Essential Vector Database Operations Using Chroma DB

Estimated time needed: 10 minutes

Objectives

After completing this reading, you'll be able to:

- Understand the concept of vector databases and Chroma DB and their role in managing high-dimensional data.
- List all collections in a Chroma DB database.
- Add, update, and delete entries in a collection.
- Explain the significance of collections and data representation in vector databases.

Introduction to Chroma DB

Chroma DB is a highly optimized vector database designed to perform the operation of efficient data management along with retrieval on the basis of high-dimensional data. Similarity searches in data or handling complex queries with less computational complexity can be taken as the highlight of why this is growing more nowadays, especially when there are high demands of advanced analytics in artificial intelligence and in machine learning applications.

Key Features of Chroma DB:

- **High-Dimensional Data Management:** Chroma DB specializes in storing and querying high-dimensional vectors. It is therefore a suitable data store for applications based on similarity comparisons.
- **Scalability:** It is built to scale with your data, ensuring that performance remains optimal even as your dataset grows.
- **Real-Time Search Capabilities:** Chroma DB supports fast and efficient retrieval of similar items, enabling real-time recommendations and searches.
- **Integration with Machine Learning:** Chroma DB can be seamlessly interfaced with any machine learning framework so that simple applications of algorithms can be performed on the stored data.

Concept of Data in Vector Databases

Vector databases store data in a form that is efficient for similarity search; this makes vector databases best suited for applications like recommendation systems, image search, and natural language processing.

- **Data Representation:** Data in vector databases is represented as high-dimensional vectors. A data point, like an image, text, or user behavior, gets transformed into a numerical form, a vector that captures its features.

- **Role of Collections:** Collections are another sensible division of these vectors. For example, product images and customer reviews and user profile can be divided into their respective collections. This arranges data meaningfully to retrieve it efficiently.

- **Updating Data:** Updating data in a vector database is done by either modifying the vector representation of existing data or adding new vectors to a collection. This usually generates new embeddings for the updated text and re-indexing for search efficiency. Updating ensures that information stored in the database reflects the most current state or feedback.

- **Deleting Data:** The removal of vectors within a collection is referred to as deletion data. This prevents the collection from containing wrong or outdated information. When vectors are deleted from a database entry, re-indexing is sometimes done to improve querying performance so that queries fetch the right results.

List a collection

The following code defines an asynchronous function named **listCollections** that retrieves a list of all collections available in your Chroma DB database.

```
async function listCollections() {
    try {
        const collections = await client.listCollections(); // Check if this method exists
        console.log("Available collections:");
        collections.forEach(collection => {
            console.log(`- ${collection.name}`);
        });
    } catch (error) {
        console.error("Error retrieving collections:", error);
    }
}
// Call the function
listCollections();
```

- `async function listCollections()`: This defines an asynchronous function, allowing the await keyword inside that function, making things much easier to work with promises.
- `try...catch`: This is a block of error handling where the code will run in the try block and if an error occurs it will be caught in the catch block.
- `const collections = A call to the listCollections method on the client object. Returns a promise resolved to a list of collections. The execution is stopped until the promise is resolved with the await` keyword.
- `console.log("Available collections:")`: This is used to print a message in the console to indicate that the next thing to be listed are available collections.
- `collections.forEach(...)`: This loop iterates over each collection in the collections array and prints its name to the console.
- `console.error("Error retrieving collections:", error)`: If an error occurs, this line logs the error message for debugging.

Adding Customer Reviews

Now let's see how we can add customer reviews to a collection. Below is the code that defines an asynchronous function called **addCustomerReviews** that adds reviews to a specified collection.

```
async function addCustomerReviews() {
    try {
        const collection = await client.getOrCreateCollection({
            name: collectionName,
            embeddings: default_emd,
        });
        const reviews = [
            { id: "review_1", text: "This product is amazing! Highly recommend it." },
            { id: "review_2", text: "Not what I expected. Disappointed." },
            // Add more reviews as needed
        ];
        const reviewTexts = reviews.map((review) => review.text);
        const embeddingsData = await default_emd.generate(reviewTexts);
        await collection.add({
            ids: reviews.map((review) => review.id),
            documents: reviewTexts,
            embeddings: embeddingsData,
        });
        console.log("Customer reviews added to collection.");
    } catch (error) {
        console.error("Error adding customer reviews:", error);
    }
}
// Call the function
addCustomerReviews();
```

- `async function addCustomerReviews()`: This function is defined as asynchronous, allowing for the use of await.
- `const collection = await client.getOrCreateCollection(...)`: This line attempts to get or create a collection with the name specified by collectionName, using the default_emd embedding function for the collection.
- `const reviews = [...]`: An array of the customer's reviews including id and text.
- `const reviewTexts = reviews.map((review) => review.text)`: This line extracts the review texts from the reviews array.
- `const embeddingsData = await default_emd.generate(reviewTexts)`: default_emd generate method called here to produce embeddings on the reviewTexts.
- `await collection.add(...)`: This statement inserts the reviews into the collection. It uses IDs, document texts, and the embeddings created above.
- `console.log("Customer reviews added to collection.")`: This statement prints out the success message to the console after adding the reviews
- `catch (error)`: This is the code in case of an error from adding reviews. In that case, it prints out the error message.
- `catch (error)`: This handles any errors that occur while adding reviews and logs the error message.

Updating a Customer Review

Now, let's see how to update a specific customer review using the updateReview function.

```
async function updateReview(reviewId, newText) {
    try {
        const collection = await client.getOrCreateCollection({
            name: collectionName,
            embeddings: default_emd,
        });
        const newEmbedding = await default_emd.generate([newText]);
        await collection.update({
            ids: [reviewId],
            documents: [newText],
            embeddings: [newEmbedding[0]],
        });
        console.log(`Review with ID: ${reviewId} updated.`);
    } catch (error) {
        console.error(`Error updating review: ${error.message}`);
    }
}
// Call the function to update a specific review
updateReview("review_2", "Changed my mind! It's actually quite good.");
```

- `async function updateReview(reviewId, newText)`: This function takes two parameters: reviewId (the ID of the review to be updated) and newText (the new review text).
- `const collection = await client.getOrCreateCollection(...)`: Similar to the previous function, this line retrieves or creates the collection specified by collectionName.
- `const newEmbedding = await default_emd.generate([newText])`: This line generates embeddings for the new text of the review.
- `await collection.update(...)`: This line updates the specified review in the collection, using the provided reviewId, the new text, and its embeddings.
- `console.log(...)`: This logs a success message indicating that the review was updated.
- `catch (error)`: This handles any errors during the update process and logs the error message.

Deleting a Customer Review

Finally, let's see how one would remove a customer review by a deleteReview function.

```
async function deleteReview(reviewId) {
    try {
        const collection = await client.getOrCreateCollection({
            name: collectionName,
            embeddings: default_emd,
        });
        await collection.delete({
            ids: [reviewId],
        });
        console.log(`Review with ID: ${reviewId} deleted.`);
    } catch (error) {
        console.error(`Error deleting review: ${error.message}`);
    }
}
// Call the function to delete a specific review
deleteReview("review_4");
```

- `async function deleteReview(reviewId)`: The function takes a single parameter, reviewId, that is the ID of the review to delete.
- `const collection = await client.getOrCreateCollection(.)`: This was exactly like in the other functions - retrieve or create the collection.
- `await collection.delete(.)`: This deletes the review on the collection with the provided reviewId.
- `console.log(.)`: A success message indicating that the review was deleted.
- `catch (error)`: It will catch any error while deleting and log the error message.

Here is the entire code available for refernece:

```
const { ChromaClient, DefaultEmbeddingFunction } = require("chromadb");
const client = new ChromaClient();
const default_emd = new DefaultEmbeddingFunction();
const collectionName = "customer_reviews_collection";
// Function to delete an existing collection if it exists
async function deleteCollectionIfExists() {
    try {
        const collections = await client.listCollections();
        const existingCollection = collections.find(collection => collection.name === collectionName);
        if (existingCollection) {
            await client.deleteCollection({ name: collectionName });
            console.log("Existing collection deleted.");
        }
    } catch (error) {
        console.error("Error deleting existing collection:", error);
    }
}
// Function to print reviews in the collection with a message
async function printReviews(message) {
    try {
        const collection = await client.getOrCreateCollection({
            name: collectionName,
            embeddings: default_emd,
        });

        const reviews = await collection.get({
            ids: ["review_1", "review_2", "review_3", "review_4", "review_5", "review_6", "review_7", "review_8", "review_9", "review_10"],
        });
        console.log(`${message}`);
        reviews.documents.forEach((doc, index) => {
            console.log(`ID: ${reviews.ids[index]}, Text: ${doc}`);
        });
    } catch (error) {
        console.error("Error fetching reviews:", error);
    }
}
// Function to add customer reviews
async function addCustomerReviews() {
    try {
        const collection = await client.getOrCreateCollection({
            name: collectionName,
            embeddings: default_emd,
        });
        const reviews = [
            { id: "review_1", text: "This product is amazing! Highly recommend it." },
            { id: "review_2", text: "Not what I expected. Disappointed." },
            { id: "review_3", text: "Great quality and fast shipping!" },
            { id: "review_4", text: "Okay product, but not worth the price." },
            { id: "review_5", text: "Excellent customer service!" },
            { id: "review_6", text: "Will buy again, love it!" },
            { id: "review_7", text: "The item arrived damaged, very unhappy." },
            { id: "review_8", text: "Fantastic experience from start to finish." },
            { id: "review_9", text: "Meh, it's alright, nothing special." },
            { id: "review_10", text: "I had high hopes but was let down." },
        ];
        const reviewTexts = reviews.map((review) => review.text);
        const embeddingsData = await default_emd.generate(reviewTexts);
        await collection.add({
            ids: reviews.map((review) => review.id),
            documents: reviewTexts,
            embeddings: embeddingsData,
        });
        console.log("Customer reviews added to collection.");

        // Print reviews after adding
        await printReviews("Reviews after adding:");
    } catch (error) {
        console.error("Error adding customer reviews:", error);
    }
}
// Function to update a customer review
async function updateReview(reviewId, newText) {
    try {
        const collection = await client.getOrCreateCollection({
            name: collectionName,
            embeddings: default_emd,
        });
        // Generate new embeddings for the updated text
        const newEmbedding = await default_emd.generate([newText]);
        // Update the review in the collection
        await collection.update({
            ids: [reviewId],
            documents: [newText],
            embeddings: [newEmbedding[0]],
        });
        console.log(`Review with ID: ${reviewId} updated.`);

        // Print reviews after updating
```

```
            await printReviews("Reviews after updating:");
        } catch (error) {
            console.error(`Error updating review: ${error.message}`);
        }
    }
    // Function to delete a customer review
    async function deleteReview(reviewId) {
        try {
            const collection = await client.getOrCreateCollection({
                name: collectionName,
                embeddings: default_emd,
            });
            // Delete the review from the collection
            await collection.delete({
                ids: [reviewId],
            });
            console.log(`Review with ID: ${reviewId} deleted.`);

            // Print reviews after deleting
            await printReviews("Reviews after deleting:");
        } catch (error) {
            console.error(`Error deleting review: ${error.message}`);
        }
    }
    // Execute the functions
    (async () => {
        // Delete existing collection to ensure a fresh start
        await deleteCollectionIfExists();

        // Add customer reviews
        await addCustomerReviews();

        // Update a review
        await updateReview("review_2", "Changed my mind! It's actually quite good.");

        // Delete a review
        await deleteReview("review_4");
    })();
```

Output:

- After Performing the above query, you might see an output like this:

```
Available collections:
```

- my_flower_collection
- CustomerReviews
- my_grocery_collection11
- customer_reviews_collection

```
Customer reviews added to collection.
Customer reviews after adding.
```

- ID: review_1, Text: ‘This product is amazing! Highly recommend it.’
- ID: review_2, Text: ‘Changed my mind! It’s actually quite good.’
- ID: review_3, Text: ‘Great quality and fast shipping!’
- ID: review_4, Text: ‘Okay product, but not worth the price.’
- ID: review_5, Text: ‘Excellent customer service!’
- ID: review_6, Text: ‘Will buy again, love it!’
- ID: review_7, Text: ‘The item arrived damaged, very unhappy.’
- ID: review_8, Text: ‘Fantastic experience from start to finish.’
- ID: review_9, Text: ‘Meh, it’s alright, nothing special.’
- ID: review_10, Text: ‘I had high hopes but was let down.’

```
Review with ID: review_2 updated.
Reviews after updating:
```

- ID: review_1, Text: ‘This product is amazing! Highly recommend it.’
- ID: review_2, Text: ‘Changed my mind! It’s actually quite good.’
- ID: review_3, Text: ‘Great quality and fast shipping!’
- ID: review_4, Text: ‘Okay product, but not worth the price.’
- ID: review_5, Text: ‘Excellent customer service!’
- ID: review_6, Text: ‘Will buy again, love it!’
- ID: review_7, Text: ‘The item arrived damaged, very unhappy.’
- ID: review_8, Text: ‘Fantastic experience from start to finish.’
- ID: review_9, Text: ‘Meh, it’s alright, nothing special.’
- ID: review_10, Text: ‘I had high hopes but was let down.’

```
Review with ID: review_4 deleted.
Reviews after deleting:
```

- ID: review_1, Text: ‘This product is amazing! Highly recommend it.’
- ID: review_2, Text: ‘Changed my mind! It’s actually quite good.’
- ID: review_3, Text: ‘Great quality and fast shipping!’
- ID: review_5, Text: ‘Excellent customer service!’
- ID: review_6, Text: ‘Will buy again, love it!’
- ID: review_7, Text: ‘The item arrived damaged, very unhappy.’
- ID: review_8, Text: ‘Fantastic experience from start to finish.’
- ID: review_9, Text: ‘Meh, it’s alright, nothing special.’
- ID: review_10, Text: ‘I had high hopes but was let down.’

Explanation of Output:

Available Colletion:

- This part of the output indicates that the listCollections function successfully retrieved the names of all available collections in the Chroma DB instance.
- Each collection is a natural grouping of data, and in this case, the collections may be associated with various customer feedback or product information.
- The collections listed include:
  my_flower_collection:
  CustomerReviews:
  my_grocery_collection11:
  customer_reviews_collection: This is the collection we use in the sample code to add and handle customer reviews.

Customer reviews added to collection

- This message indicates that the addCustomerReviews function was executed successfully and added the above customer reviews into the `customer_reviews_collection`.
- The operation also generates embeddings for review texts so that they can be stored efficiently and retrieved later.
- Reviews will enrich the dataset, which could be available for analysis or retrieval based on customer feedback.

Reviews after adding
This section lists the remaining reviews in the collection after the deletion:
ID: review_1, Text: This product is amazing! Highly recommend it.
ID: review_2, Text: Changed my mind! It's actually quite good. (Updated text)
ID: review_3, Text: Great quality and fast shipping!
ID: review_5, Text: Excellent customer service!
ID: review_6, Text: Will buy again, love it!
ID: review_7, Text: The item arrived damaged, very unhappy.
ID: review_8, Text: Fantastic experience from start to finish.
ID: review_9, Text: Meh, it's alright, nothing special.
ID: review_10, Text: I had high hopes but was let down.

Review with ID: review_2 updated
This message indicates that the review with ID `review_2` has been successfully updated. The previous text for this review, **"Not what I expected. Disappointed,"** has been changed to **"Changed my mind! It's actually quite good."**

Review with ID: review_4 marked for deletion
This line indicates that the review with ID `review_4` has been successfully deleted from the collection. The review stated, **"Okay product, but not worth the price."**

Summary

In this reading, we explored the functionalities of Chroma DB, a powerful vector database designed for high-dimensional data management. Key takeaways include:

Understanding Vector Databases: We discussed the significance of data representation in the form of high-dimensional vectors, which facilitates efficient similarity searches, making it ideal for applications like recommendation systems.

Collections in Chroma DB:
Collections are a collection of logical vectors clumped together to make better and easy search results in comparison to one another: customer reviews, among others.

Core Operations:

1. **Collection List:** The method get_collection_list retrieves the database collections.
2. **Customer Reviews:** One may use the addCustomerReview function to add various reviews of a collection then store their embeddings for the convenient storage and retrieval
3. **Update a Review:** update_review: The updateReview can update the reviews. It performs so by updating the text and hence the embeddings of reviews.
4. **Delete Reviews:** The deleteReview function removes specific reviews from the collection in an integrated manner.
5. **Error Handling:** We've used **try.catch** blocks within the entire code to handle any potential errors that would occur when operations are conducted.

Using these functions, you can now process and utilize customer review information in Chroma DB to really power your applications with excellent data understanding and real-time access.

Author(s)

Richa Arora

Other Contributor(s)

Pratiksha Verma

Skills Network