Stage 3 — Inverted Index: The Heart of a Search Engine

After building the lexicon (Stage 1) and the forward index (Stage 2), we are now at the **most critical part** of a search engine: the **inverted index**. This is what makes searching fast and efficient.

What is an Inverted Index?

Think of it like a book's index:

- A normal book index tells you the **page numbers** where a topic appears.
- In a search engine, the inverted index tells us the **document IDs** where a term appears, along with additional info like frequency and positions.

Example:

If we have three documents:

Doc ID	Content
0	"apple banana apple"
1	"banana fruit"
2	"apple fruit apple"

The inverted index would look like:

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

Explanation:

- Each term points to a **posting list**.
- A posting contains:
  - `doc_id` → which document it appears in.
  - `term_frequency` → how many times it appears.
  - `positions` → locations inside the document (useful for phrase search).

Step 3.1 — Build Posting Lists

We loop through **all terms in the lexicon**, and for each term:

1. Scan all documents (from the forward index).
2. Record the documents containing the term.
3. Track term frequency and positions.

> This is the core “mapping” from term → documents.

Step 3.2 — Sort Posting Lists by `doc_id`

Why sort?

- Allows **fast merging** when processing multiple terms.
- Makes **AND/OR queries** much faster.
- Supports **ranked retrieval** efficiently.

Step 3.3 — Compress Posting Lists

Raw posting lists can be huge. Compression helps:

1. **Delta Encoding**
    - Store differences between consecutive `doc_id`s instead of the full number.
    - Example: `[3, 10, 15]` → `[3, 7, 5]`.
2. **Variable-Byte Encoding**
    - Encode integers using fewer bytes when small.
3. **Block-Based Compression**
    - Split postings into fixed-size blocks (e.g., 128 docs).
4. **Skip Lists (Optional but Recommended)**
    - Add shortcuts every √n entries.
    - Helps quickly skip large portions when merging lists for queries.

Compression makes your inverted index **lightweight** and **fast to read from disk**.

Step 3.4 — Add Skip Pointers

Skip pointers are like **fast-forward buttons** in a posting list:

- Instead of scanning every document sequentially, you can jump ahead using these pointers.
- Greatly accelerates boolean queries (AND/OR).
- Standard practice: skip every √n entries.

Step 3.5 — Save Segmented Inverted Index

Large indexes are saved in **segments**:

```
inverted_index_segment_0.bin
inverted_index_segment_1.bin
...
```

- Avoids rewriting huge files every time you update.
- Each segment contains a portion of terms.
- Lexicon keeps a **pointer to each term's posting list**, so we can quickly load it when searching.

Summary

Stage 3 transforms your forward index (document → terms) into the **inverted index (term → documents)**:

- Enables **fast search**.
- Supports **ranking** and **phrase search**.
- Compresses data for **efficient storage**.
- Prepares the engine for **query processing**.

After this stage, your search engine is **ready to answer real user queries**. 🎯