CS4221 Tutorial 4:
Vector Databases - Milvus

Yao LU

2024 Semester 2

National University of Singapore
School of Computing

Objective

By the end of this tutorial, you will:

- Set up Milvus server using Docker.
- Prepare the virtual environment and interact with Milvus server.
- Develop a fundamental understanding of schema design principles in Milvus.
- Perform basic operations such as create, insert, search and delete data.
- Explore various index types, fine-tune parameters, and evaluate search performance.

Setup

Step 1: Install Milvus

- Download the provided docker-compose file.
- Start up the docker container and verify Milvus server is running.

Step 2: Prepare the environment

- Make sure you have installed the *anaconda*
- Download the provided environment file and set up the virtual environment, or follow the documentation to manually download required dependencies.
- Execute quickStart.ipynb to make sure everything is ok.

Prerequisites

Before starting, ensure you have gone through :

- Milvus concept overview:
  - https://milvus.io/docs/manage-collections.md
  - https://milvus.io/docs/schema.md

- Index related documentations
  - IVF_FLAT: https://milvus.io/docs/ivf-flat.md
  - HNSW: https://milvus.io/docs/hnsw.md
  - Blog: https://zilliz.com/learn/how-to-pick-a-vector-index-in-milvus-visual-guide

Prerequisites

Before starting, ensure you have gone through :

- **Milvus concept overview**

- https://milvus.io/docs/manage-collections.md
- https://milvus.io/docs/schema.md

- **Index related documentations**

- Metric type: https://milvus.io/docs/metric.md
- IVF_FLAT: https://milvus.io/docs/ivf-flat.md
- HNSW: https://milvus.io/docs/hnsw.md
- Blog: https://zilliz.com/learn/how-to-pick-a-vector-index-in-milvus-visual-guide

Hello-World

After you start the provided docker-compose, you will get

```
lingze@worker-012:~/cs4221/vector_db_tutorial$ docker compose up -d
[+] Running 23/23
 ✔ standalone 7 layers [⣿⣿⣿⣿⣿⣿⣿]      0B/0B      Pulled
   ✔ d5fd17ec1767 Pull complete
   ✔ 0f5a22c44678 Pull complete
   ✔ 72ad4f350efb Pull complete
   ✔ 2f5ee08a99b8 Pull complete
   ✔ 3fe28e251347 Pull complete
   ✔ 1f27396f6efc Pull complete
   ✔ fe556ec02776 Pull complete
 ✔ etcd 7 layers [⣿⣿⣿⣿⣿⣿⣿]      0B/0B      Pulled
   ✔ dbba69284b27 Pull complete
   ✔ 270b322b3c62 Pull complete
   ✔ 7c21e2da1038 Pull complete
   ✔ cb4f77bfee6c Pull complete
   ✔ e5485096ca5d Pull complete
   ✔ 3ea3736f61e1 Pull complete
   ✔ 1e815a2c4f55 Pull complete
 ✔ minio 6 layers [⣿⣿⣿⣿⣿⣿]      0B/0B      Pulled
   ✔ c7e856e03741 Pull complete
   ✔ c1ff217ec952 Pull complete
   ✔ b12cc8972a67 Pull complete
   ✔ 4324e307ea00 Pull complete
   ✔ 152089595ebc Pull complete
   ✔ 05f217fb8612 Pull complete
[+] Running 4/4
 ✔ Network milvus                Created
 ✔ Container milvus-etcd         Started
 ✔ Container milvus-minio        Started
 ✔ Container milvus-standalone   Started
```

Hello-World

Follow the documentation to prepare you environment.

Before executing the scripts, something to note:

- Update the server address (HOST)
- No restriction on the database name

```
HOST = '10.10.10.250'
PORT = 19530
DB_NAME = 'testdb'
URL="http://"+HOST+':'+str(PORT)
# if you deployed the standalone milvus, and connect to the database server
# if you deployed in your local machine, use "http://localhost:19530"
client = MilvusClient(URL)
```

Hello-World

Before executing the scripts, something to note:

- For the embedding model example, ensure *pymilvus[model]* dependency installed.
- Otherwise, refer to the following synthetic data example.

```python
# generate vectors for the documents
# pre-requisite is to download pymilvus[model] first
embedding_fn = model.DefaultEmbeddingFunction()
vectors = embedding_fn.encode_documents(docs)
print("Dim:", embedding_fn.dim, vectors[0].shape)

# Each record is named "entity"m entity has id, vector representation, raw text, and a subject label that we use
# the usage is similar to the NoSQL database like mongodb.
data = [
    {"id":i, "vector": vectors[i], "text": docs[i], "subject":"history"}
    for i in range(len(docs))
]

print("Data has", len(data), "entities, each with fields: ", data[0].keys())
print("Vector dim:", len(data[0]["vector"]))
```

Example

In this tutorial, we explore different search indexes and fine-tune them to evaluate retrieval performance on a real-world dataset.

For details, refer to *fine_tune_index.ipynb*.

We focus on two fundamental indexes: **IVF_FLAT** and **HNSW**, using the **Glove-25-angular** dataset. This dataset has a dimension of **25**, with **1,183,514** training samples and **10,000** test samples

The schema design is as follows:

```
# ------------------- 2. Create a collection with customized schema
# we are going to create a collection with 2 fields
# +-+-------------+-------------+------------------+------------------------------+
# | | field name  | field type  | other attributes |       field description      |
# +-+-------------+-------------+------------------+------------------------------+
# |1|    "pk"     |   VarChar   |  is_primary=True |      "primary field"         |
# +-+-------------+-------------+------------------+------------------------------+
# |2|"embeddings"| FloatVector|     dim=dim      |"float vector with specific dim"|
# +-+-------------+-------------+------------------+------------------------------+
COLLECTION_NAME = "glove_25_anugular"
schema = MilvusClient.create_schema(
    auto_id = False,
    enable_dynamic_field = True,
)

schema.add_field(field_name = 'pk', datatype=DataType.INT64, is_primary=True, auto_id = False)
schema.add_field(field_name = 'embeddings', datatype=DataType.FLOAT_VECTOR, dim=dim)
```

Example

Take the Inverted File Flat index (**IVF_FLAT**) as an example.

IVF_FLAT offers two tunable hyperparameters:

- **nlist**: the number of partitions to create using the k-means algorithm.
- **nprobe**: the number of partitions to consider during the search for candidate

The **nlist** parameter is set when building the **IVF_FLAT** index, while **nprobe** is adjusted dynamically for each query request.

Example

First, we fix **nlist** and tune **nprobe** to evaluate query performance in terms of **latency** and **recall**.

We build the index by specifying:

- The field for the **index_type** (in this case, **"embedding"**).
- The **metric_type** (here, we use **COSINE**).
- The **index type** and **nlist** parameter.

```
# first, we fix nlist, tune nprobe to check the query performance (latency and recall)
nlist = 1024
index_params = MilvusClient.prepare_index_params()
index_params.add_index(
    field_name="embeddings",
    metric_type = "COSINE",
    # related distance metric to angular is CONSINE
    index_type = "IVF_FLAT",
    index_name = "vector_index",
    params = {
        "nlist":nlist
    }
)
start_time = time.time()
client.create_index(collection_name = COLLECTION_NAME,
                   index_params = index_params,
                   sync = True)
end_time = time.time()
print(f"create index time: {end_time-start_time:.4f}s")
```

```
create index time: 17.5975s
```

Example

Next, we tune **nprobe** while executing the same search queries to analyze its impact on query latency and average recall.

```python
# ------------------ 4[IVF_FLAT]. Search with different nprobe, check the performance
for nprobe in [1,16, 64, 256, 1024]:
    start_time = time.time()
    res = client.search(
        collection_name=COLLECTION_NAME,
        data=query_embedding,
        limit = 100,
        search_params={
            "params" : {"nprobe":nprobe}
        }
    )
    end_time = time.time()
    print(fmt.format(f"------------------ nprobe={nprobe} search --------------------"))
    print(search_latency_fmt.format(end_time-start_time))

    # calculate the Mean Average Recall.
    # Recall@K = (# of true positive in top K) / (# of true positive)
    # MAR (Mean Average Recall) = 1/C * sum(Recall@K). C is the number of queries
    mar_ = []
    for i, candidate_res in enumerate(res):
        y = neighbors[i]
        y_ = [j['id'] for j in candidate_res]
        y, y_ = set(y), set(y_)
        mar_.append(1.0 * len(y & y_) / len(y))
    mar = np.mean(mar_)
    print(f"Mean Average Recall = {mar:.4f}")
```

```
=== ------------------ nprobe=1 search -------------------- ===

search latency = 3.8174s
Mean Average Recall = 0.3681

=== ------------------ nprobe=16 search -------------------- ===

search latency = 4.2593s
Mean Average Recall = 0.8873

=== ------------------ nprobe=64 search -------------------- ===

search latency = 6.7913s
Mean Average Recall = 0.9804

=== ------------------ nprobe=256 search -------------------- ===

search latency = 13.6901s
Mean Average Recall = 0.9994

=== ------------------ nprobe=1024 search -------------------- ===

search latency = 43.0186s
Mean Average Recall = 1.0000
```

Example

From this result, we can observe that as the **nprobe** increases, the **search latency** increase while the average recall degrades.

Next, we fix the **nprobe** and build the index with different **nlist** values to analyze its impact

```python
# ------------------ 4[IVF_FLAT]. Search with different nprobe, check the performance
for nprobe in [1,16, 64, 256, 1024]:
    start_time = time.time()
    res = client.search(
        collection_name=COLLECTION_NAME,
        data=query_embedding,
        limit = 100,
        search_params={
            "params" : {"nprobe":nprobe}
        }
    )
    end_time = time.time()
    print(fmt.format(f"----------------- nprobe={nprobe} search --------------------"))
    print(search_latency_fmt.format(end_time-start_time))

    # calculate the Mean Average Recall.
    # Recall@K = (# of true positive in top K) / (# of true positive)
    # MAR (Mean Average Recall) = 1/C * sum(Recall@K). C is the number of queries
    mar_ = []
    for i, candidate_res in enumerate(res):
        y = neighbors[i]
        y_ = [j['id'] for j in candidate_res]
        y, y_ = set(y), set(y_)
        mar_.append(1.0 * len(y & y_) / len(y))
    mar = np.mean(mar_)
    print(f"Mean Average Recall = {mar:.4f}")
```

```
=== ----------------- nprobe=1 search -------------------- ===

search latency = 3.8174s
Mean Average Recall = 0.3681

=== ----------------- nprobe=16 search -------------------- ===

search latency = 4.2593s
Mean Average Recall = 0.8873

=== ----------------- nprobe=64 search -------------------- ===

search latency = 6.7913s
Mean Average Recall = 0.9804

=== ----------------- nprobe=256 search -------------------- ===

search latency = 13.6901s
Mean Average Recall = 0.9994

=== ----------------- nprobe=1024 search -------------------- ===

search latency = 43.0186s
Mean Average Recall = 1.0000
```

Homework

In this homework, we try to tune the parameters of HNSW and analyze the impact on the query efficiency and accuracy.

HNSW offers three tunable hyperparameters:

- **M**: the maximum number of connections for each node in the graph
- **efConstruction**: the size of the dynamic candidate list which controls index search speed/build tradeoff.
- **ef**: the size of the dynamic candidate list during search

Task:

Follow the process of analyzing **IVF_FLAT**, explore the impact and summarize the trends observed with varying values for these parameters.