Vector Databases for Recommendation Systems and RAG Cheat Sheet

Estimated Reading Time: 15 minutes

What is RAG?

RAG is a framework that enhances language models by retrieving relevant information from external sources and using it to generate more accurate grounded responses, and thereby reducing the amount of potential hallucinations that may occur.

Core Problems RAG Solves:

- LLMs have limited context windows - not possible to include all information in a single prompt
- Knowledge is frozen at the point at which they are trained
- LLMs can hallucinate facts

RAG Pipeline Steps

1. **Source documents** that are relevant to the use case are provided and potentially split into smaller chunks
2. **Source documents or their chunks are embedded**
3. **Sources and their embeddings are stored** in a vector database such as Chroma DB
4. **User's prompt is received**
5. **User's prompt is embedded**
6. **Retriever selects the chunks** from the vector store that best match the user's prompt
7. **Retrieved text is combined** with the user's original prompt to produce an augmented prompt
8. **Augmented prompt is passed to the LLM** to produce a context-aware response

Vector Database Responsibilities in RAG

Vector databases can handle several key responsibilities:

- **Embedding** both source documents and user prompts
- **Storing** those embeddings
- **Retrieving** the most relevant matches
- **Supplying** the retrieved content for prompt augmentation

Note: Some RAG pipeline steps, such as embedding the documents and prompts (steps 2 and 5), can also be performed externally. In such cases, the vector database is used primarily for storing and retrieving vectors.

Why Use Vector Databases for RAG Steps

1. **Prevents critical mistakes** - Such as accidentally using different embedding models for source documents and user prompts, or incorrectly linking embeddings to their corresponding source documents.

2. **Development becomes faster and cleaner** - With fewer moving parts and less custom logic, your codebase stays simpler and easier to maintain, which in turn makes it faster to implement and debug.

3. **Performance is a major advantage** - Vector databases are built for high-speed, scalable semantic searches using advanced indexing algorithms. Custom-built alternatives typically cannot match this performance without significant optimization effort.

Common RAG Pipeline Pitfalls

Critical Mistakes to Avoid:

- **Using different embedding models** for your documents and queries can break retrieval entirely.
    - **Solution:** Use the same embedding model throughout. Vector databases usually handle this automatically.
- **Poor chunking strategy** - creating chunks that are either too large or too small.
    - **Solution:** Choose a chunk size that is long enough to preserve meaning without including too much irrelevant content.
- **Forgetting to re-embed content** after changing the distance metric or embedding model.
    - **Note:** For some databases, such as Chroma DB, this cannot be done on an existing database and might necessitate cloning your collection.
- **Assuming that the retrieved result is always the best answer.**
    - **Solution:** Always test your results, because a little tuning can make a big difference.

Chroma DB Operations

Creating Collections

```
import chromadb
import chromadb.utils.embedding_functions as embedding_functions
# Define the embedding model
sentence_transformer_ef = embedding_functions.SentenceTransformerEmbeddingFunction(
    model_name="all-MiniLM-L6-v2"
)
# Define chromadb client
client = chromadb.Client()
# Create collection
collection = client.create_collection(
    name="my_collection",
    metadata={"description": "A collection for storing user data"},
    configuration={
        "embedding_function": sentence_transformer_ef
    }
)
```

Connecting to Existing Collections

```
# Connect to existing collection
collection = client.get_collection(name="my_collection")
```

Modifying Collections

```
# Alter collection using modify method
collection.modify(
    name="new_collection_name",
    metadata={"key": "value"}
)
```

Important: Certain changes, such as modifying the embedding model or distance metric, cannot be made with an existing collection. To apply these changes, you need to clone a collection, which can be an expensive operation computationally if you have a substantial amount of data stored in your collection.

Adding Documents

```
# Add documents to collection
collection.add(
    documents=[
        "This is a document about LangChain",
        "This is a document about LlamaIndex"
    ],
    metadatas=[
        {"source": "langchain.com", "version": "0.2"},
        {"source": "llamaindex.ai", "version": "0.12"}
    ],
    ids=["id1", "id2"]
)
```

Note: Make sure to include an ID for each document in a list passed to the `ids` parameter.

Retrieving Documents

```
# Get all documents (returns Python dictionary)
results = collection.get()
# Get specific documents by ID
results = collection.get(ids=["id1"])
# Include embeddings in results
results = collection.get(include=['embeddings'])
```

Note: The get method does not return the embeddings by default to keep the output clean. However, the embeddings are actually stored in the collection.

Updating Documents

```
# Update existing documents
collection.update(
    ids=["id1"],
    metadatas=[{"source": "langchain.com", "version": "0.3"}],
    documents=["This an updated document about LangChain"]
)
```

Important: Chroma DB handles re-embedding in the background, re-computing the embeddings for the document as soon as the update is submitted.

Deleting Documents

```
# Delete by IDs
collection.delete(ids=["id1"])
# Delete using metadata filter
collection.delete(where={"source": "doc_to_delete.pdf"})
# Combine IDs and filters
collection.delete(
    ids=["id1"],
    where={"version": "1.0"}
)
```

Distance Functions in Chroma DB

Chroma DB uses the **Hierarchical Navigable Small World (HNSW)** algorithm to perform approximate nearest neighbor searches.

The `space` parameter defines the distance function used in the embedding space:

- `l2` (default) - squared L2 norm
- `cosine` - cosine distance
- `ip` - inner product or dot product distance

Configuring Distance Function

```
# Specify distance function at collection creation
collection = client.create_collection(
    name="my_collection",
    metadata={"description": "A collection for storing user data"},
    configuration={
        "embedding_function": sentence_transformer_ef,
        "hnsw": {"space": "cosine"}
    }
)
```

What Vector Databases Don't Handle

Some RAG pipeline tasks usually happen outside the database:

- **Chunking** is usually done before data enters the vector database
- **Extra retrieval logic** such as filtering and re-ranking may require extra tools
- **Prompt augmentation** is typically handled outside the database
- **Integration with LLMs** is not built into most vector databases

RAG Frameworks

Tools such as LangChain and LlamaIndex wrap around your vector database and help manage the pipeline from document prep to final response.

These frameworks:

- Provide additional structure.
- Simplify the development and deployment of your RAG application even further.
- Fill the gaps by connecting all the pieces.

Key Takeaways

- **Vector databases are the foundation** that makes retrieval augmented generation work.
- **RAG enhances LLM response quality** by retrieving relevant external information, helping the model generate more accurate and well-supported outputs.
- **Using a vector database for all relevant RAG steps** helps prevent critical mistakes, speeds up application development, and optimizes performance.
- **Collections are a way to organize your data** within Chroma DB.
- **Metadata helps you keep track** of the purpose and contents of your collection.
- **Always test your results** - a little tuning can make a big difference.

Author(s)

Wojciech "Victor" Fulmyk