Search Engine Construction: Stage 1 → Stage 3

Imagine building a simple search engine step by step. The goal: take raw text documents and turn them into a system that can answer “which documents contain this word?”

Stage 1: Lexicon Construction (Vocabulary Dictionary)

Goal: Map each unique word in your dataset to a numeric ID and record basic info.

Process:

1. Read all documents.
2. Extract **tokens** (words) and compute **document frequency (DF)** — how many documents contain each token.
3. Assign each token a unique **term_id**.
4. Store mappings:
    - `term → term_id`
    - `term_id → term` (reverse mapping)
    - `term_id → document frequency`

Example:

Documents:

```
Doc0: apple banana apple
Doc1: banana fruit
Doc2: apple fruit apple
```

Lexicon:

term_id	term	df
0	apple	2
1	banana	2
2	fruit	2

Output: Lexicon saved on disk as binary + JSON + reverse map.

Stage 2: Forward Index (Document → Terms)

Goal: Record internal structure of each document — what terms appear, how often, and where.

Process:

1. For each document:
   - `doc_id` → unique numeric ID
   - `term_ids[]` → list of token IDs in the document
   - `term_freqs` → frequency of each token
   - `positions[]` → optional: where each token occurs (for phrase search)
   - `length` → document length (used in ranking formulas like BM25)
2. Compress the index:
   - Variable-byte encoding
   - Delta encoding for positions
   - Optional: LZ4/Snappy
3. Save **segmented forward index** (e.g., `forward_index_0.bin`, `forward_index_1.bin`) to avoid large file rewrites.

Example:

Forward index for Doc0 (`apple banana apple`):

```
doc_id = 0
term_ids = [0,1,0]        // apple=0, banana=1
term_freqs = [2,1]
positions = [0,1,2]
length = 3
```

Visualization:

```
[Document 0] → apple, banana, apple
       |
       ▼
Forward Index → doc_id=0, term_ids=[0,1,0], freqs=[2,1],
positions=[0,1,2], length=3
```

Stage 3: Inverted Index (Term → Documents)

Goal: Build the actual searchable index: for each term, list all documents that contain it.

Process:

1. Iterate through forward index.
2. For each term, create a **posting list**:

```
term → [(doc_id, term_freq, positions[]), ...]
```

Example:

```
apple → [(0,2,[0,2]), (2,2,[0,2])]
banana → [(0,1,[1]), (1,1,[0])]
fruit → [(1,1,[1]), (2,1,[1])]
```

3. Sort postings by `doc_id` → faster merging, skipping, ranking.

4. Compress postings:
   - **Delta encoding** (store doc_id differences)
   - **Variable-byte encoding** for small numbers
   - Optional **skip pointers** for fast boolean queries
5. Save **segmented inverted index** (e.g., `inverted_index_segment_0.bin`) to avoid huge files.

Visualization:

```
Forward Index:
Doc0 → apple, banana, apple
Doc1 → banana, fruit
Doc2 → apple, fruit, apple
       |
       ▼
Build Posting Lists:
apple → [(0,2,[0,2]), (2,2,[0,2])]
banana → [(0,1,[1]), (1,1,[0])]
fruit → [(1,1,[1]), (2,1,[1])]
       |
       ▼
Sort & Compress → Delta + VByte
       |
       ▼
Segmented Storage → inverted_index_segment_0.bin, segment_1.bin, ...
```

🔑 Logical Flow Summary

Stage	Input	Output	Purpose
1	Raw documents	Lexicon (term ↔ term_id)	Assign IDs, track DF, enable indexing

2	Lexicon + Documents	Forward index (doc → terms)	Store doc structure, term positions, frequencies
3	Forward index + Lexicon	Inverted index (term → docs)	Enable fast searching, ranking, boolean queries

Data Flow Diagram (Text Version)

```
Raw Documents
      |
      ▼
Stage 1 → Lexicon
      |
      ▼
Stage 2 → Forward Index
      |
      ▼
Stage 3 → Inverted Index
      |
      ▼
Search Engine Queries (fast term → doc retrieval)
```

✅ After Stage 3, your search engine is ready to handle queries. You can now efficiently ask:

- Which documents contain "apple"? → lookup `apple` in inverted index
- How often does "banana" appear in Doc1? → check posting list frequency
- Phrase searches → use token positions