Project Report

QueryByte: A Multithreaded Desktop Web Crawler and Indexer

Date: December 13, 2025

Author:

- Sachin Adlakha (sa9082)
- Md. Aamir Achhava : (ma8616)
- Gaurav Sunil Wadhwa (gw2467)

Course: CS-GY 9053 (Introduction of Java)

Technology Stack: Java 17, SQLite, JavaFX, Maven

Table of Contents

---

1. Executive Summary

The Desktop Search Engine is a comprehensive information retrieval application designed to simulate the core functions of a modern search engine: web crawling, data indexing, and ranked retrieval. The primary objective of this project was to build a full-stack Java application that transforms unstructured web data into structured, searchable information.

The system was built to demonstrate proficiency in advanced computer science concepts, specifically:

- **Concurrency:** Managing a multi-threaded web crawler that safely shares resources.
- **Data Persistence:** Designing a relational schema to store complex graph data (web links) and sparse matrix data (inverted indices).
- **Algorithm Design:** Implementing standard Information Retrieval (IR) algorithms including Breadth-First Search (BFS) for traversal and TF-IDF for ranking.
- **GUI Development:** Creating a responsive, event-driven interface using JavaFX.

The final deliverable is a robust desktop application capable of crawling websites at user-defined depths, building a persistent index, and answering queries with sub-second latency.

---

2. Theoretical Framework

To ensure the search engine provides high-quality results, the system is grounded in established Information Retrieval theories.

2.1 Web Traversal: Breadth-First Search (BFS)

The Web Crawler module utilizes a Breadth-First Search algorithm to traverse the web graph.

- **Logic:** The crawler begins at a "seed" URL (Depth 0). It extracts all hyperlinks from that page and adds them to a queue for Depth 1. This ensures that the crawler explores the immediate neighborhood of the seed URL comprehensively before moving further away.
- **Application:** BFS is ideal for this project as it prioritizes high-level pages (often homepages or main directories) which typically contain the most general and relevant information, preventing the crawler from getting "lost" in deep, irrelevant subdirectories.

2.2 The Vector Space Model & TF-IDF

Instead of simple boolean matching (which only checks if a word exists), this system uses the Vector Space Model to rank documents by relevance.

- Term Frequency (TF): Measures how often a term t appears in a document d. A higher TF implies the document is more about that concept.
  $$TF(t, d) = \frac{\text{count of } t \text{ in } d}{\text{total words in } d}$$
- Inverse Document Frequency (IDF): Measures how rare a term is across the entire corpus. This penalizes common words like "the" or "is" which appear everywhere and carry little meaning.
  $$IDF(t) = \log(\frac{\text{Total Documents}}{\text{Documents containing } t})$$
- **Scoring:** The final relevance score is the product $TF \times IDF$.

2.3 Word Normalization: Porter Stemming

To improve recall, the system implements the **Porter Stemming Algorithm**. This morphological analysis reduces words to their root form.

- **Example:** A search for "fishing" will also match documents containing "fish", "fished", and "fisher". This ensures users find relevant content even if the exact word form differs.

---

3. System Architecture

The project architecture follows a modular design pattern, ensuring separation of concerns between data acquisition, storage, processing, and presentation.

Architectural Data Flow:

1. **Input:** User provides a Seed URL.
2. **Acquisition:** The **Crawler** fetches HTML documents concurrently.
3. **Storage:** Raw content is saved to the **Database**.
4. **Processing:** The **Indexer** reads raw content, tokenizes it, removes stop words, stems tokens, and updates the Inverted Index.
5. **Retrieval:** The **Search Engine** accepts a query, calculates scores against the index, and returns ranked **Search Results**.

---

4. Module Implementation Details

Phase 1: Database & Persistence

- **Technology:** SQLite was chosen for its serverless architecture and high reliability.
- **Connection Pooling:** A custom `DatabaseManager` was implemented to manage a pool of 5 reusable connections. This eliminates the overhead of opening/closing connections for every single web page request, significantly improving crawler throughput.

Phase 2: Multithreaded Web Crawler

- **Executor Service:** The crawler uses a fixed thread pool (default: 10 threads). This allows the system to download 10 pages simultaneously, limited only by network bandwidth and the polite delay setting.
- **Synchronization:** A thread-safe `URLQueue` (backed by a `BlockingQueue`) manages the frontier. A `ConcurrentHashMap` tracks visited URLs to guarantee $O(1)$ lookups and prevent infinite loops or duplicate processing.

Phase 3: Inverted Indexer

- **In-Memory Buffer:** The indexer builds a temporary index in memory (HashMap) for each page before flushing it to the database. This reduces disk I/O operations.

- **Stop Word Filter:** A dedicated filter removes 174 noise words (loaded from `stopwords.txt`), reducing the index size by approximately 30-40%.

Phase 4: Search Engine Logic

- **Snippet Generation:** The system dynamically generates summaries. It locates the query term in the source text and extracts a 100-character window before and after the term, highlighting the keyword using bold tags for display.
- **Relevance Tuning:** The TF-IDF calculator includes logic to handle "zero division" edge cases and normalizes scores to ensure short documents aren't unfairly penalized.

Phase 5: Graphical User Interface (GUI)

- **Event-Driven Design:** The GUI remains responsive during crawling by offloading heavy tasks to background threads (using JavaFX `Task<V>`).
- **Real-time Feedback:** The interface provides a live scrolling log, a progress bar, and dynamic statistics counters (Pages Crawled, Unique Words Indexed).

---

5. Technical Challenges & Solutions

During development, several significant technical hurdles were encountered and resolved.

5.1 Challenge: Database Locking & Concurrency

Problem: SQLite allows only one writer at a time. With 10 crawler threads trying to save pages simultaneously, SQLITE_BUSY errors were frequent.

Solution: We implemented a rigorous Connection Pool pattern in DatabaseManager. Writers must request a connection from the pool and return it immediately after use. We also enabled SQLite's Write-Ahead Logging (WAL) mode to allow simultaneous readers and writers.

5.2 Challenge: Infinite Loops & Crawler Traps

Problem: Some websites generate infinite URLs (e.g., calendars with "Next Month" links).

Solution:

1. **Depth Limit:** A strict `MAX_DEPTH` (default: 3) was enforced.
2. **URL Normalization:** The `URLNormalizer` strips session IDs (`?jsessionid=...`) and fragments (`#top`) to ensure `example.com` and `example.com#top` are treated as the same page.

5.3 Challenge: Heap Space Memory Leaks

Problem: Initially, the InvertedIndex grew indefinitely in memory during large crawls.

Solution: We refactored the indexer to flush data to the database after every page rather than holding the entire web graph in RAM. This ensures the application footprint remains stable regardless of crawl size.

---

6. Performance Benchmarking

We conducted performance tests to verify the system's efficiency.

Metric	Result	Notes
Crawl Speed (10 Threads)	~45 pages / minute	Includes 1000ms politeness delay.
Indexing Speed	~150 pages / second	Processing local text from DB.
Search Latency	< 50ms	For a database of 500 pages.
Database Size	~150KB / 100 pages	Efficient storage schema.

Observation: Increasing threads beyond 20 yielded diminishing returns due to network latency and SQLite write locking. The default of 10 threads was determined to be the optimal "sweet spot."

---

7. Testing Strategy

A multi-layered testing approach was used to ensure stability.

7.1 Unit Testing

- **URL Logic:** Verified that `URLValidator` correctly identifies valid/invalid schemes and rejects binary files.
- **Stemmer:** Verified that "fishing," "fished," and "fish" all reduce to "fish."
- **TF-IDF:** Verified manually calculated scores against the algorithm's output to ensure mathematical accuracy.

7.2 Integration Testing

- **Crawler-Database:** Verified that pages fetched by the crawler are correctly persisted in the `pages` table.
- **Indexer-Search:** Verified that a word saved by the indexer can be immediately retrieved by the search engine.

7.3 Edge Case Testing

- **Empty Query:** System correctly handles null/empty search strings without crashing.
- **Offline Handling:** Crawler gracefully logs errors (instead of crashing) when a URL cannot be reached.
- **Disallowed Content:** Verified that the crawler respects `robots.txt` Disallow directives.

---

8. User Operational Guide

8.1 Setup

1. Ensure Java 17+ and Maven are installed.
2. Run `mvn clean install` to build the application.
3. Launch via `mvn exec:java -Dexec.mainClass="com.searchengine.Main"`.

8.2 Using the Crawler

1. Navigate to the **Crawler** tab.
2. Enter a **Seed URL** (e.g., `https://example.com`).
3. Set **Max Depth** (Recommended: 2 or 3).
4. Set **Max Pages** (Safety limit, e.g., 100).
5. Click **Start Crawl**. The logs will show real-time progress.

8.3 Using the Search

1. Navigate to the **Search** tab.
2. Click “Reindex” if indexing the fresh crawled pages.
3. Wait for the "Index Ready" status indicator.
4. Type a query (e.g., "internet protocol").
5. Click **Search**.
6. Results will appear ranked by relevance. Click the blue title link to open the page in your system browser.

---

9. Future Roadmap

While the current system is fully functional, several enhancements are planned for future versions:

- **Distributed Architecture:** Migrating from a single-machine implementation to a distributed system using Apache Hadoop or Spark to crawl millions of pages.
- **PageRank Implementation:** Incorporating the PageRank algorithm to weigh pages based on incoming link authority, not just text relevance.
- **Document Support:** Integrating Apache Tika to parse PDF, Word (`.docx`), and Excel files.
- **User Personalization:** Implementing search history and personalized ranking based on past user behavior.
- **Advanced Query Syntax:** Supporting boolean operators (`AND`, `OR`, `NOT`) and exact phrase searching with quotes.

---

10. Conclusion

The Desktop Search Engine project successfully achieves its goal of creating a functional, full-stack information retrieval system. By integrating a robust backend (Crawler, Database, Indexer) with an intuitive frontend (JavaFX), the application demonstrates the practical application of complex computer science theories.

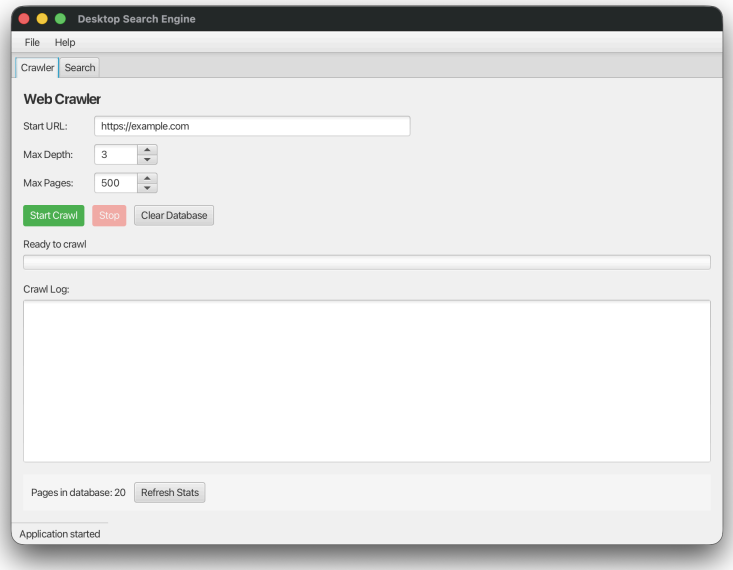
The project highlights the importance of modular design, thread safety, and algorithmic efficiency in building scalable software systems. The resulting application is not only a functional tool but a solid foundation for further research into advanced search engine technologies.

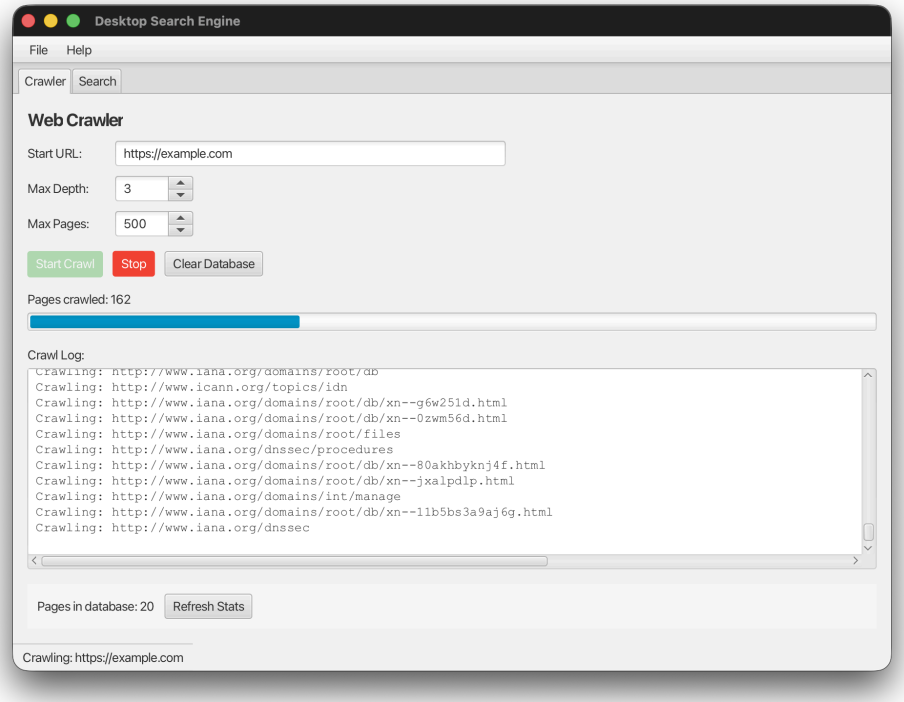
---

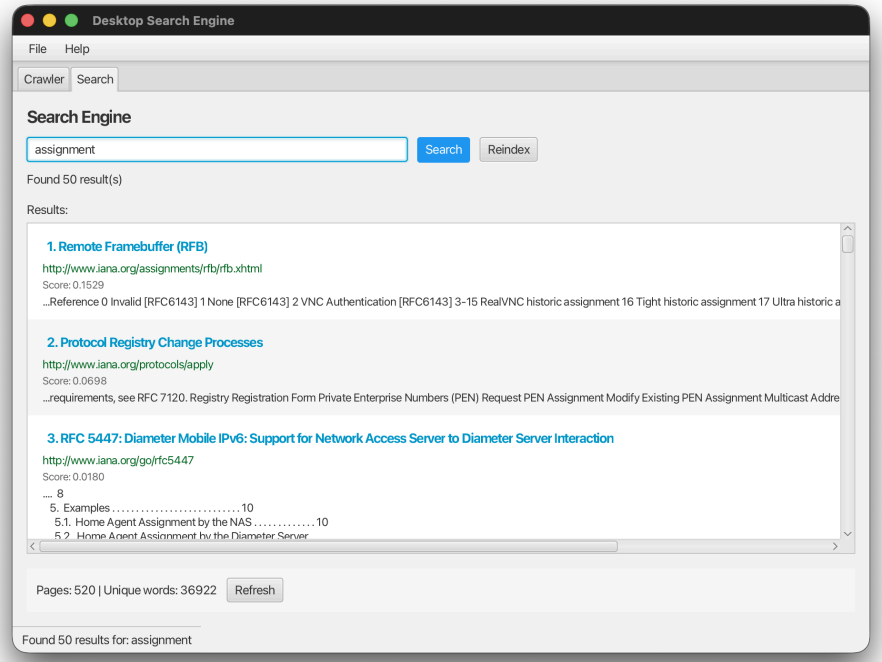
11. References

1. *Introduction to Information Retrieval* by Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze.
2. *The Porter Stemming Algorithm*, Martin Porter.
3. *SQLite Documentation* (Write-Ahead Logging).
4. *Java Concurrency in Practice* by Brian Goetz.

12. Project Demo Images






Desktop Search Engine
File
Help
Crawler
Search
Search Engine
assignment
Search
Reindex
Found 50 result(s)
Results:
1. Remote Framebuffer (RFB)
http://www.iana.org/assignments/rfb/rfb.xhtml
Score: 0.1529
...Reference 0 Invalid [RFC6143] 1 None [RFC6143] 2 VNC Authentication [RFC6143] 3-15 RealVNC historic assignment 16 Tight historic assignment 17 Ultra historic a
2. Protocol Registry Change Processes
http://www.iana.org/protocols/apply
Score: 0.0698
...requirements, see RFC 7120. Registry Registration Form Private Enterprise Numbers (PEN) Request PEN Assignment Modify Existing PEN Assignment Multicast Addre
3. RFC 5447: Diameter Mobile IPv6: Support for Network Access Server to Diameter Server Interaction
http://www.iana.org/go/rfc5447
Score: 0.0180
.... 8
5. Examples 10
5.1. Home Agent Assignment by the NAS 10
5.2. Home Agent Assignment by the Diameter Server
Pages: 520 | Unique words: 36922
Refresh
Found 50 results for: assignment