香港中文大學(深圳)
The Chinese University of Hong Kong, Shenzhen

CSC3170
Database System

Vector-Database-Powered Semantic GIF Search

Group Member:	*Student Number:*
Zhecheng Ren	121090464
Zhiyuan Gao	121090136
Yueyan Wu	121090620
Hitomi Tee Jin Ling	121010011
Kee Cheng	121040041

December 15, 2023

Table of Contents

Abstract: The popularization of Graphics Interchange Format (GIF) images in fields like social media and business has witnessed the thrivingness of GIF search technologies. However, although advanced searching techniques, such as catalog retrieval, keyword query, and vectorized data search, have been extensively employed by prior GIF search technologies, some noticeable drawbacks still exist. Among them, the most significant issue is the subjectivity in the pre-classification of data and users's searches, which can lead to flaws in searching speed, accuracy, and functionality. Following the rapid development momentum of an emerging technology —— vector database, this project aims to utilize this powerful tool to fix the abovementioned flaws and establish a high-performance semantic GIF search system. By incorporating Microsoft's Natural Language Processing transformer model MPNet and a vector database Qdrant into the GIF search system, this project exploits the vector database's advantages in indexing and vector operations for high-speed similarity search and database functionality. As the outcome of this project, our vector-database-powered semantic GIF search system can conduct a real-time and high-accuracy similarity search by returning 10 GIFs, according to the user's query descriptions, among over 100K GIFs. Moreover, this system also implements basic database operations like insertion and deletion, making it capable of conducting large-scale management over the stored data.

1. Background

GIF, a media format with unique advantages in expressing emotions and opinions, is thriving in social media and communication applications. However, the explosive increase in the number of GIFs on the Internet has also led to challenges and booming demand in real-time GIF searching for considerable social media enterprises. Therefore, high-performance GIF search systems have become indispensable nowadays.

The significance of developing such GIF search systems is apparent. It can significantly enhance the user experience by providing convenient retrieval tools that facilitate users finding GIFs that align with their emotional and expressive needs. Consolidating and offering access to GIF content from diverse social media sources also plays a pivotal role in establishing a connection between the production of GIFs and their consumption by users, thus cultivating a broader market.

However, considerable challenges behind these benefits are still awaiting to be addressed. Amidst the disorder and dispersion of information flow, the sheer amount

of GIF makes indexing and retrieval considerably challenging. Moreover, the absence of a structured information flow in GIF files makes it challenging for the search system to identify the target data precisely. For example, while Weibo serves as a hub for GIFs, searching for GIFs on Weibo proves much more challenging due to the ultra-large data stream and the folding of contents. Users often find it difficult to search for the target GIF if the textual content in the search key of a GIF is not explicitly mentioned.

Given these challenges, several retrieval techniques are employed by the existing GIF search systems to tackle them. Among the techniques, four primary categories are concluded: catalog retrieval, keyword query, instance search, and attributive retrieval. Catalog retrieval refines images through a classification structure, which works effectively but requires extensive manual classification (Li, 2014). Keyword queries that are widely used in large search engines enable users to enter relevant keywords, which can then be manually indexed and automatically extracted to retrieve results. Instance search is generally employed in content-based search engines to identify similar images based on the features or attributes of images provided by the user. Despite the fact that similar images can be easily found, users are expected to supply images. Attribute search, on the other hand, is often used in small-scale professional databases, allowing users to search based on external characteristics like author, photo date, etc. Due to its uniform information management and standardization, it is preferable for indexing and searching in relational databases.

These approaches do, however, exhibit certain shortcomings. Due to subjectivity, manual classification in catalog retrieval not only demands a significant investment of time and resources but also potentially leads to inaccurate classification. When adjustments are needed for classification structure, system maintenance can be extremely complicated, especially when inserting new images. Furthermore, keyword queries greatly depend on the user's input. In other words, imprecise search results often occur when inaccurate or subjective keywords are input. Moreover, a semantic understanding of image content may be insufficient to cover all user needs. For instance, the Baidu search tool often yields repeated results when the system finds it hard to satisfy the user’s needs. Similar to keyword queries, attributive retrieval heavily relies on attribute information provided by the users, where inaccurate or missing information may result in a decrease in retrieval accuracy. In addition, updating attribute information can sometimes be cumbersome.

Besides the retrieval technique, another approach that applies an independent vector index without using a vector database is currently being used to improve the efficiency of vector embedding search and retrieval. One of the most popular representatives of this method is Facebook AI Similarity Search (FAISS). Although it increases the accuracy of searching, it necessitates additional work and storage schemes to manage and maintain vector data. Also, applying standalone vector indexes is time-consuming and computationally expensive, as it requires a complete re-indexing process to integrate new data.

As we can see, there are still many issues that remain unsolved. While data scientists are trying their best to solve these problems, vector databases have entered the public eye as one of the most popular database topics recently. A vector database is designed explicitly for storing and searching vectors or vector collections. Its primary objective is to store, index, and retrieve large-scale vector data efficiently. In comparison to a standard database, vector databases have various advantages. Firstly, they support similarity search, which efficiently identifies vectors that are similar to a given vector. Secondly, vector databases accommodate high-dimensional vectors, allowing them to adapt to complicated data structures and schemas. Apart from that, they exhibit commendable real-time performance, which is critical for applications that require rapid response. Lastly, they employ unique indexing techniques like Approximate Nearest Neighbors (ANN) to speed up similarity searches (Han et al., 2023).

By implementing vector databases, we proposed a newly designed project, Vector-Database-Powered Semantic GIF Search Tool, to address all the aforementioned problems. The proposed project eliminates the need for tedious manual classification through flexible backstage data management functions like insertion, updating, and deletion. It supports metadata storage, enabling users to associate metadata with each vector to enhance retrieval accuracy. Besides that, it facilitates dynamic modifications to stored information and accomplishes real-time updates to data. Aligned with the project objectives to resolve the deficiencies of traditional indexing methods, our system aims to significantly improve people's experiences in online communication by achieving the slogan "seeing GIF like seeing me."

2. Description of system

In this project, we employ the vector database as the cornerstone of our semantic GIF search system. In the first part of this section, we will introduce some basic operation concepts and mechanisms of the vector database, including vector embedding, indexing strategies, and similarity measures. In the second part of this section, we will illustrate the system structure, which reveals the design layout and the working pipeline of our vector-database-powered semantic GIF search system.

2.1 Methods

2.1.1 Vector Database Mechanism - Vector Embedding

To construct a vector database, we first use an AI model, which is called the retriever in a vector database, to transform the original data into vector forms capable of revealing fundamental patterns, potential relationships, and underlying structures.

In the context of language processing, some Natural Language Processing (NLP) transformer models serve as the retriever. They can transform keywords or sentences into vectors while maintaining semantic information, which is critical for subsequent comparison operations. In other words, semantically similar sentences will be transformed into similar vectors in the vector space, and vice versa.

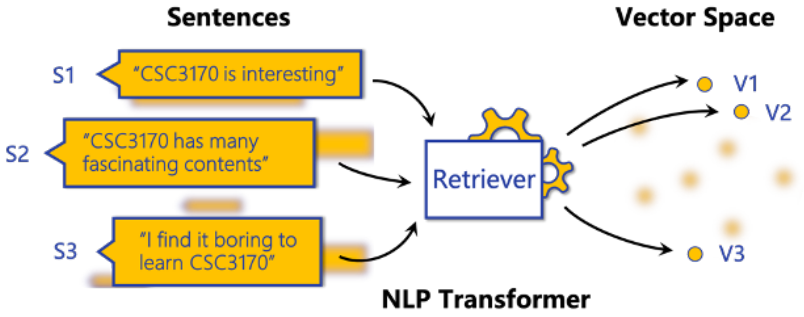


Figure 1. The retriever in the context of language processing

2.1.2 Vector Database Mechanism - Indexing

The approximate search results of the vector database can lead to trade-offs between accuracy and speed or storage. However, by organizing vector data with an optimized indexing strategy, the vector database can ensure both superior efficiency and near-perfect accuracy (Jegou, Douze & Schmid, 2010). A combination of different Approximate Nearest Neighbor (ANN) algorithms is utilized to generate such a decent indexing strategy. These algorithms can effectively enhance the performance of database search and storage through quantization, specific hashing like Locality-Sensitive Hashing (LSH), and advanced graph-based searching methods like Hierarchical Navigable Small World (HNSW). In the following subsections, we will emphatically introduce one of the hottest algorithms assembled in the vector database: Product Quantization (PQ).

PQ is commonly utilized in high-dimensional vector data processing to remove the heavy searching time and storage burden. According to James's (2022) test, the PQ algorithm drastically compresses high-dimensional vector data sets by *97%* and quintuples the searching speed, resulting in only a two percent compromise in accuracy at the same time (which can be removed by combining the PQ with the Inverted File System). The procedure of the PQ algorithm can be broken down into the following four steps: splitting, training, encoding, and querying.

1. **Splitting:** Every high-dimensional vector is sliced into *n* low-dimensional segments.
2. **Training:** For the *ith* segment, we perform k-means clustering on the *ith* segments of all vectors. We assign a specific "code" for each of the *k* centroids.

   Notice that when clustering, the number of centroids needed for high-dimensional vectors is exponentially larger than that for low-dimensional vectors. Therefore, we can use much fewer and shorter "codes" to represent the centroids for vector segments, thus reducing the space cost compared to the case without quantization.
3. **Encoding:** For the *ith* segment of each vector, we assign a "code" to it by sharing the "code" of its nearest centroid, then combine these "codes" into a "codebook," which maps the vector segment and its corresponding "code."

After this step, each vector can be represented by a shorter "code vector" containing *n* codes. And *n* "codebooks" are generated, containing each vector's "code" information.

4. **Querying:** When querying, the algorithm first breaks down the query vectors into *n* segments. We can find each segment's nearest centroid and corresponding "code," thus representing the query vector with a "code vector." Finally, with the help of "codebooks," we can use this "code vector" to find the nearest vectors to the query vector.

   Notice that the querying results also depend on the choice of the distance computation method (symmetric distance computation or asymmetric distance computation), which is omitted in this report.

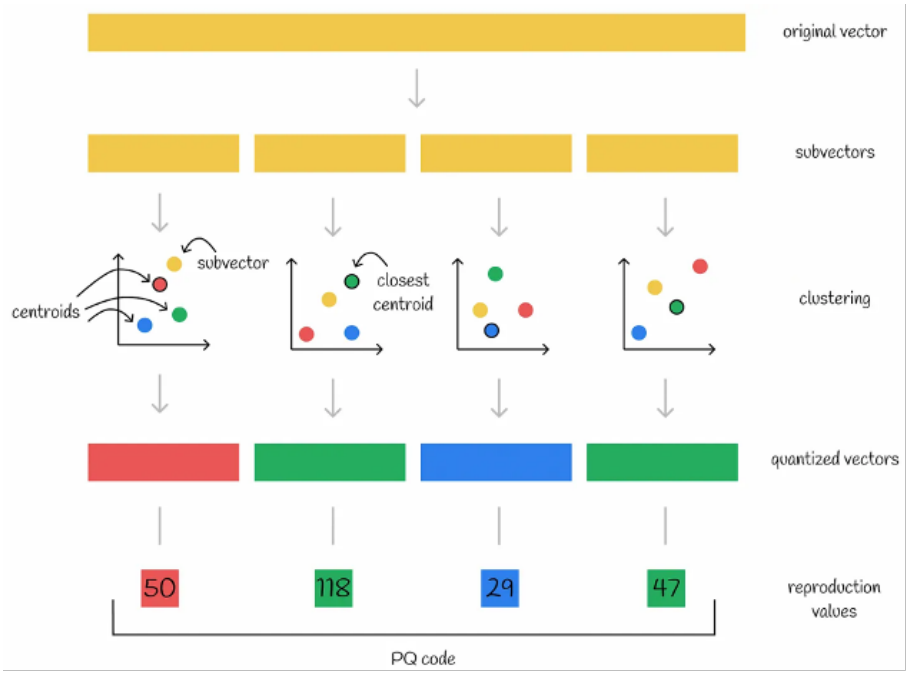


Figure 2. Generating the "code vector" of the query vector

2.1.3 Vector Database Mechanism - Similarity Measures

Similarity measures can mathematically determine how similar two vectors are, which enables the vector database to compare and identify several relevant data points for a given query vector. Some commonly used similarity measures are shown below.

- **Cosine similarity:** It measures the cosine of the angle between two given vectors in the vector space. This similarity ranges between *-1* and *1*, where *1* represents the identical vectors and *-1* represents the opposite vectors.
- **Euclidean distance:** It measures the straight-line distance between two given vectors in the vector space. This similarity ranges between *0* and $+\infty$, where *0* represents the identical vectors, and a larger value represents a larger dissimilarity between two vectors.

- **Dot product:** It measures the product of the magnitudes and the cosine of two given vectors in the vector space. This similarity ranges between -∞ and +∞, where *0* represents the orthogonality between two vectors.

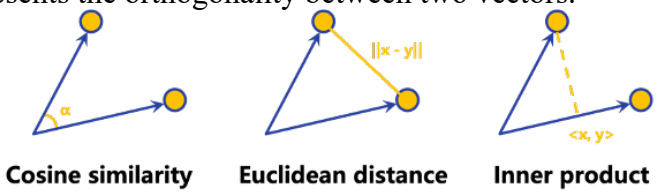


Figure 3. Some commonly used similarity measures

2.2 System Structure

To achieve an efficient semantic GIF search, we design the system structure shown below.

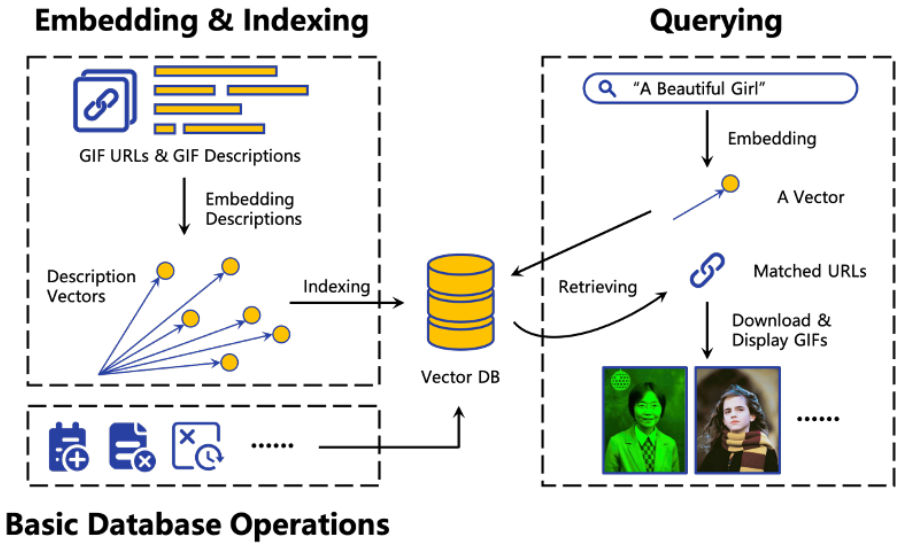


Figure 4. The structure of the system

- **Embedding & indexing:** The raw data of our system is the dataset containing GIF URLs and their corresponding descriptions. By employing an NLP transformer model as the retriever of our system, we can transform the GIF descriptions into vector forms. These vector data will be subsequently stored in our system's vector database through the indexing strategies mentioned above.
- **Querying:** In the querying stage, the client of our system can input the description of wanted GIFs on our website. The same retriever will transform the received query description into a vector and compare it with the vectors stored in the vector database. After retrieving several similar vectors, the website will utilize the URLs corresponding to these vectors to download and display GIFs. We can pre-define the number of similar GIFs returned by our system. All the returned GIFs are ordered according to their similarity to the query description.
- **Basic database operations:** As a database system, our system can also implement basic database operations like addition and deletion.

3. Implementation of system

In this project, we import the Tumblr GIF (TGIF) dataset, which contains 100K animated GIFs and 120K sentences describing the visual content of the animated GIFs. The GIFs were collected from randomly selected posts published between May and June of 2015 on Tumblr. The dataset does not contain the actual GIF files, but the URLs to access these GIF files; thus, we do not need to store or download any GIF file. Using this dataset, we build the Vector-Database-Powered semantic GIF search tool with two components.

1. A retriever to embed GIF descriptions
2. A vector database to store GIF description embeddings and retrieve relevant GIFs

3.1 Implementation of System Components

3.1.1 Retriever - SentenceTransformer

The retriever is used to generate embeddings for all GIF descriptions and queries in a way that the queries and GIF descriptions with similar meanings are categorized in a similar vector space. To find the most relevant GIF to the query, cosine similarity is preferable to compute the similarity between the query and context embeddings for its low complexity. We choose the SentenceTransformer model trained based on Microsoft's MPNet as our retriever; it performs well out of the box when searching based on generic semantic similarity.

3.1.2 Vector Database - Qdrant

The vector database stores vector representations of the GIF descriptions that can be retrieved from another query vector. Here, we use Qdrant, a fully managed Open-Source Vector Database that provides a user-friendly API to store, search, and manage high-dimensional points enriched with metadata called payloads that contain information about the vectors. Qdrant's modules that are used in this system include "QdrantClient" and "models."

3.1.2.1 Module 1 - "QdrantClient"

"QdrantClient" connects the Qdrant server to run an in-memory database. To start with it, we need to instantiate the client using a host and a port. Six functions are used in this module.

1. **recreate_collection**(collection_name, vectors_config) can be used more than once to create new collections with or without the same name. Parameters include collection_name and vectors_config. The former refers to the name of the collection to recreate; the latter has two components: size and distance. The vector_size is the size of the vectors in the collection, while the distance parameter specifies the function used to measure the distance between two points.
2. **get_collection**(collection_name) gets the collection object and extracts related information from an existing collection by the parameter collection_name.

3. **upsert**(collection_name, points) is used to insert a new record or update an existing one. Parameters include collection_name and points. The former refers to which collection to upsert; the latter contains records consisting of a vector(s), an optional ID(s) of the vector, and an optional payload(s). As mentioned, the payload is additional information alongside vectors. Since Qdrant can only take in native Python iterables like lists and tuples, the argument of the parameter "vectors" is usually given after attaching the .tolist() method.
4. **count**(collection_name, exact) counts points(vectors) in the collection matching the filter by the parameter collection_name, name of the collection to count point in. When the bool parameter exact is 'True,' it returns the exact count of points matching the filter.
5. **delete**(collection_name, points_selector) is used to delete a specified vector from the collection without affecting the payload. Parameters include collection_name and points_selector; the latter selects points based on a list of IDs or filters.
6. **search**(collection_name, query_vector, limit) searches for the closest vectors in the collection taking into account filtering conditions with parameters collection_name, query_vector, and limit. The parameter query_vector is a reference for which we search for a vector closest to; the limit determines the number of returned results. This function returns a list of found close points with similarity scores, from the closest to the farthest.

3.1.2.2 Module 2 - "models"

The module "models" allows the users to access most if not all features and functionalities to interact with Qdrant. Six classes are used in this module (as shown in the table).

Class	**Functionality**
Batch(id, vector, payload)	Allow users to give a list of the vectors(points) instead of a vector(point) as an argument.
PointStruct(id, vector, payload)	Create a new point with id, vector, and payload.
MatchValue(value)	Return the exact match of the given value.
FieldCondition(key, match)	Compile all possible payload filtering conditions given the key and the exact match.
Filter(must)	Require a filter to match all conditions (must).
FilterSelector(filter)	Select filters based on all the given conditions.

3.2 Integration of System

In this section, we provide pseudocodes of the main algorithm, followed by explanations.

3.2.1 Data Processing

Initialize Qdrant Database

```
client ← QdrantClient(host="localhost", port=6333)    #instantiate the client
collection  ← "gif_search"
client.recreate_collection(collection_name=collection,
      vectors_config=models.VectorParams(size=384, distance=models.Distance.COSINE))
collection_info ← client.get_collection(collection)
```

Firstly, we instantiate the client with localhost and port 6333 to be the default port to connect with the Docker image. Then, we specify the name of the collection where we will store our GIF descriptions and their URLs as "gif_search." We set the embedding dimension of the vectors as "384" and the similarity metric as "cosine."

Initialize Retriever

```
retriever ← SentenceTransformer("../models/all-MiniLM-L6-v2")
```

We use a sentence-transformers model that maps sentences and paragraphs to a 384-dimensional dense vector space. Therefore, the dimension of the vector is consistent.

Generate Embeddings & Upsert

```
batch _size ← 64
For each batch
      emb ← retriever.encode(description).tolist()    #generate embeddings for batch
      meta ← get records from the dataset and apply .todict() method
      ids ← IDs in the batch
      client.upsert(collection, models.Batch(ids=ids, vectors=emb, payloads=meta))
```

To increase the efficiency of generating embeddings, we utilize the concept of batches. The retriever will generate 64 GIF descriptions at once and send a single API call for each batch of 64. Next, we pass the documents to Qdrant by specifying parameter id as "ids," vectors as "emb" (embeddings for the GIF descriptions), and payloads as "meta" (metadata for each document representing GIFs); "meta" is a dictionary containing a URL and description.

3.2.2 Backstage Management

Insert (given url, description)

```
emb ← retriever.encode(description).tolist()
meta  ← {url, description}
client.upsert(collection, models.PointStruct(id=1, vector=emb, payload=meta))
```

After receiving the query to insert a new GIF given its URL and description, we create a new point with index 1, the embeddings for the GIF descriptions and metadata containing the URL and description. Then, we upsert the new point to the targeted collection.

Delete (given url)

```
client.delete(collection,
    models.FilterSelector(filter=models.Filter(
        must=[models.FieldCondition(
            key="url", match=models.MatchValue(value=url))])))
```

We use a combination of MatchValue, FieldCondition, Filter, and FilterSelector classes to select filters based on the given URL of the GIF as the condition. The delete function of Qdrant will select a point(s) in the targeted collection based on the filters and delete it.

3.2.3 Main Interface

Please note that the codes for creating a simple interactive web app interface by using Streamlit are omitted in the pseudocode below. We focus on the search tool here.

Search on Main Interface

```
query ← input text
if press button("Search")
    results ← client.search(collection, retriever.encode(query).tolist(), limit=10)
    urls ← []
    for vector in results
        url ← vector.payload["url"]
        urls.append(url)
    use streamlit to display the figures of all 10 GIFs
```

After the users input the GIF description and press the button "Search" on our main interface, we first generate the embedding for the query and search for the 10 closest vectors from the TGIF dataset. Then, we get the URLs of these vectors and display all the GIFs on our main interface from most relevant to the description to least relevant.

For more details on the system, please visit the GitHub website created by us for this group project. There are some tutorials to illustrate the elementary Qdrant Python Client API and codes regarding the implementation of system components and integration of the system.

https://github.com/langshangyuan/vdb3170

4. Testing of system

This part of the report consists of a testing plan and a performance evaluation. The testing plan consists of two parts: system setup and function testing. For performance evaluation, we include how vector database outperforms scalar database system and benchmarks test results of the vector database.

4.1 Testing plan

4.1.1 System setup

Before setting up the system, we check the software and hardware environments, database connection, and network configuration to avoid getting errors.

To initialize our application, we run the following commands in the terminal.

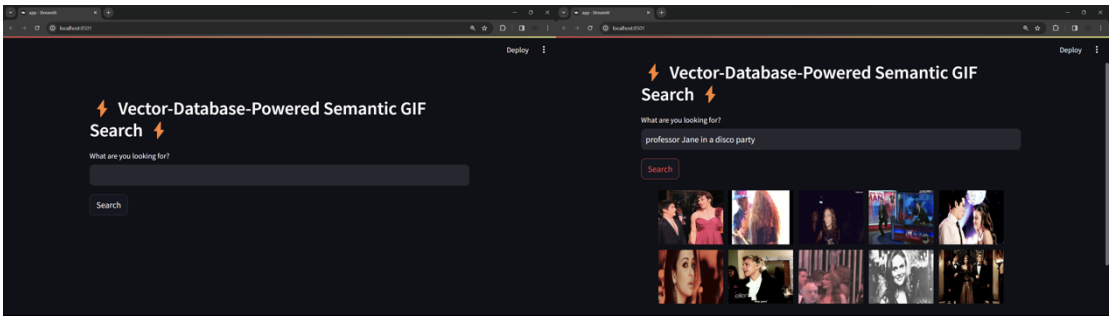
```
# Pull the Qdrant Docker Image
docker pull qdrant/qdrant
# Create Virtual Environment
poetry install
# Start the Qdrant Server
docker run -p 6333:6333 -v $(pwd)/data/qdrant:/qdrant/storage qdrant/qdrant
# Active the Virtual Python Environment
poetry shell
# Start the Streamlit Web Server
streamlit run vdb3170/app.py
```

4.1.2 Function testing

We test by searching, inserting, and deleting GIFs to ensure our database system works.

4.1.2.1 Searching

Enter a prompt or description of desired GIFs into the search bar on the interface. If successful, GIFs from our database with descriptions most related to the prompt will appear on the web interface.

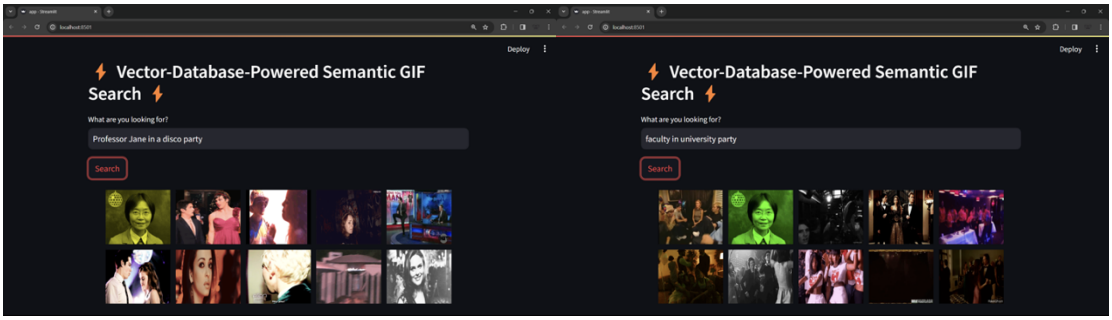


4.1.2.2 Inserting

Before inserting, we created a new GIF. Next, we insert it by using the insert_gif() function with arguments as the GIF's URL and a description of the new GIF:

```
insert_gif("https://media4.giphy.com/media/v1.Y2lkPTc5MGI3NjExOGI3aGk2c3ZnZnpwazcyamx1cjIybHBjYXlnbm44dDlsOTB6ZTkybCZlcD12MV9pbnRlcm5hbF9naWZfYnlfaWQmY3Q9Zw/q1CIZgFvnmJEtsNXkK/giphy.gif", "Professor Jane You in disco")
```

To verify that the new GIF is in our database, return to the search bar and enter a prompt with the words from or similar to its description. Inserting is a success if the new GIF appears from our search prompt.

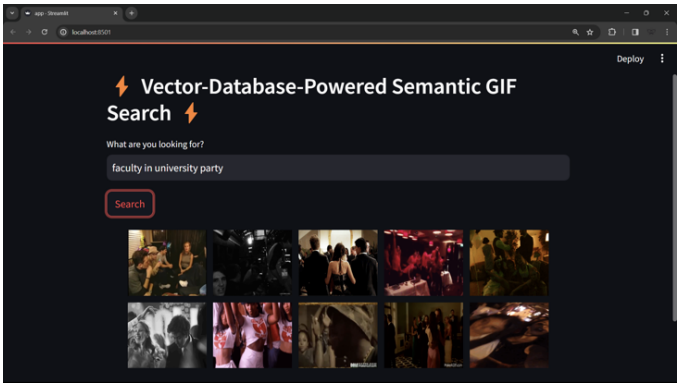


4.1.2.3 Deleting

Delete the GIF from our database by using delete_gif() with the GIF's URL as the argument.

```
delete_gif("https://media4.giphy.com/media/v1.Y2lkPTc5MGI3NjExOGI3aGk2c3ZnZnpwazcyamx1cjIybHBjYXlnbm44dDlsOTB6ZTkybCZlcD12MV9pbnRlcm5hbF9naWZfYnlfaWQmY3Q9Zw/q1CIZgFvnmJEtsNXkK/giphy.gif")
```

To verify that the GIF is deleted from our database, return to the search bar and enter a prompt with the words from or similar to its description. Deleting is a success if the GIF no longer appears from any search prompt.



4.2 Performance evaluation

4.2.1 Compare searching (GIFs) with scalar database

Vector database allows users flexibility in their input of prompts and gives better search results. As shown in the testing plan example earlier, our database will show

the results by inputting prompts with related or similar words. In contrast, traditional databases will not show any results without the exact prompts.

4.2.2 Monitor and benchmark database

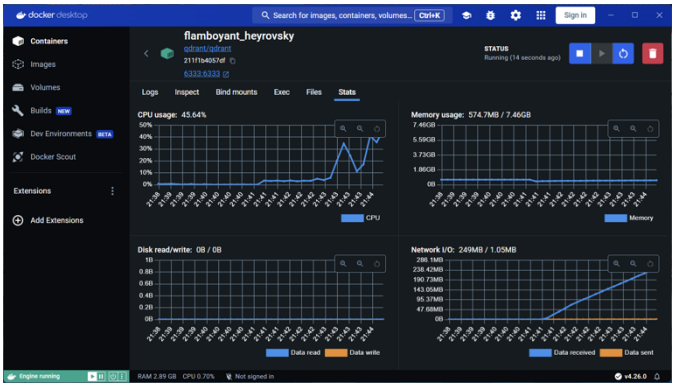
4.2.2.1 Monitoring

According to Roie Schwaber-Cohen (2023), for us to effectively manage and maintain a vector database, we need a robust monitoring system that tracks the important aspects of the database's performance, health, and overall status. Monitoring is critical for detecting potential problems, optimizing performance, and ensuring smooth production operations.

a) Usage: We can monitor the resource usage of the vector database such as CPU utilization, memory usage, disk space, and network activity. Resource usage information enables the identification of potential issues or resource constraints that could affect the performance of the database.
    a. CPU utilization: Track the CPU utilization of the database server to ensure that it is operating within acceptable limits.
    b. Memory usage: Monitor the memory consumption of database processes to prevent memory leaks or shortages.
    c. Disk space: Check disk space regularly to ensure that the database has enough space to store data and indexes.
    d. Network activity: Monitor network communication between the database and other systems to ensure that the network connection is normal.

b) Query Performance: It allows us to monitor the query latency, throughput, and error rates, which may indicate potential systemic issues.
    a. Query latency: Track the response time of queries to ensure they are completed within a reasonable time frame.
    b. Throughput: Evaluate how fast the database can process queries to ensure that the system can handle the expected load.
    c. Error rate: Monitor the error rate of queries and abnormal behavior to identify and resolve potential problems in a timely manner.

Docker Desktop application allows us to monitor our resource usage.



4.2.2.2 Benchmarks

Benchmarks are implemented in server-client mode, meaning that the server runs on a single machine and the client runs on another. Due to this project's time constraints and resource limitations, we did not perform benchmark tests locally but instead included benchmark results of VectorDBBench by Zilliztech (*Vector Database Benchmark Tool*, 2023).

We compare the throughput measured through query per second (QPS), the response speed measured through P99 Latency, and the search accuracy measured through recall.

QPS: A vector database's capability to handle concurrent queries per second. Higher QPS values indicate better vector database performance.

P99 Latency: The time that 99% of queries take to complete. Lower latency values indicate better vector database performance.

Recall: The vector search accuracy of a vector database. Higher recall rates correspond to more accurate vector search results.

a) The Performance Ranking (QPS) table

Performance Ranking(QPS) | Performance Ranking(P99 Latency) | Cost Ranking

Rankings	Databases with different hardware resources	QPS Scores	QPS/Recall Medium OpenAI None Filter	QPS/Recall Medium Cohere None Filter
1	QdrantCloud-4c16g-5node	100	633.603 / 0.919	537.498 / 0.89
2	QdrantCloud-4c16g-1node	36.5158	188.644 / 0.918	240.721 / 0.889
3	WeaviateCloud-standard	9.3208	46.862 / 0.996	67.912 / 0.991
4	WeaviateCloud-bus_crit	9.3139	43.502 / 0.996	63.137 / 0.991
5	ElasticCloud-upTo2.5c8g	2.2472	11.295 / 0.996	15.227 / 0.989

b) The Performance Ranking (P99 Latency)

Performance Ranking(QPS) | Performance Ranking(P99 Latency) | Cost Ranking

Rankings	Databases with different hardware resources	Latency Scores	Latency/Recall Medium OpenAI None Filter	Latency/Recall Medium Cohere None Filter
1	QdrantCloud-4c16g-5node	58.8235	24.6ms / 0.919	17.4ms / 0.889
2	WeaviateCloud-standard	12.9261	123ms / 0.996	18.9ms / 0.89
3	QdrantCloud-4c16g-1node	11.6682	228.7ms / 0.996	145.7ms / 0.991
4	WeaviateCloud-bus_crit	8.7459	917.5ms / 0.918	179.5ms / 0.991
5	ElasticCloud-upTo2.5c8g	1.0469	3611.2ms / 0.996	861.8ms / 0.989

The results show that Qdrant is superior compared to other databases; it achieves the highest QPS and lowest latencies. Although recall may not be the highest, it outperforms in QPS and latencies by a large margin.

5. Conclusion and future work

In summary, through this study, we have successfully implemented a semantic GIF search engine based on a vector database, which provides users with a more intelligent and efficient GIF retrieval experience. It has three main characters. The first one is the implementation of Semantic GIF search, in which we have successfully integrated vector databases to implement semantic search for GIF images. This allows users to find GIFs that match their emotions and needs more precisely and improves the intelligence of the search. The second one is the processing of high-dimensional vectors. Using vector databases enables us to process high-dimensional vector data efficiently, improving the representation accuracy of image features and making the search results more accurate. The third one is the real-time performance. Our system performs well in handling large-scale data with excellent real-time performance, allowing users to obtain GIF images that match their search intent quickly.

We have successfully solved some challenges of traditional methods in similarity search and high-dimensional vector processing. Vector database supports similarity searches. The introduction of vector databases allows us to support similarity searches effectively, providing users with a wider range of choices and thus improving the flexibility and accuracy of searches. Another advantage of vector databases is their high efficiency in high-dimensional data retrieval. The vector database can complete complex similarity searches and range queries in a short time. According to some benchmark tests, the query speed of some commercial vector databases is several times or even tens of times faster than that of traditional databases when dealing with large-scale, high-dimensional data.

However, there exist several limitations in this project. Vector databases require efficient construction and searching of vectors in billions of dimensions, which presents substantial computational and storage challenges. Thus, vector databases must employ specialized techniques to reduce complexity and improve the accuracy of vector similarity searches. Furthermore, the dataset we use could be more extensive. The size and diversity of the dataset play a crucial role in the model's performance. A broader and larger dataset will provide a more comprehensive range of patterns for the model, allowing it to generalize better and improve accuracy in real-world scenarios. In addition, the application lacks some user features, such as saving GIFs, allowing interactive user feedback, and so on. Finally, searches may return slow if the users have a weak network connection due to the downloading speeds, as the GIFs are stored in URLs.

In the future, we can focus on improving our vector database and enriching the application's user experience. We can optimize the application to process and cache GIFs locally more efficiently. In addition, we can mitigate the effects of network fluctuations, ensuring a smoother experience for users even when network conditions are poor. Furthermore, by integrating more features and functions into the application, we can improve the appeal and usability of the application. Moreover, integrating

with messaging platforms such as WeChat and Instagram could be a crucial strategy to achieve wider user engagement and increase app exposure.

References

Han, Y., Liu, C., & Wang, P. (2023). A Comprehensive Survey on Vector Database: Storage and Retrieval Technique, Challenge. arXiv preprint arXiv:2310.11703.

James, B. (2022, March 2). *Faiss: The Missing Manual*. Pinecone. https://www.pinecone.io/learn/series/faiss/

Jegou, H., Douze, M., & Schmid, C. (2010). Product quantization for nearest neighbor search. *IEEE transactions on pattern analysis and machine intelligence*, *33*(1), 117-128. https://doi.org/10.1109/TPAMI.2010.57

Li, C. (2014). Research on image search engine based on text. Wireless interconnection technology (02),140-141.

Roie Schwaber-Cohen. (2023). *What is a Vector Database & How Does it Work? Use Cases + Examples | Pinecone*. Pinecone.io. https://www.pinecone.io/learn/vector-database/

Vector Database Benchmark Tool. (2023). Zilliz.com. https://zilliz.com/vector-database-benchmark-tool?database=WeaviateCloud%2CElasticCloud%2CQdrantCloud&dataset=medium&filter=none