується час, за який для кожного з цих 11 слів по черзі виводиться *maxResults=певному значенню,* і виводиться в консоль.

Маємо відповідну функцію, що виконує це завдання:

def run\_batch\_searches(self):

        search\_words = ["python", "c++", "java", "javascript", "animal",

                        "history", "nature", "machine learning", "quantum physics", "love", "zymurgy"]  # Приклад списку ключових слів

        max\_results\_list = [5, 10, 20, 40]

        for max\_results in max\_results\_list:

            print(f"\n--- Searching with maxResults={max\_results} ---")

            start\_time = time.perf\_counter()  # Починаємо вимір для всіх запитів разом

            for query in search\_words:

                print(f"Searching for '{query}' ...")

                url = f"https://www.googleapis.com/books/v1/volumes?q={query}&maxResults={max\_results}"

                response = requests.get(url)

                if response.status\_code != 200:

                    print(f"Error fetching data for query '{query}'")

                    continue

                data = response.json()

                # Тут можна додати обробку даних, якщо потрібно

            end\_time = time.perf\_counter()  # Кінець виміру для всіх запитів

            elapsed = end\_time - start\_time

            print(f"[Timing] Total time for maxResults={max\_results}: {elapsed:.4f} seconds")

**Результати послідовного виконання запитів**

| **maxResults** | **Загальний час виконання (сек)** |
| --- | --- |
| 5 | 6.0494 |
| 10 | 6.2482 |
| 20 | 6.3079 |
| 40 | 7.3284 |

**Тепер вимірюємо час виконання запитів ІЗ використанням мультипоточності.**

Для впровадження мультипоточності використовуємо пул потоків (ThreadPoolExecutor), де кожен потік одночасно робить свій запит.

Переписуємо нашу функцію run\_batch\_searches() так, щоб вона для кожного значення max\_results (5, 10, 20, 40) запускала пул з 5 потоків (оптимальне значення). Тоді кожен потік паралельно бере по 1 ключовому слову і виконує HTTP-запит до API Google Books.

Таким чином, одночасно працюють до 5 потоків, і загалом всі 11 запитів виконуються швидше, ніж послідовно (один за одним).

Коли один потік закінчує свій запит, пул автоматично запускає наступний запит з черги, поки всі запити не будуть виконані.

Після того, як усі запити для поточного max\_results завершилися, вимірюється загальний час, який зайняло їх паралельне виконання, і виводиться в консоль.

Ось так це виглядає в коді:

def run\_batch\_searches(self):

        search\_words = ["python", "c++", "java", "javascript", "animal",

                        "history", "nature", "machine learning", "quantum physics", "love", "zymurgy"]

        max\_results\_list = [5, 10, 20, 40]

        for max\_results in max\_results\_list:

            print(f"\n--- Searching with maxResults={max\_results} ---")

            start\_time = time.perf\_counter()

            # Функція для одного запиту

            def fetch\_data(query):

                print(f"Searching for '{query}' ...")

                url = f"https://www.googleapis.com/books/v1/volumes?q={query}&maxResults={max\_results}"

                response = requests.get(url)

                if response.status\_code != 200:

                    print(f"Error fetching data for query '{query}'")

                    return None

                return response.json()

            # Виконуємо всі запити паралельно

            with ThreadPoolExecutor(max\_workers=5) as executor:

                # Створюємо майбутні об'єкти (future) для кожного запиту

                futures = {executor.submit(fetch\_data, query): query for query in search\_words}

                for future in as\_completed(futures):

                    query = futures[future]

                    try:

                        data = future.result()

                    except Exception as exc:

                        print(f"Query '{query}' generated an exception: {exc}")

            end\_time = time.perf\_counter()

            elapsed = end\_time - start\_time

            print(f"[Timing] Total time for maxResults={max\_results}: {elapsed:.4f} seconds")

**Результати паралельного виконання запитів**

| **maxResults** | **Загальний час виконання (сек)** |
| --- | --- |
| 5 | 1.8868 |
| 10 | 1.4115 |
| 20 | 1.6902 |
| 40 | 1.7544 |

**Аналізуємо обидві таблиці**

**Паралельне виконання** дає значне прискорення у порівнянні з послідовним.

Час виконання у паралельному режимі — близько 1.4–1.9 секунд, тоді як послідовно — від 6 до понад 7 секунд.

**Це показує, що мультипоточність дозволяє істотно зменшити загальний час очікування.**

**Можемо ще перевірити результати для іншої кількості потоків, але тих самих даних, щоб переконатися, що обрана кількість (5) є оптимальною.**

| **max\_workers** | **maxResults=5** | **maxResults=10** | **maxResults=20** | **maxResults=40** |
| --- | --- | --- | --- | --- |
| 2 | 2.9917 | 3.9882 | 3.6484 | 3.9403 |
| 10 | 1.5742 | 1.0788 | 1.1166 | 1.7928 |
| 20 | 0.9618 | 1.1382 | 0.6060 | 1.3544 |

**max\_workers=2**:

Надто мала кількість потоків. Багато запитів очікують у черзі.

Система не встигає обробляти запити паралельно загальний час високий. Хоч і не такий, як при послідовному виконанні запитів, але порівняно з більшою кількістю потоків - так.

**max\_workers=20**:

Найкращі результати для maxResults=5 і 20, але вже **не стабільно**:

Для maxResults=10 — час вищий, ніж з max\_workers=10.

* для **max\_workers=10** і maxResults=10 час = **1.0788 секунд**
* для **max\_workers=20** і maxResults=10 час = **1.1382 секунд**

Тобто, коли ти **збільшуєш кількість потоків із 10 до 20**, **час обробки збільшується.**

Це вказує на **перевантаження або конкуренцію за ресурси**: надто багато потоків можуть створити затори (особливо на мережевих або системних рівнях).

**max\_workers=10**:

Найбільш **стабільний** варіант з цих:

Час виконання **не коливається сильно** для різних обсягів даних.

Найкращий час для maxResults=10, і гарний баланс у всіх інших випадках.

Результати тут майже такі самі, як і при кількості потоків 5.

**Вочевидь, саме 5-10 потоків достатньо, щоб запити оброблялися майже одночасно, без перевантаження ресурсів.**

### Подальші вдосконалення введенням мультипоточності

# Асинхронний *search()* (основний пошук)

Зараз метод search() виконує синхронний запит до API, що **блокує UI**, поки не отримає результат. Це помітно, якщо інтернет повільний.

Візьмемо кілька запитів і виміряємо час виконання кожного з них, як і раніше, за допомогою бібліотеки time:

| **Запит** | **Час виконання (сек)** |
| --- | --- |
| python | 4.1911 |
| animal | 4.1287 |
| machine learning | 4.0851 |
| love | 4.2308 |

**Оптимізація**

Винесемо всю логіку API-запиту та обробки в окремий клас SearchWorker, який працює в окремому потоці.

class WorkerSignals(QObject):

    finished = pyqtSignal(dict, float)

    error = pyqtSignal(str)

class SearchWorker(QRunnable):

    def \_\_init\_\_(self, query, max\_results=20):

        super().\_\_init\_\_()

        self.query = query

        self.max\_results = max\_results

        self.signals = WorkerSignals()

    @pyqtSlot()

    def run(self):

        try:

            start\_time = time.perf\_counter()

            url = f"https://www.googleapis.com/books/v1/volumes?q={self.query}&maxResults={self.max\_results}"

            response = requests.get(url)

            if response.status\_code != 200:

                self.signals.error.emit("Error fetching data from Google Books API.")

                return

            data = response.json()

            elapsed = time.perf\_counter() - start\_time

            self.signals.finished.emit(data, elapsed)

        except Exception as e:

            self.signals.error.emit(str(e))

Оновлюємо метод search()

def search(self):

        self.save\_current\_state\_as\_memento()

        self.clear\_results()

        query = self.search\_box.text().strip()

        if not query:

            return

        self.thread\_pool = QThreadPool.globalInstance()

        worker = SearchWorker(query)

        worker.signals.finished.connect(self.handle\_search\_results)

        worker.signals.error.connect(self.handle\_search\_error)

        self.thread\_pool.start(worker)

Решту логіки колишнього search() винесемо в окрему функцію.

def handle\_search\_results(self, data, elapsed):

        group\_mode = self.grouping\_box.currentText()

        grouped = {}

        ungrouped = []

        for item in data.get('items', []):

            info = item.get('volumeInfo', {})

            title = info.get('title', 'N/A')

            published\_date = info.get('publishedDate', 'N/A')

            rating = info.get('averageRating', 'N/A')

            image = info.get('imageLinks', {}).get('thumbnail', '')

            authors = info.get('authors', [])

            self.notifier.notify(title)

            leaf = BookLeaf(title, image, published\_date, rating, authors)

            if group\_mode == "Group by Year":

                key = published\_date.split('-')[0] if published\_date != 'N/A' else "Unknown"

            elif group\_mode == "Group by Rating":

                key = str(rating) if rating != 'N/A' else "No Rating"

            elif group\_mode == "Group by First Letter":

                key = title[0].upper() if title and title[0].isalpha() else "#"

            elif group\_mode == "Group by Author":

                key = authors[0] if authors else "Unknown Author"

            else:

                key = None

            if key is None:

                ungrouped.append(leaf)

            else:

                if key not in grouped:

                    grouped[key] = BookComposite(key)

                grouped[key].add(leaf)

        if group\_mode == "No Grouping":

            for leaf in ungrouped:

                leaf.display(

                    self.results\_layout,

                    show\_date=self.check\_var.isChecked(),

                    show\_rating=self.check\_var2.isChecked()

                )

        else:

            for key in sorted(grouped.keys()):

                grouped[key].display(

                    self.results\_layout,

                    show\_date=self.check\_var.isChecked(),

                    show\_rating=self.check\_var2.isChecked()

                )

        print(f"[Timing] Search for '{self.search\_box.text().strip()}' took {elapsed:.4f} seconds")

Додамо метод обробки помилок:

  def handle\_search\_error(self, message):

        QMessageBox.critical(self, "Помилка пошуку", message)

**Клас SearchWorker** наслідується від QRunnable, що дозволяє його запускати у пулі потоків (QThreadPool).

**Завдяки QThreadPool + QRunnable ми переносимо цю операцію у фоновий потік. Робимо пошук асинхронним.**

Метод run() виконується в окремому потоці, не блокуючи основний GUI-потік.

В цьому методі відбувається HTTP-запит до Google Books.

thread\_pool.start(worker) запускає нашого SearchWorker у вільному потоці з пулу.

Це означає, що основний потік GUI залишається вільним, інтерфейс не "зависне" і може реагувати на дії користувача.

WorkerSignals — клас, який визначає сигнали finished і error.

Після завершення роботи в потоці SearchWorker емінтує сигнал finished з результатами (data, elapsed).

Ці сигнали приймаються в основному потоці і викликають метод handle\_search\_results.

Аналогічно, у випадку помилки емінтується сигнал error, який викликає handle\_search\_error.

**Результати**

| **Запит** | **Час виконання (сек)** |
| --- | --- |
| python | 0.6572 |
| animal | 0.9237 |
| machine learning | 1.2120 |
| love | 0.7400 |

**Аналізуємо обидві таблиці**

У першій таблиці час виконання близько 4 секунд для кожного запиту — це послідовне виконання, без асинхронності, що призводить до значно більших затримок.

У другому блоці час виконання запитів від 0.65 до 1.2 секунд — це швидке виконання, що свідчить про асинхронне опрацювання, яке дозволяє не блокувати основний потік і одночасно обробляти кілька запитів.

Різниця в часах показує, що застосування асинхронності через search() суттєво покращує продуктивність і скорочує час відповіді на запити.