Documentation: Local.ch Scraper

Code Source: GitHub Repo Link Click Here

- **indexDataBaseRunTime.py**
  - This is the Main Code which follow the database updation approach that the client have been mentioned
- **indexDataBaseEnd.py**
  - The new code optimizes performance by deferring database updates. Instead of updating the database after each URL is processed, the script accumulates all successful scrapes in memory and performs a bulk insertion at the end. Additionally, URLs that fail to be scraped are stored in a "Missing.txt" file for later review.
- **indexExtractUrls.py**
  - This code will responsible to fetch all urls from Xml file and store them into extracted_urls.txt file for further process.
- **txtComparison.py**
  - This code will use for testing in last,how many urls have not parse,actually this code perform comaparison operation between Original Urls file and final Data urls File
- **DatabaseCombine.py**
  - As my internet slow,so i have run code in small bunch upto 50k urls,and in last ,by the help of this code,i had join all database to Final Database

Data Source: Drive Link Click Here

- Click here to get CSV and Database for Final Data approx. 654580

Explanation of Code Performance: Old Approach vs. New Approach

1. Overview of the Two Approaches

- **Old Approach**:
  - In the previous code, each time a web page was successfully scraped, the database was updated immediately. This meant that for each of the 600,000+ URLs, the script performed a database operation, which significantly slowed down the overall execution.
- **New Approach**:
  - The new code optimizes performance by deferring database updates. Instead of updating the database after each URL is processed, the script accumulates all successful scrapes in memory and performs a bulk insertion at the end. Additionally, URLs that fail to be scraped are stored in a "Missing.txt" file for later review.

2. Why the Previous Code is Slower

- **Frequent Database Updates**:
  - In the previous approach, the script performs a database update for every URL processed. With over 600,000 URLs, this means 600,000+ separate database

transactions. Each transaction involves locking the database, writing to the disk, and unlocking the database, which is resource-intensive and time-consuming.

- **Thread Contention**:
    - Due to frequent updates, multiple threads are constantly trying to access the database simultaneously, leading to thread contention. The use of a lock (threading.Lock()) mitigates potential data corruption, but it also forces threads to wait, further slowing down the overall process.

3. Why the New Code is Faster

- **Deferred Database Updates**:
    - By deferring database updates until all URLs are processed, the new approach reduces the number of database transactions to just a single bulk insertion. This greatly reduces the overhead associated with disk I/O operations and database locking.
- **Reduced Thread Contention**:
    - Since the database is updated only once at the end, the need for thread locking during the scraping process is minimized. This allows threads to operate more independently and efficiently, leading to faster completion times.
- **Efficient Error Handling**:
    - Instead of retrying or logging failed URLs immediately, the new approach simply records them in memory and writes them to a file at the end. This prevents the scraping process from being interrupted or slowed down by failed requests.

4. Conclusion

- **Old Approach**:
    - The primary drawback of the old approach is its inefficiency in handling large-scale database operations. Frequent database updates slow down the entire scraping process, making it unsuitable for large datasets.
- **New Approach**:
    - The new approach is designed for speed and efficiency. By deferring database operations and handling errors more gracefully, it significantly reduces the overall execution time, making it ideal for large-scale web scraping tasks.

Code Flow Summary

1. Overview

This Python script is designed to scrape data from specified web pages, process the data concurrently using threading, and store the results in a local SQLite database. Finally, it exports the data into a CSV file for further analysis or usage.

2. Imports and Dependencies

- **requests**: Used for making HTTP requests to web pages.
- **json**: Used for handling JSON data (though not utilized in the provided code).
- **BeautifulSoup**: From the bs4 library, used for parsing HTML content and extracting information from web pages.
- **pandas**: For handling data manipulation and converting database results into a DataFrame to be exported as a CSV file.
- **ThreadPoolExecutor and as_completed**: From the concurrent.futures module, used for managing threading and handling asynchronous tasks.
- **alive_bar**: For providing a progress bar to track the progress of the scraping tasks.
- **threading**: Provides thread management, specifically for locking mechanisms to handle shared resources.
- **dataset**: Used for interacting with a SQLite database, allowing easy storage and retrieval of scraped data.
- **time**: Used for implementing pauses or delays in execution.

3. Function Definitions

3.1. retrivePages(url, id)

- **Purpose**: Scrape a specific web page for details such as name, address, email, website, and contact numbers.
- **Process**:
    1. Sends a GET request to the provided url using predefined headers.
    2. If the response status is 200 (OK), it parses the HTML using BeautifulSoup.
    3. Extracts various details such as name, address, and contact information (Telefon, Mobiltelefon, WhatsApp, Email, Website).
    4. Returns a dictionary with the extracted data.

3.2. read_db_to_dict_list(table)

- **Purpose**: Fetch unscreaped data from the SQLite database table.
- **Process**:
    1. Connects to the SQLite database and retrieves rows marked as Scraped=False.
    2. Converts these rows into a list of dictionaries.
    3. Returns the list for further processing.

3.3. Threading_Page_Urls(PageUrlsData, Scrapedtable, UpdatedTable)

- **Purpose**: Manages the concurrent scraping of multiple web pages using threads.
- **Process**:

1. Initializes a thread pool with a maximum of 50 workers.
2. Submits scraping tasks (retrivePages) for each URL in the provided list (PageUrlsData).
3. Uses a thread lock to safely update the database with scraped data.
4. Displays progress using alive_bar.
5. Returns the list of results after all tasks are completed.

3.4. sql_table_to_csv(UpdatedTable, csv_file_path)

- **Purpose**: Export the data stored in the SQLite table to a CSV file.
- **Process**:
    1. Retrieves all rows from the UpdatedTable.
    2. Converts the data into a pandas DataFrame.
    3. Writes the DataFrame to a CSV file at the specified path (csv_file_path).

4. Main Function

- **Purpose**: Orchestrates the entire process from database interaction to data export.
- **Process**:
    1. Connects to the SQLite database ScrapedData.db and initializes the required tables.
    2. Fetches data to be scraped using read_db_to_dict_list.
    3. Calls Threading_Page_Urls to start scraping the fetched URLs.
    4. Exports the final scraped data to a CSV file using sql_table_to_csv.

5. Execution Flow

- The script starts executing from the main() function when run directly.
- It connects to the database, fetches data, scrapes web pages concurrently, updates the database with the results, and finally exports the results to a CSV file.

Conclusion

This script effectively handles web scraping of multiple URLs concurrently, ensuring efficiency and safe data handling. It leverages threading for faster execution and includes mechanisms for progress tracking and database management.