Caching integrations

This notebook covers how to cache results of individual LLM calls.

```
import langchain
from langchain.llms import OpenAI

# To make the caching really obvious, lets use a slower model.
llm = OpenAI(model_name="text-davinci-002", n=2, best_of=2)
```

In Memory Cache

```
from langchain.cache import InMemoryCache

langchain.llm_cache = InMemoryCache()
```

```
# The first time, it is not yet in cache, so it should take longer
llm("Tell me a joke")
```

```
    CPU times: user 35.9 ms, sys: 28.6 ms, total: 64.6 ms
    Wall time: 4.83 s
    "\n\nWhy couldn't the bicycle stand up by itself? It was...two tired!"
```

```
# The second time it is, so it goes faster
llm("Tell me a joke")
```

```
    CPU times: user 238 µs, sys: 143 µs, total: 381 µs
    Wall time: 1.76 ms





    '\n\nWhy did the chicken cross the road?\n\nTo get to the other side.'
```

SQLite Cache

```
rm .langchain.db
```

```
# We can do the same thing with a SQLite cache
from langchain.cache import SQLiteCache

langchain.llm_cache = SQLiteCache(database_path=".langchain.db")
```

```
# The first time, it is not yet in cache, so it should take longer
llm("Tell me a joke")
```

```
    CPU times: user 17 ms, sys: 9.76 ms, total: 26.7 ms
    Wall time: 825 ms





    '\n\nWhy did the chicken cross the road?\n\nTo get to the other side.'
```

```
# The second time it is, so it goes faster
llm("Tell me a joke")
```

```
    CPU times: user 2.46 ms, sys: 1.23 ms, total: 3.7 ms
    Wall time: 2.67 ms





    '\n\nWhy did the chicken cross the road?\n\nTo get to the other side.'
```

Redis Cache

Standard Cache

Use Redis to cache prompts and responses.

```
# We can do the same thing with a Redis cache
# (make sure your local Redis instance is running first before running this example)
from redis import Redis
from langchain.cache import RedisCache

langchain.llm_cache = RedisCache(redis_=Redis())
```

```
# The first time, it is not yet in cache, so it should take longer
llm("Tell me a joke")
```

```
    CPU times: user 6.88 ms, sys: 8.75 ms, total: 15.6 ms
    Wall time: 1.04 s







    '\n\nWhy did the chicken cross the road?\n\nTo get to the other side!'
```

```
# The second time it is, so it goes faster
llm("Tell me a joke")
```

```
    CPU times: user 1.59 ms, sys: 610 µs, total: 2.2 ms
    Wall time: 5.58 ms





    '\n\nWhy did the chicken cross the road?\n\nTo get to the other side!'
```

Semantic Cache

Use Redis to cache prompts and responses and evaluate hits based on semantic similarity.

```
from langchain.embeddings import OpenAIEmbeddings
from langchain.cache import RedisSemanticCache


langchain.llm_cache = RedisSemanticCache(
    redis_url="redis://localhost:6379", embedding=OpenAIEmbeddings()
)
```

```
# The first time, it is not yet in cache, so it should take longer
llm("Tell me a joke")
```

```
    CPU times: user 351 ms, sys: 156 ms, total: 507 ms
    Wall time: 3.37 s
    "\n\nWhy don't scientists trust atoms?\nBecause they make up everything."
```

```
# The second time, while not a direct hit, the question is semantically similar to the original question,
# so it uses the cached result!
llm("Tell me one joke")
```

```
    CPU times: user 6.25 ms, sys: 2.72 ms, total: 8.97 ms
    Wall time: 262 ms




    "\n\nWhy don't scientists trust atoms?\nBecause they make up everything."
```

GPTCache

We can use GPTCache for exact match caching OR to cache results based on semantic similarity

Let's first start with an example of exact match

```
from gptcache import Cache
from gptcache.manager.factory import manager_factory
from gptcache.processor.pre import get_prompt
from langchain.cache import GPTCache
```

```
import hashlib


def get_hashed_name(name):
    return hashlib.sha256(name.encode()).hexdigest()


def init_gptcache(cache_obj: Cache, llm: str):
    hashed_llm = get_hashed_name(llm)
    cache_obj.init(
        pre_embedding_func=get_prompt,
        data_manager=manager_factory(manager="map", data_dir=f"map_cache_{hashed_llm}"),
    )


langchain.llm_cache = GPTCache(init_gptcache)
```

```
# The first time, it is not yet in cache, so it should take longer
llm("Tell me a joke")
```

```
    CPU times: user 21.5 ms, sys: 21.3 ms, total: 42.8 ms
    Wall time: 6.2 s





    '\n\nWhy did the chicken cross the road?\n\nTo get to the other side!'
```

```
# The second time it is, so it goes faster
llm("Tell me a joke")
```

```
    CPU times: user 571 µs, sys: 43 µs, total: 614 µs
    Wall time: 635 µs





    '\n\nWhy did the chicken cross the road?\n\nTo get to the other side!'
```

Let's now show an example of similarity caching

```
from gptcache import Cache
from gptcache.adapter.api import init_similar_cache
from langchain.cache import GPTCache
import hashlib


def get_hashed_name(name):
    return hashlib.sha256(name.encode()).hexdigest()


def init_gptcache(cache_obj: Cache, llm: str):
    hashed_llm = get_hashed_name(llm)
    init_similar_cache(cache_obj=cache_obj, data_dir=f"similar_cache_{hashed_llm}")
```

```
langchain.llm_cache = GPTCache(init_gptcache)
```

```
# The first time, it is not yet in cache, so it should take longer
llm("Tell me a joke")
```

```
    CPU times: user 1.42 s, sys: 279 ms, total: 1.7 s
    Wall time: 8.44 s





    '\n\nWhy did the chicken cross the road?\n\nTo get to the other side.'
```

```
# This is an exact match, so it finds it in the cache
llm("Tell me a joke")
```

```
    CPU times: user 866 ms, sys: 20 ms, total: 886 ms
    Wall time: 226 ms





    '\n\nWhy did the chicken cross the road?\n\nTo get to the other side.'
```

```
# This is not an exact match, but semantically within distance so it hits!
llm("Tell me joke")
```

```
    CPU times: user 853 ms, sys: 14.8 ms, total: 868 ms
    Wall time: 224 ms





    '\n\nWhy did the chicken cross the road?\n\nTo get to the other side.'
```

Momento Cache

Use Momento to cache prompts and responses.

Requires momento to use, uncomment below to install:

```
# !pip install momento
```

You'll need to get a Momento auth token to use this class. This can either be passed in to a momento.CacheClient if you'd like to instantiate that directly, as a named parameter `auth_token` to `MomentoChatMessageHistory.from_client_params`, or can just be set as an environment variable `MOMENTO_AUTH_TOKEN`.

```python
from datetime import timedelta

from langchain.cache import MomentoCache


cache_name = "langchain"
ttl = timedelta(days=1)
langchain.llm_cache = MomentoCache.from_client_params(cache_name, ttl)
```

```python
# The first time, it is not yet in cache, so it should take longer
llm("Tell me a joke")
```

```
    CPU times: user 40.7 ms, sys: 16.5 ms, total: 57.2 ms
    Wall time: 1.73 s





    '\n\nWhy did the chicken cross the road?\n\nTo get to the other side!'
```

```python
# The second time it is, so it goes faster
# When run in the same region as the cache, latencies are single digit ms
llm("Tell me a joke")
```

```
    CPU times: user 3.16 ms, sys: 2.98 ms, total: 6.14 ms
    Wall time: 57.9 ms
    '\n\nWhy did the chicken cross the road?\n\nTo get to the other side!'
```

SQLAlchemy Cache

```
# You can use SQLAlchemyCache to cache with any SQL database supported by SQLAlchemy.

# from langchain.cache import SQLAlchemyCache
# from sqlalchemy import create_engine

# engine = create_engine("postgresql://postgres:postgres@localhost:5432/postgres")
# langchain.llm_cache = SQLAlchemyCache(engine)
```

Custom SQLAlchemy Schemas

```
# You can define your own declarative SQLAlchemyCache child class to customize the schema used for caching.
For example, to support high-speed fulltext prompt indexing with Postgres, use:

from sqlalchemy import Column, Integer, String, Computed, Index, Sequence
from sqlalchemy import create_engine
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy_utils import TSVectorType
from langchain.cache import SQLAlchemyCache
```

```python
Base = declarative_base()


class FulltextLLMCache(Base):  # type: ignore
    """Postgres table for fulltext-indexed LLM Cache"""

    __tablename__ = "llm_cache_fulltext"
    id = Column(Integer, Sequence("cache_id"), primary_key=True)
    prompt = Column(String, nullable=False)
    llm = Column(String, nullable=False)
    idx = Column(Integer)
    response = Column(String)
    prompt_tsv = Column(
        TSVectorType(),
        Computed("to_tsvector('english', llm || ' ' || prompt)", persisted=True),
    )
    __table_args__ = (
        Index("idx_fulltext_prompt_tsv", prompt_tsv, postgresql_using="gin"),
    )


engine = create_engine("postgresql://postgres:postgres@localhost:5432/postgres")
langchain.llm_cache = SQLAlchemyCache(engine, FulltextLLMCache)
```

Optional Caching

You can also turn off caching for specific LLMs should you choose. In the example below, even though global caching is enabled, we turn it off for a specific LLM

```
llm = OpenAI(model_name="text-davinci-002", n=2, best_of=2, cache=False)
```

```
llm("Tell me a joke")
```

```
    CPU times: user 5.8 ms, sys: 2.71 ms, total: 8.51 ms
    Wall time: 745 ms





    '\n\nWhy did the chicken cross the road?\n\nTo get to the other side!'
```

```
llm("Tell me a joke")
```

```
    CPU times: user 4.91 ms, sys: 2.64 ms, total: 7.55 ms
    Wall time: 623 ms





    '\n\nTwo guys stole a calendar. They got six months each.'
```

Optional Caching in Chains

You can also turn off caching for particular nodes in chains. Note that because of certain interfaces, its often easier to construct the chain first, and then edit the LLM afterwards.

As an example, we will load a summarizer map-reduce chain. We will cache results for the map-step, but then not freeze it for the combine step.

```
llm = OpenAI(model_name="text-davinci-002")
no_cache_llm = OpenAI(model_name="text-davinci-002", cache=False)
```

```
from langchain.text_splitter import CharacterTextSplitter
from langchain.chains.mapreduce import MapReduceChain

text_splitter = CharacterTextSplitter()
```

```
with open("../../../state_of_the_union.txt") as f:
    state_of_the_union = f.read()
texts = text_splitter.split_text(state_of_the_union)
```

```
from langchain.docstore.document import Document

docs = [Document(page_content=t) for t in texts[:3]]
from langchain.chains.summarize import load_summarize_chain
```

```
chain = load_summarize_chain(llm, chain_type="map_reduce", reduce_llm=no_cache_llm)
```

```
chain.run(docs)
```

```
    CPU times: user 452 ms, sys: 60.3 ms, total: 512 ms
    Wall time: 5.09 s





    '\n\nPresident Biden is discussing the American Rescue Plan and the Bipartisan Infrastructure Law, which will create jobs and help Americans. He also talks about his vision for America, which includes investing in education and infrastructure. In response to Russian aggression in Ukraine, the United States is joining with European allies to impose sanctions and isolate Russia. American forces are being mobilized to protect NATO countries in the event that Putin decides to keep moving west. The Ukrainians are bravely fighting back, but the next few weeks will be hard for them. Putin will pay a high price for his actions in the long run. Americans should not be alarmed, as the United States is taking action to protect its interests and allies.'
```

When we run it again, we see that it runs substantially faster but the final answer is different. This is due to caching at the map steps, but not at the reduce step.

```
chain.run(docs)
```

```
    CPU times: user 11.5 ms, sys: 4.33 ms, total: 15.8 ms
    Wall time: 1.04 s
    '\n\nPresident Biden is discussing the American Rescue Plan and the Bipartisan Infrastructure Law, which
will create jobs and help Americans. He also talks about his vision for America, which includes investing in
education and infrastructure.'
```

```
rm .langchain.db sqlite.db
```