ChatGPT

Using PostgresStore for LangGraph Memory (Replacing InMemoryStore)

To persist memory across sessions, you can replace the in-memory store with a PostgreSQL-backed store. The `PostgresStore` in LangGraph supports vector-based semantic search by using the **pgvector** extension in PostgreSQL. Below, we update the code snippet to use `PostgresStore` instead of `InMemoryStore` – all tool usage remains the same, only the store initialization changes [1] [2] :

```
import argparse
from dotenv import load_dotenv
from pydantic import BaseModel, Field

_ = load_dotenv()

# Use PostgresStore instead of InMemoryStore
from langgraph.store.postgres import PostgresStore
from langmem import create_manage_memory_tool, create_search_memory_tool

# Connect to a PostgreSQL database (make sure it's running and accessible)
conn_string = "postgresql://postgres:<password>@localhost:5432/<database>"

# Initialize the Postgres-backed store with vector index configuration
with PostgresStore.from_conn_string(conn_string, index={
    "dims": 1536,                              # embedding dimensions
    "embed": "openai:text-embedding-3-small",  # embedding model for vector
search
}) as store:
    store.setup()  # Run migrations to create tables and indexes (do this once)

    # Create memory tools with a namespace for emails (e.g., per user)
    manage_memory_tool = create_manage_memory_tool(
        namespace=("email_assistant", "user1", "collection"),
        store=store
    )
    search_memory_tool = create_search_memory_tool(
        namespace=("email_assistant", "user1", "collection"),
        store=store
    )

    email1 = "Reminder: Team meeting tomorrow at 10 AM."
    email2 = "Project Update – Completed the initial design draft."
```

```
# Store the emails in memory (create new memory entries)
r1 = manage_memory_tool.invoke({"action": "create", "content": email1})
print("create #1 ->", r1)

r2 = manage_memory_tool.invoke({"action": "create", "content": email2})
print("create #2 ->", r2)

# Search memories (semantic search by query meaning)
search_out = search_memory_tool.invoke({"query": "meeting", "limit": 5})
print("\nsearch ->", search_out)
```

Notes: In this updated code, we import `PostgresStore` and provide a PostgreSQL connection string. We then call `PostgresStore.from_conn_string(...)` with an `index` configuration (embedding model and dimension). This ensures that any content stored will be indexed for semantic similarity search [1]. We call `store.setup()` once to run the necessary database migrations (creating tables and indexes) before using the store [3]. The rest of the code (creating memory tools and invoking them) remains unchanged. Make sure to replace `<password>` and `<database>` in the connection string with your actual PostgreSQL credentials and database name. Also, ensure that the `pgvector` extension is installed in your database, since semantic vector search requires it [4].

Installing PostgreSQL on macOS

To run the above code, you need a local PostgreSQL server on your Mac. Here's how to install and set up PostgreSQL (including the pgvector extension for vector search):

1. **Install Homebrew (if not already installed):** Homebrew is a package manager for macOS. If you don't have it, install it by following instructions on brew.sh. Once Homebrew is ready, proceed to install PostgreSQL.

2. **Install PostgreSQL via Homebrew:** Open Terminal and run:

   ```
   brew install postgresql
   ```

   This will install the latest PostgreSQL on your system [5]. Homebrew will also initialize a default database cluster for you (with a default database, usually named `postgres`).

3. **Start the PostgreSQL service:** After installation, start the PostgreSQL server as a background service:

   ```
   brew services start postgresql
   ```

   Wait a moment and then verify it's running by checking `brew services list` (it should show PostgreSQL as started) [6]. Once running, PostgreSQL will listen on the default port 5432.

4. **[Optional] Create a database/user:** By default, Homebrew's setup creates a **default database** (often named `postgres`) and allows the current macOS user to connect with trust authentication (no password needed). You can use this default database in your connection string (for example, use `/<database>=postgres` and your macOS username as the user). Alternatively, to use the `postgres` user with a password, you may need to create that role and a database. For instance, you can open the PostgreSQL shell with `psql` and run SQL commands to create a user and database:

   ```
   CREATE ROLE postgres WITH LOGIN SUPERUSER PASSWORD 'your_password';
   CREATE DATABASE your_database OWNER postgres;
   ```

   If you use the default setup, you can skip this step and simply use the default database and user in your connection string.

5. **Install the pgvector extension:** To enable vector similarity search, install the pgvector extension via Homebrew and enable it in your database:

   ```
   brew install pgvector
   ```

   After installing, load the extension in Postgres by running:

   ```
   psql -d postgres -c "CREATE EXTENSION vector;"
   ```

   This will install the `vector` data type in the `postgres` database (replace `postgres` with your database name if different) [7]. Ensure the Postgres server is running when you execute this.

6. **Install Python dependencies (if needed):** In your Python environment, make sure you have the PostgreSQL driver installed. The `langgraph.store.postgres` uses the `psycopg` library under the hood, so install it if not already present:

   ```
   pip install psycopg[binary]
   ```

   (Many LangGraph installations include this, but if you see errors about missing `psycopg`, this step will fix it.) [8]

Once PostgreSQL is installed and running on your Mac, update the `conn_string` in the code to use your actual username, password, and database name. For example, if you created a `postgres` user with password `"mypassword"` and a database named `mydb`, your connection string might be:

```
conn_string = "postgresql://postgres:mypassword@localhost:5432/mydb"
```

Now you can run the updated script. It will connect to the PostgreSQL database, store the emails in the `PostgresStore`, and perform a semantic search. The use of `PostgresStore` ensures your data is persisted in the database between runs (unlike `InMemoryStore` which is volatile), and with pgvector enabled you can query by semantic similarity of text [4] .

References:

- LangGraph documentation on using `PostgresStore` for vector search [2] [4]
- LangChain blog – *Semantic Search for LangGraph Memory* (example of initializing PostgresStore with embeddings) [1]
- Homebrew PostgreSQL installation guide (installing and starting PostgreSQL on macOS) [5] [6]
- Stack Overflow – enabling pgvector extension on Mac (installation and `CREATE EXTENSION vector`) [7]
- *LangGraph + PostgreSQL* tutorial (prerequisites including `psycopg` driver) [8]

---

[1] Semantic Search for LangGraph Memory
https://blog.langchain.com/semantic-search-for-langgraph-memory/

[2] [3] [4] Storage
https://langchain-ai.github.io/langgraph/reference/store/

[5] [6] How to install PostgreSQL on a Mac with Homebrew
https://www.moncefbelyamani.com/how-to-install-postgresql-on-a-mac-with-homebrew-and-lunchy/

[7] postgresql - Install pgvector extension on mac - Stack Overflow
https://stackoverflow.com/questions/75664004/install-pgvector-extension-on-mac

[8] Using PostgreSQL with LangGraph for State Management and Vector Storage | by Sajith K | Medium
https://medium.com/@sajith_k/using-postgresql-with-langgraph-for-state-management-and-vector-storage-df4ca9d9b89e