Scalable Web Crawler

This project is a scalable web crawler built using Python, asyncio, and Celery. The crawler asynchronously fetches and parses web pages, extracts relevant content and stores it in a database. The project includes mechanisms for avoiding revisiting URLs, handling failed requests, and throttling requests.

Table of Contents

Installation

1. Ensure you have Python 3.9+ installed.
2. Clone the repository:

```
git clone https://github.com/aslitaser/web-crawler.git
```

3. Navigate to the project directory and install the required packages:

```
cd web-crawler pip install -r requirements.txt
```

4. Set up MongoDB and Redis as described in their respective documentation.

Usage

1. Start the MongoDB server (refer to the MongoDB documentation for your operating system): **https://docs.mongodb.com/manual/tutorial/manage-mongodb-processes/**

```
mongod
```

2. Start the Redis server by running the following command in your terminal:

```
redis-server
```

3. Start the Celery worker for distributed crawling:

```
celery -A celery_config worker --loglevel=info --concurrency=4 -Q crawl_worker
```

4. In a separate terminal, run the following command to start the web crawler with the seed URL:

```
main.py
```

To monitor the Celery worker and tasks, you can use Celery Flower. To start Flower, run the following command in a separate terminal:

```
celery flower -A web_crawler.tasks.crawl_worker
```

```
```

Then, navigate to http://localhost:5555 to access the Flower dashboard.

Project Structure

The project has the following structure:

```
.
├── README.md
├── celery_config.py
├── fetcher
│   └── fetcher.py
├── logger_config.py
├── logstash.conf
├── main.py
├── parsers
│   ├── base_parser.py
│   ├── default_parser.py
│   ├── image_parser.py
│   └── product_parser.py
├── requirements.txt
├── storage
│   └── storage.py
├── tests
│   ├── test_fetcher.py
│   ├── test_integration.py
│   ├── test_redis_url_frontier.py
│   ├── test_storage.py
│   └── test_url_deduplicator.py
├── url_deduplication
│   └── url_deduplication.py
├── url_frontier
│   └── redis_url_frontier.py
└── web_crawler
    ├── __init__.py
    └── tasks.py
```

The main components of the project are:

Modules

Fetcher

The Fetcher module (`fetcher/fetcher.py`) is responsible for asynchronously fetching URLs using the `httpx` library. It includes a retry mechanism and rate limiting functionality.

Parsers

The Parsers module contains various parser classes for extracting content and links from the fetched HTML content. The `BaseParser` class is an abstract class that serves as the foundation for other parser classes.

- `base_parser.py`: Contains the `BaseParser` class
- `default_parser.py`: Contains the `DefaultParser` class, which extracts links from the HTML content.
- `image_parser.py`: Contains the `ImageParser` class, which extracts image sources and titles from the HTML content.
- `product_parser.py`: Contains the `ProductParser` class, which extracts product-specific content from the HTML content.

Storage

The storage module (`storage/storage.py`) provides a way to store the extracted content in a MongoDB database. It includes methods for inserting content into the database and closing the database connection.

URL Deduplication

The URL Deduplication module (`url_deduplication/url_deduplication.py`) implements a Bloom Filter to prevent revisiting URLs that have already been visited. This module provides methods for adding URLs, checking if a URL is in the filter, and serializing the filter state for distributed use.

URL Frontier

The URL Frontier module (`url_frontier/redis_url_frontier.py`) manages the URL queue using Redis. This module provides methods for adding URLs, checking if a URL is in the queue, and retrieving the next URL to crawl. It also includes functionality for domain-specific wait times and adding URLs only if they haven't been seen before.

Web Crawler Task

The Web Crawler Task module (`web_crawler/tasks.py`) defines the main task for the web crawler, which is executed by Celery workers. This task fetches URLs, parses the content, adds new URLs to the queue, and stores the extracted content in the database.

Testing

The `tests` directory contains unit and integration tests for various modules of the project. To run the tests, execute the following command:

```
pytest tests
```

Logging

The logging configuration is set up in `logger_config.py`. Logs are generated with different levels (DEBUG, INFO, WARNING, ERROR) and are sent to different output streams based on their severity. The log files are stored in the `logs` directory.