Chroma DB Tutorial: A Step-By-Step Guide

With the rise of Large Language Models (LLMs) and their applications, we have seen an increase in the popularity of integration tools, LLMOps frameworks, and vector databases. This is because working with LLMs requires a different approach than traditional machine learning models.

One of the core enabling technologies for LLMs is vector embeddings. While computers cannot directly understand text, embeddings represent text numerically. All user-provided text is converted to embeddings, which are used to generate responses.

Converting text into embedding is a time-consuming process. To avoid that, we have vector databases explicitly designed for efficient storage and retrieval of vector embeddings.

In this tutorial, we will learn about vector stores and Chroma DB, an open-source database for storing and managing embeddings. Moreover, we will learn how to add and remove documents, perform similarity searches, and convert our text into embeddings.

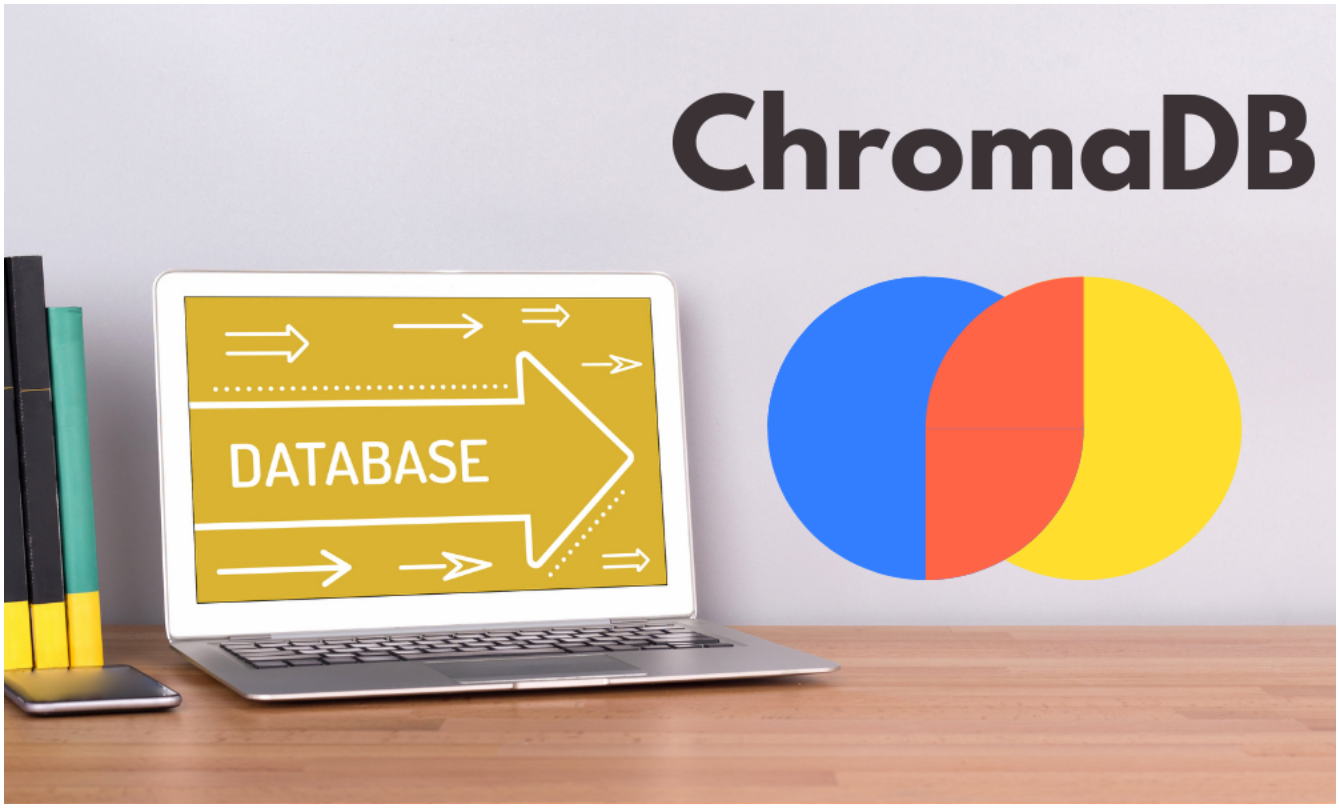

Image by author

What Are Vector Stores?

Vector stores are databases explicitly designed for storing and retrieving vector embeddings efficiently. They are needed because traditional databases like SQL are not optimized for storing and querying large vector data.

Embeddings represent data (usually unstructured data like text) in numerical vector formats within a high-dimensional space. Traditional relational databases are not well-suited to storing and searching these vector representations.

Vector stores can index and quickly search for similar vectors using similarity algorithms. It allows applications to find related vectors given a target vector query.

In the case of a personalized chatbot, the user inputs a prompt for the generative AI model. The model then searches for similar text within a

collection of documents using a similarity search algorithm. The resulting information is then used to generate a highly personalized and accurate response. It is made possible through embedding and vector indexing within vector stores.

What is Chroma DB?

Chroma DB is an open-source vector store used for storing and retrieving vector embeddings. Its main use is to save embeddings along with metadata to be used later by large language models. Additionally, it can also be used for semantic search engines over text data.

Chroma DB key features:

- Supports different underlying storage options like DuckDB for standalone or ClickHouse for scalability.
- Provides SDKs for Python and JavaScript/TypeScript.
- Focuses on simplicity, speed, and enabling analysis.

Chroma DB offers a self-hosted server option. If you need a managed vector database platform, check out the Pinecone Guide for Mastering Vector Databases.

Image from Chroma

How does Chroma DB work?

1. First, you have to create a collection similar to the tables in the relations database. By default, Chroma converts the text into the embeddings using `all-MiniLM-L6-v2`, but you can modify the collection to use another embedding model.
2. Add text documents to the newly created collection with metadata and a unique ID. When your collection receives the text, it automatically

converts it into embedding.
3. Query the collection by text or embedding to receive similar documents. You can also filter out results based on metadata.

In the next part, we will use Chroma and OpenAI API to create our own vector DB.

Getting Started With Chroma DB

In this section, we will create a vector database, add collections, add text to the collection, and perform a query search.

First, we will install `chromadb` for the vector database and `openai` for a better embedding model. Make sure you have set up the OpenAI API key.

Note: Chroma requires SQLite version 3.35 or higher. If you experience problems, either upgrade to Python 3.11 or install an older version of `chromadb`.

```
!pip install chromadb openai
```

You can create an in-memory DB for testing by creating a Chroma DB client without settings.

In our case, we will create a persistent database that will be stored in the “db/” directory and use DuckDB on the backend.

```
import chromadb
from chromadb.config import Settings


client = chromadb.Client(Settings(chroma_db_impl="duckdb+parquet",
                                  persist_directory="db/"
                                ))
```

After that, we will create a collection object using the client. It is similar to creating a table in a traditional database.

```
collection = client.create_collection(name="Students")
```

To add text to our collection, we need to generate random text about a student, club, and university. You can generate random text using ChatGPT. It is pretty simple.

```
student_info = """
Alexandra Thompson, a 19-year-old computer science sophomore with a 3.7 GPA,
is a member of the programming and chess clubs who enjoys pizza, swimming, and h
in her free time in hopes of working at a tech company after graduating from the
"""

club_info = """
The university chess club provides an outlet for students to come together and e
the classic strategy game of chess. Members of all skill levels are welcome, fro
the rules to experienced tournament players. The club typically meets a few time
participate in tournaments, analyze famous chess matches, and improve members' s
"""

university_info = """
The University of Washington, founded in 1861 in Seattle, is a public research u
with over 45,000 students across three campuses in Seattle, Tacoma, and Bothell.
As the flagship institution of the six public universities in Washington state,
UW encompasses over 500 buildings and 20 million square feet of space,
including one of the largest library systems in the world.
```

Now, we will use the `add` function to add text data with metadata and unique IDs. After that, Chroma will automatically download the `all-MiniLM-L6-v2` model to convert the text into embeddings and store it in the “Students” collection.

```
collection.add(
    documents = [student_info, club_info, university_info],
    metadatas = [{"source": "student info"},{"source": "club info"},{'source':'u
    ids = ["id1", "id2", "id3"]
)
```

To run a similarity search, you can use the `query` function and ask questions in natural language. It will convert the query into embedding and use similarity algorithms to come up with similar results. In our case, it is returning two similar results.

```
results = collection.query(
    query_texts=["What is the student name?"],
    n_results=2
)

results
```

Embeddings

You can use any high-performing embedding model from the embedding list. You can even create your custom embedding functions.

In this section, we will use the line OpenAI embedding model called “text-embedding-ada-002” to convert text into embedding.

After creating the OpenAI embedding function, you can add the list of text documents to generate embeddings.

Discover how to use the OpenAI API for Text Embeddings and create text classifiers, information retrieval systems, and semantic similarity detectors.

```
from chromadb.utils import embedding_functions
openai_ef = embedding_functions.OpenAIEmbeddingFunction(
                model_name="text-embedding-ada-002"
            )
students_embeddings = openai_ef([student_info, club_info, university_info])
print(students_embeddings)
```

```
[[-0.01015068031847477, 0.0070903063751757145, 0.010579396970570087, -0.04118313
```

Instead of using the default embedding model, we will load already created embedding directly to the collections.

1. We will use the `get_or_create_collection` function to create a new collection called "Students2". This function is different from `create_collection`. It will get a collection or create if it doesn't exist already.
2. We will now add embedding, text documents, metadata, and IDs to our newly created collection.

```
collection2 = client.get_or_create_collection(name="Students2")

collection2.add(
    embeddings = students_embeddings,
    documents = [student_info, club_info, university_info],
    metadatas = [{"source": "student info"},{"source": "club info"},{'source':'u
    ids = ["id1", "id2", "id3"]
)
```

There is another, more straightforward method, too. You can add an OpenAI embedding function while creating or accessing the collection. Apart from

OpenAI, you can use Cohere, Google PaLM, HuggingFace, and Instructor models.

In our case, adding new text documents will run an OpenAI embedding function instead of the default model to convert text into embeddings.

```
collection2 = client.get_or_create_collection(name="Students2",embedding_functio

collection2.add(
    documents = [student_info, club_info, university_info],
    metadatas = [{"source": "student info"},{"source": "club info"},{'source':'u
    ids = ["id1", "id2", "id3"]
)
```

Let's see the difference by running a similar query on the new collection.

```
results = collection2.query(
    query_texts=["What is the student name?"],
    n_results=2
)

results
```

Our results have improved. The similarity search now returns information about the university instead of a club. Additionally, the distance between the vectors is lower than the default embedding model, which is a good thing.

Updating and Removing Data

Just like relational databases, you can update or remove the values from the collections. To update the text and metadata, we will provide the specific ID for the record and new text.

```
collection2.update(
    ids=["id1"],
    documents=["Kristiane Carina, a 19-year-old computer science sophomore with
    metadatas=[{"source": "student info"}],
)
```

Run a simple query to check if the changes have been made successfully.

```
results = collection2.query(
    query_texts=["What is the student name?"],
    n_results=2
)

results
```

As we can see, instead of Alexandra, we got Kristiane.

To remove a record from the collection, we will use the `delete` function and specify a unique ID.

```
collection2.delete(ids = ['id1'])
```

```
results = collection2.query(
    query_texts=["What is the student name?"],
    n_results=2
)

results
```

The student information text has been removed; instead of that, we get the next best results.

Collection Management

In this section, we will learn about the collection utility function that will make our lives much easier.

We will create a new collection called “vectordb” and add the information about the Chroma DB cheat sheet, documentation, and JS API with metadata.

```
vector_collections = client.create_collection("vectordb")
```

```
vector_collections.add(
    documents=["This is Chroma DB CheatSheet",
               "This is Chroma DB Documentation",
               "This document Chroma JS API Docs"],
    metadatas=[{"source": "Chroma Cheatsheet"},
    {"source": "Chroma Doc"},
    {'source':'JS API Doc'}],
    ids=["id1", "id2", "id3"]
)
```

We will use the `count()` function to check how many records the collection has.

```
vector_collections.count()
```

```
3
```

To view all the records from the collection, use the `.get()` function.

```
vector_collections.get()
```

```
{'ids': ['id1', 'id2', 'id3'],
 'embeddings': None,
 'documents': ['This is Chroma DB CheatSheet',
  'This is Chroma DB Documentation',
  'This document Chroma JS API Docs'],
 'metadatas': [{'source': 'Chroma Cheatsheet'},
  {'source': 'Chroma Doc'},
  {'source': 'JS API Doc'}]}
```

To change the collection name, use the `modify()` function. To view all collection names, use `list_collections()`.

```
vector_collections.modify(name="chroma_info")
```

```
client.list_collections()
```

It appears that we have effectively renamed "vectordb" as "chroma_info".

```
[Collection(name=Students),
 Collection(name=Students2),
 Collection(name=chroma_info)]
```

To access any new collection, you can use `get_collection` with the collection's name.

```
vector_collections_new = client.get_collection(name="chroma_info")
```

We can delete a collection using the client function `delete_collection` and specify the collection name.

```
client.delete_collection(name="chroma_info")
```

```
client.list_collections()
```

```
[Collection(name=Students), Collection(name=Students2)]
```

We can delete the entire database collection by using `client.reset()`. However, it is not recommended as there is no way to restore the data after deletion.

```
client.reset()
client.list_collections()
```

```
[]
```

Conclusion

Vector stores like Chroma DB are becoming essential components of large language model systems. By providing specialized storage and efficient retrieval of vector embeddings, they enable fast access to relevant semantic information to power LLMs.

In this Chroma DB tutorial, we covered the basics of creating a collection, adding documents, converting text to embeddings, querying for semantic similarity, and managing the collections.

The next step in the learning process is to integrate vector databases into your generative AI application. You can easily ingest, manage, and retrieve private and domain-specific data for your AI application by following the LlamaIndex tutorial, which is a data framework for Large Language Model (LLM) based applications. Additionally, you can follow the How to Build LLM Applications with LangChain tutorial to dive into the world of LLMOps.

Our certification programs help you stand out and prove your skills are job-ready to potential employers.