Information Retrieval Systems

Mini Keyword-Based Search Engine

Zbigniew Dendzik

March 2026

Lecture Plan

- Introduction to IR systems
- Text document processing
- Document class – document representation
- Inverted index
- Single and multi-word queries
- AND and OR operators
- Document ranking by relevance
- Extensions: TF-IDF
- Modern IR systems and AI

Information Retrieval (IR) Systems

Information Retrieval – systems that return documents best matching a query.

Application examples:

- Web search engines (Google, Bing)
- Technical documentation search
- Email or notes search
- Digital library search

Modern systems:

- Use large language models (LLM)
- Understand context and semantic relations
- Based on vector embeddings

Our System – Mini Search Engine

Goal: Build a simple keyword-based system.

Main components:

- **Document class** – represents a single text document
- **Inverted index** – for each word a list of documents where it appears
- **Queries** – single and two-word search
- **Ranking** – sorting results by relevance

Starting point:

- Simple keyword matching
- Counting word frequencies
- Foundation for understanding advanced IR systems

Document Representation – Document Class

Document consists of:

Private fields:

- `int id` – unique identifier
- `String title` – document title
- `String body` – document content

Public methods:

- `getId()`, `getTitle()`, `getBody()` – getters
- `getWords()` – returns array of normalized words
- `wordFrequency(String word)` – word frequency
- `wordSet()` – set of unique words
- `toString()` – text representation

Document Class Constructor

```
class Document
{
    private int id;
    private String title;
    private String body;

    public Document(int id, String title, String body)
    {
        this.id = id;
        this.title = title;
        this.body = body;
    }

    public int getId()
    {
        return id;
    }
```

Text Normalization – getWords()

```
public String[] getWords()
{
    String txt = body.toLowerCase();
    txt = txt.replaceAll("[^a-z0-9 ]", " ");
    String[] words = txt.split("\\s+");
    return words;
}
```

Normalization process:

- `toLowerCase()` – convert to lowercase
- `replaceAll("[`$^a - z0 - 9$`]","")` – – *removespecialcharacters*

- `split("`
  `textbackslash`
  `textbackslash s+") -- split into words`

Word Frequency – wordFrequency()

```
public int wordFrequency(String word)
{
    String[] words = getWords();
    int count = 0;
    for(String s : words)
    {
        if(s.equals(word.toLowerCase()))
            count++;
    }
    return count;
}
```

Application:

- Computing document weight relative to query
- Ranking documents by keyword frequency

Unique Word Set – wordSet()

```
public java.util.Set<String> wordSet()
{
    java.util.Set<String> set =
        new java.util.HashSet<String>();

    for(String s : getWords())
    {
        if(s.length() > 0)
            set.add(s);
    }
    return set;
}
```

Application:

- Building inverted index
- Eliminating duplicate words

Inverted Index

Inverted Index – data structure storing:

For each word $\rightarrow$ **list of document IDs** where it appears.

Example:

- Word `"java"` $\rightarrow$ documents [1, 3, 7, 12]
- Word `"python"` $\rightarrow$ documents [2, 5, 7]
- Word `"algorithm"` $\rightarrow$ documents [3, 5, 9]

Advantages:

- Fast search for documents containing given word
- Efficient for large document collections
- Foundation of web search engines

Index Class – Structure

```
import java.util.*;

class Index
{
    private List<Document> documents;
    private Map<String, List<Integer>> index;

    public Index()
    {
        documents = new ArrayList<Document>();
        index = new HashMap<String, List<Integer>>();
    }
}
```

Fields:

- documents – list of all documents
- index – map: word → list of document IDs

Adding Document – addDocument()

```
public void addDocument(Document d)
{
    documents.add(d);

    for(String word : d.wordSet())
    {
        List<Integer> list = index.get(word);
        if(list == null)
        {
            list = new ArrayList<Integer>();
            index.put(word, list);
        }
        if(!list.contains(d.getId()))
            list.add(d.getId());
    }
}
```

Adding Document – Logic

Addition process:

1. **Add document to list**
   - `documents.add(d)`
2. **For each word in document**
   - Get list of documents for this word from index
   - If list doesn't exist – create new one
   - If document ID not in list – add it

Result:

- Each word mapped to list of documents
- No duplicate IDs in lists

Searching Documents – searchWord()

```
public List<Document> searchWord(String word)
{
    List<Document> result = new ArrayList<Document>();
    List<Integer> idList =
        index.get(word.toLowerCase());

    if(idList != null)
    {
        for(Integer id : idList)
        {
            Document d = findDocumentById(id);
            if(d != null)
                result.add(d);
        }
    }
    return result;
}
```

Search – Step by Step

Word search process:

1. **Query normalization**
   - `word.toLowerCase()` – convert to lowercase
2. **Get ID list from index**
   - `index.get(word)` – list of identifiers
3. **Convert ID → Document objects**
   - For each ID: `findDocumentById(id)`
   - Add found documents to result
4. **Return document list**
   - `List<Document>`

Helper Method – findDocumentById()

```
private Document findDocumentById(int id)
{
    for(Document d : documents)
    {
        if(d.getId() == id)
            return d;
    }
    return null;
}
```

Application:

- Convert ID $\rightarrow$ Document object
- Linear search through document list
- Complexity: O(n) where n = number of documents

SearchEngine – Extended System

SearchEngine class – advanced queries:

Features:

- Stopwords filtering
- Single query: `query(String word)`
- AND query: `queryAND(String w1, String w2)`
- OR query: `queryOR(String w1, String w2)`
- Relevance ranking: `sortByRelevance()`

Stopwords:

- Frequently occurring words without semantic meaning
- Examples: `"and"`, `"the"`, `"or"`, `"a"`
- Ignored during search

SearchEngine Constructor

```
class SearchEngine
{
    private Index index;
    private Set<String> stopWords;

    public SearchEngine(Index index)
    {
        this.index = index;
        stopWords = new HashSet<String>();
        stopWords.add("and");
        stopWords.add("the");
        stopWords.add("or");
        stopWords.add("a");
    }
}
```

Single Query – query()

```
public List<Document> query(String word)
{
    word = normaliseWord(word);

    if(stopWords.contains(word))
        return new ArrayList<Document>();

    return index.searchWord(word);
}
```

Logic:

- Word normalization
- Check if stopword
- If yes – return empty list
- If no – search in index

AND Query – queryAND()

```
public List<Document> queryAND(String w1, String w2)
{
    w1 = normaliseWord(w1);
    w2 = normaliseWord(w2);

    List<Document> l1 = index.searchWord(w1);
    List<Document> l2 = index.searchWord(w2);

    List<Document> result = new ArrayList<Document>();
    for(Document d1 : l1)
    {
        if(l2.contains(d1))
            result.add(d1);
    }
    return result;
}
```

AND Query – Semantics

AND operator: documents containing **both** words.

Algorithm:

- Search documents containing `w1` $\rightarrow$ list `l1`
- Search documents containing `w2` $\rightarrow$ list `l2`
- Find intersection: `l1` $\cap$ `l2`
- Return documents in both lists

Example:

- Query: `queryAND("java", "algorithm")`
- Result: documents about Java algorithms

OR Query – queryOR()

```
public List<Document> queryOR(String w1, String w2)
{
    w1 = normaliseWord(w1);
    w2 = normaliseWord(w2);

    List<Document> l1 = index.searchWord(w1);
    List<Document> l2 = index.searchWord(w2);

    List<Document> result = new ArrayList<Document>(l1);
    for(Document d2 : l2)
    {
        if(!result.contains(d2))
            result.add(d2);
    }
    return result;
}
```

OR Query – Semantics

OR operator: documents containing **at least one** word.

Algorithm:

- Search documents containing `w1` $\rightarrow$ list `l1`
- Search documents containing `w2` $\rightarrow$ list `l2`
- Create union: `l1` ∪ `l2`
- Return all unique documents from both lists

Example:

- Query: `queryOR("java", "python")`
- Result: documents about Java OR Python (or both)

Document Ranking

Problem: Search may return many documents.

Question: Which documents are most relevant?

Simple ranking criterion:

- Frequency of keyword occurrence in document
- Higher frequency $\rightarrow$ higher ranking
- Document with 10 occurrences more important than with 2

Method:

- `sortByRelevance(List<Document>, String word)`
- Sorts documents descending by word frequency

Ranking – Implementation (1)

```
public List<Document> sortByRelevance(
    List<Document> list, String word)
{
    word = normaliseWord(word);

    Collections.sort(list,
        new Comparator<Document>()
        {
            public int compare(Document d1, Document d2)
            {
                int f1 = d1.wordFrequency(word);
                int f2 = d2.wordFrequency(word);
                return Integer.compare(f2, f1);
            }
        });
```

Ranking – Implementation (2)

```
    return list;
}
```

Explanation:

- `Collections.sort()` – in-place sorting
- `Comparator<Document>` – anonymous class defining order
- `wordFrequency()` – word frequency in document
- `Integer.compare(f2, f1)` – descending sort (f2 before f1)

System Usage Example

Scenario:

1. **Create index and engine**
   - `Index index = new Index();`
   - `SearchEngine se = new SearchEngine(index);`
2. **Add documents**
   - `index.addDocument(d1);`
   - `index.addDocument(d2);`
   - ...
3. **Execute query**
   - `List<Document> results = se.query("java");`
   - `results = se.sortByRelevance(results, "java");`
4. **Display results**
   - Iterate through `results` list
   - Print document titles

Advanced Extensions

TF-IDF (Term Frequency - Inverse Document Frequency)

Word weight in document:

- **TF** (Term Frequency) – word frequency in document
- **IDF** (Inverse Document Frequency) – $\log(\frac{N}{df})$
  - `N` – total number of documents
  - `df` – number of documents containing word
- **Weight:** TF · IDF

Intuition:

- Words frequent in document but rare globally → higher weight
- Words appearing everywhere (e.g. "the") → low weight

TF-IDF – Example

We have 100 documents:

Word "algorithm":

- TF = 5 (occurs 5 times in document D)
- df = 10 (occurs in 10 documents)
- IDF = $\log(100/10) = \log(10) \approx 2.3$
- Weight = $5 \cdot 2.3 = 11.5$

Word "the":

- TF = 20 (occurs 20 times in document D)
- df = 98 (occurs in 98 documents)
- IDF = $\log(100/98) \approx 0.02$
- Weight = $20 \cdot 0.02 = 0.4$

Other Extensions

Additional features to implement:

1. **Multi-word queries**
   - Parsing queries with multiple operators
   - Example: `"(java OR python) AND algorithm"`
2. **Phrases**
   - Searching for word sequences
   - Example: `"machine learning"`
3. **Result highlighting**
   - Displaying document fragments with highlighted keywords
4. **Positional indexing**
   - Storing word positions in documents
   - Supporting proximity queries

Modern IR Systems and AI

From keywords to semantics:

Vector embeddings:

- Representing words and documents as vectors
- Semantic similarity instead of literal matching
- Models: Word2Vec, GloVe, BERT

Large Language Models (LLM):

- GPT, BERT, T5
- Understanding context and meaning
- Generating answers, not just searching

Retrieval-Augmented Generation (RAG):

- Combining retrieval and generation
- Knowledge base + generative model

Approach Comparison

Aspect	**Keyword IR**	**Semantic IR (AI)**
Matching	Exact words	Meaning/context
"car" vs "automobile"	Different	Same
Complexity	Low	Very high
Speed	Very fast	Slower
Understanding	Superficial	Deep
Application	Simple search	Q&A, chatbots

Conclusion:

- Keyword IR – foundation
- Semantic IR – future
- Often used together (hybrid search)

Summary

- **IR systems** – searching documents by queries
- **Document class** – representation and text processing
- **Inverted index** – efficient search
- **AND/OR queries** – combining conditions
- **Ranking** – sorting by relevance
- **TF-IDF** – advanced word weighting
- **Semantic IR** – future with AI and LLM