Building a Search Engine: Stage 1 — Lexicon Construction (Professional, Single-File Approach)

When building a search engine, the very first step is to construct a Lexicon — a compact, efficient representation of all unique words in your dataset, along with essential metadata. Stage 1 lays the foundation for fast searching and indexing.

Why Stage 1 Is Important

Before a search engine can retrieve documents for a query like “machine learning”, it needs to know:

1. Which documents contain “machine”?
2. Which documents contain “learning”?
3. How many documents contain each word (Document Frequency, DF)?
4. Where the posting list for each word is stored on disk?

All of this is stored in a lexicon, which is what Stage 1 builds.

Single-File Pipeline Overview

We implemented Stage 1 in a single C++ file for simplicity and efficiency. The workflow is:

Traverse Documents → Build Lexicon → Optimize → Save to Disk

This single-file approach is professional and ready for real search engine applications.

1️⃣ Lexicon Data Structure

We define a `LexiconEntry` to store:

- term_id → sequential integer ID for compact storage
- term → the token string itself
- df → document frequency
- posting_ptr → placeholder for pointer in inverted index

```
struct LexiconEntry {
    int term_id;
    std::string term;
    int df;
    long posting_ptr;
};
```

2️⃣ Lexicon Class

We encapsulate everything in a Lexicon class:

- Traverse documents and count DF
- Build entries and assign sequential IDs
- Sort alphabetically for fast binary search
- Save lexicon in binary, JSON, and reverse map formats

Traverse Documents & Count DF

Each document is read line-by-line. Unique tokens are counted per document to compute document frequency (DF).

```
void build_from_documents(const std::string& filepath);
```

- Uses an unordered_map to map token → DF
- Uses unordered_set to count unique tokens per document
- Ensures controlled vocabulary and minimal noise

Build Lexicon & Assign IDs

- Convert the DF map into a vector of `LexiconEntry`
- Assign sequential term IDs
- Build a token → term_id map for fast lookup
- Sort alphabetically for binary search efficiency

```
std::sort(entries.begin(), entries.end(), [](const LexiconEntry& a, const
LexiconEntry& b) {
    return a.term < b.term;
});
```

Save Lexicon to Disk

We save in three formats:

1. Binary → efficient for search engine loading (`lexicon.lexicon`)
2. JSON → readable for debugging (`lexicon.json`)
3. Reverse lookup → `term_id` → `token` (`term_map.term_map`)

```
void save_to_disk(const std::string& binary_file,
                  const std::string& json_file,
                  const std::string& reverse_map_file) const;
```

- Binary file stores: term_id, token string, df, posting pointer
- JSON file for readability
- Reverse map ensures you can map IDs back to tokens

3️⃣ Main Function (Pipeline Controller)

```
int main() {
    try {
        Lexicon lex;
        lex.build_from_documents("C:\\Users\\Muhammad
Haris\\OneDrive\\Desktop\\DSA
PROJECT\\cpp\\sample_tokens_final_clean.txt");
        lex.save_to_disk("lexicon.lexicon", "lexicon.json",
"term_map.term.map");
        std::cout << "Stage 1 completed successfully.\n";
    } catch (const std::exception& ex) {
        std::cerr << "Error: " << ex.what() << "\n";
    }
    return 0;
}
```

- Reads your cleaned dataset
- Builds a professional lexicon
- Saves all necessary files for the next stage

4️⃣ Dry Run Example

Suppose your dataset has three documents:

```
apple banana apple
banana mango
apple mango mango
```

Document Frequency Calculation:

Term	DF	Term ID
apple	2	0
banana	2	1
mango	2	2

Files produced:

JSON:

```
[
  {"term_id":0,"term":"apple","df":2,"posting_ptr":0},
  {"term_id":1,"term":"banana","df":2,"posting_ptr":0},
  {"term_id":2,"term":"mango","df":2,"posting_ptr":0}
]
```

Reverse Map (`term_map.term.map`):

```
0 apple
1 banana
2 mango
```

5️⃣ Why Alphabetical Lexicon + Term IDs Matters

1. Fast Binary Search: Lookup time reduces from O(n) to O(log n)
2. Memory Efficiency: Store integer term IDs instead of strings in posting lists
3. Stable IDs: Same token always has same ID → predictable indexing
4. Ready for Stage 2: Posting pointers can now be used to link terms to document lists

---

6️⃣ Stage 1 Output

```
Lexicon built. Total unique tokens: 46763
Lexicon saved to disk (binary + JSON + reverse map).
Stage 1 completed successfully.
```

- Your single-file implementation completed Stage 1 professionally
- 46k+ terms are ready for efficient indexing and search

✅ Key Takeaways

- Single-file professional design
- Controlled, compact vocabulary
- Alphabetical lexicon → fast binary search
- Sequential IDs → memory-efficient indexing
- Binary + JSON + reverse map → versatile storage
- Stage 1 completed, ready for inverted index (Stage 2)

Haris, this blog now reflects exactly what you impleme