# Short Report

## 1. Overview

This document provides a concise overview of the Java-based web crawler and information retrieval system you have implemented. It covers four main areas:

**1. Approach to Designing the Crawler**

**2. Construction of the Inverted Index**

**3. Computation of TF–IDF and Cosine Similarity**

**4. Challenges Encountered and Solutions**

## 2. Approach to Designing the Crawler

The *WebCrawler* class uses *Jsoup* for HTML fetching and parsing. The key design points are:

* **Depth-First Recursion with Limits:** A recursive *crawl(String url, int depth)* method tracks both the remaining depth and a global page count. Crawling stops when either the maximum depth is reached or the configured *maxPages* limit is met.
* **URL Normalization & Deduplication**: URLs are stored in a *HashSet<String> visited* to avoid revisiting the same page. Each URL is normalized by removing the protocol and replacing non-alphanumeric characters when naming the output file.
* **Storage of Page Content**: Crawled pages are stored as text files under a *pages* directory. Each file begins with the URL followed by the page’s raw body text, obtained via *doc.body().text().*
* **Link Extraction:** After saving the current page, the crawler selects all `<a href>` links, resolves them to absolute URLs via `l.absUrl("href")`, and recurses.

This design balances simplicity and robustness, allowing easy extension (e.g., politeness delays, robots.txt compliance) if needed.

---

## 3. Building the Inverted Index

The `InvertedIndex` class encapsulates all logic for building and querying a term-based index:

1. \*\*Directory Traversal\*\*: `BuildInvertedIndx(String folderPath)` scans the `pages` directory for `.txt` files.

2. **Content Reading & Tokenization**:

* `ReadContent(File file)` reads entire files into a single string.
* `TextProcessing.processText(String text)`:
* Removes Wikipedia-style citations (e.g., `[1]`).
* Normalizes tokens by lowercasing and stripping special characters.
* Filters out stop words, purely numeric tokens, and tokens shorter than three characters.
* Applies Porter stemming (`Stemmer` class) to reduce each token to its root form.

3**. Term Frequency Counting:** For each document, a `Map<String,Integer> TF` collects the frequency of each stemmed token.

4. **Posting List Construction:** For each term, a `Posting` object `(docId, TF)` is created and added to `InvertedIndx` under that term’s entry.

This inverted index structure enables efficient retrieval of all documents containing a given term, along with term frequency information needed for ranking.

---

## 4. TF–IDF and Cosine Similarity Computation

The `TFIDFCalculator` class provides methods to weight terms and rank documents.

**1. Document Vector Computation (`computeDocumentVector`) :**

* **Term Frequency Weighting:** For each term `t` in document `d`, compute (tf\_{t,d} = 1 + log10(TF)).
* **Inverse Document Frequency:** With (N) total documents and document frequency (df\_t) (# of postings for term (t)), compute (idf\_t = log10(N/df\_t)).
* **TF–IDF Weight:** Multiply to get weight (w\_{t,d} = tf\_{t,d} \* idf\_t).

**2. Query Vector Computation (`computeQueryVector`) :**

* Tokenizes the input query, computes TF for each query term, applies the same weighting formula.

**3. Cosine Similarity (`cosineSimilarity`) :**

* Given two vectors (v) (document) and (q) (query), compute

cos(θ) = (Σ v\_t q\_t) / (||v|| ||q||).

* Terms not present default to zero weight. This normalized dot product ranks how closely a document matches the query.

**4. Ranking (`rankDocuments`) :**

* Computes the similarity score for each document, sorts in descending order, and returns the top 10 document IDs.

These computations enable a classic vector-space retrieval model using TF–IDF weighting.

---

## 5. Challenges and Solutions

**1. Handling HTML Noise:** Early tests showed that raw text extraction included navigational and boilerplate text. Solution: Focused on `doc.body().text()` and relied on stop words + stemming to minimize noise impact.

**2. Term Normalization:** Documents contained punctuation, citations, and mixed casing. Solution: A dedicated `TextProcessing` pipeline with regex filters, lowercasing, and a Porter stemmer ensured consistent token forms.

**3. Recursion Depth vs. Page Limit:** Without careful checks, crawling could exceed desired limits or recuse infinitely. Solution: Combined both `depth` parameter and a global `currentCount` against `maxPages`, with immediate returns if exceeded.

**4. Efficient Similarity Calculation:** Computing all pairwise similarities naively can be slow for many documents. While our implementation processes each document on demand, a production system might precompute norms or use sparse vector structures.

---

# User Manual

## Prerequisites

* Java 8+
* Jsoup library on the classpath

## Setup

1. Compile all `.java` files under `com.example` and `invertedIndex` packages.

2. Create an empty folder named `pages` in the working directory.

## Running the Crawler

java com.example.WebCrawler <start-URL> <max-depth> <max-pages>

## Building the Index and Querying

java com.example.InvertedIndex pages

java com.example.MainQueryApp "your search terms here"

## Example Main Program

When you run:

java com.example.Main

It crawls, builds the index, prints the inverted index, computes TF–IDF for a document and query, and ranks the top 10 documents.