g++ -std=c++17 src/main.cpp src/stage1_lexicon.cpp src/stage2_forward_index.cpp src/stage3_inverted_index.cpp -I include -o search_engine.exe

🔍 Building a Mini Search Engine From Scratch — Stage 1 to Stage 3 (Full Guide)

By Haris — Machine Learning & Systems Development

Modern search engines rely on powerful indexing pipelines to store, organize, and retrieve information efficiently. In this project, I built a complete **three-stage indexing system** — from raw tokens to a fully structured **inverted index** with **byte-offset posting pointers**.

This article explains the entire pipeline **step-by-step**, without skipping any detail.

⚙️ Project Overview

The project builds three critical components:

1. **Lexicon (Stage 1)**
2. **Forward Index (Stage 2)**
3. **Inverted Index (Stage 3)**

Each stage processes tokens and saves structured data to disk — exactly like industrial search engines (Lucene, Elasticsearch, Sphinx, Vespa).

The final output consists of:

- `lexicon/lexicon.json` (with posting_ptr and document frequencies)
- `forward_index/*.bin` segments
- `inverted_index/*.bin` segments

- Correct byte offsets for every posting list

This makes the system ready for **Stage 4: Query Execution**, where we can seek directly to any posting list in the binary file.

🧩 STAGE 1 — Building the Lexicon

🎯 Goal:

Extract all unique terms from the token file and assign each term:

- `term_id` (integer)
- `df` (document frequency = initialized to 0)
- placeholder `posting_ptr = 0` (final value will be updated in Stage 3)

📥 Input:

`data/sample_tokens_final_clean.txt`

This contains 50k tokens in the form:

```
doc_id term
doc_id term
doc_id term
...
```

📤 Output:

`output/lexicon/lexicon.json`

🛠️ How Stage 1 Works

1. Load all tokens line-by-line.

2. For each term, insert it into an ordered map.
3. Assign increasing `term_id` values.
4. Store only the term — df will be updated later.
5. Save the lexicon to disk.

✔ Result

A file like:

```
[
  {"term_id": 11031, "term": "aa", "df": 0, "posting_ptr": 0},
  {"term_id": 17206, "term": "aaa", "df": 0, "posting_ptr": 0},
  {"term_id": 19799, "term": "aaaa", "df": 0, "posting_ptr": 0}
]
```

Lexicon is now ready for Stage 2.

🧱 STAGE 2 — Building the Forward Index

🎯 Goal:

For every document, store the list of terms it contains.

📥 Input:

- Tokens file
- Lexicon from Stage 1

📤 Output:

Binary files:

`output/forward_index/segment_0.bin`

```
output/forward_index/segment_1.bin
...
```

Each segment contains **1000 documents**.

🛠️ How Stage 2 Works

1. Read tokens sequentially.
2. For each document:
    - Map term → term_id using lexicon
    - Count term frequency
3. Build a structure like:

```
doc_id: { term_id, term_frequency }
```

4. Write 1000 documents at a time to a binary segment:
    - doc_id
    - term count
    - for each term: (term_id, frequency)
5. Update each term's `df` (document frequency).

✔ Result Example

In binary format (conceptually):

```
[doc_id][term_count][term_id][tf][term_id][tf]...
```

Forward index now exists → ready for Stage 3.

🔁 STAGE 3 — Building the Inverted Index

🎯 Goal:

Reverse the structure:

Instead of:

```
doc → terms
```

We build:

```
term → list_of_docs(and frequency)
```

This is the heart of every search engine.

📥 Input:

- Forward index segments
- Lexicon (with df from Stage 2)

📤 Output:

```
output/inverted_index/inverted_0.bin
output/inverted_index/inverted_1.bin
...
output/lexicon/lexicon.json (updated)
```

🛠️ How Stage 3 Works (Important)

Initialize posting lists:

```
vector< vector<Posting> > inverted_index;
```

1.
2. Read each forward segment.
3. For each `term_id` inside each document:

    - Append `(doc_id, tf)` to the term's posting list.
4. Open one binary output file.
5. Write posting lists sequentially **in order of term_id**, not alphabetical order.

Track the **byte offset** before writing each posting list:

```
posting_ptr = current_file_offset;
```

6.
7. After writing all postings:
    - Update lexicon with final posting_ptr.
    - Save updated lexicon to JSON.

🎉 FINAL OUTPUT AFTER STAGE 3

Your lexicon entries now look like this:

```
{"term_id":11031,"term":"aa","df":7,"posting_ptr":0},
{"term_id":17206,"term":"aaa","df":11,"posting_ptr":32},
{"term_id":19799,"term":"aaaa","df":1,"posting_ptr":132}
```

This means:

- `"aa"` posting list starts at byte **0**
- `"aaa"` starts at byte **32**
- `"aaaa"` starts at byte **132**

Every posting list is now directly seekable.

This is the behavior of real search engines like Lucene.

📂 Folder Structure After Stage 3

```
search_engine_project/
│
├── data/
│   └── sample_tokens_final_clean.txt
│
├── output/
│   ├── lexicon/
│   │   └── lexicon.json
│   ├── forward_index/
│   │   ├── segment_0.bin
│   │   ├── segment_1.bin
│   │   └── ...
│   └── inverted_index/
│       ├── inverted_0.bin
│       ├── inverted_1.bin
│       └── ...
```

Everything is now stored exactly like a production indexing engine.

🏁 Where We Stand After Stage 3

You now have:

✔ A complete lexicon with:

- `term_id`
- `df`
- correct `posting_ptr` (byte offset)

✔ Full forward index

✔ Full inverted index

This is a complete indexing pipeline.

Next step is Stage 4 → **Query Engine**.