Vector Stores in LangChain and RAG Applications

1. Introduction: The Need for Vector Stores in RAG Applications

Vector stores are a "very important component" of Retrieval Augmented Generation (RAG) applications. *Why* vector stores are needed, moving from a simple keyword-matching recommendation system to a more sophisticated semantic understanding of data.

The Problem with Keyword Matching: Initially, building a movie recommendation system based on keyword matching (director, actor, genre, release date). While seemingly straightforward, this approach has significant flaws:

- **Inaccurate Similarity:** Movies with similar keywords might be semantically very different (e.g., "My Name Is Khan" and "Kabhi Alvida Naa Kehna" share director/actors but have distinct plots).
- **Missed Similarities:** Movies with different keywords might be semantically very similar (e.g., "Taare Zameen Par" and "A Beautiful Mind" share a central theme of a brilliant character struggling with a problem, but have no common keywords).

This highlights that "underlying principle jo apne use kiya hai woh bahut simple hai keyword matching ka" (the underlying principle you used is very simple, keyword matching). To provide quality recommendations, the system needs to understand the *meaning* or *plot* of movies, not just surface-level keywords.

The Solution: Semantic Comparison using Embeddings: The improved approach involves comparing the *plot* or *story* of two movies. This requires transforming textual data (movie plots) into a numerical format that computers can understand and compare meaningfully.

- **Embeddings:** "Embeddings is basically technique jiski help se aap kisi bhi piece of text ka jo semantic meaning hai usko numbers ke form mein represent kar sakte ho" (Embeddings is basically a technique with the help of which you can represent the semantic meaning of any piece of text in the form of numbers). A neural network processes text and represents its meaning as a vector (a list of numbers).
- **Vector Similarity:** Once texts are converted to vectors, "ab in numbers ke beech mein similarity nikalna kind of aasan hai" (now it's kind of easy to find similarity between these numbers). This is typically done by calculating the angular distance (e.g., cosine similarity) between vectors in a high-dimensional coordinate system. A smaller angular distance indicates higher similarity.

This process involves:

1. Collecting movie plots.
2. Generating embedding vectors for each plot.
3. Calculating similarities between these embedding vectors.
4. Recommending movies with high similarity scores.

2. Challenges in Building Semantic Search Systems

While the conceptual solution of using embeddings is clear, building such a system at scale presents three major challenges:

1. **Generating Embedding Vectors:** For millions of movies, generating embeddings for each is a massive computational task.

- **Storage of Embedding Vectors:**These vectors need to be stored efficiently for repeated use.
- "Embedding vectors ko aap normal relational databases ke andar store nahi karte jaise ki MySQL ya Oracle" (You don't store embedding vectors in normal relational databases like MySQL or Oracle) because relational databases cannot calculate similarity between them.

- A "different type of database" is required.
- **Semantic Search (Efficient Retrieval):**Given a query vector, finding the most similar vectors among millions by performing a linear, vector-by-vector comparison is "computationally bahut bhaari kaam hai" (a computationally very heavy task) and "will take a lot of time," leading to a slow application.
- An "intelligent semantic search" mechanism is needed to find similar vectors quickly without comparing every single one.

Vector Stores Address These Challenges.

3. What is a Vector Store? Core Features.

A vector store is defined as "a system designed to store and retrieve data represented as numerical vectors." It's essential for applications requiring vector storage and retrieval.

Key features of a vector store:

1. **Storage:**

- The primary feature is to store vectors and their associated metadata (e.g., movie ID, name, etc.).
- Offers **in-memory** storage (for quick lookups, non-persistent) and **on-disk** storage (for durability, persistent storage in a database or hard drive). "Agar aap chhota-mota application bana rahe hain to aap in-memory use karoge agar aap ek proper enterprise level application bana rahe ho to aap on-disk use karoge" (If you are making a small application, you will use in-memory; if you are making a proper enterprise-level application, you will use on-disk).

1. **Similarity Search:**

- Enables the retrieval of vectors most similar to a given query vector by calculating a similarity score. "It helps retrieve the vectors most similar to a query vector."

1. **Indexing:**

- This is the most "interesting feature" and crucial for fast similarity searches on high-dimensional vectors.
- "Vector stores provide a data structure or a method that enables fast similarity searches on high dimensional vectors."
- **How it works (clustering example):** Instead of comparing a query vector with all 10 lakh (1 million) stored vectors (which is O(N) complexity), indexing organizes vectors into clusters. The query vector is first compared with the centroids of these clusters (e.g., 10 comparisons for 10 clusters). Once the most similar cluster is identified, the search is narrowed down to only the vectors within that cluster (e.g., 1 lakh comparisons). This drastically reduces computation.
- Techniques like Approximate Nearest Neighbour (ANN) lookup are used for this.

1. **CRUD Operations:**

- Allows standard Create, Retrieve, Update, and Delete operations for vectors, similar to traditional databases.

4. Vector Store vs. Vector Database

The terms "vector store" and "vector database" are often used interchangeably, but there's a crucial distinction:

- **Vector Store:**
- A "lightweight library or service" primarily focused on "storing vectors and performing similarity search."
- May *not* include traditional database features like transactions, rich query languages (SQL-like), or role-based access control.

- Ideal for "prototyping and smaller scale applications."
- **Example:** FAISS (Facebook AI Similarity Search).
- **Vector Database:**
- A "full-fledged database system" designed to store and query vectors.
- "Offers additional database-like features" such as:
- Distributed architecture (for scalability).
- Durability and persistence (backup/restore).
- Metadata handling.
- ACID (or near-ACID) guarantees for transactions.
- Concurrency control (multiple users).
- Authentication and authorization (security).
- Used in "production environments where you need significant scaling or are dealing with very large datasets."
- **Examples:** Milvus, Qdrant, Weaviate, Pinecone.

The Key Relationship: "A vector database is effectively a vector store with extra database features." Therefore, "har vector database hota vector store hi hai" (every vector database is, after all, a vector store), but "har vector store vector database nahi hota" (not every vector store is a vector database).

5. Use Cases for Vector Stores

Vector stores are widely used in applications that involve vector data:

- **Recommender Systems:** (As demonstrated in the video).
- **Semantic Search:** Any application requiring semantic understanding and search between vectors.
- **RAG (Retrieval Augmented Generation):** Central to RAG applications, which the video will cover in detail later.
- **Image/Multimedia Search:** When multimedia content is converted into embeddings for similarity-based retrieval.

In essence, if your application deals with "vectors ko use kar raha hai, unko store kar raha hai, unko retrieve kar raha hai" (using vectors, storing them, retrieving them), vector stores will perform "much much better than traditional relational databases."

6. Vector Store Implementation in LangChain

LangChain, as a framework for LLM-based applications, provides extensive support for various vector stores due to the pervasive use of embedding vectors in such applications.

- **Built-in Support:** LangChain offers "wrappers" for all major vector stores like Pinecone, Chroma, Qdrant, Weaviate, etc.
- **Common Interface:** A key design principle in LangChain is that these wrappers "share the same method signatures." This means common functions (e.g., from_documents, add_documents, similarity_search) work identically across different vector stores.
- **Interchangeability:** This common interface allows developers to easily switch between vector stores (e.g., from FAISS to Pinecone) with minimal code changes, making development flexible and scalable. "Aap jhat se Fay ko replace kar sakte ho with Pinecone aur aapko apne existing code mein kuch bhi zyada change nahi

karna padega" (You can quickly replace FAISS with Pinecone and you won't have to make many changes to your existing code).

7. Practical Demonstration with ChromaDB in LangChain

Using **ChromaDB** with LangChain. Chroma is highlighted as a "lightweight open source vector database that is especially friendly for local development and small to medium scale production needs." It sits "between a vector store and a vector database" offering database-like features while remaining lightweight.

ChromaDB Data Organization:

- **Tenant:** Top-level user/organization.
- **Database:** Multiple databases per tenant.
- **Collection:** Equivalent to a table in RDBMS, storing multiple documents.
- **Document:** Contains the embedding vector and its associated metadata.

LangChain Operations with ChromaDB:

1. **Installation and Imports:** Install necessary libraries (langchain-openai, chromadb). Import OpenAIEmbeddings and Chroma.
2. **Creating Document Objects:** LangChain uses Document objects, which contain page_content (actual text) and metadata (associated data).
3. **Initializing Vector Store:** Create a Chroma object, specifying:

- embedding_function: The model to generate embeddings (e.g., OpenAIEmbeddings).
- persist_directory: Local path to store the vector database (e.g., my_chroma_db). Chroma stores data as a SQLite3 database.
- collection_name: Name of the collection (table) within Chroma (e.g., sample).
- **Adding Documents (add_documents):**Documents are added to the vector store.
- Each document automatically gets a unique ID, or custom IDs can be provided.

1. **Retrieving Documents (get):** View stored documents, their IDs, embeddings, and metadata.

- **Similarity Search (similarity_search):**Pass a query string and k (number of top similar results to return).
- The system semantically searches the embeddings to find the most relevant documents. For example, querying "Who among these are a bowler?" returns Jasprit Bumrah and Ravindra Jadeja.

1. **Similarity Search with Score (similarity_search_with_score):** Returns the similarity score alongside each result. A lower score indicates greater similarity (as it represents distance).
2. **Metadata Filtering:** Perform searches while filtering based on metadata (e.g., find players from "Chennai Super Kings"). This is a powerful feature for refining search results.
3. **Updating Documents (update_documents):** Update existing documents by providing their ID and the new Document object.
4. **Deleting Documents (delete):** Delete documents by providing their IDs.