Google Cloud SQL for PostgreSQL - `PostgresVectorStore`

> Cloud SQL is a fully managed relational database service that offers high performance, seamless integration, and impressive scalability. It offers MySQL, PostgreSQL, and SQL Server database engines. Extend your database application to build AI-powered experiences leveraging Cloud SQL's LlamaIndex integrations.

This notebook goes over how to use `Cloud SQL for PostgreSQL` to store vector embeddings with the `PostgresVectorStore` class.

Learn more about the package on GitHub.

Before you begin

To run this notebook, you will need to do the following:

- Create a Google Cloud Project
- Enable the Cloud SQL Admin API.
- Create a Cloud SQL instance.
- Create a Cloud SQL database.
- Add a User to the database.

🦙 Library Installation

Install the integration library, `llama-index-cloud-sql-pg`, and the library for the embedding service, `llama-index-embeddings-vertex`.

```
%pip install --upgrade --quiet llama-index-cloud-sql-pg llama-index-embeddings-vertex llama-index-llms-vertex llama-index
```

Colab only: Uncomment the following cell to restart the kernel or use the button to restart the kernel. For Vertex AI Workbench you can restart the terminal using the button on top.

```
# # Automatically restart kernel after installs so that your
environment can access the new packages
# import IPython

# app = IPython.Application.instance()
# app.kernel.do_shutdown(True)
```

🔐 Authentication

Authenticate to Google Cloud as the IAM user logged into this notebook in order to access your Google Cloud Project.

- If you are using Colab to run this notebook, use the cell below and continue.
- If you are using Vertex AI Workbench, check out the setup instructions here.

```
from google.colab import auth

auth.authenticate_user()
```

☁️ Set Your Google Cloud Project

Set your Google Cloud project so that you can leverage Google Cloud resources within this notebook.

If you don't know your project ID, try the following:

- Run `gcloud config list`.
- Run `gcloud projects list`.
- See the support page: Locate the project ID.

```
# @markdown Please fill in the value below with your Google Cloud
project ID and then run the cell.

PROJECT_ID = "my-project-id"  # @param {type:"string"}
```

```
# Set the project id
!gcloud config set project {PROJECT_ID}
```

Basic Usage

Set Cloud SQL database values

Find your database values, in the Cloud SQL Instances page.

```
# @title Set Your Values Here { display-mode: "form" }
REGION = "us-central1"  # @param {type: "string"}
INSTANCE = "my-primary"  # @param {type: "string"}
DATABASE = "my-database"  # @param {type: "string"}
TABLE_NAME = "vector_store"  # @param {type: "string"}
USER = "postgres"  # @param {type: "string"}
PASSWORD = "my-password"  # @param {type: "string"}
```

PostgresEngine Connection Pool

One of the requirements and arguments to establish Cloud SQL as a vector store is a `PostgresEngine` object. The `PostgresEngine` configures a connection pool to your Cloud SQL database, enabling successful connections from your application and following industry best practices.

To create a `PostgresEngine` using `PostgresEngine.from_instance()` you need to provide only 4 things:

1. `project_id` : Project ID of the Google Cloud Project where the Cloud SQL instance is located.
2. `region` : Region where the Cloud SQL instance is located.
3. `instance` : The name of the Cloud SQL instance.
4. `database` : The name of the database to connect to on the Cloud SQL instance.

By default, IAM database authentication will be used as the method of database authentication. This library uses the IAM principal belonging to the Application Default Credentials (ADC) sourced from the envionment.

For more informatin on IAM database authentication please see:

- Configure an instance for IAM database authentication
- Manage users with IAM database authentication

Optionally, built-in database authentication using a username and password to access the Cloud SQL database can also be used. Just provide the optional `user` and `password` arguments to `PostgresEngine.from_instance()`:

- `user` : Database user to use for built-in database authentication and login
- `password` : Database password to use for built-in database authentication and login.

Note: This tutorial demonstrates the async interface. All async methods have corresponding sync methods.

```
from llama_index_cloud_sql_pg import PostgresEngine

engine = await PostgresEngine.afrom_instance(
    project_id=PROJECT_ID,
    region=REGION,
    instance=INSTANCE,
    database=DATABASE,
    user=USER,
    password=PASSWORD,
)
```

Initialize a table

The `PostgresVectorStore` class requires a database table. The `PostgresEngine` engine has a helper method `init_vector_store_table()` that can be used to create a table with the proper schema for you.

```
await engine.ainit_vector_store_table(
    table_name=TABLE_NAME,
    vector_size=768,  # Vector size for VertexAI
model(textembedding-gecko@latest)
)
```

Optional Tip: 💡

You can also specify a schema name by passing `schema_name` wherever you pass `table_name`.

```
SCHEMA_NAME = "my_schema"

await engine.ainit_vector_store_table(
    table_name=TABLE_NAME,
    schema_name=SCHEMA_NAME,
    vector_size=768,
)
```

Create an embedding class instance

You can use any Llama Index embeddings model. You may need to enable Vertex AI API to use `VertexTextEmbeddings`. We recommend setting the embedding model's version for production, learn more about the Text embeddings models.

```
# enable Vertex AI API
!gcloud services enable aiplatform.googleapis.com
```

```
from llama_index.core import Settings
from llama_index.embeddings.vertex import VertexTextEmbedding
from llama_index.llms.vertex import Vertex
import google.auth

credentials, project_id = google.auth.default()
Settings.embed_model = VertexTextEmbedding(
    model_name="textembedding-gecko@003",
    project=PROJECT_ID,
    credentials=credentials,
)

Settings.llm = Vertex(model="gemini-1.5-flash-002",
project=PROJECT_ID)
```

Initialize a default PostgresVectorStore

```
from llama_index_cloud_sql_pg import PostgresVectorStore
```

```
vector_store = await PostgresVectorStore.create(
    engine=engine,
    table_name=TABLE_NAME,
    # schema_name=SCHEMA_NAME
)
```

Download data

```
!mkdir -p 'data/paul_graham/'
!wget 'https://raw.githubusercontent.com/run-llama/llama_index/main/docs/docs/examples/data/paul_graham/paul_graham_essay.txt' -O 'data/paul_graham/paul_graham_essay.txt'
```

Load documents

```
from llama_index.core import SimpleDirectoryReader

documents = SimpleDirectoryReader("./data/paul_graham").load_data()
print("Document ID:", documents[0].doc_id)
```

Use with VectorStoreIndex

Create an index from the vector store by using `VectorStoreIndex`.

Initialize Vector Store with documents

The simplest way to use a Vector Store is to load a set of documents and build an index from them using `from_documents`.

```
from llama_index.core import StorageContext, VectorStoreIndex

storage_context = StorageContext.from_defaults(vector_store=vector_store)
index = VectorStoreIndex.from_documents(
    documents, storage_context=storage_context, show_progress=True
)
```

Query the index

```
query_engine = index.as_query_engine()
response = query_engine.query("What did the author do?")
print(response)
```

Create a custom Vector Store

A Vector Store can take advantage of relational data to filter similarity searches.

Create a new table with custom metadata columns. You can also re-use an existing table which already has custom columns for a Document's id, content, embedding, and/or metadata.

```
from llama_index_cloud_sql_pg import Column

# Set table name
TABLE_NAME = "vectorstore_custom"
# SCHEMA_NAME = "my_schema"

await engine.ainit_vector_store_table(
    table_name=TABLE_NAME,
    # schema_name=SCHEMA_NAME,
    vector_size=768,  # VertexAI model: textembedding-gecko@003
    metadata_columns=[Column("len", "INTEGER")],
)


# Initialize PostgresVectorStore
custom_store = await PostgresVectorStore.create(
    engine=engine,
    table_name=TABLE_NAME,
    # schema_name=SCHEMA_NAME,
    metadata_columns=["len"],
)
```

Add documents with metadata

Document `metadata` can provide the LLM and retrieval process with more information. Learn more about different approaches for extracting and adding metadata.

```
from llama_index.core import Document

fruits = ["apple", "pear", "orange", "strawberry", "banana",
"kiwi"]
documents = [
    Document(text=fruit, metadata={"len": len(fruit)}) for fruit in
fruits
]

storage_context =
StorageContext.from_defaults(vector_store=custom_store)
custom_doc_index = VectorStoreIndex.from_documents(
    documents, storage_context=storage_context, show_progress=True
)
```

Search for documents with metadata filter

You can apply pre-filtering to the search results by specifying a `filters` argument

```
from llama_index.core.vector_stores.types import (
    MetadataFilter,
    MetadataFilters,
    FilterOperator,
)

filters = MetadataFilters(
    filters=[
        MetadataFilter(key="len", operator=FilterOperator.GT,
value="5"),
    ],
)

query_engine = custom_doc_index.as_query_engine(filters=filters)
res = query_engine.query("List some fruits")
print(str(res.source_nodes[0].text))
```

Add a Index

Speed up vector search queries by applying a vector index. Learn more about vector indexes.

```
from llama_index_cloud_sql_pg.indexes import IVFFlatIndex

index = IVFFlatIndex()
await vector_store.aapply_vector_index(index)
```

Re-index

```
await vector_store.areindex()  # Re-index using default index name
```

Remove an index

```
await vector_store.adrop_vector_index()  # Delete index using
default name
```