Project Title: Multithreaded Web Crawler

By: Amaar Iftikhar 35441 & Naqi Turab

Submitted to: Sir Tajamul Shahzad

Introduction

The Internet has given birth to an ever-growing knowledge base. In order to retrieve information from these repositories, a web crawler is used, which is an automated tool to surf web pages methodically.

This project will deal with the development of a **multithreaded web crawler** that should be efficient in fetching web pages and storing data in the database for analysis.
The project will handle concurrency with multithreading, manage URLs using a queue system, and store the crawled data systematically in an SQLite database. Such systems are used in search engines, data mining, and content analysis.

Goals and Objectives

The key goals of this project are:
Fast Crawling: Design a Crawler that will fetch content from the web at high speed without overloading the servers.
Concurrency: Implement multithreading, allowing simultaneous crawling of web pages.
Database Integration: The crawled data should be stored in a structured database, comprising the content of web pages and the link relationships between them.
Queue Management: The crawling order can be managed through a queue mechanism in order to avoid any duplicate URLs.
Scalability: The system should support larger datasets either by increasing the thread count or optimizing the database.

Challenges

The following are some of the challenges experienced during development:
Concurrency Management: Thread-safe access of shared resources, like visited URLs, done by multiple threads.
Error Handling: How to handle network timeouts, broken links, and unexpected HTTP response codes.
URLs Validation: Checking and normalizing URLs to avoid redundancies or invalid entries.
Database Design: This involves the design of the database for high insertion and query rates.
Politeness: It introduces delays between requests to avoid burdening web servers.

Proposed Methodology

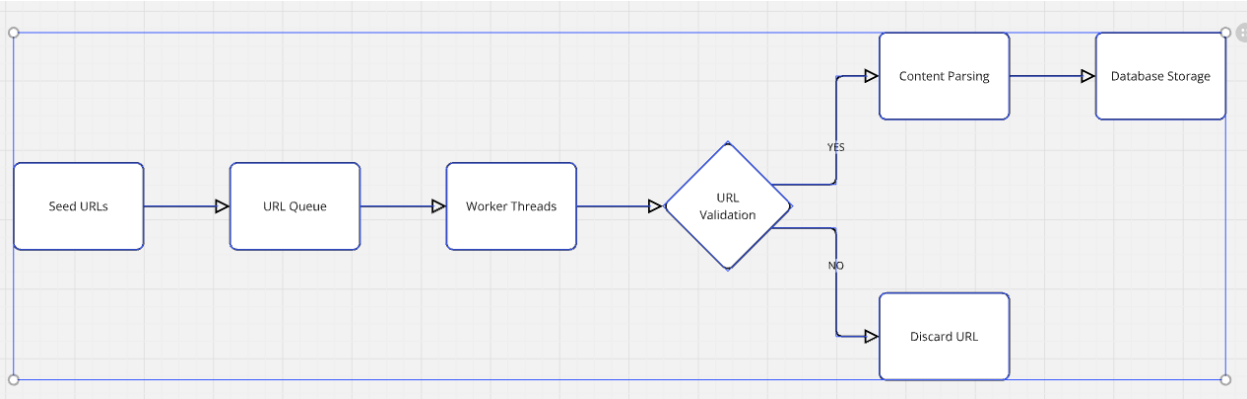
The system consists of several core components working together:
Seed URLs: These are the locations at which the crawl will begin and start from. Initial URLs are added in a queue for processing.
Multithreading: Each thread fetches URLs from the queue and processes them concurrently.
Locking mechanisms provide thread safety during shared resource operations.
URL queue: A synchronized queue manages URLs waiting to be crawled.
Avoids duplication, controls the order of crawl.

Web Scraping: HTML content is fetched using “requests”.
“Beautiful Soup” parses the HTML and extracts links.

Database Storage: URLs crawled, their text content, and link relationships are stored in an SQLite database.
A thread-local connection provides consistency for data.

System Design

Flowchart



Class Structure

WebCrawler: Handles URL crawling, multithreading, and coordination of tasks.
Database Manager: Manages database connections and operations such as URL insertions and link mappings.
URL Queue: Provides thread-safe queue that holds the URLs.

Code Description:

WebCrawler Class: The `WebCrawler` class manages the overall crawling

Initialization: sets seed URLs, maximum depth and number of threads.

Crawl Method: Fetches a web page, parses its content, and stores data in the database.

Worker Threads: A number of threads of execution run concurrently, fetching and processing URLs.

Start Crawling: Starts the crawling process and waits for all threads to be finished.

Database Manager Class

Handles the interaction with the SQLite database:

Initialize: Creates database tables for URLs and links.

Insert URL: Stores URLs that have been crawled, and their content. -

Insert Link: It maps the relationship between the source and target URLs.

Update Status: Provides the status of each URL pending, crawled, or failed.

URL Queue Class

A thread-safe queue to handle URLs:

Insert URL: Inserts new URLs without duplication.

Get URL: Gets next URL for processing.

Task Processed: Marks a URL as processed.

Conclusion:

This project will be used to demonstrate the use of a multithreaded web crawler and integrate it with an SQLite database. It should fetch the web content, store it for analysis, and should be scalable because of its modular design.

Future Improvements:

Use dynamic depth depending on page importance.

Add support for robots.txt compliance.

Improve the performance of the database by using advanced indexing techniques or hosted on a distributed database.

Integrate advanced features for scraping: JavaScript-heavy websites using software like Selenium.