HLD Case Study 4 (Elastic Search)

Contents

[Building a Search Feature 2](#_Toc184730154)

[Why Standard Relational and NoSQL Queries are Inefficient 2](#_Toc184730155)

[Full Text Search and Inverted Index Creation 4](#_Toc184730156)

[Lucene's Indexing Process 6](#_Toc184730157)

[What is Elasticsearch? 7](#_Toc184730158)

[Elasticsearch Full-Text Search and Architecture 10](#_Toc184730159)

[Elasticsearch – sharding 13](#_Toc184730160)

[Elasticsearch - Reads and Writes 15](#_Toc184730161)

## Building a Search Feature

**Context**

* The goal is to implement a search feature similar to those on platforms like LinkedIn, Google, or Facebook.
* This feature will allow users to search for posts containing specific keywords.
* The results will include the **top matching posts** rather than all possible matches to avoid overwhelming the user.

**Key Requirements**

1. **Search Query Handling**:
   * Users input text queries (e.g., "good culture" or "good working culture").
   * The system should match and retrieve posts containing these keywords or closely related content.
2. **Search Scope**:
   * The focus is on **posts** but may extend to profiles, company names, etc., as part of future enhancements.
3. **Result Prioritization**:
   * Display **top matching posts** that are most relevant to the query.
   * Avoid overwhelming the user with too many results.
4. **Comparison with Other Platforms**:
   * **LinkedIn**: Includes posts, profiles, and companies in search results.
   * **Google Search**: Finds web pages matching the search keywords.
   * **WhatsApp Search**: Offers a simpler search model within chat content.

## Why Standard Relational and NoSQL Queries are Inefficient

1. **Relational DB (MySQL)**:
   * Searching for a substring (e.g., "good work culture") using LIKE '%good work culture%' triggers a **full table scan**, even if there is an index on the content column.
   * Indexes like B-Trees are optimized for exact matches or range queries but not for substrings.
   * Every row must be scanned and checked for the substring, leading to **O(N \* substring\_search\_complexity)**.
2. **NoSQL Key-Value Stores**:
   * While efficient for direct key lookups, they do not natively support searching within the value (content).
   * Creating a reverse index with words as keys may help but requires additional processing for phrase matching.
3. **Document Stores**:
   * Provide better support for full-text indexing compared to relational DBs or key-value stores.
   * Still need enhancements like inverted indices for optimal performance.

**Proposed Solution: Inverted Index with Positional Information**

**1. Inverted Index**

* Create a hash map (or trie) where:
  + **Key**: Each unique word in the posts.
  + **Value**: A list of entries containing:
    - Post ID(s) where the word appears.
    - Positions of the word in the post.
* For example:

|  |
| --- |
| "good" -> [(1, [4, 10]), (5, [7])]  "work" -> [(1, [5]), (5, [8])]  "culture" -> [(1, [6]), (5, [9])] |

**2. Phrase Matching**

* To search for the phrase "good work culture":
  + Retrieve the lists for "good," "work," and "culture."
  + Perform an intersection of the Post IDs.
  + Check positions for continuity: For post 1, ensure positions [4, 5, 6] align.

**3. Storage Structure**

* Use a **trie** or **hash map** for the inverted index:
  + A trie allows efficient prefix-based queries.
  + A hash map is simpler and faster for direct word lookups.
* Store the post content in a separate database/document store, as it is not needed for quick searches.

**4. Updates**

* For new posts:
  + Tokenize the content into words.
  + Update the inverted index by adding new words or appending to existing entries.

## Full Text Search and Inverted Index Creation

**Introduction to Full Text Search (FTS):**

1. **Definition:**
   * Full Text Search is the ability to search text-based data efficiently within large documents or collections of posts.
   * Example Use Cases:
     + Searching for posts on LinkedIn containing phrases like *"good work culture"* or *"run marathon"*.
     + Google Search or Facebook post search.
2. **Challenges with Conventional Databases:**
   * **SQL Databases:**
     + Not optimized for searching large documents or posts.
     + Searching for substrings in SQL databases has high time complexity: O(N×substring)O(N \times \text{substring})O(N×substring).
   * **NoSQL Databases:**
     + Though scalable for diverse schemas, they also lack optimization for full-text search.
3. **Real-World Systems for FTS:**
   * Systems like **Elasticsearch** and **Apache Solr** are specialized for full-text search.
   * Internally rely on **Apache Lucene** for core functionalities like building an **inverted index**.

**What is an Inverted Index?**

1. **Concept:**
   * A data structure that maps terms (words) to the documents they appear in.
   * Example

|  |
| --- |
| "good work culture" -> [doc1, doc2]  "ran marathon" -> [doc3] |

1. **Purpose:**
   * Facilitates quick retrieval of documents containing specific terms or phrases.
2. **Optimization Challenges:**
   * **Stemming/Lemmatization:** Handle variations of the same word (e.g., *run* and *ran* should be treated as identical).
   * **Synonyms:** Words like *best* and *good* should be treated as equivalent.

**Step-by-Step Process to Build an Inverted Index:**

1. **Input:**
   * A collection of documents/posts.
2. **Preprocessing:**
   * Tokenization:
     + Break down text into individual words (tokens).
   * Case normalization:
     + Convert all words to lowercase to ensure uniformity.
   * Remove stop words:
     + Exclude common words like *is, the, a*, etc., to reduce the index size.
   * Stemming/Lemmatization:
     + Reduce words to their base forms (*running* -> *run*, *ran* -> *run*).
3. **Index Creation:**
   * For each word in the processed document:
     + Add the document ID to the list of documents associated with that word in the index.

## Lucene's Indexing Process

1. **Word Elimination**:
   * Removes insignificant words (e.g., prepositions and pronouns like *a*, *the*, *on*).
   * Ensures phrases like *"run a marathon"* and *"run marathon"* are treated equivalently.
2. **Tokenization**:
   * Splits the remaining text into individual words (tokens).
   * Leaves the original document intact but determines what goes into the inverted index.
3. **Normalization**:
   * **Stemming and Lemmatization**:
     + Reduces words to their root forms (e.g., *running*, *ran* → *run*; *better*, *best* → *good*).
     + Stemming crudely truncates words, potentially causing inaccuracies (*caring* → *car*).
     + Lemmatization uses a dictionary to derive context-aware root words, preserving meaning.
   * Removes special characters.
4. **Stop Words**:
   * Filters out specific words that shouldn't appear in search results or indexing (e.g., offensive or irrelevant terms).
5. **Synonyms and Spelling Correction**:
   * Lucene might not inherently handle synonyms, but advanced systems like Google do.
   * Spell correction involves calculating the **edit distance** between words (e.g., *recieve* → *receive*).

**Edit Distance and Spell Correction**

Edit distance determines the number of operations needed to transform one word into another. Operations include:

* **Insert** a character.
* **Remove** a character.
* **Replace** a character.

**Optimization Techniques:**

* Use **tries** or **hash maps** for efficient word lookups.
* Restrict comparison to words with common prefixes or similar lengths.
* Consider **keyboard proximity** for replacement costs (e.g., *m* → *n* costs less than *q* → *n*).

**Enhancing Search Efficiency:**

* Implement heuristics or preprocessing steps to reduce the number of potential matches.
* Utilize advanced data structures and algorithms to optimize brute-force edit distance calculations.

## What is Elasticsearch?

1. **Type of Database:**
   * Elasticsearch is a **document-oriented database**, part of the NoSQL family.
   * Documents are stored in JSON format, each having a unique ID.
2. **Core Functionality:**
   * It excels in **full-text search** and **real-time query execution**.
   * Internally uses **Apache Lucene** for indexing and searching.
3. **Storage and Retrieval:**
   * Stores data as documents.
   * Allows queries to fetch documents based on content or specific fields.
4. **Indexing:**
   * When a document is added, fields like content can be indexed for fast searching.
   * Indexing involves breaking the content into words, removing stop words, and applying **stemming** or **lemmatization** (reducing words to their root forms).

**How Elasticsearch Works**

1. **Add Data:**
   * Use HTTP PUT to add documents into Elasticsearch.
   * For example, adding a review:

|  |
| --- |
| {    "reviewID": 2534,    "content": "I ran a marathon last weekend and it was awesome",    "authorID": 123  } |

* Elasticsearch creates a unique ID if none is provided and stores this document.

1. **Search Data:**

* Perform HTTP GET queries.
* Example: Find all documents where the content contains "ran marathon":

|  |
| --- |
| {    "query": {      "match": {        "content": "ran marathon"      }    }  } |

* + Elasticsearch supports advanced queries like **AND**, **OR**, and **fuzzy matching**.

1. **Lucene Integration:**
   * Behind the scenes, Lucene performs **reverse indexing**, enabling fast search capabilities.

**Common Use Cases**

1. **Applicant Tracking System (ATS):**
   * **Scenario:** A recruiter wants to search resumes for skills like "Java" or "Spring Boot."
   * **Solution:**
     + Parse resumes into plain text and store them as documents in Elasticsearch.
     + When a recruiter searches for keywords, Elasticsearch quickly fetches matching resumes by their IDs.
2. **Log Aggregation and Search (e.g., Distributed Systems):**
   * **Scenario:** A system processes requests across multiple microservices, and you need to debug where a failure occurred.
   * **Solution:**
     + Assign a unique ID (e.g., traceID) to every request.
     + Send logs from all microservices to Elasticsearch.
     + Use queries to search logs by traceID or error details to pinpoint the issue.
   * **Example Workflow:**
     + Logs might include:

|  |
| --- |
| {    "traceID": "abc123",    "component": "notificationService",    "status": "failure"  } |

* + - Querying traceID reveals which components logged data and identifies where the failure occurred.

1. **Search Across Large Text Data:**
   * Systems like WhatsApp, Windows search, or even messaging platforms use Elasticsearch for efficient phrase or keyword searches.

**Tools Built Around Elasticsearch**

1. **Kibana:**
   * A UI tool for visualizing Elasticsearch data.
   * Allows you to perform queries, view logs, and create dashboards.
2. **ELK Stack:**
   * Combination of **Elasticsearch**, **Logstash** (for log processing), and **Kibana**.

**Simplified Workflow**

1. Store documents using PUT.
2. Index fields (e.g., content) for fast search.
3. Perform advanced queries using GET.
4. Use tools like **Kibana** for visualization and debugging.

## Elasticsearch Full-Text Search and Architecture

**Introduction to Full-Text Search**

1. **Definition**:
   * Full-text search allows searching through large collections of text (e.g., websites, resumes, LinkedIn posts).
   * Commonly implemented using tools like Elasticsearch.
   * Operates on the concept of indexing documents for fast retrieval.
2. **Use Cases**:
   * Searching content on websites.
   * Resume searches by recruiters.
   * Searching posts or user-generated content (e.g., LinkedIn).
3. **Technology Overview**:
   * Elasticsearch leverages **Apache Lucene** for indexing and querying.
   * Lucene builds an **inverse index**, which maps keywords to their document locations.
4. **Why Not Traditional Databases?**
   * MySQL or Cassandra is not optimized for full-text search.
   * Elasticsearch is specialized for such use cases and provides better performance.

**Core Terminologies**

1. **Document**:
   * A JSON object representing data.
   * Example: A resume JSON containing attributes like id and content.
2. **Index**:
   * A collection of documents stored together.
   * Functions like a database in Elasticsearch.
   * Example: Separate indexes for resumes and LinkedIn posts.
   * By default, every index has an **inverse index** for fast searches.
3. **Node**:
   * A physical or virtual machine running Elasticsearch.
   * In architecture discussions, "node" refers to individual machines.

**Non-Functional Requirements of Elasticsearch**

1. **Fault Tolerance**:
   * System must avoid single points of failure (e.g., a single machine crashing).
   * Data and processes need replication across nodes.
2. **High Availability**:
   * Critical for search use cases where downtime is unacceptable.
   * Consistency is less critical; occasional stale results are acceptable as long as search remains operational.
3. **Read-Heavy System**:
   * Majority of operations are searches (reads) compared to writes (indexing).
   * Example: Index resumes once, but search multiple times.

**Challenges with a Single Node Setup**

1. **Single Point of Failure**:
   * A single machine crash results in system downtime.
2. **Storage Limits**:
   * Limited capacity for document storage.
3. **Performance Bottlenecks**:
   * Overloaded machine slows down search operations.

**Elasticsearch Scalability & Architecture**

1. **Multiple Nodes**:
   * Multiple nodes eliminate single points of failure.
   * Load distribution ensures better performance and fault tolerance.
2. **Indexes as Logical Databases**:
   * Elasticsearch doesn't differentiate between tables or collections like traditional databases.
   * Instead, uses indexes to separate types of documents (e.g., resumes vs LinkedIn posts).
3. **Index Creation**:
   * Use a PUT command to define an index (e.g., resumes or posts).
   * Each index has its own metadata and configuration.

## Elasticsearch – sharding

**1. Why Use Sharding?**

When dealing with a massive volume of data, a single machine will eventually run out of space or processing power. **Sharding** solves this by distributing data across multiple machines:

* Each shard is a subset of the data.
* Machines can independently process their shards, improving scalability.

**2. Replica Benefits**

* **Fault Tolerance**: If one machine fails, replicas ensure no data is lost.
* **Load Distribution**: Replicas help balance **read-heavy traffic**, as queries can be served by any replica of the shard.

**3. Sharding Strategies**

There are two common approaches: **Sharding by Document ID** and **Sharding by Keywords**.

**a. Sharding by Document ID:**

* Assigns documents to shards based on the hash of the document ID modulo the number of shards (hash(docID) % number\_of\_shards).
* Advantages:
  + Cleaner and deterministic routing.
  + No redundant storage across shards.
* Challenges:
  + Queries still need to access all shards for keyword-based searches, which can be slow.

**b. Sharding by Keywords:**

* Distributes data based on the content (keywords).
* Each shard stores documents that share specific keywords.
* Challenges:
  + **Redundant Storage**: A document might be duplicated across multiple shards if it contains multiple keywords.
  + **Complex Query Processing**: Queries require combining results from multiple shards, involving extra network communication and merging operations.

**ElasticSearch chooses document ID sharding** due to its simplicity and predictable data distribution, even though keyword sharding might seem to align better with query patterns.

**4. Elasticsearch Shard and Replica Configuration**

* **Fixed Number of Shards**:
  + The number of shards is set during index creation and cannot be easily changed later.
  + To accommodate future growth, it's advisable to start with more shards than initially required.
* **Replicas**:
  + The number of replicas can also be configured initially.
  + Each replica is a full copy of its shard, providing redundancy and scalability.

**5. Routing in Elasticsearch**

* Elasticsearch uses a straightforward routing algorithm:
  + Compute hash(document\_id) % number\_of\_shards.
  + Route the document or query to the calculated shard.
  + As the number of shards is fixed, consistent hashing is not required.

This approach ensures that any given document is deterministically assigned to a specific shard and its replicas.

**6. Considerations for Performance**

* **Query Performance**: Even though document ID-based sharding is clean, queries for keywords need to scan all shards, which can become a bottleneck.
* **Replica Optimization**: Adding replicas helps distribute the read load but increases storage requirements.

**7. Implementation Details**

* Shards can be assigned to the same or different machines, depending on the cluster configuration.
* It is possible to have multiple shards on a single machine during the early stages of deployment.

**8. Why This Design Choice?**

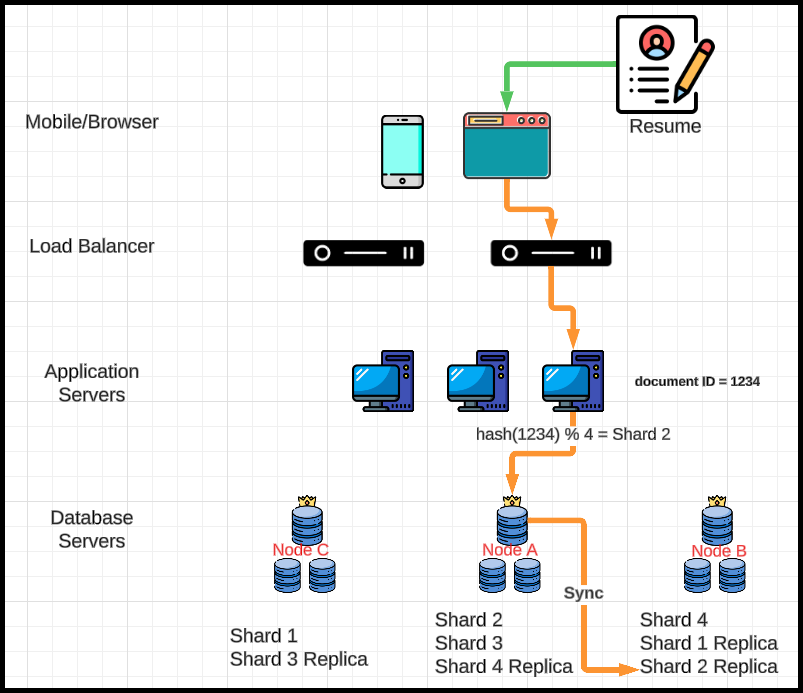
The Elasticsearch approach is a trade-off:

* **Document ID-based sharding** is efficient for writes and clean in terms of data placement.
* While **keyword-based sharding** might optimize search queries, its storage redundancy and query complexity can outweigh its benefits.

## Elasticsearch - Reads and Writes

**Write Flow in Elasticsearch**

1. **Client Request**:
   * A document (e.g., JSON with resume\_id and content) is sent to an application server.
   * The application server determines the **document ID** (e.g., 1234).
2. **Shard Determination**:
   * A hash function (e.g., hash(1234) % 4) determines which shard should store the document.
   * Example: If hash(1234) % 4 = 2, the document is routed to **Shard 2**.
3. **Shard Mapping**:
   * Elasticsearch nodes maintain a **cluster state** that maps shards to physical or virtual machines.
   * The application server looks up where **Shard 2** resides (e.g., Node A).
4. **Write to Primary Shard**:
   * The document is sent to the **primary shard** (e.g., Shard 2 on Node A).
   * The primary shard processes the write by updating its inverted index and storing the document.



1. **Replica Update**:
   * The change is propagated to **Shard 2’s replicas** on other nodes.
   * This ensures redundancy and high availability.
2. **Acknowledgment**:
   * Once the write is complete on the primary shard and replicas, the client receives confirmation.

**Read Flow in Elasticsearch**

1. **Search Request**:
   * A search query (e.g., "Ran Marathon") arrives at the application server.
2. **Broadcast Query**:
   * The query is broadcast to all shards in the index.
   * For each shard, either the primary shard or one of its replicas is randomly selected to handle the query.
3. **Query Execution**:
   * Each selected shard executes the query against its local data, considering filters, term matching, and ranking logic.
4. **Parallel Responses**:
   * Results are sent back from all shards to the coordinating node.
   * Elasticsearch merges the responses into a unified result set.
5. **Timeout Handling**:
   * If one or more shards fail to respond within a configured timeout, Elasticsearch uses results from the responding shards.
   * This ensures high availability, though it may lead to partial results.
6. **Result Ranking**:
   * Results are ranked by relevance scores (e.g., term proximity, exact match vs. partial match).
   * Scores allow fuzzy matching and prioritization of highly relevant documents.

**Scaling in Elasticsearch**

1. **Sharding**:
   * Data is divided into **shards** for distribution across nodes.
   * Each shard is an independent Lucene index.
2. **Replication**:
   * Each shard has **replicas** for fault tolerance and parallel processing.
3. **Cluster Coordination**:
   * A **master node** manages shard allocation and cluster state.
4. **Query Optimization**:
   * Queries are distributed across nodes and processed in parallel, leveraging replicas to balance the load.

**Strengths and Limitations**

1. **Strengths**:
   * High availability: Results are served even with partial shard responses.
   * Scalability: Shards and replicas allow handling large datasets and high query volumes.
   * Flexible search: Supports full-text, fuzzy, and ranked search with rich querying capabilities.
2. **Limitations**:
   * **Write performance**: Writes can be slower due to index updates and replication overhead.
   * **Initial indexing**: Bulk indexing or rebuilding indexes can be time-consuming.
   * **Inconsistencies**: In rare cases, replicas may return stale data during transient states.

**Use Case Considerations**

* **Ideal for Read-Heavy Workloads**:
  + Elasticsearch excels in scenarios where search operations dominate, such as logging, analytics, and resume searches.
* **Avoid Write-Intensive Use Cases**:
  + High-frequency writes may stress Elasticsearch’s indexing mechanism.
* **Batch Writes**:
  + To optimize performance, write in batches (e.g., log aggregation every hour).