Mini-Elasticsearch: Detailed Step-by-Step Build Guide

==========

INTRODUCTION

This document provides a clear, beginner-friendly but academically solid step-by-step guide to building a simplified distributed search engine in Python, inspired by Elasticsearch. It focuses on practical implementation steps rather than deep theoretical explanations.

The goal is to give you a roadmap so you can confidently build the project from start to finish.

SECTION 1 — PROJECT OVERVIEW

You will build a distributed system consisting of:

1. Coordinator Node

- Receives CRUD operations (index documents, search).
- Routes requests to appropriate Data Nodes.
- Merges search results.
- Tracks cluster health.

2. Data Nodes

- Store documents.
- Build and store inverted indexes.
- Respond to search queries.
- Participate in sharding and replication.

3. Optional Components

- Heartbeat & failure detection.

- Docker / Kubernetes deployment.

- Persistent storage on disk.

SECTION 2 — MINIMUM FEATURES NEEDED FOR A FUNCTIONAL PRODUCT

---

1. Accept JSON documents via REST API.

2. Tokenize text fields.

3. Build an inverted index.

4. Split documents across nodes using sharding.

5. Query nodes and merge results.

6. Run each node as its own HTTP server.

7. Add basic heartbeat signals.

This is enough to impress a professor and demonstrate real distributed systems knowledge.

SECTION 3 — STEP-BY-STEP IMPLEMENTATION PLAN (THE MOST IMPORTANT PART)

==========

STEP 1 — Define the Document Model

==========

All documents will be JSON with a mandatory "id" field.

Example:

{

"id": "123",

"title": "Distributed Systems",

"body": "This is a test document."

}

Store them on each Data Node as simple JSON files or in-memory dicts first.

==========

STEP 2 — Implement Tokenization

==========

Before indexing, convert text into searchable tokens:

- lowercase

- remove punctuation

- split by spaces

Example:

"Hello World!" → ["hello", "world"]

Implement a tokenizer function in Python.

==========

STEP 3 — Build the Inverted Index Logic (LOCAL ONLY)

==========

Your inverted index is simply:

token -> [list of document IDs that contain that token]

Example:

"distributed" -> ["123", "200"]

"systems" -> ["123", "900"]

Data Node stores this in memory:

index = { token: set(doc_ids) }

==========

STEP 4 — Add Document Indexing API (LOCAL ONLY)

==========

Use FastAPI or Flask.

Route:

POST /index

{

"id": "...",

"title": "...",

"body": "..."

}

Pipeline:

1. Receive document

2. Tokenize text fields

3. Update inverted index

4. Store raw document

At this point: single-node working.

==========

STEP 5 — Build the Search API (LOCAL ONLY)

==========

GET /search?q=distributed+systems

Pipeline:

1. Tokenize query

2. Look up each token in inverted index

3. Intersect or union results

4. Return list of document IDs

Now you have a basic search engine.

==========

STEP 6 — Introduce MULTIPLE DATA NODES

==========

Run multiple Python servers on different ports:

Node 1 → localhost:9001

Node 2 → localhost:9002

Node 3 → localhost:9003

Each has:

- Its own inverted index

- Its own documents

- Its own REST API

==========

STEP 7 — Implement SHARDING

==========

Use hashing:

shard_id = hash(doc_id) % NUM_SHARDS

If NUM_SHARDS=3:

- doc_id=10 → shard 1

- doc_id=22 → shard 1

- doc_id=15 → shard 0

The Coordinator maps shards to Data Nodes.

==========

STEP 8 — Coordinator Node

==========

Coordinator handles:

1. POST /index → determines shard and forwards request to correct Data Node.

2. GET /search → broadcasts request to ALL nodes, merges results.

3. GET /cluster/status → reports node health.

Coordinator must know:

node_map = {

0: "localhost:9001",

1: "localhost:9002",

2: "localhost:9003"

}

==========

STEP 9 — QUERY DISTRIBUTION & MERGING

==========

Search process:

1. Coordinator receives search query.

2. It forwards query to ALL data nodes.

3. Nodes respond with document IDs.

4. Coordinator merges IDs, removes duplicates.

5. Coordinator returns combined result.

==========

STEP 10 — ADD REPLICATION (OPTIONAL BUT IMPRESSIVE)

==========

Assign each shard two nodes:

shard_replicas = {

0: ["node1", "node2"],

1: ["node2", "node3"],

2: ["node3", "node1"]

}

Coordinator writes document to ALL replicas.

==========

STEP 11 — HEARTBEAT & FAILURE DETECTION

==========

Coordinator runs a background loop:

- Every 3 seconds, ping each node.

- If no reply → node marked DEAD.

- If node is dead → send traffic only to replicas.

==========

STEP 12 — DOCKERIZE THE NODE PROCESSES

==========

Make a Dockerfile:

- Install FastAPI

- Expose port

- Start server

Run containers:

docker run -p 9001:9001 datanode

==========

STEP 13 — DEPLOY TO KUBERNETES (OPTIONAL)

==========

Use:

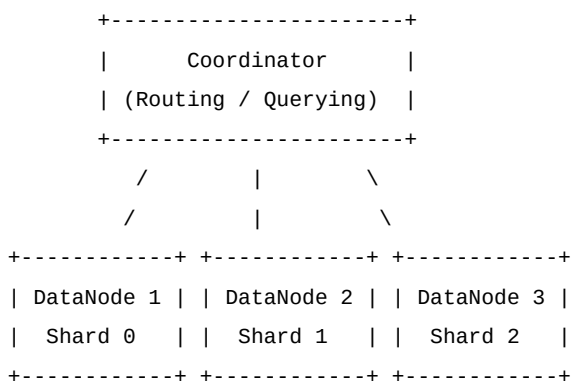
- Deployment for Coordinator

- StatefulSet for Data Nodes

- Service for discovery

==========

SECTION 4 — ASCII ARCHITECTURE DIAGRAM



SECTION 5 — FINAL RECOMMENDATIONS

---

1. Build single-node version first.

2. Then add multiple nodes without sharding.

3. Then add sharding logic.

4. Then add replication.

5. Then add heartbeat.

6. Only THEN try Docker / Kubernetes.

This ensures you avoid complexity too early.

END OF DOCUMENT
