Media Search Challenge

Project Overview

This media Search Challenge is a media search platform designed to handle media items search with unstructured metadata. The system provides fast search capabilities across images with metadata, including captions, photographers, dates, and technical information. It combines a React frontend with a Node.js/Express backend and Elasticsearch for powerful search capabilities.

Key Features:

- **Flexible Keyword Search**: Supports single-term and multi-term queries with boosting for exact matches.
- **Photographer/Agency Filtering**: Users can filter results by specifying a photographer's name.
- **Date Range Filtering**
- **Sorting Options**: Results can be sorted by relevance (_score) or specific fields (e.g., date).
- **Pagination**: Results are paginated with customizable page and limit values.
- **Performance Logging**: Measures response time for search queries.
- **Highlighting**: Returns highlighted text snippets for keyword matches in search results.
- **Frontend Caching**: Implements React Query for transparent caching of search results, significantly reducing redundant API calls

Setup and Installation

Prerequisites

- Node.js (v18+)
- npm
- Elasticsearch (v8.x)

Cloning and Environment Setup

1. Clone the repository:

```
git clone https://github.com/japel/challenge-c4.git
```

2. Configure Elasticsearch connection:
   Create a .env file in the root directory with the following variables:

```
ELASTICSEARCH_NODE=https://your-elasticsearch-host:9200
ELASTICSEARCH_USERNAME=your_username
ELASTICSEARCH_PASSWORD=your_password
ELASTICSEARCH_INDEX=imago
PORT=3000
```

Backend Setup

1. Navigate to the root directory:

```
cd challenge-c4
```

2. Install dependencies:

   ```
   npm install
   ```

Frontend Setup

1. Navigate to the frontend directory:

   ```
   cd frontend
   ```

2. Install dependencies:

   ```
   npm install
   ```

Running the Application

Development Mode

1. Start the backend server:

   ```
   # From the root directory
   npm run dev
   ```

2. Start the frontend development server:

   ```
   # From the frontend directory
   npm run dev
   ```

3. Access the application:

   - Frontend: http://localhost:5173
   - Backend API: http://localhost:3000/api

Production Mode

1. Build the frontend:

   ```
   # From the frontend directory
   npm run build
   ```

2. Build the backend:

   ```
   # From the root directory
   npm run build
   ```

3. Start the backend server:

   ```
   npm start
   ```

Testing

1. Run unit tests:

   ```
   # From the root directory
   npm run test
   ```

Search Method

The search functionality for a media archive allows users to search for media items using various parameters, including keywords, photographer names, date ranges, and further filtering constraints. The search queries are optimized for relevance, supporting both exact phrase matches and multi-term searches with weighted boosting.

Since the dataset contains unstructured metadata, the search function must:

- Handle inconsistent metadata.
- Support exact and partial matches.
- Prioritize highly relevant results while avoiding excessive noise.
- Scale efficiently as new media items are continuously added.

Single-term query

This query applies when the user searches with a single keyword (e.g., "Jackson" or "Manchester").
Boosts Exact Match (term) by a factor of 5, ensuring that exact matches rank at the top.
Partial Matching (match) uses Elasticsearch's standard text analysis to find similar words and boosts these by a factor of 3 ensuring these results are included but ranked lower than exact matches. (Helps when users enter a part of a word or an imperfect term, e.g., "Jackson" instead of "Michael Jackson".
Flexible Matching (minimum_should_match: 1) ensures at least one condition (exact or partial) is satisfied.

Multi-Term Query

When a user searches with multiple words (e.g., "Michael Jackson 1995"), we need a different approach:
Boosted Exact Phrase Match (match_phrase) ensures that if a user searches an exact phrase, it is prioritized and boosts by a factor of 6 to show results at the top.
Strict Multi-Word Matching (match with "and") ensures that all words in the query are present and boosts by a factor of 4 to keep this results high in ranking.
Flexible Matching (match with operator: "or") allows results containing at least one of the words boosted by a factor of 2, still appearing in the result but ranked lower.

The solution uses a Boolean query (bool), which allows combining multiple conditions while maintaining flexibility. We implement different strategies for single-term and multi-term searches to improve search relevance.

Filtering

In Elasticsearch, search and filtering serve distinct purposes that significantly impact performance and relevance. Search queries, such as match or match_phrase, are used for full-text searches where relevance scoring is important, meaning Elasticsearch must analyze, tokenize, and rank documents based on how well they match the query. This process is computationally expensive, as it involves TF-IDF or BM25 calculations to determine the best results. In contrast, filters (term, range, exists, etc.) are used for exact matching or numeric constraints and do not contribute to scoring. Because filters narrow down the dataset first, Elasticsearch avoids unnecessary scoring computations, improving efficiency. Additionally, filtered results are cached, meaning repeated queries with the same filters run much faster. This distinction matters in large-scale search systems, as applying filters before full-text searches can drastically reduce the number of documents that need scoring, leading to faster and more scalable search performance.

Scaling to Millions of Media Items

Scaling this solution to handle millions of media items presents several core challenges:

Preprocessing for Metadata Consistency

Metadata preprocessing pipeline significantly improves search accuracy through a comprehensive approach to transforming raw, unstructured metadata into clean, searchable data:

- **Entity Extraction and Classification**
  We could employ natural language processing (NLP) techniques to identify named entities within descriptive text fields to extract and classify data from descriptions.
  For instance, when processing a caption like "Manchester United players celebrate at Old Trafford after winning the 1999 FA Cup final", the system could extract:
  "Manchester United" (organization)
  "Old Trafford" (location)
  "1999" (date)
  "FA Cup final" (event)
  This structured metadata improves searchability without requiring manual tagging.

- **Image-Based Entity Extraction and Classification**
  By leveraging advanced picture-to-text models, we could extract structured metadata directly from images using targeted prompting techniques. These models analyze visual content to identify key elements such as people, locations, objects, and events.
  For example, when processing an image of a football match celebration, the system could generate descriptive text and extract:
  “Manchester United” (organization) from team jerseys or logos
  “Old Trafford” (location) from stadium architecture and signage
  “1999” (date) from scoreboard details or contextual clues
  “FA Cup final” (event) inferred from trophy recognition and crowd reactions
  This approach enhances metadata enrichment by complementing textual descriptions with visual context, improving searchability and categorization. These models are still in quite early stages and there are still huge challenges like testing, model biases, hallucinations.

- **Hierarchical Taxonomies and Controlled Vocabularies**
  To address terminology inconsistencies, we could develop domain-specific hierarchical taxonomies for sports, news, entertainment, and other content categories. During preprocessing, our system could map free-text descriptors to these controlled vocabularies, ensuring consistent categorization across the media library.
  This enables powerful faceted navigation and ensures that searches for higher-level concepts (e.g., "team sports") retrieve relevant content tagged with specific instances (e.g., "football", "basketball").

- **Multilingual Processing and Cross-lingual Search**
  The collection contains metadata in multiple languages. Our preprocessing pipeline should incorporate language detection to identify the primary language of each text field. We then apply language-specific analysis chains, e.g. including stemming, stop word removal, compound word handling
  Additionally, cross-language mappings for key entities could be built, allowing users to search in their preferred language and find relevant results, regardless of the original metadata language.

- **Metadata Quality Scoring and Enhancement**
  Each incoming media item receives a metadata quality score based on completeness, specificity, consistency
  Items with lower scores are flagged for either automated enhancement (using contextual inference

from similar items) as well as manual review.
This scoring system ensures continuous metadata improvement and helps prioritize enhancement efforts.

- **Indexing Strategies and general normalization**
  Ensure all dates are consistently formatted in DD.MM.YYYY before conversion to ISO-8601 (YYYY-MM-DD). Any non-standard formats are detected and normalized before indexing. Standardizing casing and punctuation in text fields (e.g., "U.S.A." → "USA") as well as normalizing spelling variations (e.g., "color" vs. "colour") could also improve search experience.

Architecture to Scale for Millions

Building a search infrastructure that can efficiently handle millions of users and continuous media ingestion requires a scalable and resilient architecture. This document outlines the core components of the system, focusing on search, backend services, ingestion pipelines, and scaling strategies.

Elasticsearch Core

At the heart of the search functionality lies an Elasticsearch cluster, optimized to support large-scale queries and indexing operations.

Optimized Index Design

A well-structured index is crucial for performance and relevancy. The system should employ custom mappings and analyzers tailored to handle media metadata efficiently. Given the multilingual nature of the content, text analysis pipelines should be carefully designed to accommodate language-specific tokenization, stemming, and stop-word removal, ensuring high-quality search results across different languages.

Sharding Strategy

To distribute the load effectively, the main index should follow a calculated sharding approach based on projected data volume and expected query patterns. By ensuring that each shard holds a balanced subset of the data, performance bottlenecks can be prevented and enable smooth horizontal scaling of the Elasticsearch cluster.

Field Data Management

Optimizing which fields to index and how they are analyzed significantly impacts query speed and memory consumption. Some fields require full-text search capabilities, while others only need keyword indexing for exact matching. Additionally, frequently queried fields should be carefully managed to reduce field data cache pressure, ensuring that memory/compute-intensive operations do not degrade cluster performance.

Ingestion Pipeline

Given the continuous addition of new media every minute, the system should employ a robust ingestion pipeline designed for real-time processing.

Event-Driven Intake

A scalable event-driven architecture ensures that new media additions are processed reliably. When new content is meant to be ingested, events should be published to a distributed queue, allowing workers to handle them asynchronously. This prevents bottlenecks and ensures smooth processing even during traffic spikes.

Preprocessing Service

Before data is to be indexed into Elasticsearch, it should undergo preprocessing, which includes validation, normalization, and metadata enrichment. This step guarantees data consistency and enhances search capabilities by extracting relevant features such as tags, categories, and language-specific attributes.

Batched Operations

To maximize indexing efficiency, bulk operations can be utilized instead of indexing individual documents. This reduces the overhead associated with frequent writes and improves cluster performance.

Parallel Processing

The ingestion pipeline should support multi-threaded processing, allowing the system to handle high peak in media uploads efficiently. By distributing tasks across multiple workers, we ensure that indexing remains performant even as data volume increases.

Scaling Strategy

The architecture should be designed for horizontal scalability at multiple levels, ensuring that the system can grow with increasing demand.

Application Tier Scaling

The API servers should be stateless, allowing them to scale horizontally behind a load balancer. This ensures that increasing request volumes can be handled by adding more instances as needed, without requiring major architectural changes.

Elasticsearch Cluster Growth

The Elasticsearch cluster is built for long-term scalability, with careful shard allocation and node specialization. Dedicated master, data, and ingest nodes can improve efficiency by separating indexing workloads from query execution, reducing contention and improving response times.

Resource Optimization

Certain compute-intensive tasks, such as batch indexing and analytics should run on separate dedicated resources to prevent them from impacting real-time query performance. This separation ensures that the system remains responsive even under heavy indexing loads.

Caching Layer

A tiered caching approach could help offload queries from Elasticsearch. Frequently accessed search results and aggregations could be cached, reducing redundant processing and improving response times for popular queries.

Monitoring and Maintenance

To ensure continued reliability and optimal performance, the system should incorporate a comprehensive monitoring and maintenance strategy.

Performance Metrics

Real-time metrics track key performance indicators such as query latency, indexing throughput, and resource utilization. This data helps identify potential bottlenecks and allows for proactive scaling and optimization.

Index Lifecycle Management

Automated policies manage index optimization, merging, and archival. Older data that is less frequently queried could be transitioned to lower-cost storage solutions, preserving performance for active datasets.

Search Analytics

User query patterns and behavior should be continuously analyzed to refine search relevancy and improve the ranking of results. Insights gathered from these analytics help to craft adjustments to tokenization strategies, boosting rules, and synonym handling, etc.

Health Checks

Regular health checks should monitor Elasticsearch cluster status, node availability, and performance anomalies. Automated alerts should notify administrators of potential issues before they impact users, ensuring high availability and minimal downtime.