TiDB is a MySQL-compatible database, and SQLAlchemy is a popular Python SQL toolkit and Object Relational Mapper (ORM).

In this tutorial, you can learn how to use TiDB and SQLAlchemy to accomplish the following tasks:

- Set up your environment.
- Connect to your TiDB cluster using SQLAlchemy.
- Build and run your application. Optionally, you can find sample code snippets for basic CRUD operations.

## Connect to TiDB

```python
from sqlalchemy import create_engine, URL
from sqlalchemy.orm import sessionmaker

def get_db_engine():
    connect_args = {}
    if ${ca_path}:
        connect_args = {
            "ssl_verify_cert": True,
            "ssl_verify_identity": True,
            "ssl_ca": ${ca_path},
        }
    return create_engine(
        URL.create(
            drivername="mysql+pymysql",
            username=${tidb_user},
            password=${tidb_password},
            host=${tidb_host},
            port=${tidb_port},
            database=${tidb_db_name},
        ),
        connect_args=connect_args,
    )

engine = get_db_engine()
Session = sessionmaker(bind=engine)
```

When using this function, you need to replace `${tidb_host}`, `${tidb_port}`, `${tidb_user}`, `${tidb_password}`, `${tidb_db_name}` and `${ca_path}` with the actual values of your TiDB cluster.

## Define a table

```python
from sqlalchemy import Column, Integer, String
from sqlalchemy.orm import declarative_base

Base = declarative_base()
```

```
class Player(Base):
    id = Column(Integer, primary_key=True)
    name = Column(String(32), unique=True)
    coins = Column(Integer)
    goods = Column(Integer)

    __tablename__ = "players"
```

### Insert data

```
with Session() as session:
    player = Player(name="test", coins=100, goods=100)
    session.add(player)
    session.commit()
```

### Query data

```
with Session() as session:
    player = session.query(Player).filter_by(name == "test").one()
    print(player)
```

### Update data

```
with Session() as session:
    player = session.query(Player).filter_by(name == "test").one()
    player.coins = 200
    session.commit()
```

### Delete data

```
with Session() as session:
    player = session.query(Player).filter_by(name == "test").one()
    session.delete(player)
    session.commit()
```

# Integrate TiDB Vector Search with SQLAlchemy

This tutorial walks you through how to use SQLAlchemy to interact with TiDB Vector Search, store embeddings, and perform vector search queries.

## Sample code snippets

You can refer to the following sample code snippets to develop your application.

## Create vector tables

### Connect to TiDB cluster

```python
import os

import dotenv



from sqlalchemy import Column, Integer, create_engine, Text

from sqlalchemy.orm import declarative_base, Session

from tidb_vector.sqlalchemy import VectorType



dotenv.load_dotenv()



tidb_connection_string = os.environ['TIDB_DATABASE_URL']

engine = create_engine(tidb_connection_string)
```

### Define a vector column

Create a table with a column named `embedding` that stores a 3-dimensional vector.

```python
Base = declarative_base()



class Document(Base):

    __tablename__ = 'sqlalchemy_demo_documents'

    id = Column(Integer, primary_key=True)

    content = Column(Text)
    embedding = Column(VectorType(3))
```

## Store documents with embeddings

```
with Session(engine) as session:

   session.add(Document(content="dog", embedding=[1, 2, 1]))

   session.add(Document(content="fish", embedding=[1, 2, 4]))

   session.add(Document(content="tree", embedding=[1, 0, 0]))

   session.commit()
```

## Search the nearest neighbor documents

Search for the top-3 documents that are semantically closest to the query vector `[1, 2, 3]` based on the cosine distance function.

```
with Session(engine) as session:

   distance = Document.embedding.cosine_distance([1, 2, 3]).label('distance')

   results = session.query(

      Document, distance

   ).order_by(distance).limit(3).all()
```

## Search documents within a certain distance

Search for documents whose cosine distance from the query vector `[1, 2, 3]` is less than 0.2.

```
with Session(engine) as session:

    distance = Document.embedding.cosine_distance([1, 2,
3]).label('distance')

    results = session.query(

        Document, distance

    ).filter(distance < 0.2).order_by(distance).limit(3).all()
```

## Get started

This section provides step-by-step instructions for integrating TiDB Vector Search with LangChain to perform semantic searches.

### Step 1. Create a new Jupyter Notebook file

In your preferred directory, create a new Jupyter Notebook file named `integrate_with_langchain.ipynb`:

```
touch integrate_with_langchain.ipynb
```

### Step 2. Install required dependencies

In your project directory, run the following command to install the required packages:

```
!pip install langchain langchain-community

!pip install langchain-openai

!pip install pymysql

!pip install tidb-vector
```

Open the `integrate_with_langchain.ipynb` file in Jupyter Notebook, and then add the following code to import the required packages:

```
from langchain_community.document_loaders import TextLoader

from langchain_community.vectorstores import TiDBVectorStore

from langchain_openai import OpenAIEmbeddings

from langchain_text_splitters import CharacterTextSplitter
```

### Step 3. Set up your environment

Configure the environment variables depending on the TiDB deployment option you've selected.

For a TiDB Cloud Starter cluster, take the following steps to obtain the cluster connection string and configure environment variables:

1. Navigate to the Clusters page, and then click the name of your target cluster to go to its overview page.
2. Click Connect in the upper-right corner. A connection dialog is displayed.
3. Ensure the configurations in the connection dialog match your operating environment.
    - Connection Type is set to `Public`.
    - Branch is set to `main`.
    - Connect With is set to `SQLAlchemy`.
    - Operating System matches your environment.
4. Click the PyMySQL tab and copy the connection string.
5. Tip
6. If you have not set a password yet, click Generate Password to generate a random password.
7. Configure environment variables.

   This document uses OpenAI as the embedding model provider. In this step, you need to provide the connection string obtained from the previous step and your OpenAI API key.

   To configure the environment variables, run the following code. You will be prompted to enter your connection string and OpenAI API key:

```
# Use getpass to securely prompt for environment variables in your terminal.

import getpass

import os



# Copy your connection string from the TiDB Cloud console.

# Connection string format:
"mysql+pymysql://<USER>:<PASSWORD>@<HOST>:4000/<DB>?ssl_ca=/etc/ssl/cert.pem&
ssl_verify_cert=true&ssl_verify_identity=true"

tidb_connection_string = getpass.getpass("TiDB Connection String:")

os.environ["OPENAI_API_KEY"] = getpass.getpass("OpenAI API Key:")
```

8.

## Step 4. Load the sample document

### Step 4.1 Download the sample document

In your project directory, create a directory named `data/how_to/` and download the sample document `state_of_the_union.txt` from the langchain-ai/langchain GitHub repository.

```
!mkdir -p 'data/how_to/'

!wget
'https://raw.githubusercontent.com/langchain-ai/langchain/master/docs/docs/how_to/state_of_the_union.txt' -O 'data/how_to/state_of_the_union.txt'
```

### Step 4.2 Load and split the document

Load the sample document from `data/how_to/state_of_the_union.txt` and split it into chunks of approximately 1,000 characters each using a `CharacterTextSplitter`.

```
loader = TextLoader("data/how_to/state_of_the_union.txt")

documents = loader.load()

text_splitter = CharacterTextSplitter(chunk_size=1000, chunk_overlap=0)

docs = text_splitter.split_documents(documents)
```

## Step 5. Embed and store document vectors

TiDB vector store supports both cosine distance (`consine`) and Euclidean distance (`l2`) for measuring similarity between vectors. The default strategy is cosine distance.

The following code creates a table named `embedded_documents` in TiDB, which is optimized for vector search.

```
embeddings = OpenAIEmbeddings()

vector_store = TiDBVectorStore.from_documents(

    documents=docs,

    embedding=embeddings,

    table_name="embedded_documents",

    connection_string=tidb_connection_string,

    distance_strategy="cosine",  # default, another option is "l2"

)
```

Upon successful execution, you can directly view and access the `embedded_documents` table in your TiDB database.

## Step 6. Perform a vector search

This step demonstrates how to query "What did the president say about Ketanji Brown Jackson" from the document `state_of_the_union.txt`.

```
query = "What did the president say about Ketanji Brown Jackson"
```

### Option 1: Use `similarity_search_with_score()`

The `similarity_search_with_score()` method calculates the vector space distance between the documents and the query. This distance serves as a similarity score, determined by the chosen `distance_strategy`. The method returns the top `k` documents with the lowest scores. A lower score indicates a higher similarity between a document and your query.

```
docs_with_score = vector_store.similarity_search_with_score(query, k=3)

for doc, score in docs_with_score:

    print("-" * 80)

    print("Score: ", score)

    print(doc.page_content)

    print("-" * 80)
```

Expected output

### Option 2: Use `similarity_search_with_relevance_scores()`

The `similarity_search_with_relevance_scores()` method returns the top `k` documents with the highest relevance scores. A higher score indicates a higher degree of similarity between a document and your query.

```
docs_with_relevance_score =
vector_store.similarity_search_with_relevance_scores(query, k=2)

for doc, score in docs_with_relevance_score:
    print("-" * 80)
    print("Score: ", score)
    print(doc.page_content)
    print("-" * 80)
```

Expected output

## Use as a retriever

In Langchain, a retriever is an interface that retrieves documents in response to an unstructured query, providing more functionality than a vector store. The following code demonstrates how to use TiDB vector store as a retriever.

```
retriever = vector_store.as_retriever(
   search_type="similarity_score_threshold",
   search_kwargs={"k": 3, "score_threshold": 0.8},
)
docs_retrieved = retriever.invoke(query)
for doc in docs_retrieved:
   print("-" * 80)
   print(doc.page_content)
   print("-" * 80)
```

The expected output is as follows:

```
--------------------------------------------------------------------------------
Tonight. I call on the Senate to: Pass the Freedom to Vote Act. Pass the John Lewis Voting Rights Act. And while you're at it, pass the Disclose Act so Americans can know who is funding our elections.

Tonight, I'd like to honor someone who has dedicated his life to serve this country: Justice Stephen Breyer—an Army veteran, Constitutional scholar, and
```

```
retiring Justice of the United States Supreme Court. Justice Breyer, thank
you for your service.


One of the most serious constitutional responsibilities a President has is
nominating someone to serve on the United States Supreme Court.


And I did that 4 days ago, when I nominated Circuit Court of Appeals Judge
Ketanji Brown Jackson. One of our nation’s top legal minds, who will continue
Justice Breyer’s legacy of excellence.
-----------------------------------------------------------------------------
---
```

## Remove the vector store

To remove an existing TiDB vector store, use the `drop_vectorstore()` method:

```
vector_store.drop_vectorstore()
```

## Search with metadata filters

To refine your searches, you can use metadata filters to retrieve specific nearest-neighbor results that match the applied filters.

## Supported metadata types

Each document in the TiDB vector store can be paired with metadata, structured as key-value pairs within a JSON object. Keys are always strings, while values can be any of the following types:

- String
- Number: integer or floating point
- Boolean: `true` or `false`

For example, the following is a valid metadata payload:

```
{
  "page": 12,
  "book_title": "Siddhartha"
}
```

## Metadata filter syntax

Available filters include the following:

- `$or`: Selects vectors that match any one of the specified conditions.
- `$and`: Selects vectors that match all the specified conditions.
- `$eq`: Equal to the specified value.
- `$ne`: Not equal to the specified value.
- `$gt`: Greater than the specified value.
- `$gte`: Greater than or equal to the specified value.
- `$lt`: Less than the specified value.
- `$lte`: Less than or equal to the specified value.
- `$in`: In the specified array of values.
- `$nin`: Not in the specified array of values.

If the metadata of a document is as follows:

```
{
  "page": 12,
  "book_title": "Siddhartha"
}
```

The following metadata filters can match this document:

```
{ "page": 12 }
```

```
{ "page": { "$eq": 12 } }
```

```
{
  "page": {
    "$in": [11, 12, 13]
  }
}
```

```
{ "page": { "$nin": [13] } }
```

```
{ "page": { "$lt": 11 } }
```

```
{
  "$or": [{ "page": 11 }, { "page": 12 }],
  "$and": [{ "page": 12 }, { "page": 13 }]
}
```

In a metadata filter, TiDB treats each key-value pair as a separate filter clause and combines these clauses using the `AND` logical operator.

## Example

The following example adds two documents to `TiDBVectorStore` and adds a `title` field to each document as the metadata:

```
vector_store.add_texts(
    texts=[
        "TiDB Vector offers advanced, high-speed vector processing capabilities, enhancing AI workflows with efficient data handling and analytics support.",
        "TiDB Vector, starting as low as $10 per month for basic usage",
    ],
    metadatas=[
        {"title": "TiDB Vector functionality"},
        {"title": "TiDB Vector Pricing"},
    ],
)
```

The expected output is as follows:

```
[UUID('c782cb02-8eec-45be-a31f-fdb78914f0a7'),
 UUID('08dcd2ba-9f16-4f29-a9b7-18141f8edae3')]
```

Perform a similarity search with metadata filters:

```
docs_with_score = vector_store.similarity_search_with_score(
    "Introduction to TiDB Vector", filter={"title": "TiDB Vector functionality"}, k=4
)
for doc, score in docs_with_score:
    print("-" * 80)
    print("Score: ", score)
    print(doc.page_content)
    print("-" * 80)
```

The expected output is as follows:

```
--------------------------------------------------------------------------------
Score:  0.12761409169211535
TiDB Vector offers advanced, high-speed vector processing capabilities, enhancing AI workflows with efficient data handling and analytics support.
--------------------------------------------------------------------------------
```

## Advanced usage example: travel agent

This section demonstrates a use case of integrating vector search with Langchain for a travel agent. The goal is to create personalized travel reports for clients, helping them find airports with specific amenities, such as clean lounges and vegetarian options.

The process involves two main steps:

1. Perform a semantic search across airport reviews to identify airport codes that match the desired amenities.
2. Execute a SQL query to merge these codes with route information, highlighting airlines and destinations that align with user's preferences.

## Prepare data

First, create a table to store airport route data:

```
# Create a table to store flight plan data.

vector_store.tidb_vector_client.execute(

    """CREATE TABLE airplan_routes (

        id INT AUTO_INCREMENT PRIMARY KEY,

        airport_code VARCHAR(10),

        airline_code VARCHAR(10),

        destination_code VARCHAR(10),

        route_details TEXT,

        duration TIME,

        frequency INT,

        airplane_type VARCHAR(50),

        price DECIMAL(10, 2),

        layover TEXT

    );"""

)


# Insert some sample data into airplan_routes and the vector table.

vector_store.tidb_vector_client.execute(

    """INSERT INTO airplan_routes (

        airport_code,

        airline_code,
        destination_code,

        route_details,

        duration,

        frequency,

        airplane_type,

        price,

        layover

    ) VALUES

    ('JFK', 'DL', 'LAX', 'Non-stop from JFK to LAX.', '06:00:00', 5, 'Boeing
777', 299.99, 'None'),

    ('LAX', 'AA', 'ORD', 'Direct LAX to ORD route.', '04:00:00', 3, 'Airbus
A320', 149.99, 'None'),

    ('EFGH', 'UA', 'SEA', 'Daily flights from SFO to SEA.', '02:30:00', 7,
'Boeing 737', 129.99, 'None');

    """

)

vector_store.add_texts(

    texts=[

        "Clean lounges and excellent vegetarian dining options. Highly
recommended.",

        "Comfortable seating in lounge areas and diverse food selections,
including vegetarian.",

        "Small airport with basic facilities.",

    ],

    metadatas=[

        {"airport_code": "JFK"},

        {"airport_code": "LAX"},

        {"airport_code": "EFGH"},

    ],

)
```

The expected output is as follows:

```
[UUID('6dab390f-acd9-4c7d-b252-616606fbc89b'),

 UUID('9e811801-0e6b-4893-8886-60f4fb67ce69'),

 UUID('f426747c-0f7b-4c62-97ed-3eeb7c8dd76e')]
```

## Perform a semantic search

The following code searches for airports with clean facilities and vegetarian options:

```
retriever = vector_store.as_retriever(

    search_type="similarity_score_threshold",

    search_kwargs={"k": 3, "score_threshold": 0.85},

)

semantic_query = "Could you recommend a US airport with clean lounges and
good vegetarian dining options?"

reviews = retriever.invoke(semantic_query)

for r in reviews:

    print("-" * 80)

    print(r.page_content)

    print(r.metadata)

    print("-" * 80)
```

The expected output is as follows:

```
-----------------------------------------------------------------------------
---

Clean lounges and excellent vegetarian dining options. Highly recommended.

{'airport_code': 'JFK'}

-----------------------------------------------------------------------------
---
```

```
--------------------------------------------------------------------------------
---

Comfortable seating in lounge areas and diverse food selections, including vegetarian.

{'airport_code': 'LAX'}

--------------------------------------------------------------------------------
---
```

## Retrieve detailed airport information

Extract airport codes from the search results and query the database for detailed route information:

```
# Extracting airport codes from the metadata

airport_codes = [review.metadata["airport_code"] for review in reviews]



# Executing a query to get the airport details

search_query = "SELECT * FROM airplan_routes WHERE airport_code IN :codes"

params = {"codes": tuple(airport_codes)}



airport_details = vector_store.tidb_vector_client.execute(search_query, params)

airport_details.get("result")
```

The expected output is as follows:

```
[(1, 'JFK', 'DL', 'LAX', 'Non-stop from JFK to LAX.', datetime.timedelta(seconds=21600), 5, 'Boeing 777', Decimal('299.99'), 'None'),

 (2, 'LAX', 'AA', 'ORD', 'Direct LAX to ORD route.', datetime.timedelta(seconds=14400), 3, 'Airbus A320', Decimal('149.99'), 'None')]
```

## Streamline the process

Alternatively, you can streamline the entire process using a single SQL query:

```
search_query = f"""

    SELECT

        VEC_Cosine_Distance(se.embedding, :query_vector) as distance,

        ar.*,

        se.document as airport_review

    FROM

        airplan_routes ar

    JOIN

        {TABLE_NAME} se ON ar.airport_code =
JSON_UNQUOTE(JSON_EXTRACT(se.meta, '$.airport_code'))

    ORDER BY distance ASC

    LIMIT 5;

"""

query_vector = embeddings.embed_query(semantic_query)

params = {"query_vector": str(query_vector)}

airport_details = vector_store.tidb_vector_client.execute(search_query,
params)

airport_details.get("result")
```

The expected output is as follows:

```
[(0.1219207353407008, 1, 'JFK', 'DL', 'LAX', 'Non-stop from JFK to LAX.',
datetime.timedelta(seconds=21600), 5, 'Boeing 777', Decimal('299.99'),
'None', 'Clean lounges and excellent vegetarian dining options. Highly
recommended.'),

 (0.14613754359804654, 2, 'LAX', 'AA', 'ORD', 'Direct LAX to ORD route.',
datetime.timedelta(seconds=14400), 3, 'Airbus A320', Decimal('149.99'),
'None', 'Comfortable seating in lounge areas and diverse food selections,
including vegetarian.'),
 (0.19840519342700513, 3, 'EFGH', 'UA', 'SEA', 'Daily flights from SFO to
SEA.', datetime.timedelta(seconds=9000), 7, 'Boeing 737', Decimal('129.99'),
'None', 'Small airport with basic facilities.')]
```

## Clean up data

Finally, clean up the resources by dropping the created table:

```
vector_store.tidb_vector_client.execute("DROP TABLE airplan_routes")
```

The expected output is as follows:

```
{'success': True, 'result': 0, 'error': None}
```