Basic Vector Search Engine

1. Project Overview

Develop a basic vector search engine that allows users to search for documents based on vectorized representations. The backend will handle storing documents as vector embeddings, and querying those vectors to find the closest matches to a search query.

2. Functional Requirements

2.1 Document Ingestion

- Allow the backend to receive documents (text-based) via an API endpoint.
- Vectorize the incoming documents into embeddings (e.g., using TF-IDF, Word2Vec, or a pre-trained transformer model like BERT).
- Store the document's metadata and its corresponding vector representation in a database.

2.2 Search Query

- Accept a user's search query through an API endpoint.
- Vectorize the search query into an embedding using the same method as the document ingestion.
- Perform a similarity search between the query vector and the stored document vectors.
  - Use a similarity metric such as **cosine similarity** or **Euclidean distance**.
- Return the top-N most similar documents based on the similarity score.

2.3 Results Pagination

- Implement pagination for search results to allow users to browse results in chunks.

2.4 Optional Features

- **Filtering**: Add support for filtering results based on metadata (e.g., document category, date).
- **Ranking**: Rank the results based on the similarity score.

3. Non-Functional Requirements

3.1 Efficiency

- Ensure that the search process can efficiently handle large numbers of documents.
- Use an indexing method (e.g., FAISS or HNSW) to speed up vector search for large datasets if the dataset grows.

3.2 Scalability

- The system should scale to handle an increasing number of documents and queries without significant performance degradation.
- Consider handling large datasets in chunks or using batch processing for vectorizing and storing documents.

3.3 Security

- Ensure that document ingestion and search queries are secured via HTTPS.
- Protect sensitive information, if applicable, during storage and querying.

4. Technical Requirements

4.1 Technology Stack

- **Backend**: Node.js (Express), Python (Flask, Django), or any backend framework of your choice.
- **Vectorization**:
  - Use a vectorization method like **TF-IDF** (for simplicity), **Word2Vec**, or a pre-trained transformer model (e.g., BERT, SentenceBERT).
- **Similarity Calculation**: Cosine similarity or Euclidean distance for measuring vector closeness.
- **Database**:
  - Use a NoSQL database like **MongoDB** or a vector-database solution like **Pinecone**, **Weaviate**, or **FAISS** (for high-performance vector search).
  - Optionally, a traditional database like PostgreSQL can store document metadata.
- **API Endpoints**:
  - `/ingest`: Endpoint to upload and vectorize a document.
  - `/search`: Endpoint to perform a vector search based on a query string.
  - `/document/:id`: Optional endpoint to retrieve a specific document by ID.

4.2 Vectorization Method

- **TF-IDF**: Vectorize using term frequency-inverse document frequency for simplicity.
- **Word Embeddings**: Use pre-trained embeddings like Word2Vec or GloVe.

- **Transformers**: Optionally, use a deep learning model like BERT to generate contextual embeddings for more accurate vector representations.

4.3 Endpoints

- `POST /ingest`: Upload a document (text) and vectorize it.
- `GET /search`: Accept a query and return the top-N similar documents, sorted by similarity.
- `GET /document/:id`: (Optional) Fetch a document by its unique identifier.

5. User Stories

5.1 Ingest Documents

As a backend service, I should ingest documents and store their vectorized representations to enable vector-based search.

5.2 Search Documents

As a user, I want to input a search query and receive the most similar documents based on vector representations.

5.3 Paginate Results

As a user, I want to paginate through search results so I can browse the top similar documents in chunks.