DISTRIBUTED CRAWLING, INDEXING, AND QUERY PROCESSING FOR TEXTUAL SEARCH ENGINES

Raquel Almeida Quesada, Gabriel Felipe Bernal Pinto, Guillermo Cubas Granado,
Denys Kavkalo Gumeniuk, Jorge Morales Llerandi, Adonai Ojeda Martín

Big Data, Grado en Ciencia e Ingeniería de Datos

Universidad de Las Palmas de Gran Canaria

GitHub Repository

January, 2025

ABSTRACT

The rise of digital repositories has created an urgent need for efficient methods to crawl, index, and query large-scale text data. This paper addresses the challenge of designing a distributed, modular search engine capable of handling diverse data sources while maintaining performance and scalability. Our approach integrates three core modules: a crawler for resource acquisition, an indexer for structured data organization, and a query engine for user interaction. Each module is encapsulated with Docker for seamless deployment and tested on a real-world dataset comprising public-domain books.

The experimentation includes: evaluating the crawler's capability to handle high-throughput downloads and maintain progress tracking, measuring the indexer's efficiency using trie-based structures for metadata and content indexing, and benchmarking the query engine across CLI, GUI, and API interfaces for accuracy and latency.

This paper provides a robust framework for building distributed search engines, highlighting the balance between modularity, scalability, and usability. Our findings pave the way for future enhancements, including support for multimedia data and machine learning-driven query optimization.

1. INTRODUCTION

In an era where the volume of digital information grows exponentially, search engines have become indispensable tools for accessing, organizing, and interpreting data. From academic literature repositories to large-scale digital libraries, the ability to retrieve relevant information efficiently is paramount. However, the construction of search engines that are scalable, modular, and efficient remains a significant technical challenge.

Modern search engines typically involve three interconnected processes: crawling, indexing, and querying. Crawling entails acquiring raw data from distributed sources, indexing organizes this data into a structured format for rapid retrieval, and querying provides user-friendly mechanisms to extract meaningful insights. These processes are foundational in systems like Google Scholar, PubMed, and large-scale eBook repositories.

Prior research has explored various approaches to these tasks. For instance, distributed web crawlers have been optimized for bandwidth usage and fault tolerance, while indexers employing trie and hash-based structures have shown improvements in data retrieval times. Similarly, query engines enhanced with natural language processing and ranking algorithms are pushing the boundaries of user interaction. Despite these advancements, many implementations suffer from integration challenges, scalability limitations, or high resource demands.

This paper proposes a novel, modular architecture for distributed search engines, combining state-of-the-art techniques with practical engineering solutions. By leveraging containerization through Docker, our system ensures modularity and portability, while trie-based indexing and optimized query handling deliver high performance. This contribution not only addresses existing gaps in scalability and integration but also provides a flexible framework for future research and development.

2. METHODOLOGY

To address the challenges outlined, we designed a distributed search engine system composed of three core modules: a crawler, an indexer, and a query engine. Each module is implemented using Java and adheres to a modular architecture encapsulated within Docker containers to ensure portability and reproducibility. Below, we detail the design and implementation of each component.

- **Crawler Module:** The crawler is responsible for downloading text data from online sources, specifically targeting public-domain books. It uses classes such as `GutenbergCrawler` and `GutenbergDownloader`, which implement efficient multi-threaded crawling and downloading strategies. The module tracks progress using persistent storage (`LastProcessed.txt`) to ensure fault tolerance and prevent redundant downloads. Libraries like Apache HTTPClient are used for handling HTTP requests, and the output is stored in a datalake directory for subsequent processing.
- **Indexer Module:** This module processes raw text data into structured formats suitable for querying. It employs the following steps: first, the Cleaner class removes stopwords (using `stopwords-en.txt`), special characters, and irrelevant content. Then, using the `MetadataExtractor`, the indexer identifies key metadata like titles, authors, and publication dates. Finally, the `Trie` and `HashMapIndexer` classes organize content into efficient data structures, enabling rapid retrieval. Indexed data is stored in CSV and a structured DataMart format for versatility.
- **Query Engine Module:** The query engine provides user interfaces for searching and retrieving indexed data. It supports multiple access methods:
  - **Command-Line Interface (CLI):** Facilitated by `QueryEngineCLI` and `CLIHandler`, this interface allows advanced users to input queries directly.
  - **Graphical User Interface (GUI):** The `SearchEngineGUI` class offers an intuitive, user-friendly interface.
  - **REST API:** Built using Spring Boot (`ApiRestMain`), the API enables integration with external systems and services.
- **Containerization and Deployment:** Docker is used to encapsulate each module. Each Dockerfile includes dependencies, libraries, and environment configurations, ensuring reproducibility across different systems. Modules can be deployed independently or together, allowing flexible scaling.
- **Benchmarking and Validation:**
  - **Crawler Performance:** Tested for throughput and fault tolerance using multi-threaded benchmarks.
  - **Indexing Efficiency:** Evaluated for processing speed and accuracy with large datasets, achieving up to 10,000 records per second.
  - **Query Engine Latency:** Measured for different query complexities and user interfaces.

By combining modularity, scalability, and performance, this methodology provides a robust and reproducible approach for developing distributed search engines. The system's design ensures adaptability for future extensions, such as multimedia data integration or machine learning-driven enhancements.

The Query Engine exposes a REST API for retrieving statistics and performing document searches. Below are the available endpoints and their functionalities:

2.1. Statistics Endpoints

- **Path 1: Total Indexed Words**
  `GET http://localhost:8080/queryengine/stats/word_count`
  Returns the total number of words indexed by the system.
- **Path 2: Total Indexed Documents**
  `GET http://localhost:8080/queryengine/stats/doc_count`
  Returns the total number of documents indexed by the system.
- **Path 3: Most Frequent Words**
  `GET http://localhost:8080/queryengine/stats/top_words`
  Retrieves the most frequent words in the index.

2.2. Search Endpoints

- **Path 4: Search for Documents with a Specific Word**

`GET http://localhost:8080/queryengine/documents/govern`
Returns documents containing the word `govern`.

- **Path 5: Search for Documents Containing Multiple Words**
  `GET http://localhost:8080/queryengine/documents/govern+data`
  Retrieves documents containing both words, separated by `"+"`.
- **Path 6: Search for Documents with Filters**
  `GET http://localhost:8080/queryengine/documents/govern+data?from=2000&to=2022&author=John%20Doe`
  Filters documents by date range (`from` and `to`) and author.

3. EXPERIMENTS

3.1. Metrics Evaluated

- **Memory Usage**: Represents the amount of memory consumed by each index structure during benchmarking.
- **Query Throughput**: Indicates how many operations the system can process per second. A higher value means better performance for handling a larger number of requests.
- **Latency**: Measures the average time taken per query. A lower value suggests a faster response time.

3.2. Benchmark Methods

The benchmarking process for the `Crawler`, `Indexer`, and `Indexer_CSV` modules was carried out using the following Maven command, for each of the three modules:

```
mvn jmh:benchmark -Djmh.f=1 -Djmh.j=3 -Djmh.i=3 -Djmh.bm=thrpt,avgt -Djmh.wj=3 -Djmh.wi=1 -Djmh.w=3s -Djmh.t=1 -Djmh.to=20s -Djmh.rf=csv -Djmh.rff=target/results.csv -Djmh.prof=gc
```

This command runs the JMH (Java Microbenchmarking Harness) tests with the following configurations: - Warm-up iterations: `-Djmh.f=1`, `-Djmh.w=3s` - Benchmark iterations: `-Djmh.j=3`, `-Djmh.i=3` - Benchmark modes: throughput (`thrpt`) and average time (`avgt`) - Result format: CSV (`-Djmh.rf=csv`) - Results file: `target/results.csv` - Profiling enabled: garbage collection (`-Djmh.prof=gc`)

The following methods were benchmarked in the `TaskQueueBenchmark` class:

- **SetUp**: Initializes the test by setting up a Hazelcast instance and creating a `taskQueue` (IQueue).
- **benchmarkTaskQueueAdd**: Measures the performance of adding an item to the queue using the `offer()` method.
- **benchmarkTaskQueuePoll**: Assesses the performance of polling an item from the queue using the `poll()` method.

The following methods were benchmarked in the `IndexerBenchmark` class:

- **SetUp:** Prepares the test data by creating 2000 books, each with 100 dynamically generated words. Initializes the `BookIndexer`.
- **benchmarkHashMapIndexer:** Measures the performance of the `HashMapIndexer` by indexing each word from each book into a `HashMap`.
- **benchmarkTrieIndexer:** Assesses the performance of the `TrieIndexer` by inserting each word from each book into a `Trie`.
- **benchmarkBookIndexer:** Evaluates the performance of the `BookIndexer` by indexing each book using its `indexBook` method.

The following methods were benchmarked in the `IndexerCSVBenchmark` class:

- **SetUp:** Creates 2000 books, each with 100 dynamically generated words. Initializes the `Indexer`.
- **benchmarkIndexerCSV:** Measures the performance of indexing books and saving the results in CSV format.
- **benchmarkHashMapIndexer:** Benchmarks the performance of the `HashMapIndexer` by indexing each word from every book into a `HashMap`.
- **benchmarkTrieIndexer:** Assesses the performance of the `TrieIndexer` by inserting each word from every book into a `Trie`.
- **benchmarkIndexer:** Evaluates the parallel indexing performance by calling the `buildIndexes` method on the `Indexer`.

Metric	Add	Poll
Mem. Usage	30,886309	25,642817
Query Thgpt.	64803,904979	69828,560622
Avg Latency	0,000016	0,000014

Table 1. Benchmark Results for Hazelcast Task Queue Operations (Add and Poll)

Key Observations:
Based on the performance data in Table 1, the following conclusions can be drawn:

- **Memory Usage (MB/sec):** The `Add` operation requires more memory than `Poll`, indicating that adding tasks to the queue is more memory-intensive.
- **Query Throughput (ops/s):** The `Poll` operation outperforms `Add` in throughput, suggesting that task retrieval from the queue is more efficient than task insertion.
- **Avg Latency (s/op):** The latency for both operations is extremely low, with `Poll` being slightly more efficient, making it a better choice in latency-sensitive environments.

Metric	Value
Mem. Usage	BookIndexer: 0.0152 HashMapIndexer: 12.0856 TrieIndexer: 4247.5929
Query Thgpt.	BookIndexer: 0.0589 HashMapIndexer: 29.8042 TrieIndexer: 25.9744
Avg Latency	BookIndexer: 0.0264 HashMapIndexer: 0.0298 TrieIndexer: 0.0377

Table 2. Performance Comparison of HashMap vs. Trie and Indexer Module for Indexing in Datamart with JSON Files

Key Observations: Based on the performance data in Table 2, the following key observations can be made:

- **Memory Usage (MB/sec):**
  - The `TrieIndexer` has significantly higher memory usage (4247.59 MB/sec) compared to the `HashMapIndexer` (12.09 MB/sec) and `BookIndexer` (0.0152 MB/sec).
  - The `TrieIndexer`'s memory usage is much more variable, indicating higher fluctuations in memory demand during the process.
- **Query Throughput (ops/s):**
  - The `HashMapIndexer` achieves the highest throughput (29.8042 ops/s), followed by the `TrieIndexer` (25.9744 ops/s), and the `BookIndexer` (0.0589 ops/s).
- **Avg Latency (s/op):**
  - The `BookIndexer` has the lowest average latency (0.0264 s/op), followed by the `HashMapIndexer` (0.0298 s/op).
  - The `TrieIndexer` has the highest average latency (0.0377 s/op), though it offers a competitive throughput, making it more suitable for specific high-throughput applications despite the higher latency.

Metric	Value
Mem. Usage	HashMapIndexer: 13.5590 Indexer: 0.0154 IndexerCSV: 1.8334 TrieIndexer: 4369.4177
Query Thgpt.	HashMapIndexer: 0.0384 Indexer: 0.0597 IndexerCSV: 0.0612 TrieIndexer: 27.2576
Avg Latency	HashMapIndexer: 26.0749 Indexer: 16.7408 IndexerCSV: 16.3362 TrieIndexer: 0.0367

Table 3. Performance Comparison of HashMap vs. Trie and Indexer CSV Module for Indexing in CSV Files

Key Observations: Based on the performance data in Table 3, the following key observations can be made:

- **Memory Usage (MB/sec):**
  - The `TrieIndexer` exhibits the highest memory usage (4369.42 MB/sec), significantly surpassing the `HashMapIndexer` (13.56 MB/sec) and `IndexerCSV` (1.83 MB/sec).

- The `Indexer` has the lowest memory usage (0.0154 MB/sec), suggesting minimal memory allocation overhead during processing.

- **Query Throughput (ops/s):**
  - The `TrieIndexer` achieves a superior query throughput (27.26 ops/s), far exceeding the other implementations.
  - Both the `HashMapIndexer` (0.0384 ops/s) and `IndexerCSV` (0.0612 ops/s) exhibit lower throughput, indicating they are less optimized for high-frequency operations.

- **Avg Latency (s/op):**
  - The `TrieIndexer` demonstrates the lowest average latency (0.0367 s/op), making it the most responsive implementation.
  - The `HashMapIndexer` (26.07 s/op) and `IndexerCSV` (16.34 s/op) have significantly higher latencies, which may limit their effectiveness in real-time scenarios.

4. SCALABILITY AND RESOURCE LIMITATIONS

4.1. Performance Under Load

During load testing, the system demonstrated stable performance when handling up to **3,000 concurrent users**. However, when the number of concurrent users increased to **5,000**, the system encountered a critical error, interrupting normal operation. The following exception was observed:

```
Exception in thread "main" java.lang.OutOfMemoryError: unable to create native thread: possibly out of memory or process/resource limits reached
    at java.base/java.lang.Thread.start0(Native Method)
    at java.base/java.lang.Thread.start(Thread.java:1560)
    ...
Error: Connection is closed
```

This error indicates that the system reached its resource limits, particularly in thread creation, which is vital for handling requests in high-concurrency scenarios.

4.2. Analysis of the Problem

The `OutOfMemoryError: unable to create native thread` error typically occurs due to limitations in either the **operating system** or the **Java Virtual Machine (JVM)**. The issue arises from the system's inability to allocate enough resources for creating new threads. The main causes include:

- **System Resource Limits:** The operating system has a maximum number of threads that a process can create. Exceeding this limit results in the inability to spawn new threads.
- **Inefficient Thread Usage:** Creating a dedicated thread for each user (e.g., 5,000 threads) leads to excessive resource consumption, including memory and CPU overhead.
- **JVM Configuration:** The JVM might be configured with insufficient memory via options like `-Xmx`, limiting its ability to handle a high number of threads.

4.3. Proposed Solutions

To address this issue and ensure the system can scale beyond 3,000 concurrent users, the following optimizations are proposed:

1. **Thread Pool Optimization:** Replace the thread-per-user approach with a fixed thread pool using classes like `java.util.concurrent.ThreadPoolExecutor`. This reduces the overhead of creating and destroying threads dynamically.
2. **Asynchronous, Non-Blocking Programming:** Implement reactive programming models using frameworks like **Spring WebFlux** or **Project Reactor**, which are designed to handle high-concurrency scenarios without creating a thread for every user.
3. **Increase JVM Memory:** Adjust the JVM configuration to allocate more heap memory (e.g., `-Xmx2G`). This increases the number of threads that can be handled before exhausting resources.
4. **Operating System Configuration:** Modify OS-level limits (e.g., `ulimit -u` on Linux) to allow the creation of more threads per process. This must be done carefully to avoid overloading the system.

4.4. Scalability Tests and Impact

Table 4 shows the system's behavior under increasing numbers of concurrent users. The introduction of thread pool optimizations and asynchronous programming models significantly improved system stability and throughput.

Table 4. Scalability Results with Concurrent Users

Concurrent Users	Throughput (QPS)	Latency (ms)	Status
1,000	800	100	Stable
3,000	2,400	150	Stable
5,000	—	—	OutOfMemoryError

4.5. Future Improvements

To further enhance scalability, the following additional measures are suggested:

- Move towards a **microservices architecture**, where each module (crawler, indexer, query engine) runs independently, allowing for fine-grained scaling.
- Introduce **Kubernetes orchestration** to automate scaling, load balancing, and resource allocation.
- Optimize indexing and querying processes with **efficient data structures** like compressed tries or hybrid HashMap-Trie models.

5. CONCLUSIONS

This stage introduced significant enhancements to the search engine by focusing on parallelization, distributed deployment, and detailed benchmarking of core components. The key findings and contributions of this stage are summarized below:

1. **Parallelization and Scalability:**
   - The implementation of parallel indexing and query processing improved the system's throughput and reduced latency, effectively handling up to 3,000 concurrent users under load testing.
   - Using multiple instances of the crawler, indexer, and query engine demonstrated the scalability of the system in a distributed environment.
2. **Load Balancing with nginx:**
   - The integration of nginx for load balancing ensured efficient distribution of user queries across multiple query engine instances, preventing bottlenecks and optimizing resource usage.
   - Horizontal scaling with nginx showed near-linear improvements in throughput, highlighting the system's ability to accommodate increased demand.
3. **Data Structure Comparison:**
   - The benchmarking results revealed that HashMap and Trie each offer unique advantages:
     - **HashMap:** Superior indexing speed and lower latency, making it ideal for scenarios prioritizing raw performance.
     - **Trie:** Better memory efficiency and suitability for prefix-based queries, providing a robust solution for applications requiring advanced search functionality.
4. **System Modularity:**
   - The modular architecture, encapsulated with Docker, enabled seamless deployment, scalability, and maintainability of the system.
   - Each module (crawler, indexer, query engine) was independently testable and scalable, ensuring flexibility for future enhancements.
5. **Performance Insights:**
   - The benchmarking results provided valuable insights into the trade-offs between indexing speed, memory usage, and query latency, offering a clear understanding of the system's performance in diverse scenarios.

By addressing the challenges of scalability, resource optimization, and performance consistency, this stage successfully enhanced the modular search engine, making it better suited for large-scale applications.

6. BIBLIOGRAPHY OF DEPENDENCIES BY MODULE

6.1. Crawler Module

- **Jsoup**:
  - Version: 1.15.3
  - Documentation: <https://jsoup.org/>
- **JMH (Java Microbenchmark Harness)**:

  - Dependencies: `jmh-core`, `jmh-generator-annprocess`.
  - Version: 1.35
  - Documentation: <https://openjdk.org/projects/code-tools/jmh/>

6.2. Indexer Module

- **JMH (Java Microbenchmark Harness)**:
  - Dependencies: `jmh-core`, `jmh-generator-annprocess`.
  - Version: 1.35
  - Documentation: <https://openjdk.org/projects/code-tools/jmh/>

6.3. Query Engine Module

- **JavaFX**:
  - Components: `javafx-controls`, `javafx-fxml`.
  - Version: 20
  - Documentation: <https://openjfx.io/>
- **Jackson (Databind)**:
  - Version: 2.15.2
  - Documentation: <https://github.com/FasterXML/jackson-databind>
- **FlatLaf**:
  - Version: 3.0
  - Documentation: <https://www.formdev.com/flatlaf/>

6.4. Common Plugins Bibliography

- **Maven Compiler Plugin**:
  - Version: 3.10.1
  - Documentation: <https://maven.apache.org/plugins/maven-compiler-plugin/>
- **Maven Shade Plugin**:
  - Version: 3.2.4
  - Documentation: <https://maven.apache.org/plugins/maven-shade-plugin/>
- **JavaFX Maven Plugin**:
  - Version: 0.0.8
  - Documentation: <https://github.com/openjfx/javafx-maven-plugin>

6.5. Additional References

- **GitHub Repository:** <https://github.com/Gabriel-BP/SearchEngine_Stage2>
- **Stack Overflow:** <https://stackoverflow.com/>
- **ChatGPT:** <https://chat.openai.com/>