🧪 Usage Guide

Python Javascript

Initiating a persistent Chroma client

python Copy Code

```python
import chromadb
```

You can configure Chroma to save and load the database from your local machine. Data will be persisted automatically and loaded on start (if it exists).

python Copy Code

```python
client = chromadb.PersistentClient(path="/path/to/save/to")
```

The `path` is where Chroma will store its database files on disk, and load them on start.

The client object has a few useful convenience methods.

python Copy Code

```
client.heartbeat() # returns a nanosecond heartbeat. Useful for making sure
client.reset() # Empties and completely resets the database. ⚠ This is dest
```

Running Chroma in client/server mode

Chroma can also be configured to run in client/server mode. In this mode, the Chroma client connects to a Chroma server running in a separate process.

To start the Chroma server, run the following command:

Command Line Copy Code

```
chroma run --path /db_path
```

Then use the Chroma HTTP client to connect to the server:

python Copy Code

```python
import chromadb
chroma_client = chromadb.HttpClient(host='localhost', port=8000)
```

That's it! Chroma's API will run in `client-server` mode with just this change.

Chroma also provides an async HTTP client. The behaviors and method signatures are identical to the synchronous client, but all methods that would block are now async. To use it, call `AsyncHttpClient` instead:

python Copy Code

```python
import asyncio
import chromadb

async def main():
    client = await chromadb.AsyncHttpClient()
    collection = await client.create_collection(name="my_collection")

    await collection.add(
        documents=["hello world"],
        ids=["id1"]
```

```
    )

asyncio.run(main())
```

Using the Python HTTP-only client

If you are running Chroma in client-server mode, you may not need the full Chroma library. Instead, you can use the lightweight client-only library. In this case, you can install the `chromadb-client` package. This package is a lightweight HTTP client for the server with a minimal dependency footprint.

python Copy Code

```python
pip install chromadb-client
```

python Copy Code

```python
import chromadb
# Example setup of the client to connect to your chroma server
client = chromadb.HttpClient(host='localhost', port=8000)

# Or for async usage:
async def main():
    client = await chromadb.AsyncHttpClient(host='localhost', port=8000)
```

Note that the `chromadb-client` package is a subset of the full Chroma library and does not include all the dependencies. If you want to use the full Chroma library, you can install the `chromadb` package instead. Most importantly, there is no default embedding function. If you add() documents without embeddings, you must have manually specified an embedding function and installed the dependencies for it.

Using collections

Chroma lets you manage collections of embeddings, using the `collection` primitive.

Creating, inspecting, and deleting Collections

Chroma uses collection names in the url, so there are a few restrictions on naming them:

- The length of the name must be between 3 and 63 characters.

- The name must start and end with a lowercase letter or a digit, and it can contain dots, dashes, and underscores in between.
- The name must not contain two consecutive dots.
- The name must not be a valid IP address.

Chroma collections are created with a name and an optional embedding function. If you supply an embedding function, you must supply it every time you get the collection.

python Copy Code

```python
collection = client.create_collection(name="my_collection", embedding_funct
collection = client.get_collection(name="my_collection", embedding_function
```

> If you later wish to `get_collection`, you MUST do so with the embedding function you supplied while creating the collection

The embedding function takes text as input, and performs tokenization and embedding. If no embedding function is supplied, Chroma will use **sentence transformer** as a default.

You can learn more about 🧬 **embedding functions**, and how to create your own.

Existing collections can be retrieved by name with `.get_collection`, and deleted with `.delete_collection`. You can also use `.get_or_create_collection` to get a collection if it exists, or create it if it doesn't.

python Copy Code

```python
collection = client.get_collection(name="test") # Get a collection object f
collection = client.get_or_create_collection(name="test") # Get a collectio
client.delete_collection(name="my_collection") # Delete a collection and al
```

Collections have a few useful convenience methods.

python Copy Code

```python
collection.peek() # returns a list of the first 10 items in the collection
collection.count() # returns the number of items in the collection
collection.modify(name="new_name") # Rename the collection
```

Changing the distance function

`create_collection` also takes an optional `metadata` argument which can be used to customize the distance method of the embedding space by setting the value of `hnsw:space` .

python Copy Code

```python
collection = client.create_collection(
        name="collection_name",
        metadata={"hnsw:space": "cosine"} # l2 is the default
    )
```

Valid options for `hnsw:space` are "l2", "ip, "or "cosine". The **default** is "l2" which is the squared L2 norm.

Distance	parameter	Equation
Squared L2	`l2`	$d = \sum{(A_i - B_i)^2}$
Inner product	`ip`	$d = 1.0 - \sum{(A_i \times B_i)}$
Cosine similarity	`cosine`	$d = 1.0 - \frac{\sum(A_i \times B_i)}{\sqrt{\sum(A_i^2)} \cdot \sqrt{\sum(B_i^2)}}$

Adding data to a Collection

Add data to Chroma with `.add` .

Raw documents:

python Copy Code

```python
collection.add(
    documents=["lorem ipsum...", "doc2", "doc3", ...],
    metadatas=[{"chapter": "3", "verse": "16"}, {"chapter": "3", "verse": "
    ids=["id1", "id2", "id3", ...]
)
```

If Chroma is passed a list of `documents` , it will automatically tokenize and embed them with the collection's embedding function (the default will be used if none was supplied at collection creation). Chroma will also store the `documents` themselves. If the documents are too large to embed using the chosen embedding function, an exception will be raised.

Each document must have a unique associated `id`. Trying to `.add` the same ID twice will result in only the initial value being stored. An optional list of `metadata` dictionaries can be supplied for each document, to store additional information and enable filtering.

Alternatively, you can supply a list of document-associated `embeddings` directly, and Chroma will store the associated documents without embedding them itself.

python Copy Code

```python
collection.add(
    documents=["doc1", "doc2", "doc3", ...],
    embeddings=[[1.1, 2.3, 3.2], [4.5, 6.9, 4.4], [1.1, 2.3, 3.2], ...],
    metadatas=[{"chapter": "3", "verse": "16"}, {"chapter": "3", "verse": "
    ids=["id1", "id2", "id3", ...]
)
```

If the supplied `embeddings` are not the same dimension as the collection, an exception will be raised.

You can also store documents elsewhere, and just supply a list of `embeddings` and `metadata` to Chroma. You can use the `ids` to associate the embeddings with your documents stored elsewhere.

python Copy Code

```python
collection.add(
    embeddings=[[1.1, 2.3, 3.2], [4.5, 6.9, 4.4], [1.1, 2.3, 3.2], ...],
    metadatas=[{"chapter": "3", "verse": "16"}, {"chapter": "3", "verse": "
    ids=["id1", "id2", "id3", ...]
)
```

Querying a Collection

Chroma collections can be queried in a variety of ways, using the `.query` method.

You can query by a set of `query_embeddings`.

python Copy Code

```python
collection.query(
    query_embeddings=[[11.1, 12.1, 13.1],[1.1, 2.3, 3.2], ...],
    n_results=10,
    where={"metadata_field": "is_equal_to_this"},
```

```
    where_document={"$contains":"search_string"}
)
```

The query will return the `n_results` closest matches to each `query_embedding`, in order. An optional `where` filter dictionary can be supplied to filter by the `metadata` associated with each document. Additionally, an optional `where_document` filter dictionary can be supplied to filter by contents of the document.

If the supplied `query_embeddings` are not the same dimension as the collection, an exception will be raised.

You can also query by a set of `query_texts`. Chroma will first embed each `query_text` with the collection's embedding function, and then perform the query with the generated embedding.

python Copy Code

```python
collection.query(
    query_texts=["doc10", "thus spake zarathustra", ...],
    n_results=10,
    where={"metadata_field": "is_equal_to_this"},
    where_document={"$contains":"search_string"}
)
```

You can also retrieve items from a collection by `id` using `.get`.

python Copy Code

```python
collection.get(
 ids=["id1", "id2", "id3", ...],
 where={"style": "style1"}
)
```

`.get` also supports the `where` and `where_document` filters. If no `ids` are supplied, it will return all items in the collection that match the `where` and `where_document` filters.

Choosing which data is returned

When using get or query you can use the include parameter to specify which data you want returned - any of `embeddings`, `documents`, `metadatas`, and for query, `distances`. By default, Chroma will return the `documents`, `metadatas` and in the case of query, the

`distances` of the results. `embeddings` are excluded by default for performance and the `ids` are always returned. You can specify which of these you want returned by passing an array of included field names to the includes parameter of the query or get method.

javascript Copy Code

```

# Only get documents and ids
collection.get(
    include=["documents"]
)

collection.query(
    query_embeddings=[[11.1, 12.1, 13.1],[1.1, 2.3, 3.2], ...],
    include=["documents"]
)
```

Using Where filters

Chroma supports filtering queries by `metadata` and `document` contents. The `where` filter is used to filter by `metadata` , and the `where_document` filter is used to filter by `document` contents.

Filtering by metadata

In order to filter on metadata, you must supply a `where` filter dictionary to the query. The dictionary must have the following structure:

python Copy Code

```
{
    "metadata_field": {
        <Operator>: <Value>
    }
}
```

Filtering metadata supports the following operators:

- `$eq` - equal to (string, int, float)
- `$ne` - not equal to (string, int, float)
- `$gt` - greater than (int, float)
- `$gte` - greater than or equal to (int, float)

- `$lt` - less than (int, float)
- `$lte` - less than or equal to (int, float)

Using the $eq operator is equivalent to using the `where` filter.

python Copy Code

```python
{
    "metadata_field": "search_string"
}

# is equivalent to

{
    "metadata_field": {
        "$eq": "search_string"
    }
}

```

Where filters only search embeddings where the key exists. If you search `collection.get(where={"version": {"$ne": 1}})`. Metadata that does not have the key `version` will not be returned.

Filtering by document contents

In order to filter on document contents, you must supply a `where_document` filter dictionary to the query. We support two filtering keys: `$contains` and `$not_contains`. The dictionary must have the following structure:

python Copy Code

```python
# Filtering for a search_string
{
    "$contains": "search_string"
}
```

python Copy Code

```python
# Filtering for not contains
{
    "$not_contains": "search_string"
}
```

Using logical operators

You can also use the logical operators `$and` and `$or` to combine multiple filters.

An `$and` operator will return results that match all of the filters in the list.

python Copy Code

```python
{
    "$and": [
        {
            "metadata_field": {
                <Operator>: <Value>
            }
        },
        {
            "metadata_field": {
                <Operator>: <Value>
            }
        }
    ]
}
```

An `$or` operator will return results that match any of the filters in the list.

python Copy Code

```python
{
    "$or": [
        {
            "metadata_field": {
                <Operator>: <Value>
            }
        },
        {
            "metadata_field": {
                <Operator>: <Value>
            }
        }
    ]
}
```

Using inclusion operators (`$in` and `$nin`)

The following inclusion operators are supported:

- `$in` - a value is in predefined list (string, int, float, bool)
- `$nin` - a value is not in predefined list (string, int, float, bool)

An `$in` operator will return results where the metadata attribute is part of a provided list:

```json
{
  "metadata_field": {
    "$in": ["value1", "value2", "value3"]
  }
}
```

An `$nin` operator will return results where the metadata attribute is not part of a provided list:

```json
{
  "metadata_field": {
    "$nin": ["value1", "value2", "value3"]
  }
}
```

Practical examples

For additional examples and a demo how to use the inclusion operators, please see provided notebook **here**

Updating data in a collection

Any property of items in a collection can be updated using `.update` .

```python
collection.update(
    ids=["id1", "id2", "id3", ...],
```

```
    embeddings=[[1.1, 2.3, 3.2], [4.5, 6.9, 4.4], [1.1, 2.3, 3.2], ...],
    metadatas=[{"chapter": "3", "verse": "16"}, {"chapter": "3", "verse": "
    documents=["doc1", "doc2", "doc3", ...],
)
```

If an `id` is not found in the collection, an error will be logged and the update will be

will be recomputed with the collection's embedding function.

If the supplied `embeddings` are not the same dimension as the collection, an exception will be raised.

Chroma also supports an `upsert` operation, which updates existing items, or adds them if they don't yet exist.

python Copy Code

```python
collection.upsert(
    ids=["id1", "id2", "id3", ...],
    embeddings=[[1.1, 2.3, 3.2], [4.5, 6.9, 4.4], [1.1, 2.3, 3.2], ...],
    metadatas=[{"chapter": "3", "verse": "16"}, {"chapter": "3", "verse": "
    documents=["doc1", "doc2", "doc3", ...],
)
```

If an `id` is not present in the collection, the corresponding items will be created as per `add`. Items with existing `id`s will be updated as per `update`.

Deleting data from a collection

Chroma supports deleting items from a collection by `id` using `.delete`. The embeddings, documents, and metadata associated with each item will be deleted. ⚠️ Naturally, this is a destructive operation, and cannot be undone.

python Copy Code

```python
collection.delete(
    ids=["id1", "id2", "id3",...],
 where={"chapter": "20"}
)
```

`.delete` also supports the `where` filter. If no `ids` are supplied, it will delete all items in the collection that match the `where` filter.

Edit this page on GitHub