LlamaIndex

Jerry Liu

Apr 28, 2023

GETTING STARTED

LlamaIndex (GPT Index) is a project that provides a central interface to connect your LLM's with external data.

- Github: https://github.com/jerryjliu/llama_index
- **PyPi:**
  - LlamaIndex: https://pypi.org/project/llama-index/.
  - GPT Index (duplicate): https://pypi.org/project/gpt-index/.
- Twitter: https://twitter.com/gpt_index
- Discord https://discord.gg/dGcwcsnxhU

CHAPTER

ONE

ECOSYSTEM

- LlamaHub: https://llamahub.ai
- LlamaLab: https://github.com/run-llama/llama-lab

1.1 Overview

CHAPTER

TWO

CONTEXT

- LLMs are a phenomenonal piece of technology for knowledge generation and reasoning. They are pre-trained on large amounts of publicly available data.
- How do we best augment LLMs with our own private data?
- One paradigm that has emerged is *in-context* learning (the other is finetuning), where we insert context into the input prompt. That way, we take advantage of the LLM's reasoning capabilities to generate a response.

To perform LLM's data augmentation in a performant, efficient, and cheap manner, we need to solve two components:

- Data Ingestion
- Data Indexing

CHAPTER

THREE

PROPOSED SOLUTION

That's where the **LlamaIndex** comes in. LlamaIndex is a simple, flexible interface between your external data and LLMs. It provides the following tools in an easy-to-use fashion:

- Offers data connectors to your existing data sources and data formats (API's, PDF's, docs, SQL, etc.)
- Provides **indices** over your unstructured and structured data for use with LLM's. These indices help to abstract away common boilerplate and pain points for in-context learning:
  - Storing context in an easy-to-access format for prompt insertion.
  - Dealing with prompt limitations (e.g. 4096 tokens for Davinci) when context is too big.
  - Dealing with text splitting.
- Provides users an interface to **query** the index (feed in an input prompt) and obtain a knowledge-augmented output.
- Offers you a comprehensive toolset trading off cost and performance.

3.1 Installation and Setup

3.1.1 Installation from Pip

You can simply do:

```
pip install llama-index
```

3.1.2 Installation from Source

Git clone this repository: `git clone git@github.com:jerryjliu/gpt_index.git`. Then do:

- `pip install -e .` if you want to do an editable install (you can modify source files) of just the package itself.
- `pip install -r requirements.txt` if you want to install optional dependencies + dependencies used for development (e.g. unit testing).

3.1.3 Environment Setup

By default, we use the OpenAI GPT-3 `text-davinci-003` model. In order to use this, you must have an OPENAI_API_KEY setup. You can register an API key by logging into OpenAI's page and creating a new API token.

You can customize the underlying LLM in the *Custom LLMs How-To* (courtesy of Langchain). You may need additional environment keys + tokens setup depending on the LLM provider.

3.2 Starter Tutorial

Here is a starter example for using LlamaIndex. Make sure you've followed the *installation* steps first.

3.2.1 Download

LlamaIndex examples can be found in the `examples` folder of the LlamaIndex repository. We first want to download this `examples` folder. An easy way to do this is to just clone the repo:

```
$ git clone https://github.com/jerryjliu/gpt_index.git
```

Next, navigate to your newly-cloned repository, and verify the contents:

```
$ cd gpt_index
$ ls
LICENSE                 data_requirements.txt  tests/
MANIFEST.in             examples/              pyproject.toml
Makefile                experimental/          requirements.txt
README.md               gpt_index/             setup.py
```

We now want to navigate to the following folder:

```
$ cd examples/paul_graham_essay
```

This contains LlamaIndex examples around Paul Graham's essay, "What I Worked On". A comprehensive set of examples are already provided in `TestEssay.ipynb`. For the purposes of this tutorial, we can focus on a simple example of getting LlamaIndex up and running.

3.2.2 Build and Query Index

Create a new `.py` file with the following:

```
from llama_index import GPTSimpleVectorIndex, SimpleDirectoryReader

documents = SimpleDirectoryReader('data').load_data()
index = GPTSimpleVectorIndex.from_documents(documents)
```

This builds an index over the documents in the `data` folder (which in this case just consists of the essay text). We then run the following

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

You should get back a response similar to the following: `The author wrote short stories and tried to program on an IBM 1401.`

3.2.3 Viewing Queries and Events Using Logging

In a Jupyter notebook, you can view info and/or debugging logging using the following snippet:

```
import logging
import sys

logging.basicConfig(stream=sys.stdout, level=logging.DEBUG)
logging.getLogger().addHandler(logging.StreamHandler(stream=sys.stdout))
```

You can set the level to `DEBUG` for verbose output, or use `level=logging.INFO` for less.

3.2.4 Saving and Loading

To save to disk and load from disk, do

```
# save to disk
index.save_to_disk('index.json')
# load from disk
index = GPTSimpleVectorIndex.load_from_disk('index.json')
```

3.2.5 Next Steps

That's it! For more information on LlamaIndex features, please check out the numerous "Guides" to the left. If you are interested in further exploring how LlamaIndex works, check out our *Primer Guide*.

Additionally, if you would like to play around with Example Notebooks, check out *this link*.

3.3 A Primer to using LlamaIndex

At its core, LlamaIndex contains a toolkit designed to easily connect LLM's with your external data. LlamaIndex helps to provide the following:

- A set of **data structures** that allow you to index your data for various LLM tasks, and remove concerns over prompt size limitations.
- Data connectors to your common data sources (Google Docs, Slack, etc.).
- Cost transparency + tools that reduce cost while increasing performance.

Each data structure offers distinct use cases and a variety of customizable parameters. These indices can then be *queried* in a general purpose manner, in order to achieve any task that you would typically achieve with an LLM:

- Question-Answering
- Summarization
- Text Generation (Stories, TODO's, emails, etc.)
- and more!

The guides below are intended to help you get the most out of LlamaIndex. It gives a high-level overview of the following:

1. The general usage pattern of LlamaIndex.
2. Mapping Use Cases to LlamaIndex data Structures
3. How Each Index Works

3.3.1 LlamaIndex Usage Pattern

The general usage pattern of LlamaIndex is as follows:

1. Load in documents (either manually, or through a data loader)
2. Parse the Documents into Nodes
3. Construct Index (from Nodes or Documents)
4. [Optional, Advanced] Building indices on top of other indices
5. Query the index

1. Load in Documents

The first step is to load in data. This data is represented in the form of `Document` objects. We provide a variety of *data loaders* which will load in Documents through the `load_data` function, e.g.:

```
from llama_index import SimpleDirectoryReader

documents = SimpleDirectoryReader('data').load_data()
```

You can also choose to construct documents manually. LlamaIndex exposes the `Document` struct.

```
from llama_index import Document

text_list = [text1, text2, ...]
documents = [Document(t) for t in text_list]
```

A Document represents a lightweight container around the data source. You can now choose to proceed with one of the following steps:

1. Feed the Document object directly into the index (see section 3).
2. First convert the Document into Node objects (see section 2).

2. Parse the Documents into Nodes

The next step is to parse these Document objects into Node objects. Nodes represent "chunks" of source Documents, whether that is a text chunk, an image, or more. They also contain metadata and relationship information with other nodes and index structures.

Nodes are a first-class citizen in LlamaIndex. You can choose to define Nodes and all its attributes directly. You may also choose to "parse" source Documents into Nodes through our `NodeParser` classes.

For instance, you can do

```
from llama_index.node_parser import SimpleNodeParser

parser = SimpleNodeParser()

nodes = parser.get_nodes_from_documents(documents)
```

You can also choose to construct Node objects manually and skip the first section. For instance,

```
from llama_index.data_structs.node_v2 import Node, DocumentRelationship

node1 = Node(text="<text_chunk>", doc_id="<node_id>")
node2 = Node(text="<text_chunk>", doc_id="<node_id>")
# set relationships
node1.relationships[DocumentRelationship.NEXT] = node2.get_doc_id()
node2.relationships[DocumentRelationship.PREVIOUS] = node1.get_doc_id()
```

3. Index Construction

We can now build an index over these Document objects. The simplest high-level abstraction is to load-in the Document objects during index initialization (this is relevant if you came directly from step 1 and skipped step 2).

```
from llama_index import GPTSimpleVectorIndex

index = GPTSimpleVectorIndex.from_documents(documents)
```

You can also choose to build an index over a set of Node objects directly (this is a continuation of step 2).

```
from llama_index import GPTSimpleVectorIndex

index = GPTSimpleVectorIndex(nodes)
```

Depending on which index you use, LlamaIndex may make LLM calls in order to build the index.

Reusing Nodes across Index Structures

If you have multiple Node objects defined, and wish to share these Node objects across multiple index structures, you can do that. Simply define a DocumentStore object, add the Node objects to the DocumentStore, and pass the DocumentStore around.

```
from gpt_index.docstore import SimpleDocumentStore

docstore = SimpleDocumentStore()
docstore.add_documents(nodes)

index1 = GPTSimpleVectorIndex(nodes, docstore=docstore)
index2 = GPTListIndex(nodes, docstore=docstore)
```

NOTE: If the `docstore` argument isn't specified, then it is implicitly created for each index during index construction. You can access the docstore associated with a given index through `index.docstore`.

Inserting Documents or Nodes

You can also take advantage of the `insert` capability of indices to insert Document objects one at a time instead of during index construction.

```
from llama_index import GPTSimpleVectorIndex

index = GPTSimpleVectorIndex([])
for doc in documents:
    index.insert(doc)
```

If you want to insert nodes on directly you can use `insert_nodes` function instead.

```
from llama_index import GPTSimpleVectorIndex

# nodes: Sequence[Node]
index = GPTSimpleVectorIndex([])
index.insert_nodes(nodes)
```

See the *Update Index How-To* for details and an example notebook.

Customizing LLM's

By default, we use OpenAI's `text-davinci-003` model. You may choose to use another LLM when constructing an index.

```
from llama_index import LLMPredictor, GPTSimpleVectorIndex, PromptHelper, ServiceContext
from langchain import OpenAI

...

# define LLM
llm_predictor = LLMPredictor(llm=OpenAI(temperature=0, model_name="text-davinci-003"))

# define prompt helper
# set maximum input size
max_input_size = 4096
# set number of output tokens
num_output = 256
# set maximum chunk overlap
max_chunk_overlap = 20
prompt_helper = PromptHelper(max_input_size, num_output, max_chunk_overlap)

service_context = ServiceContext.from_defaults(llm_predictor=llm_predictor, prompt_
↪helper=prompt_helper)

index = GPTSimpleVectorIndex.from_documents(
```

(continues on next page)

(continued from previous page)

```
    documents, service_context=service_context
)
```

See the *Custom LLM's How-To* for more details.

Customizing Prompts

Depending on the index used, we used default prompt templates for constructing the index (and also insertion/querying). See *Custom Prompts How-To* for more details on how to customize your prompt.

Customizing embeddings

For embedding-based indices, you can choose to pass in a custom embedding model. See *Custom Embeddings How-To* for more details.

Cost Predictor

Creating an index, inserting to an index, and querying an index may use tokens. We can track token usage through the outputs of these operations. When running operations, the token usage will be printed. You can also fetch the token usage through `index.llm_predictor.last_token_usage`. See *Cost Predictor How-To* for more details.

[Optional] Save the index for future use

To save to disk and load from disk, do

```
# save to disk
index.save_to_disk('index.json')
# load from disk
index = GPTSimpleVectorIndex.load_from_disk('index.json')
```

NOTE: If you had initialized the index with a custom `ServiceContext` object, you will also need to pass in the same ServiceContext during `load_from_disk`.

```
service_context = ServiceContext.from_defaults(llm_predictor=llm_predictor)

# when first building the index
index = GPTSimpleVectorIndex.from_documents(documents, service_context=service_context)

...

# when loading the index from disk
index = GPTSimpleVectorIndex.load_from_disk("index.json", service_context=service_
↪context)
```

4. [Optional, Advanced] Building indices on top of other indices

You can build indices on top of other indices!

```
from llama_index import GPTSimpleVectorIndex, GPTListIndex

index1 = GPTSimpleVectorIndex.from_documents(documents1)
index2 = GPTSimpleVectorIndex.from_documents(documents2)

# Set summary text
# you can set the summary manually, or you can
# generate the summary itself using LlamaIndex
index1.set_text("summary1")
index2.set_text("summary2")

index3 = GPTListIndex([index1, index2])
```

Composability gives you greater power in indexing your heterogeneous sources of data. For a discussion on relevant use cases, see our *Query Use Cases*. For technical details and examples, see our *Composability How-To*.

5. Query the index.

After building the index, you can now query it. Note that a "query" is simply an input to an LLM - this means that you can use the index for question-answering, but you can also do more than that!

To start, you can query an index without specifying any additional keyword arguments, as follows:

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

response = index.query("Write an email to the user given their background information.")
print(response)
```

However, you also have a variety of keyword arguments at your disposal, depending on the type of index being used. A full treatment of all the index-dependent query keyword arguments can be found *here*.

Setting `mode`

An index can have a variety of query modes. For instance, you can choose to specify `mode="default"` or `mode="embedding"` for a list index. `mode="default"` will a create and refine an answer sequentially through the nodes of the list. `mode="embedding"` will synthesize an answer by fetching the top-k nodes by embedding similarity.

```
index = GPTListIndex.from_documents(documents)
# mode="default"
response = index.query("What did the author do growing up?", mode="default")
# mode="embedding"
response = index.query("What did the author do growing up?", mode="embedding")
```

The full set of modes per index are documented in the *Query Reference*.

Setting `response_mode`

Note: This option is not available/utilized in `GPTTreeIndex`.

An index can also have the following response modes through `response_mode`:

- `default`: For the given index, "create and refine" an answer by sequentially going through each Node; make a separate LLM call per Node. Good for more detailed answers.
- `compact`: For the given index, "compact" the prompt during each LLM call by stuffing as many Node text chunks that can fit within the maximum prompt size. If there are too many chunks to stuff in one prompt, "create and refine" an answer by going through multiple prompts.
- `tree_summarize`: Given a set of Nodes and the query, recursively construct a tree and return the root node as the response. Good for summarization purposes.

```
index = GPTListIndex.from_documents(documents)
# mode="default"
response = index.query("What did the author do growing up?", response_mode="default")
# mode="compact"
response = index.query("What did the author do growing up?", response_mode="compact")
# mode="tree_summarize"
response = index.query("What did the author do growing up?", response_mode="tree_
↪summarize")
```

Setting `required_keywords` and `exclude_keywords`

You can set `required_keywords` and `exclude_keywords` on most of our indices (the only exclusion is the GPTTreeIndex). This will preemptively filter out nodes that do not contain `required_keywords` or contain `exclude_keywords`, reducing the search space and hence time/number of LLM calls/cost.

```
index.query(
    "What did the author do after Y Combinator?", required_keywords=["Combinator"],
    exclude_keywords=["Italy"]
)
```

5. Parsing the response

The object returned is a *`Response` object*. The object contains both the response text as well as the "sources" of the response:

```
response = index.query("<query_str>")

# get response
# response.response
str(response)

# get sources
response.source_nodes
# formatted sources
response.get_formatted_sources()
```

Query Index

```
# try verbose=True for more detailed outputs
response = index.query("What did the author do growing up?", verbose=True)
```

```
display(Markdown(f"<b>{response}</b>"))
```

Growing up, the author wrote short stories, programmed on an IBM 1401, wrote simple games and a word processor on a TRS-80, studied philosophy in college, learned Lisp, reverse-engineered SHRDLU, wrote a book about Lisp hacking, took art classes at Harvard, and was disappointed by the lack of teaching and learning in the painting department at the Accademia.

Get Sources

```
print(response.get_formatted_sources())
```

```
>Source: 1782e65e-2b85-44bf-80e9-7198d273feb2:

What I Worked On

February 2021

Before college the two main things I worked on, outside of s...
```

An example is shown below.

3.3.2 How Each Index Works

This guide describes how each index works with diagrams. We also visually highlight our "Response Synthesis" modes.

Some terminology:

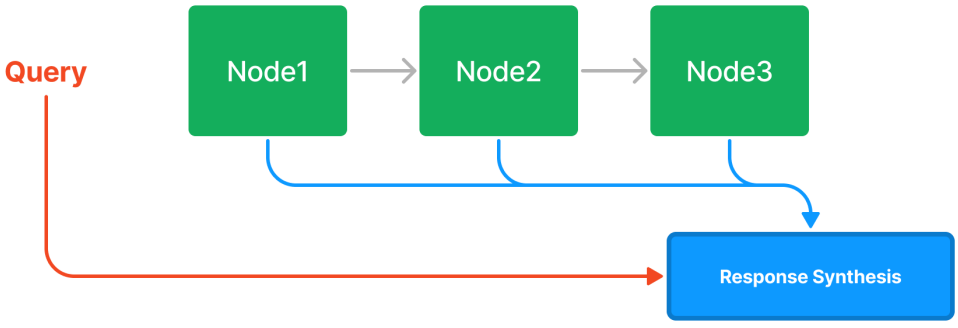
- **Node**: Corresponds to a chunk of text from a Document. LlamaIndex takes in Document objects and internally parses/chunks them into Node objects.
- **Response Synthesis**: Our module which synthesizes a response given the retrieved Node. You can see how to *specify different response modes* here. See below for an illustration of how each response mode works.

List Index

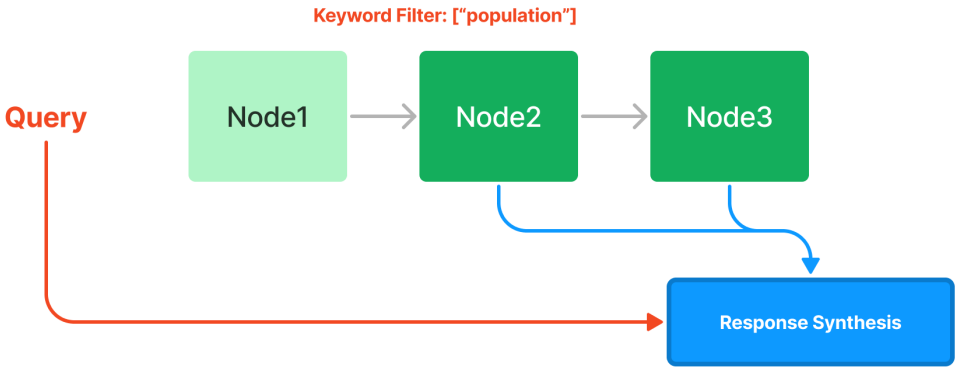
The list index simply stores Nodes as a sequential chain.

Querying

During query time, if no other query parameters are specified, LlamaIndex simply loads all Nodes in the list into our Response Synthesis module.

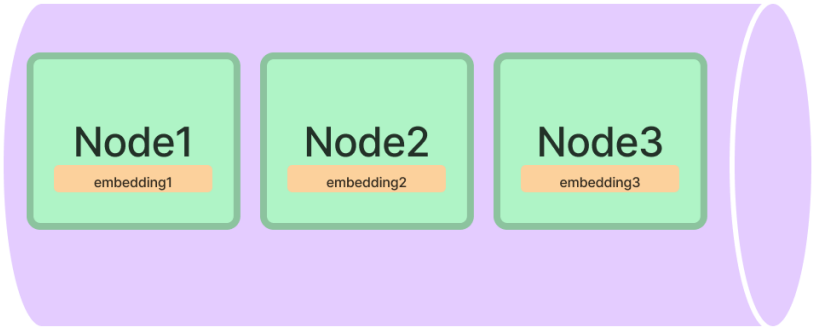


The list index does offer numerous ways of querying a list index, from an embedding-based query which will fetch the top-k neighbors, or with the addition of a keyword filter, as seen below:



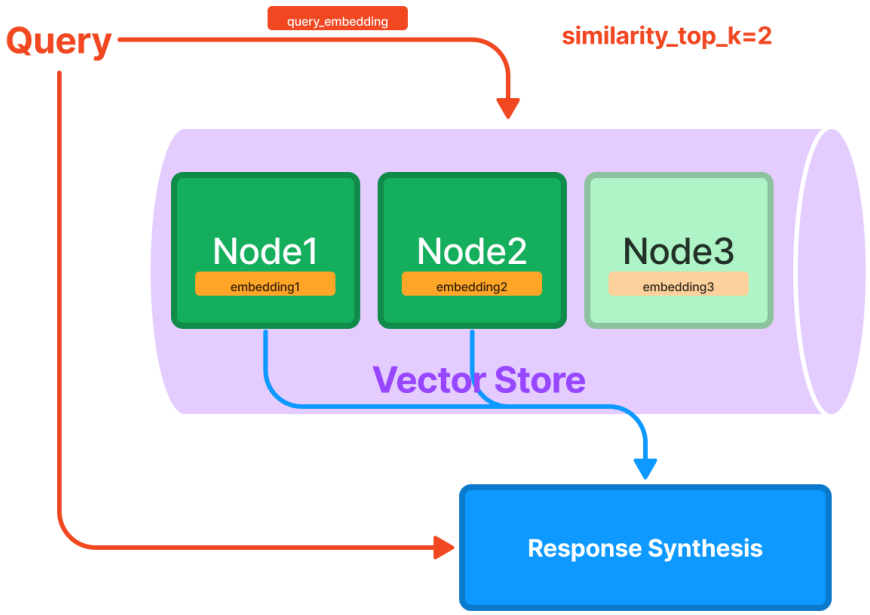
Vector Store Index

The vector store index stores each Node and a corresponding embedding in a *Vector Store*.



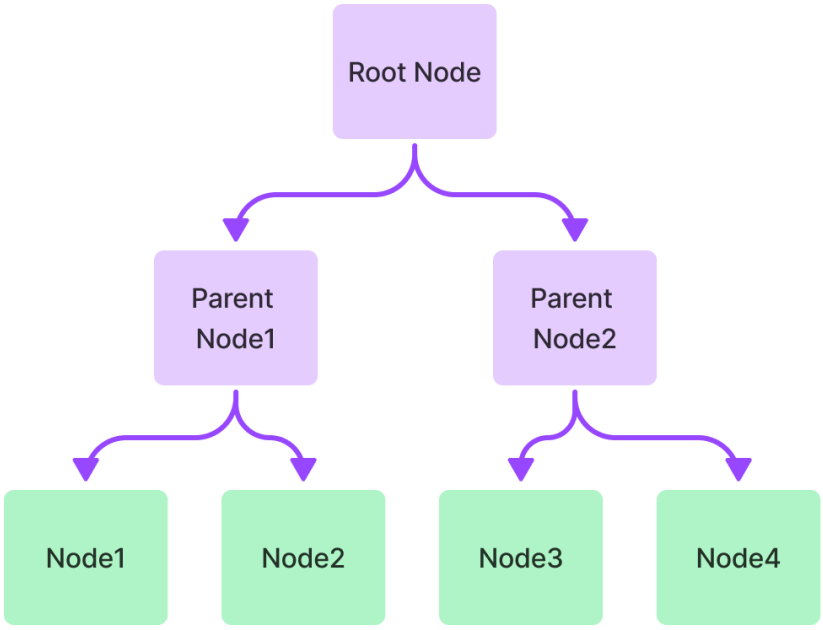
Querying

Querying a vector store index involves fetching the top-k most similar Nodes, and passing those into our Response Synthesis module.



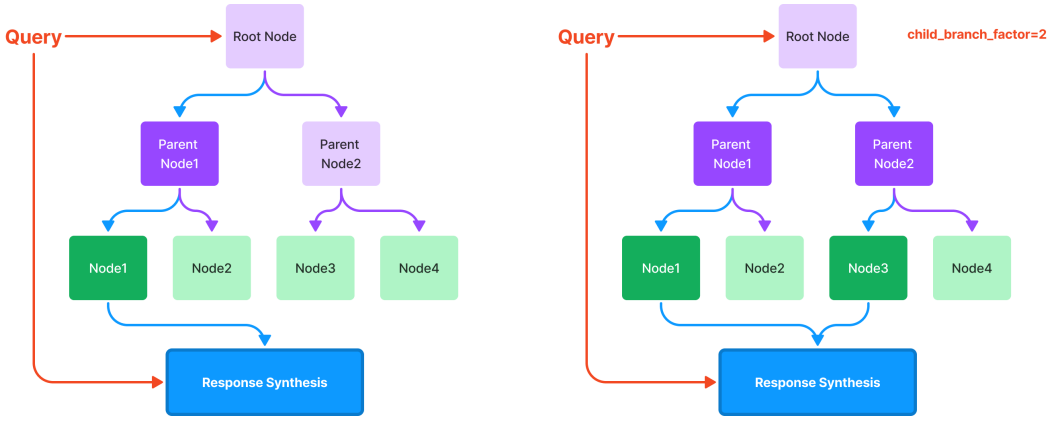
Tree Index

The tree index builds a hierarchical tree from a set of Nodes (which become leaf nodes in this tree).



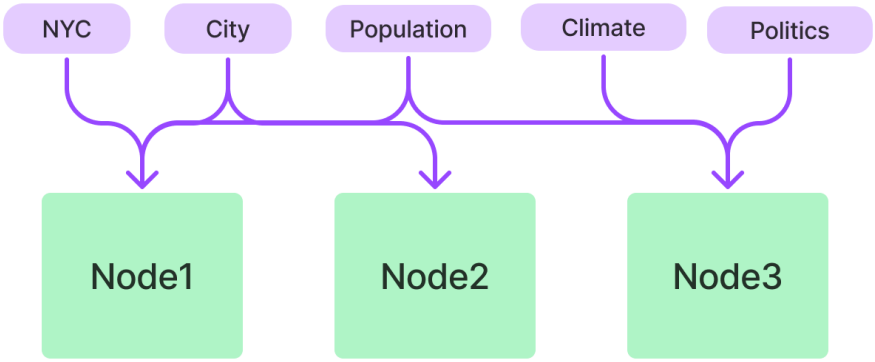
Querying

Querying a tree index involves traversing from root nodes down to leaf nodes. By default, (`child_branch_factor=1`), a query chooses one child node given a parent node. If `child_branch_factor=2`, a query chooses two child nodes per parent.



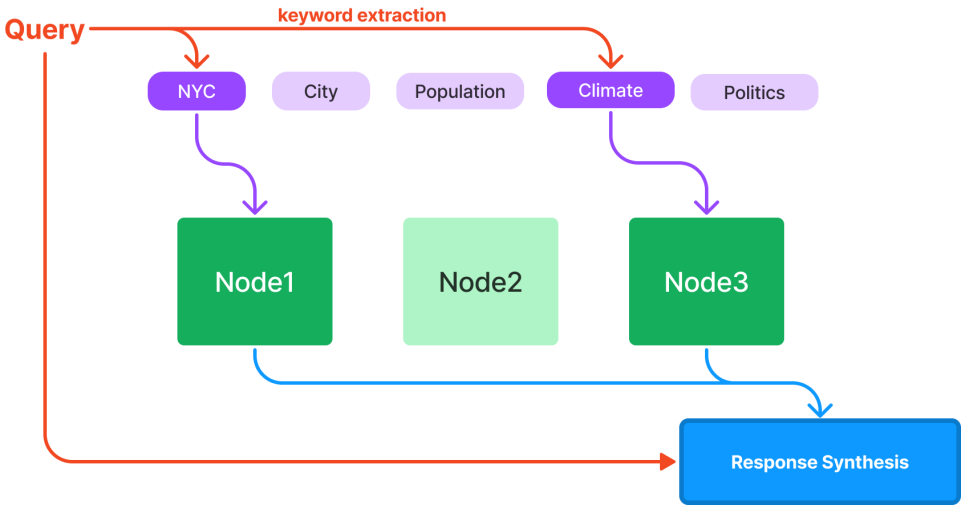
Keyword Table Index

The keyword table index extracts keywords from each Node and builds a mapping from each keyword to the corresponding Nodes of that keyword.



Querying

During query time, we extract relevant keywords from the query, and match those with pre-extracted Node keywords to fetch the corresponding Nodes. The extracted Nodes are passed to our Response Synthesis module.

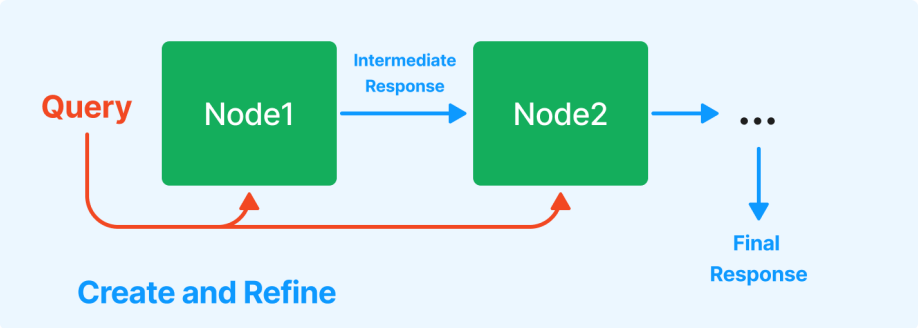


Response Synthesis

LlamaIndex offers different methods of synthesizing a response. The way to toggle this can be found in our *Usage Pattern Guide*. Below, we visually highlight how each response mode works.

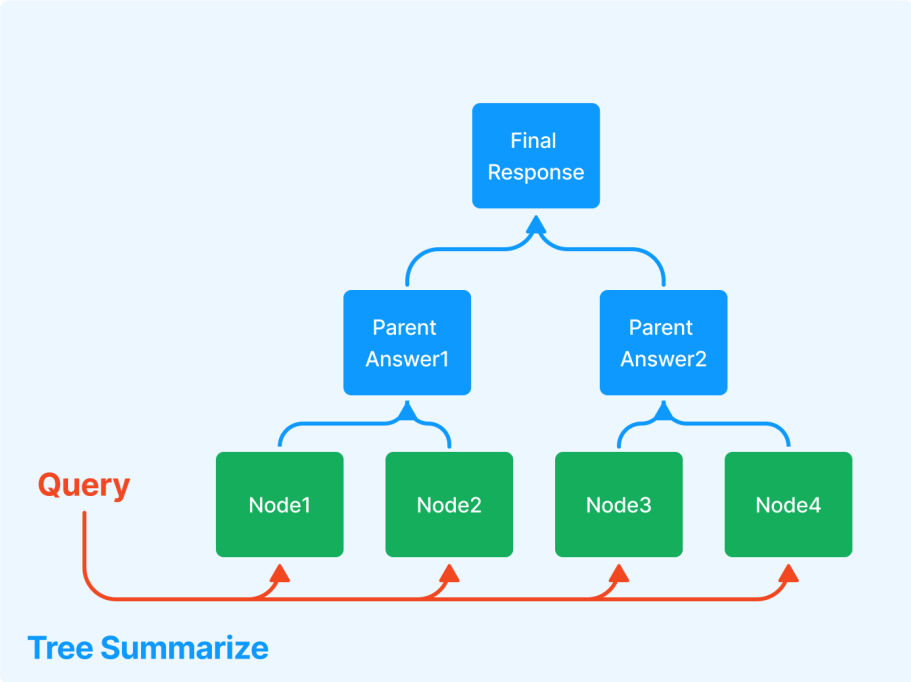
Create and Refine

Create and refine is an iterative way of generating a response. We first use the context in the first node, along with the query, to generate an initial answer. We then pass this answer, the query, and the context of the second node as input into a "refine prompt" to generate a refined answer. We refine through N-1 nodes, where N is the total number of nodes.



Tree Summarize

Tree summarize is another way of generating a response. We essentially build a tree index over the set of candidate nodes, with a *summary prompt* seeded with the query. The tree is built in a bottoms-up fashion, and in the end the root node is returned as the response.



3.4 Tutorials

This section contains a list of in-depth tutorials on how to best utilize different capabilities of LlamaIndex within your end-user application.

They include a broad range of LlamaIndex concepts:

- Semantic search
- Structured data support
- Composability/Query Transformation

They also showcase a variety of application settings that LlamaIndex can be used, from a simple Jupyter notebook to a chatbot to a full-stack web application.

3.4.1 How to Build a Chatbot

LlamaIndex is an interface between your data and LLM's; it offers the toolkit for you to setup a query interface around your data for any downstream task, whether it's question-answering, summarization, or more.

In this tutorial, we show you how to build a context augmented chatbot. We use Langchain for the underlying Agent/Chatbot abstractions, and we use LlamaIndex for the data retrieval/lookup/querying! The result is a chatbot agent that has access to a rich set of "data interface" Tools that LlamaIndex provides to answer queries over your data.

Note: This is a continuation of some initial work building a query interface over SEC 10-K filings - check it out here.

Context

In this tutorial, we build an "10-K Chatbot" by downloading the raw UBER 10-K HTML filings from Dropbox. The user can choose to ask questions regarding the 10-K filings.

Ingest Data

Let's first download the raw 10-k files, from 2019-2022.

```
# NOTE: the code examples assume you're operating within a Jupyter notebook.
# download files
!mkdir data
!wget "https://www.dropbox.com/s/948jr9cfs7fgj99/UBER.zip?dl=1" -O data/UBER.zip
!unzip data/UBER.zip -d data
```

We use the Unstructured library to parse the HTML files into formatted text. We have a direct integration with Unstructured through LlamaHub - this allows us to convert any text into a Document format that LlamaIndex can ingest.

```
from llama_index import download_loader, GPTSimpleVectorIndex, ServiceContext
from pathlib import Path

years = [2022, 2021, 2020, 2019]
UnstructuredReader = download_loader("UnstructuredReader", refresh_cache=True)

loader = UnstructuredReader()
doc_set = {}
all_docs = []
for year in years:
    year_docs = loader.load_data(file=Path(f'./data/UBER/UBER_{year}.html'), split_
↪documents=False)
    # insert year metadata into each year
    for d in year_docs:
        d.extra_info = {"year": year}
    doc_set[year] = year_docs
    all_docs.extend(year_docs)
```

Setting up Vector Indices for each year

We first setup a vector index for each year. Each vector index allows us to ask questions about the 10-K filing of a given year.

We build each index and save it to disk.

```
# initialize simple vector indices + global vector index
service_context = ServiceContext.from_defaults(chunk_size_limit=512)
index_set = {}
for year in years:
    cur_index = GPTSimpleVectorIndex.from_documents(doc_set[year], service_
↪context=service_context)
    index_set[year] = cur_index
    cur_index.save_to_disk(f'index_{year}.json')
```

To load an index from disk, do the following

```
# Load indices from disk
index_set = {}
for year in years:
    cur_index = GPTSimpleVectorIndex.load_from_disk(f'index_{year}.json')
    index_set[year] = cur_index
```

Composing a Graph to Synthesize Answers Across 10-K Filings

Since we have access to documents of 4 years, we may not only want to ask questions regarding the 10-K document of a given year, but ask questions that require analysis over all 10-K filings.

To address this, we compose a "graph" which consists of a list index defined over the 4 vector indices. Querying this graph would first retrieve information from each vector index, and combine information together via the list index.

```
from llama_index import GPTListIndex, LLMPredictor, ServiceContext
from langchain import OpenAI
from llama_index.indices.composability import ComposableGraph

# describe each index to help traversal of composed graph
index_summaries = [f"UBER 10-k Filing for {year} fiscal year" for year in years]

# define an LLMPredictor set number of output tokens
llm_predictor = LLMPredictor(llm=OpenAI(temperature=0, max_tokens=512))
service_context = ServiceContext.from_defaults(llm_predictor=llm_predictor)

# define a list index over the vector indices
# allows us to synthesize information across each index
graph = ComposableGraph.from_indices(
    GPTListIndex,
    [index_set[y] for y in years],
    index_summaries=index_summaries,
    service_context=service_context,
)

# [optional] save to disk
```

(continues on next page)

(continued from previous page)

```
graph.save_to_disk('10k_graph.json')
# [optional] load from disk, so you don't need to build graph from scratch
graph = ComposableGraph.load_from_disk(
    '10k_graph.json',
    service_context=service_context,
)
```

Setting up the Tools + Langchain Chatbot Agent

We use Langchain to setup the outer chatbot agent, which has access to a set of Tools. LlamaIndex provides some wrappers around indices and graphs so that they can be easily used within a Tool interface.

```
# do imports
from langchain.agents import Tool
from langchain.chains.conversation.memory import ConversationBufferMemory
from langchain.chat_models import ChatOpenAI
from langchain.agents import initialize_agent

from llama_index.langchain_helpers.agents import LlamaToolkit, create_llama_chat_agent,␣
↪IndexToolConfig, GraphToolConfig
```

We want to define a separate Tool for each index (corresponding to a given year), as well as the graph. We can define all tools under a central `LlamaToolkit` interface.

Below, we define a `GraphToolConfig` for our graph. Note that we also import a `DecomposeQueryTransform` module for use within each vector index within the graph - this allows us to "decompose" the overall query into a query that can be answered from each subindex. (see example below).

```
# define a decompose transform
from llama_index.indices.query.query_transform.base import DecomposeQueryTransform
decompose_transform = DecomposeQueryTransform(
    llm_predictor, verbose=True
)

# define query configs for graph
query_configs = [
    {
        "index_struct_type": "simple_dict",
        "query_mode": "default",
        "query_kwargs": {
            "similarity_top_k": 1,
            # "include_summary": True
        },
        "query_transform": decompose_transform
    },
    {
        "index_struct_type": "list",
        "query_mode": "default",
        "query_kwargs": {
            "response_mode": "tree_summarize",
            "verbose": True
```

(continues on next page)

(continued from previous page)

```
        }
    },
]
# graph config
graph_config = GraphToolConfig(
    graph=graph,
    name=f"Graph Index",
    description="useful for when you want to answer queries that require analyzing␣
↪multiple SEC 10-K documents for Uber.",
    query_configs=query_configs,
    tool_kwargs={"return_direct": True}
)
```

Besides the `GraphToolConfig` object, we also define an `IndexToolConfig` corresponding to each index:

```
# define toolkit
index_configs = []
for y in range(2019, 2023):
    tool_config = IndexToolConfig(
        index=index_set[y],
        name=f"Vector Index {y}",
        description=f"useful for when you want to answer queries about the {y} SEC 10-K␣
↪for Uber",
        index_query_kwargs={"similarity_top_k": 3},
        tool_kwargs={"return_direct": True}
    )
    index_configs.append(tool_config)
```

Finally, we combine these configs with our `LlamaToolkit`:

```
toolkit = LlamaToolkit(
    index_configs=index_configs,
    graph_configs=[graph_config]
)
```

Finally, we call `create_llama_chat_agent` to create our Langchain chatbot agent, which has access to the 5 Tools we defined above:

```
memory = ConversationBufferMemory(memory_key="chat_history")
llm=OpenAI(temperature=0)
agent_chain = create_llama_chat_agent(
    toolkit,
    llm,
    memory=memory,
    verbose=True
)
```

Testing the Agent

We can now test the agent with various queries.

If we test it with a simple "hello" query, the agent does not use any Tools.

```
agent_chain.run(input="hi, i am bob")
```

```
> Entering new AgentExecutor chain...

Thought: Do I need to use a tool? No
AI: Hi Bob, nice to meet you! How can I help you today?

> Finished chain.
'Hi Bob, nice to meet you! How can I help you today?'
```

If we test it with a query regarding the 10-k of a given year, the agent will use the relevant vector index Tool.

```
agent_chain.run(input="What were some of the biggest risk factors in 2020 for Uber?")
```

```
> Entering new AgentExecutor chain...

Thought: Do I need to use a tool? Yes
Action: Vector Index 2020
Action Input: Risk Factors
...

Observation:

Risk Factors

The COVID-19 pandemic and the impact of actions to mitigate the pandemic has adversely␣
↪affected and continues to adversely affect our business, financial condition, and␣
↪results of operations.

...
'\n\nRisk Factors\n\nThe COVID-19 pandemic and the impact of actions to mitigate the␣
↪pandemic has adversely affected and continues to adversely affect our business,
```

Finally, if we test it with a query to compare/contrast risk factors across years, the agent will use the graph index Tool.

```
cross_query_str = (
    "Compare/contrast the risk factors described in the Uber 10-K across years. Give␣
↪answer in bullet points."
)
agent_chain.run(input=cross_query_str)
```

```
> Entering new AgentExecutor chain...

Thought: Do I need to use a tool? Yes
Action: Graph Index
Action Input: Compare/contrast the risk factors described in the Uber 10-K across years.>
```

(continues on next page)

(continued from previous page)

```
↪ Current query: Compare/contrast the risk factors described in the Uber 10-K across␣
↪years.
> New query:  What are the risk factors described in the Uber 10-K for the 2022 fiscal␣
↪year?
> Current query: Compare/contrast the risk factors described in the Uber 10-K across␣
↪years.
> New query:  What are the risk factors described in the Uber 10-K for the 2022 fiscal␣
↪year?
INFO:llama_index.token_counter.token_counter:> [query] Total LLM token usage: 964 tokens
INFO:llama_index.token_counter.token_counter:> [query] Total embedding token usage: 18␣
↪tokens
> Got response:
The risk factors described in the Uber 10-K for the 2022 fiscal year include: the␣
↪potential for changes in the classification of Drivers, the potential for increased␣
↪competition, the potential for...
> Current query: Compare/contrast the risk factors described in the Uber 10-K across␣
↪years.
> New query:  What are the risk factors described in the Uber 10-K for the 2021 fiscal␣
↪year?
> Current query: Compare/contrast the risk factors described in the Uber 10-K across␣
↪years.
> New query:  What are the risk factors described in the Uber 10-K for the 2021 fiscal␣
↪year?
INFO:llama_index.token_counter.token_counter:> [query] Total LLM token usage: 590 tokens
INFO:llama_index.token_counter.token_counter:> [query] Total embedding token usage: 18␣
↪tokens
> Got response:
1. The COVID-19 pandemic and the impact of actions to mitigate the pandemic have␣
↪adversely affected and may continue to adversely affect parts of our business.

2. Our business would be adversely ...
> Current query: Compare/contrast the risk factors described in the Uber 10-K across␣
↪years.
> New query:  What are the risk factors described in the Uber 10-K for the 2020 fiscal␣
↪year?
> Current query: Compare/contrast the risk factors described in the Uber 10-K across␣
↪years.
> New query:  What are the risk factors described in the Uber 10-K for the 2020 fiscal␣
↪year?
INFO:llama_index.token_counter.token_counter:> [query] Total LLM token usage: 516 tokens
INFO:llama_index.token_counter.token_counter:> [query] Total embedding token usage: 18␣
↪tokens
> Got response:
The risk factors described in the Uber 10-K for the 2020 fiscal year include: the timing␣
↪of widespread adoption of vaccines against the virus, additional actions that may be␣
↪taken by governmental ...
> Current query: Compare/contrast the risk factors described in the Uber 10-K across␣
↪years.
> New query:  What are the risk factors described in the Uber 10-K for the 2019 fiscal␣
↪year?
> Current query: Compare/contrast the risk factors described in the Uber 10-K across␣
↪years.
```

(continues on next page)

(continued from previous page)

```
> New query:  What are the risk factors described in the Uber 10-K for the 2019 fiscal␣
↪year?
INFO:llama_index.token_counter.token_counter:> [query] Total LLM token usage: 1020 tokens
INFO:llama_index.token_counter.token_counter:> [query] Total embedding token usage: 18␣
↪tokens
INFO:llama_index.indices.common.tree.base:> Building index from nodes: 0 chunks
> Got response:
Risk factors described in the Uber 10-K for the 2019 fiscal year include: competition␣
↪from other transportation providers; the impact of government regulations; the impact␣
↪of litigation; the impac...
INFO:llama_index.token_counter.token_counter:> [query] Total LLM token usage: 7039 tokens
INFO:llama_index.token_counter.token_counter:> [query] Total embedding token usage: 72␣
↪tokens

Observation:
In 2020, the risk factors included the timing of widespread adoption of vaccines against␣
↪the virus, additional actions that may be taken by governmental authorities, the␣
↪further impact on the business of Drivers

...
```

Setting up the Chatbot Loop

Now that we have the chatbot setup, it only takes a few more steps to setup a basic interactive loop to converse with our SEC-augmented chatbot!

```
while True:
    text_input = input("User: ")
    response = agent_chain.run(input=text_input)
    print(f'Agent: {response}')
```

Here's an example of the loop in action:

```
User:  What were some of the legal proceedings against Uber in 2022?
Agent:

In 2022, legal proceedings against Uber include a motion to compel arbitration, an␣
↪appeal of a ruling that Proposition 22 is unconstitutional, a complaint alleging that␣
↪drivers are employees and entitled to protections under the wage and labor laws, a␣
↪summary judgment motion, allegations of misclassification of drivers and related␣
↪employment violations in New York, fraud related to certain deductions, class actions␣
↪in Australia alleging that Uber entities conspired to injure the group members during␣
↪the period 2014 to 2017 by either directly breaching transport legislation or␣
↪commissioning offenses against transport legislation by UberX Drivers in Australia,␣
↪and claims of lost income and decreased value of certain taxi. Additionally, Uber is␣
↪facing a challenge in California Superior Court alleging that Proposition 22 is␣
↪unconstitutional, and a preliminary injunction order prohibiting Uber from classifying␣
↪Drivers as independent contractors and from violating various wage and hour laws.
```

(continues on next page)

(continued from previous page)

```
User:
```

Notebook

Take a look at our corresponding notebook.

3.4.2 A Guide to Building a Full-Stack Web App with LLamaIndex

LlamaIndex is a python library, which means that integrating it with a full-stack web application will be a little different than what you might be used to.

This guide seeks to walk through the steps needed to create a basic API service written in python, and how this interacts with a TypeScript+React frontend.

All code examples here are available from the llama_index_starter_pack in the flask_react folder.

The main technologies used in this guide are as follows:

- python3.11
- llama_index
- flask
- typescript
- react

Flask Backend

For this guide, our backend will use a Flask API server to communicate with our frontend code. If you prefer, you can also easily translate this to a FastAPI server, or any other python server library of your choice.

Setting up a server using Flask is easy. You import the package, create the app object, and then create your endpoints. Let's create a basic skeleton for the server first:

```python
from flask import Flask

app = Flask(__name__)

@app.route("/")
def home():
    return "Hello World!"

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=5601)
```

flask_demo.py

If you run this file (`python flask_demo.py`), it will launch a server on port 5601. If you visit `http://localhost:5601/`, you will see the "Hello World!" text rendered in your browser. Nice!

The next step is deciding what functions we want to include in our server, and to start using LlamaIndex.

To keep things simple, the most basic operation we can provide is querying an existing index. Using the paul graham essay from LlamaIndex, create a documents folder and download+place the essay text file inside of it.

Basic Flask - Handling User Index Queries

Now, let's write some code to initialize our index:

```
import os
from llama_index import SimpleDirectoryReader, GPTSimpleVectorIndex

# NOTE: for local testing only, do NOT deploy with your key hardcoded
os.environ['OPENAI_API_KEY'] = "your key here"

index = None

def initialize_index():
  global index
   if os.path.exists(index_name):
      index = GPTSimpleVectorIndex.load_from_disk(index_name)
  else:
      documents = SimpleDirectoryReader("./documents").load_data()
      index = GPTSimpleVectorIndex.from_documents(documents)
      index.save_to_disk(index_name)
```

This function will initialize our index. If we call this just before starting the flask server in the `main` function, then our index will be ready for user queries!

Our query endpoint will accept `GET` requests with the query text as a parameter. Here's what the full endpoint function will look like:

```
from flask import request

@app.route("/query", methods=["GET"])
def query_index():
  global index
  query_text = request.args.get("text", None)
  if query_text is None:
    return "No text found, please include a ?text=blah parameter in the URL", 400
  response = index.query(query_text)
  return str(response), 200
```

Now, we've introduced a few new concepts to our server:

- a new `/query` endpoint, defined by the function decorator
- a new import from flask, `request`, which is used to get parameters from the request
- if the `text` parameter is missing, then we return an error message and an appropriate HTML response code
- otherwise, we query the index, and return the response as a string

A full query example that you can test in your browser might look something like this: `http://localhost:5601/query?text=what did the author do growing up` (once you press enter, the browser will convert the spaces into "%20" characters).

Things are looking pretty good! We now have a functional API. Using your own documents, you can easily provide an interface for any application to call the flask API and get answers to queries.

Advanced Flask - Handling User Document Uploads

Things are looking pretty cool, but how can we take this a step further? What if we want to allow users to build their own indexes by uploading their own documents? Have no fear, Flask can handle it all :muscle:.

To let users upload documents, we have to take some extra precautions. Instead of querying an existing index, the index will become **mutable**. If you have many users adding to the same index, we need to think about how to handle concurrency. Our Flask server is threaded, which means multiple users can ping the server with requests which will be handled at the same time.

One option might be to create an index for each user or group, and store and fetch things from S3. But for this example, we will assume there is one locally stored index that users are interacting with.

To handle concurrent uploads and ensure sequential inserts into the index, we can use the `BaseManager` python package to provide sequential access to the index using a separate server and locks. This sounds scary, but it's not so bad! We will just move all our index operations (initializing, querying, inserting) into the `BaseManager` "index_server", which will be called from our Flask server.

Here's a basic example of what our `index_server.py` will look like after we've moved our code:

```
import os
from multiprocessing import Lock
from multiprocessing.managers import BaseManager
from llama_index import SimpleDirectoryReader, GPTSimpleVectorIndex, Document

# NOTE: for local testing only, do NOT deploy with your key hardcoded
os.environ['OPENAI_API_KEY'] = "your key here"

index = None
lock = Lock()

def initialize_index():
  global index

  with lock:
    # same as before ...
  ...

def query_index(query_text):
  global index
  response = index.query(query_text)
  return str(response)

if __name__ == "__main__":
    # init the global index
    print("initializing index...")
    initialize_index()

    # setup server
    # NOTE: you might want to handle the password in a less hardcoded way
    manager = BaseManager(('', 5602), b'password')
    manager.register('query_index', query_index)
    server = manager.get_server()

    print("starting server...")
```

(continues on next page)

(continued from previous page)

```
    server.serve_forever()
```

index_server.py

So, we've moved our functions, introduced the `Lock` object which ensures sequential access to the global index, registered our single function in the server, and started the server on port 5602 with the password `password`.

Then, we can adjust our flask code as follows:

```
from multiprocessing.managers import BaseManager
from flask import Flask, request

# initialize manager connection
# NOTE: you might want to handle the password in a less hardcoded way
manager = BaseManager(('', 5602), b'password')
manager.register('query_index')
manager.connect()

@app.route("/query", methods=["GET"])
def query_index():
  global index
  query_text = request.args.get("text", None)
  if query_text is None:
    return "No text found, please include a ?text=blah parameter in the URL", 400
  response = manager.query_index(query_text)._getvalue()
  return str(response), 200

@app.route("/")
def home():
    return "Hello World!"

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=5601)
```

flask_demo.py

The two main changes are connecting to our existing `BaseManager` server and registering the functions, as well as calling the function through the manager in the `/query` endpoint.

One special thing to note is that `BaseManager` servers don't return objects quite as we expect. To resolve the return value into it's original object, we call the `_getvalue()` function.

If we allow users to upload their own documents, we should probably remove the Paul Graham essay from the documents folder, so let's do that first. Then, let's add an endpoint to upload files! First, let's define our Flask endpoint function:

```
...
manager.register('insert_into_index')
...

@app.route("/uploadFile", methods=["POST"])
def upload_file():
    global manager
    if 'file' not in request.files:
        return "Please send a POST request with a file", 400
```

(continues on next page)

(continued from previous page)

```
    filepath = None
    try:
        uploaded_file = request.files["file"]
        filename = secure_filename(uploaded_file.filename)
        filepath = os.path.join('documents', os.path.basename(filename))
        uploaded_file.save(filepath)

        if request.form.get("filename_as_doc_id", None) is not None:
            manager.insert_into_index(filepath, doc_id=filename)
        else:
            manager.insert_into_index(filepath)
    except Exception as e:
        # cleanup temp file
        if filepath is not None and os.path.exists(filepath):
            os.remove(filepath)
        return "Error: {}".format(str(e)), 500

    # cleanup temp file
    if filepath is not None and os.path.exists(filepath):
        os.remove(filepath)

    return "File inserted!", 200
```

Not too bad! You will notice that we write the file to disk. We could skip this if we only accept basic file formats like `txt` files, but written to disk we can take advantage of LlamaIndex's `SimpleDirectoryReader` to take care of a bunch of more complex file formats. Optionally, we also use a second `POST` argument to either use the filename as a doc_id or let LlamaIndex generate one for us. This will make more sense once we implement the frontend.

With these more complicated requests, I also suggest using a tool like Postman. Examples of using postman to test our endpoints are in the repository for this project.

Lastly, you'll notice we added a new function to the manager. Let's implement that inside `index_server.py`:

```
def insert_into_index(doc_text, doc_id=None):
    global index
    document = SimpleDirectoryReader(input_files=[doc_text]).load_data()[0]
    if doc_id is not None:
        document.doc_id = doc_id

    with lock:
        index.insert(document)
        index.save_to_disk(index_name)

...
manager.register('insert_into_index', insert_into_index)
...
```

Easy! If we launch both the `index_server.py` and then the `flask_demo.py` python files, we have a Flask API server that can handle multiple requests to insert documents into a vector index and respond to user queries!

To support some functionality in the frontend, I've adjusted what some responses look like from the Flask API, as well as added some functionality to keep track of which documents are stored in the index (LlamaIndex doesn't currently support this in a user-friendly way, but we can augment it ourselves!). Lastly, I had to add CORS support to the server

using the `Flask-cors` python package.

Check out the complete `flask_demo.py` and `index_server.py` scripts in the repository for the final minor changes, the`requirements.txt` file, and a sample `Dockerfile` to help with deployment.

React Frontend

Generally, React and Typescript are one of the most popular libraries and languages for writing webapps today. This guide will assume you are familiar with how these tools work, because otherwise this guide will triple in length :smile:.

In the repository, the frontend code is organized inside of the `react_frontend` folder.

The most relevant part of the frontend will be the `src/apis` folder. This is where we make calls to the Flask server, supporting the following queries:

- `/query` – make a query to the existing index
- `/uploadFile` – upload a file to the flask server for insertion into the index
- `/getDocuments` – list the current document titles and a portion of their texts

Using these three queries, we can build a robust frontend that allows users to upload and keep track of their files, query the index, and view the query response and information about which text nodes were used to form the response.

fetchDocuments.tsx

This file contains the function to, you guessed it, fetch the list of current documents in the index. The code is as follows:

```
export type Document = {
  id: string;
  text: string;
};

const fetchDocuments = async (): Promise<Document[]> => {
  const response = await fetch("http://localhost:5601/getDocuments", {
    mode: "cors",
  });

  if (!response.ok) {
    return [];
  }

  const documentList = (await response.json()) as Document[];
  return documentList;
};
```

As you can see, we make a query to the Flask server (here, it assumes running on localhost). Notice that we need to include the `mode: 'cors'` option, as we are making an external request.

Then, we check if the response was ok, and if so, get the response json and return it. Here, the response json is a list of `Document` objects that are defined in the same file.

queryIndex.tsx

This file sends the user query to the flask server, and gets the response back, as well as details about which nodes in our index provided the response.

```
export type ResponseSources = {
  text: string;
  doc_id: string;
  start: number;
  end: number;
  similarity: number;
};

export type QueryResponse = {
  text: string;
  sources: ResponseSources[];
};

const queryIndex = async (query: string): Promise<QueryResponse> => {
  const queryURL = new URL("http://localhost:5601/query?text=1");
  queryURL.searchParams.append("text", query);

  const response = await fetch(queryURL, { mode: "cors" });
  if (!response.ok) {
    return { text: "Error in query", sources: [] };
  }

  const queryResponse = (await response.json()) as QueryResponse;

  return queryResponse;
};

export default queryIndex;
```

This is similar to the `fetchDocuments.tsx` file, with the main difference being we include the query text as a parameter in the URL. Then, we check if the response is ok and return it with the appropriate typescript type.

insertDocument.tsx

Probably the most complex API call is uploading a document. The function here accepts a file object and constructs a `POST` request using `FormData`.

The actual response text is not used in the app but could be utilized to provide some user feedback on if the file failed to upload or not.

```
const insertDocument = async (file: File) => {
  const formData = new FormData();
  formData.append("file", file);
  formData.append("filename_as_doc_id", "true");

  const response = await fetch("http://localhost:5601/uploadFile", {
    mode: "cors",
    method: "POST",
```

(continues on next page)

(continued from previous page)

```
    body: formData,
  });

  const responseText = response.text();
  return responseText;
};

export default insertDocument;
```

All the Other Frontend Good-ness

And that pretty much wraps up the frontend portion! The rest of the react frontend code is some pretty basic react components, and my best attempt to make it look at least a little nice :smile:.

I encourage to read the rest of the codebase and submit any PRs for improvements!

Conclusion

This guide has covered a ton of information. We went from a basic "Hello World" Flask server written in python, to a fully functioning LlamaIndex powered backend and how to connect that to a frontend application.

As you can see, we can easily augment and wrap the services provided by LlamaIndex (like the little external document tracker) to help provide a good user experience on the frontend.

You could take this and add many features (multi-index/user support, saving objects into S3, adding a Pinecone vector server, etc.). And when you build an app after reading this, be sure to share the final result in the Discord! Good Luck! :muscle:

3.4.3 A Guide to Building a Full-Stack LlamaIndex Web App with Delphic

This guide seeks to walk you through using LlamaIndex with a production-ready web app starter template called Delphic. All code examples here are available from the Delphic repo

What We're Building

Here's a quick demo of the out-of-the-box functionality of Delphic:

https://user-images.githubusercontent.com/5049984/233236432-aa4980b6-a510-42f3-887a-81485c9644e6.mp4

Architectural Overview

Delphic leverages the LlamaIndex python library to let users to create their own document collections they can then query in a responsive frontend.

We chose a stack that provides a responsive, robust mix of technologies that can (1) orchestrate complex python processing tasks while providing (2) a modern, responsive frontend and (3) a secure backend to build additional functionality upon.

The core libraries are:

1. Django

2. Django Channels
3. Django Ninja
4. Redis
5. Celery
6. LlamaIndex
7. Langchain
8. React
9. Docker & Docker Compose

Thanks to this modern stack built on the super stable Django web framework, the starter Delphic app boasts a streamlined developer experience, built-in authentication and user management, asynchronous vector store processing, and web-socket-based query connections for a responsive UI. In addition, our frontend is built with TypeScript and is based on MUI React for a responsive and modern user interface.

System Requirements

Celery doesn't work on Windows. It may be deployable with Windows Subsystem for Linux, but configuring that is beyond the scope of this tutorial. For this reason, we recommend you only follow this tutorial if you're running Linux or OSX. You will need Docker and Docker Compose installed to deploy the application. Local development will require node version manager (nvm).

Django Backend

Project Directory Overview

The Delphic application has a structured backend directory organization that follows common Django project conventions. From the repo root, in the `./delphic` subfolder, the main folders are:

1. `contrib`: This directory contains custom modifications or additions to Django's built-in `contrib` apps.
2. `indexes`: This directory contains the core functionality related to document indexing and LLM integration. It includes:

- `admin.py`: Django admin configuration for the app
- `apps.py`: Application configuration
- `models.py`: Contains the app's database models
- `migrations`: Directory containing database schema migrations for the app
- `signals.py`: Defines any signals for the app
- `tests.py`: Unit tests for the app

3. `tasks`: This directory contains tasks for asynchronous processing using Celery. The `index_tasks.py` file includes the tasks for creating vector indexes.
4. `users`: This directory is dedicated to user management, including:
5. `utils`: This directory contains utility modules and functions that are used across the application, such as custom storage backends, path helpers, and collection-related utilities.

Database Models

The Delphic application has two core models: `Document` and `Collection`. These models represent the central entities the application deals with when indexing and querying documents using LLMs. They're defined in `./delphic/indexes/models.py`.

1. `Collection`:

- `api_key`: A foreign key that links a collection to an API key. This helps associate jobs with the source API key.
- `title`: A character field that provides a title for the collection.
- `description`: A text field that provides a description of the collection.
- `status`: A character field that stores the processing status of the collection, utilizing the `CollectionStatus` enumeration.
- `created`: A datetime field that records when the collection was created.
- `modified`: A datetime field that records the last modification time of the collection.
- `model`: A file field that stores the model associated with the collection.
- `processing`: A boolean field that indicates if the collection is currently being processed.

2. `Document`:

- `collection`: A foreign key that links a document to a collection. This represents the relationship between documents and collections.
- `file`: A file field that stores the uploaded document file.
- `description`: A text field that provides a description of the document.
- `created`: A datetime field that records when the document was created.
- `modified`: A datetime field that records the last modification time of the document.

These models provide a solid foundation for collections of documents and the indexes created from them with LlamaIndex.

Django Ninja API

Django Ninja is a web framework for building APIs with Django and Python 3.7+ type hints. It provides a simple, intuitive, and expressive way of defining API endpoints, leveraging Python's type hints to automatically generate input validation, serialization, and documentation.

In the Delphic repo, the `./config/api/endpoints.py` file contains the API routes and logic for the API endpoints. Now, let's briefly address the purpose of each endpoint in the `endpoints.py` file:

1. `/heartbeat`: A simple GET endpoint to check if the API is up and running. Returns `True` if the API is accessible. This is helpful for Kubernetes setups that expect to be able to query your container to ensure it's up and running.
2. `/collections/create`: A POST endpoint to create a new `Collection`. Accepts form parameters such as `title`, `description`, and a list of `files`. Creates a new `Collection` and `Document` instances for each file, and schedules a Celery task to create an index.

```
@collections_router.post("/create")
async def create_collection(request,
                            title: str = Form(...),
```

(continues on next page)

(continued from previous page)

```
                            description: str = Form(...),
                            files: list[UploadedFile] = File(...), ):
    key = None if getattr(request, "auth", None) is None else request.auth
    if key is not None:
        key = await key

    collection_instance = Collection(
        api_key=key,
        title=title,
        description=description,
        status=CollectionStatusEnum.QUEUED,
    )

    await sync_to_async(collection_instance.save)()

    for uploaded_file in files:
        doc_data = uploaded_file.file.read()
        doc_file = ContentFile(doc_data, uploaded_file.name)
        document = Document(collection=collection_instance, file=doc_file)
        await sync_to_async(document.save)()

    create_index.si(collection_instance.id).apply_async()

    return await sync_to_async(CollectionModelSchema)(
        ...
    )
```

3. `/collections/query` — a POST endpoint to query a document collection using the LLM. Accepts a JSON payload containing `collection_id` and `query_str`, and returns a response generated by querying the collection. We don't actually use this endpoint in our chat GUI (We use a websocket - see below), but you could build an app to integrate to this REST endpoint to query a specific collection.

```
@collections_router.post("/query",
                         response=CollectionQueryOutput,
                         summary="Ask a question of a document collection", )
def query_collection_view(request: HttpRequest, query_input: CollectionQueryInput):
    collection_id = query_input.collection_id
    query_str = query_input.query_str
    response = query_collection(collection_id, query_str)
    return {"response": response}
```

4. `/collections/available`: A GET endpoint that returns a list of all collections created with the user's API key. The output is serialized using the `CollectionModelSchema`.

```
@collections_router.get("/available",
                        response=list[CollectionModelSchema],
                        summary="Get a list of all of the collections created with my␣
↪api_key", )
async def get_my_collections_view(request: HttpRequest):
    key = None if getattr(request, "auth", None) is None else request.auth
    if key is not None:
        key = await key
```

(continues on next page)

(continued from previous page)

```
    collections = Collection.objects.filter(api_key=key)

    return [
        {
            ...
        }
        async for collection in collections
    ]
```

5. `/collections/{collection_id}/add_file`: A POST endpoint to add a file to an existing collection. Accepts a `collection_id` path parameter, and form parameters such as `file` and `description`. Adds the file as a `Document` instance associated with the specified collection.

```
@collections_router.post("/{collection_id}/add_file", summary="Add a file to a collection
↪")
async def add_file_to_collection(request,
                                 collection_id: int,
                                 file: UploadedFile = File(...),
                                 description: str = Form(...), ):
    collection = await sync_to_async(Collection.objects.get)(id=collection_id
```

Intro to Websockets

WebSockets are a communication protocol that enables bidirectional and full-duplex communication between a client and a server over a single, long-lived connection. The WebSocket protocol is designed to work over the same ports as HTTP and HTTPS (ports 80 and 443, respectively) and uses a similar handshake process to establish a connection. Once the connection is established, data can be sent in both directions as "frames" without the need to reestablish the connection each time, unlike traditional HTTP requests.

There are several reasons to use WebSockets, particularly when working with code that takes a long time to load into memory but is quick to run once loaded:

1. **Performance**: WebSockets eliminate the overhead associated with opening and closing multiple connections for each request, reducing latency.
2. **Efficiency**: WebSockets allow for real-time communication without the need for polling, resulting in more efficient use of resources and better responsiveness.
3. **Scalability**: WebSockets can handle a large number of simultaneous connections, making it ideal for applications that require high concurrency.

In the case of the Delphic application, using WebSockets makes sense as the LLMs can be expensive to load into memory. By establishing a WebSocket connection, the LLM can remain loaded in memory, allowing subsequent requests to be processed quickly without the need to reload the model each time.

The ASGI configuration file `./config/asgi.py` defines how the application should handle incoming connections, using the Django Channels `ProtocolTypeRouter` to route connections based on their protocol type. In this case, we have two protocol types: "http" and "websocket".

The "http" protocol type uses the standard Django ASGI application to handle HTTP requests, while the "websocket" protocol type uses a custom `TokenAuthMiddleware` to authenticate WebSocket connections. The `URLRouter` within the `TokenAuthMiddleware` defines a URL pattern for the `CollectionQueryConsumer`, which is responsible for handling WebSocket connections related to querying document collections.

```
application = ProtocolTypeRouter(
    {
        "http": get_asgi_application(),
        "websocket": TokenAuthMiddleware(
            URLRouter(
                [
                    re_path(
                        r"ws/collections/(?P<collection_id>\w+)/query/$",
                        CollectionQueryConsumer.as_asgi(),
                    ),
                ]
            )
        ),
    }
)
```

This configuration allows clients to establish WebSocket connections with the Delphic application to efficiently query document collections using the LLMs, without the need to reload the models for each request.

Websocket Handler

The `CollectionQueryConsumer` class in `config/api/websockets/queries.py` is responsible for handling WebSocket connections related to querying document collections. It inherits from the `AsyncWebsocketConsumer` class provided by Django Channels.

The `CollectionQueryConsumer` class has three main methods:

1. `connect`: Called when a WebSocket is handshaking as part of the connection process.
2. `disconnect`: Called when a WebSocket closes for any reason.
3. `receive`: Called when the server receives a message from the WebSocket.

Websocket connect listener

The `connect` method is responsible for establishing the connection, extracting the collection ID from the connection path, loading the collection model, and accepting the connection.

```
async def connect(self):
    try:
        self.collection_id = extract_connection_id(self.scope["path"])
        self.index = await load_collection_model(self.collection_id)
        await self.accept()

except ValueError as e:
await self.accept()
await self.close(code=4000)
except Exception as e:
pass
```

Websocket disconnect listener

The `disconnect` method is empty in this case, as there are no additional actions to be taken when the WebSocket is closed.

Websocket receive listener

The `receive` method is responsible for processing incoming messages from the WebSocket. It takes the incoming message, decodes it, and then queries the loaded collection model using the provided query. The response is then formatted as a markdown string and sent back to the client over the WebSocket connection.

```
async def receive(self, text_data):
    text_data_json = json.loads(text_data)

    if self.index is not None:
        query_str = text_data_json["query"]
        modified_query_str = f"Please return a nicely formatted markdown string to this␣
↪request:\n\n{query_str}"
        response = self.index.query(modified_query_str)

        markdown_response = f"## Response\n\n{response}\n\n"
        if response.source_nodes:
            markdown_sources = f"## Sources\n\n{response.get_formatted_sources()}"
        else:
            markdown_sources = ""

        formatted_response = f"{markdown_response}{markdown_sources}"

        await self.send(json.dumps({"response": formatted_response}, indent=4))
    else:
        await self.send(json.dumps({"error": "No index loaded for this connection."},␣
↪indent=4))
```

To load the collection model, the `load_collection_model` function is used, which can be found in `delphic/utils/collections.py`. This function retrieves the collection object with the given collection ID, checks if a JSON file for the collection model exists, and if not, creates one. Then, it sets up the `LLMPredictor` and `ServiceContext` before loading the `GPTSimpleVectorIndex` using the cache file.

```
async def load_collection_model(collection_id: str | int) -> GPTSimpleVectorIndex:
    """
    Load the Collection model from cache or the database, and return the index.

    Args:
        collection_id (Union[str, int]): The ID of the Collection model instance.

    Returns:
        GPTSimpleVectorIndex: The loaded index.

    This function performs the following steps:
    1. Retrieve the Collection object with the given collection_id.
    2. Check if a JSON file with the name '/cache/model_{collection_id}.json' exists.
    3. If the JSON file doesn't exist, load the JSON from the Collection.model FileField␣
```

(continues on next page)

(continued from previous page)

```
↪and save it to
        '/cache/model_{collection_id}.json'.
    4. Call GPTSimpleVectorIndex.load_from_disk with the cache_file_path.
    """
    # Retrieve the Collection object
    collection = await Collection.objects.aget(id=collection_id)
    logger.info(f"load_collection_model() - loaded collection {collection_id}")

    # Make sure there's a model
    if collection.model.name:
        logger.info("load_collection_model() - Setup local json index file")

        # Check if the JSON file exists
        cache_dir = Path(settings.BASE_DIR) / "cache"
        cache_file_path = cache_dir / f"model_{collection_id}.json"
        if not cache_file_path.exists():
            cache_dir.mkdir(parents=True, exist_ok=True)
            with collection.model.open("rb") as model_file:
                with cache_file_path.open("w+", encoding="utf-8") as cache_file:
                    cache_file.write(model_file.read().decode("utf-8"))

        # define LLM
        logger.info(
            f"load_collection_model() - Setup service context with tokens {settings.MAX_
↪TOKENS} and "
            f"model {settings.MODEL_NAME}"
        )
        llm_predictor = LLMPredictor(
            llm=OpenAI(temperature=0, model_name="text-davinci-003", max_tokens=512)
        )
        service_context = ServiceContext.from_defaults(llm_predictor=llm_predictor)

        # Call GPTSimpleVectorIndex.load_from_disk
        logger.info("load_collection_model() - Load llama index")
        index = GPTSimpleVectorIndex.load_from_disk(
            cache_file_path, service_context=service_context
        )
        logger.info(
            "load_collection_model() - Llamaindex loaded and ready for query..."
        )

    else:
        logger.error(
            f"load_collection_model() - collection {collection_id} has no model!"
        )
        raise ValueError("No model exists for this collection!")

    return index
```

React Frontend

Overview

We chose to use TypeScript, React and Material-UI (MUI) for the Delphic project's frontend for a couple reasons. First, as the most popular component library (MUI) for the most popular frontend framework (React), this choice makes this project accessible to a huge community of developers. Second, React is, at this point, a stable and generally well-liked framework that delivers valuable abstractions in the form of its virtual DOM while still being relatively stable and, in our opinion, pretty easy to learn, again making it accessible.

Frontend Project Structure

The frontend can be found in the `/frontend` directory of the repo, with the React-related components being in `/frontend/src` . You'll notice there is a DockerFile in the `frontend` directory and several folders and files related to configuring our frontend web server — nginx.

The `/frontend/src/App.tsx` file serves as the entry point of the application. It defines the main components, such as the login form, the drawer layout, and the collection create modal. The main components are conditionally rendered based on whether the user is logged in and has an authentication token.

The DrawerLayout2 component is defined in the`DrawerLayour2.tsx` file. This component manages the layout of the application and provides the navigation and main content areas.

Since the application is relatively simple, we can get away with not using a complex state management solution like Redux and just use React's useState hooks.

Grabbing Collections from the Backend

The collections available to the logged-in user are retrieved and displayed in the DrawerLayout2 component. The process can be broken down into the following steps:

1. Initializing state variables:

```
const[collections, setCollections] = useState < CollectionModelSchema[] > ([]);
const[loading, setLoading] = useState(true);
```

Here, we initialize two state variables: `collections` to store the list of collections and `loading` to track whether the collections are being fetched.

2. Collections are fetched for the logged-in user with the `fetchCollections()` function:

```
const
fetchCollections = async () = > {
try {
const accessToken = localStorage.getItem("accessToken");
if (accessToken) {
const response = await getMyCollections(accessToken);
setCollections(response.data);
}
} catch (error) {
console.error(error);
} finally {
setLoading(false);
```

(continues on next page)

(continued from previous page)

```
}
};
```

The `fetchCollections` function retrieves the collections for the logged-in user by calling the `getMyCollections` API function with the user's access token. It then updates the `collections` state with the retrieved data and sets the `loading` state to `false` to indicate that fetching is complete.

Displaying Collections

The latest collectios are displayed in the drawer like this:

```
< List >
{collections.map((collection) = > (
    < div key={collection.id} >
    < ListItem disablePadding >
    < ListItemButton
    disabled={
    collection.status != = CollectionStatus.COMPLETE | |
    !collection.has_model
    }
    onClick={() = > handleCollectionClick(collection)}
selected = {
    selectedCollection & &
    selectedCollection.id == = collection.id
}
>
< ListItemText
primary = {collection.title} / >
          {collection.status == = CollectionStatus.RUNNING ? (
    < CircularProgress
    size={24}
    style={{position: "absolute", right: 16}}
    / >
): null}
< / ListItemButton >
    < / ListItem >
        < / div >
))}
< / List >
```

You'll notice that the `disabled` property of a collection's `ListItemButton` is set based on whether the collection's status is not `CollectionStatus.COMPLETE` or the collection does not have a model (`!collection.has_model`). If either of these conditions is true, the button is disabled, preventing users from selecting an incomplete or model-less collection. Where the CollectionStatus is RUNNING, we also show a loading wheel over the button.

In a separate `useEffect` hook, we check if any collection in the `collections` state has a status of `CollectionStatus.RUNNING` or `CollectionStatus.QUEUED`. If so, we set up an interval to repeatedly call the `fetchCollections` function every 15 seconds (15,000 milliseconds) to update the collection statuses. This way, the application periodically checks for completed collections, and the UI is updated accordingly when the processing is done.

```
useEffect(() = > {
    let
interval: NodeJS.Timeout;
if (
    collections.some(
        (collection) = >
collection.status == = CollectionStatus.RUNNING | |
collection.status == = CollectionStatus.QUEUED
)
) {
    interval = setInterval(() = > {
    fetchCollections();
}, 15000);
}
return () = > clearInterval(interval);
}, [collections]);
```

Chat View Component

The `ChatView` component in `frontend/src/chat/ChatView.tsx` is responsible for handling and displaying a chat interface for a user to interact with a collection. The component establishes a WebSocket connection to communicate in real-time with the server, sending and receiving messages.

Key features of the `ChatView` component include:

1. Establishing and managing the WebSocket connection with the server.
2. Displaying messages from the user and the server in a chat-like format.
3. Handling user input to send messages to the server.
4. Updating the messages state and UI based on received messages from the server.
5. Displaying connection status and errors, such as loading messages, connecting to the server, or encountering errors while loading a collection.

Together, all of this allows users to interact with their selected collection with a very smooth, low-latency experience.

Chat Websocket Client

The WebSocket connection in the `ChatView` component is used to establish real-time communication between the client and the server. The WebSocket connection is set up and managed in the `ChatView` component as follows:

First, we want to initialize the the WebSocket reference:

const websocket = useRef<WebSocket | null>(null);

A `websocket` reference is created using `useRef`, which holds the WebSocket object that will be used for communication. `useRef` is a hook in React that allows you to create a mutable reference object that persists across renders. It is particularly useful when you need to hold a reference to a mutable object, such as a WebSocket connection, without causing unnecessary re-renders.

In the `ChatView` component, the WebSocket connection needs to be established and maintained throughout the lifetime of the component, and it should not trigger a re-render when the connection state changes. By using `useRef`, you ensure that the WebSocket connection is kept as a reference, and the component only re-renders when there are actual state changes, such as updating messages or displaying errors.

The `setupWebsocket` function is responsible for establishing the WebSocket connection and setting up event handlers to handle different WebSocket events.

Overall, the setupWebsocket function looks like this:

```
const setupWebsocket = () => {
  setConnecting(true);
  // Here, a new WebSocket object is created using the specified URL, which includes the
  // selected collection's ID and the user's authentication token.

  websocket.current = new WebSocket(
    `ws://localhost:8000/ws/collections/${selectedCollection.id}/query/?token=$
↪{authToken}`
  );

  websocket.current.onopen = (event) => {
    //...
  };

  websocket.current.onmessage = (event) => {
    //...
  };

  websocket.current.onclose = (event) => {
    //...
  };

  websocket.current.onerror = (event) => {
    //...
  };

  return () => {
    websocket.current?.close();
  };
};
```

Notice in a bunch of places we trigger updates to the GUI based on the information from the web socket client.

When the component first opens and we try to establish a connection, the `onopen` listener is triggered. In the callback, the component updates the states to reflect that the connection is established, any previous errors are cleared, and no messages are awaiting responses:

```
websocket.current.onopen = (event) => {
  setError(false);
  setConnecting(false);
  setAwaitingMessage(false);

  console.log("WebSocket connected:", event);
};
```

`onmessage`is triggered when a new message is received from the server through the WebSocket connection. In the callback, the received data is parsed and the `messages` state is updated with the new message from the server:

```
websocket.current.onmessage = (event) => {
  const data = JSON.parse(event.data);
```

(continues on next page)

(continued from previous page)

```
  console.log("WebSocket message received:", data);
  setAwaitingMessage(false);

  if (data.response) {
    // Update the messages state with the new message from the server
    setMessages((prevMessages) => [
      ...prevMessages,
      {
        sender_id: "server",
        message: data.response,
        timestamp: new Date().toLocaleTimeString(),
      },
    ]);
  }
};
```

`onclose`is triggered when the WebSocket connection is closed. In the callback, the component checks for a specific close code (`4000`) to display a warning toast and update the component states accordingly. It also logs the close event:

```
websocket.current.onclose = (event) => {
  if (event.code === 4000) {
    toast.warning(
      "Selected collection's model is unavailable. Was it created properly?"
    );
    setError(true);
    setConnecting(false);
    setAwaitingMessage(false);
  }
  console.log("WebSocket closed:", event);
};
```

Finally, `onerror` is triggered when an error occurs with the WebSocket connection. In the callback, the component updates the states to reflect the error and logs the error event:

```
    websocket.current.onerror = (event) => {
      setError(true);
      setConnecting(false);
      setAwaitingMessage(false);

      console.error("WebSocket error:", event);
    };
```

Rendering our Chat Messages

In the `ChatView` component, the layout is determined using CSS styling and Material-UI components. The main layout consists of a container with a `flex` display and a column-oriented `flexDirection`. This ensures that the content within the container is arranged vertically.

There are three primary sections within the layout:

1. The chat messages area: This section takes up most of the available space and displays a list of messages exchanged between the user and the server. It has an overflow-y set to 'auto', which allows scrolling when the

content overflows the available space. The messages are rendered using the `ChatMessage` component for each message and a `ChatMessageLoading` component to show the loading state while waiting for a server response.

2. The divider: A Material-UI `Divider` component is used to separate the chat messages area from the input area, creating a clear visual distinction between the two sections.
3. The input area: This section is located at the bottom and allows the user to type and send messages. It contains a `TextField` component from Material-UI, which is set to accept multiline input with a maximum of 2 rows. The input area also includes a `Button` component to send the message. The user can either click the "Send" button or press " Enter" on their keyboard to send the message.

The user inputs accepted in the `ChatView` component are text messages that the user types in the `TextField`. The component processes these text inputs and sends them to the server through the WebSocket connection.

Deployment

Prerequisites

To deploy the app, you're going to need Docker and Docker Compose installed. If you're on Ubuntu or another, common Linux distribution, DigitalOcean has a great Docker tutorial and another great tutorial for Docker Compose you can follow. If those don't work for you, try the official docker documentation.

Build and Deploy

The project is based on django-cookiecutter, and it's pretty easy to get it deployed on a VM and configured to serve HTTPs traffic for a specific domain. The configuration is somewhat involved, however — not because of this project, but it's just a fairly involved topic to configure your certificates, DNS, etc.

For the purposes of this guide, let's just get running locally. Perhaps we'll release a guide on production deployment. In the meantime, check out the Django Cookiecutter project docs for starters.

This guide assumes your goal is to get the application up and running for use. If you want to develop, most likely you won't want to launch the compose stack with the — profiles fullstack flag and will instead want to launch the react frontend using the node development server.

To deploy, first clone the repo:

```
git clone https://github.com/yourusername/delphic.git
```

Change into the project directory:

```
cd delphic
```

Copy the sample environment files:

```
mkdir -p ./.envs/.local/
cp -a ./docs/sample_envs/local/.frontend ./frontend
cp -a ./docs/sample_envs/local/.django ./.envs/.local
cp -a ./docs/sample_envs/local/.postgres ./.envs/.local
```

Edit the `.django` and `.postgres` configuration files to include your OpenAI API key and set a unique password for your database user. You can also set the response token limit in the .django file or switch which OpenAI model you want to use. GPT4 is supported, assuming you're authorized to access it.

Build the docker compose stack with the `--profiles fullstack` flag:

```
sudo docker-compose --profiles fullstack -f local.yml build
```

The fullstack flag instructs compose to build a docker container from the frontend folder and this will be launched along with all of the needed, backend containers. It takes a long time to build a production React container, however, so we don't recommend you develop this way. Follow the instructions in the project readme.md for development environment setup instructions.

Finally, bring up the application:

```
sudo docker-compose -f local.yml up
```

Now, visit `localhost:3000` in your browser to see the frontend, and use the Delphic application locally.

Using the Application

Setup Users

In order to actually use the application (at the moment, we intend to make it possible to share certain models with unauthenticated users), you need a login. You can use either a superuser or non-superuser. In either case, someone needs to first create a superuser using the console:

Why set up a Django superuser? A Django superuser has all the permissions in the application and can manage all aspects of the system, including creating, modifying, and deleting users, collections, and other data. Setting up a superuser allows you to fully control and manage the application.

How to create a Django superuser:

1 Run the following command to create a superuser:

sudo docker-compose -f local.yml run django python manage.py createsuperuser

2 You will be prompted to provide a username, email address, and password for the superuser. Enter the required information.

How to create additional users using Django admin:

1. Start your Delphic application locally following the deployment instructions.
2. Visit the Django admin interface by navigating to `http://localhost:8000/admin` in your browser.
3. Log in with the superuser credentials you created earlier.
4. Click on "Users" under the "Authentication and Authorization" section.
5. Click on the "Add user +" button in the top right corner.
6. Enter the required information for the new user, such as username and password. Click "Save" to create the user.
7. To grant the new user additional permissions or make them a superuser, click on their username in the user list, scroll down to the "Permissions" section, and configure their permissions accordingly. Save your changes.

3.4.4 A Guide to LlamaIndex + Structured Data

A lot of modern data systems depend on structured data, such as a Postgres DB or a Snowflake data warehouse. LlamaIndex provides a lot of advanced features, powered by LLM's, to both create structured data from unstructured data, as well as analyze this structured data through augmented text-to-SQL capabilities.

This guide helps walk through each of these capabilities. Specifically, we cover the following topics:

- **Inferring Structured Datapoints**: Converting unstructured data to structured data.
- **Text-to-SQL (basic)**: How to query a set of tables using natural language.
- **Injecting Context**: How to inject context for each table into the text-to-SQL prompt. The context can be manually added, or it can be derived from unstructured documents.
- **Storing Table Context within an Index**: By default, we directly insert the context into the prompt. Sometimes this is not feasible if the context is large. Here we show how you can actually use a LlamaIndex data structure to contain the table context!

We will walk through a toy example table which contains city/population/country information.

Setup

First, we use SQLAlchemy to setup a simple sqlite db:

```
from sqlalchemy import create_engine, MetaData, Table, Column, String, Integer, select,␣
↪column

engine = create_engine("sqlite:///:memory:")
metadata_obj = MetaData(bind=engine)
```

We then create a toy `city_stats` table:

```
# create city SQL table
table_name = "city_stats"
city_stats_table = Table(
    table_name,
    metadata_obj,
    Column("city_name", String(16), primary_key=True),
    Column("population", Integer),
    Column("country", String(16), nullable=False),
)
metadata_obj.create_all()
```

Now it's time to insert some datapoints!

If you want to look into filling into this table by inferring structured datapoints from unstructured data, take a look at the below section. Otherwise, you can choose to directly populate this table:

```
from sqlalchemy import insert
rows = [
    {"city_name": "Toronto", "population": 2731571, "country": "Canada"},
    {"city_name": "Tokyo", "population": 13929286, "country": "Japan"},
    {"city_name": "Berlin", "population": 600000, "country": "Germany"},
]
```

(continues on next page)

(continued from previous page)

```
for row in rows:
    stmt = insert(city_stats_table).values(**row)
    with engine.connect() as connection:
        cursor = connection.execute(stmt)
```

Finally, we can wrap the SQLAlchemy engine with our SQLDatabase wrapper; this allows the db to be used within LlamaIndex:

```
from llama_index import SQLDatabase

sql_database = SQLDatabase(engine, include_tables=["city_stats"])
```

If the db is already populated with data, we can instantiate the SQL index with a blank documents list. Otherwise see the below section.

```
index = GPTSQLStructStoreIndex(
    [],
    sql_database=sql_database,
    table_name="city_stats",
)
```

Inferring Structured Datapoints

LlamaIndex offers the capability to convert unstructured datapoints to structured data. In this section, we show how we can populate the `city_stats` table by ingesting Wikipedia articles about each city.

First, we use the Wikipedia reader from LlamaHub to load some pages regarding the relevant data.

```
from llama_index import download_loader

WikipediaReader = download_loader("WikipediaReader")
wiki_docs = WikipediaReader().load_data(pages=['Toronto', 'Berlin', 'Tokyo'])
```

When we build the SQL index, we can specify these docs as the first input; these documents will be converted to structured datapoints and inserted into the db:

```
from llama_index import GPTSQLStructStoreIndex, SQLDatabase

sql_database = SQLDatabase(engine, include_tables=["city_stats"])
# NOTE: the table_name specified here is the table that you
# want to extract into from unstructured documents.
index = GPTSQLStructStoreIndex.from_documents(
    wiki_docs,
    sql_database=sql_database,
    table_name="city_stats",
)
```

You can take a look at the current table to verify that the datapoints have been inserted!

```
# view current table
stmt = select(
    [column("city_name"), column("population"), column("country")]
).select_from(city_stats_table)

with engine.connect() as connection:
    results = connection.execute(stmt).fetchall()
    print(results)
```

Text-to-SQL (basic)

LlamaIndex offers "text-to-SQL" capabilities, both at a very basic level and also at a more advanced level. In this section, we show how to make use of these text-to-SQL capabilities at a basic level.

A simple example is shown here:

```
# set Logging to DEBUG for more detailed outputs
response = index.query("Which city has the highest population?", mode="default")
print(response)
```

You can access the underlying derived SQL query through `response.extra_info['sql_query']`. It should look something like this:

```
SELECT city_name, population
FROM city_stats
ORDER BY population DESC
LIMIT 1
```

Injecting Context

By default, the text-to-SQL prompt just injects the table schema information into the prompt. However, oftentimes you may want to add your own context as well. This section shows you how you can add context, either manually, or extracted through documents.

We offer you a context builder class to better manage the context within your SQL tables: `SQLContextContainerBuilder`. This class takes in the `SQLDatabase` object, and a few other optional parameters, and builds a `SQLContextContainer` object that you can then pass to the index during construction + query-time.

You can add context manually to the context builder. The code snippet below shows you how:

```
# manually set text
city_stats_text = (
    "This table gives information regarding the population and country of a given city.\n
↪"
    "The user will query with codewords, where 'foo' corresponds to population and 'bar'"
    "corresponds to city."
)
table_context_dict={"city_stats": city_stats_text}
context_builder = SQLContextContainerBuilder(sql_database, context_dict=table_context_
↪dict)
```

(continues on next page)

(continued from previous page)

```
context_container = context_builder.build_context_container()

# building the index
index = GPTSQLStructStoreIndex.from_documents(
    wiki_docs,
    sql_database=sql_database,
    table_name="city_stats",
    sql_context_container=context_container
)
```

You can also choose to **extract** context from a set of unstructured Documents. To do this, you can call `SQLContextContainerBuilder.from_documents`. We use the `TableContextPrompt` and the `RefineTableContextPrompt` (see the *reference docs*).

```
# this is a dummy document that we will extract context from
# in GPTSQLContextContainerBuilder
city_stats_text = (
    "This table gives information regarding the population and country of a given city.\n
↪"
)
context_documents_dict = {"city_stats": [Document(city_stats_text)]}
context_builder = SQLContextContainerBuilder.from_documents(
    context_documents_dict,
    sql_database
)
context_container = context_builder.build_context_container()

# building the index
index = GPTSQLStructStoreIndex.from_documents(
    wiki_docs,
    sql_database=sql_database,
    table_name="city_stats",
    sql_context_container=context_container,
)
```

Storing Table Context within an Index

A database collection can have many tables, and if each table has many columns + a description associated with it, then the total context can be quite large.

Luckily, you can choose to use a LlamaIndex data structure to store this table context! Then when the SQL index is queried, we can use this "side" index to retrieve the proper context that can be fed into the text-to-SQL prompt.

Here we make use of the `derive_index_from_context` function within `SQLContextContainerBuilder` to create a new index. You have flexibility in choosing which index class to specify + which arguments to pass in. We then use a helper method called `query_index_for_context` which is a simple wrapper on the `index.query` call that wraps a query template + stores the context on the generated context container.

You can then build the context container, and pass it to the index during query-time!

```
from gpt_index import GPTSQLStructStoreIndex, SQLDatabase, GPTSimpleVectorIndex
from gpt_index.indices.struct_store import SQLContextContainerBuilder
```

(continues on next page)

(continued from previous page)

```
sql_database = SQLDatabase(engine)
# build a vector index from the table schema information
context_builder = SQLContextContainerBuilder(sql_database)
table_schema_index = context_builder.derive_index_from_context(
    GPTSimpleVectorIndex,
    store_index=True
)

query_str = "Which city has the highest population?"

# query the table schema index using the helper method
# to retrieve table context
SQLContextContainerBuilder.query_index_for_context(
    table_schema_index,
    query_str,
    store_context_str=True
)
context_container = context_builder.build_context_container()

# query the SQL index with the table context
response = index.query(query_str, sql_context_container=context_container)
print(response)
```

Concluding Thoughts

This is it for now! We're constantly looking for ways to improve our structured data support. If you have any questions let us know in our Discord.

3.4.5 A Guide to Extracting Terms and Definitions

Llama Index has many use cases (semantic search, summarization, etc.) that are well documented. However, this doesn't mean we can't apply Llama Index to very specific use cases!

In this tutorial, we will go through the design process of using Llama Index to extract terms and definitions from text, while allowing users to query those terms later. Using Streamlit, we can provide an easy to build frontend for running and testing all of this, and quickly iterate with our design.

This tutorial assumes you have Python3.9+ and the following packages installed:

- llama-index
- streamlit

At the base level, our objective is to take text from a document, extract terms and definitions, and then provide a way for users to query that knowledge base of terms and definitions. The tutorial will go over features from both Llama Index and Streamlit, and hopefully provide some interesting solutions for common problems that come up.

The final version of this tutorial can be found here and a live hosted demo is available on Huggingface Spaces.

Uploading Text

Step one is giving users a way to upload documents. Let's write some code using Streamlit to provide the interface for this! Use the following code and launch the app with `streamlit run app.py`.

```
import streamlit as st

st.title(" Llama Index Term Extractor ")

document_text = st.text_area("Or enter raw text")
if st.button("Extract Terms and Definitions") and document_text:
    with st.spinner("Extracting..."):
        extracted_terms = document text  # this is a placeholder!
    st.write(extracted_terms)
```

Super simple right! But you'll notice that the app doesn't do anything useful yet. To use llama_index, we also need to setup our OpenAI LLM. There are a bunch of possible settings for the LLM, so we can let the user figure out what's best. We should also let the user set the prompt that will extract the terms (which will also help us debug what works best).

LLM Settings

This next step introduces some tabs to our app, to separate it into different panes that provide different features. Let's create a tab for LLM settings and for uploading text:

```
import os
import streamlit as st

DEFAULT_TERM_STR = (
    "Make a list of terms and definitions that are defined in the context, "
    "with one pair on each line. "
    "If a term is missing it's definition, use your best judgment. "
    "Write each line as as follows:\nTerm: <term> Definition: <definition>"
)

st.title(" Llama Index Term Extractor ")

setup_tab, upload_tab = st.tabs(["Setup", "Upload/Extract Terms"])

with setup_tab:
    st.subheader("LLM Setup")
    api_key = st.text_input("Enter your OpenAI API key here", type="password")
    llm_name = st.selectbox('Which LLM?', ["text-davinci-003", "gpt-3.5-turbo", "gpt-4"])
    model_temperature = st.slider("LLM Temperature", min_value=0.0, max_value=1.0,␣
↪step=0.1)
    term_extract_str = st.text_area("The query to extract terms and definitions with.",␣
↪value=DEFAULT_TERM_STR)

with upload_tab:
    st.subheader("Extract and Query Definitions")
    document_text = st.text_area("Or enter raw text")
    if st.button("Extract Terms and Definitions") and document_text:
        with st.spinner("Extracting..."):
```

(continues on next page)

(continued from previous page)

```
            extracted_terms = document text  # this is a placeholder!
        st.write(extracted_terms)
```

Now our app has two tabs, which really helps with the organization. You'll also noticed I added a default prompt to extract terms – you can change this later once you try extracting some terms, it's just the prompt I arrived at after experimenting a bit.

Speaking of extracting terms, it's time to add some functions to do just that!

Extracting and Storing Terms

Now that we are able to define LLM settings and upload text, we can try using Llama Index to extract the terms from text for us!

We can add the following functions to both initialize our LLM, as well as use it to extract terms from the input text.

```
from llama_index import Document, GPTListIndex, LLMPredictor, ServiceContext,␣
↪PromptHelper

def get_llm(llm_name, model_temperature, api_key, max_tokens=256):
    os.environ['OPENAI_API_KEY'] = api_key
    if llm_name == "text-davinci-003":
        return OpenAI(temperature=model_temperature, model_name=llm_name, max_tokens=max_
↪tokens)
    else:
        return ChatOpenAI(temperature=model_temperature, model_name=llm_name, max_
↪tokens=max_tokens)

def extract_terms(documents, term_extract_str, llm_name, model_temperature, api_key):
    llm = get_llm(llm_name, model_temperature, api_key, max_tokens=1024)

    service_context = ServiceContext.from_defaults(llm_predictor=LLMPredictor(llm=llm),
                                                   prompt_helper=PromptHelper(max_input_
↪size=4096,
                                                                              max_chunk_
↪overlap=20,
                                                                              num_
↪output=1024),
                                                   chunk_size_limit=1024)

    temp_index = GPTListIndex.from_documents(documents, service_context=service_context)
    terms_definitions = str(temp_index.query(term_extract_str, response_mode="tree_
↪summarize"))
    terms_definitions = [x for x in terms_definitions.split("\n") if x and 'Term:' in x␣
↪and 'Definition:' in x]
    # parse the text into a dict
    terms_to_definition = {x.split("Definition:")[0].split("Term:")[-1].strip(): x.split(
↪"Definition:")[-1].strip() for x in terms_definitions}
    return terms_to_definition
```

Now, using the new functions, we can finally extract our terms!

```
...
with upload_tab:
    st.subheader("Extract and Query Definitions")
    document_text = st.text_area("Or enter raw text")
    if st.button("Extract Terms and Definitions") and document_text:
        with st.spinner("Extracting..."):
            extracted_terms = extract_terms([Document(document_text)],
                                            term_extract_str, llm_name,
                                            model_temperature, api_key)
        st.write(extracted_terms)
```

There's a lot going on now, let's take a moment to go over what is happening.

`get_llm()` is instantiating the LLM based on the user configuration from the setup tab. Based on the model name, we need to use the appropriate class (`OpenAI` vs. `ChatOpenAI`).

`extract_terms()` is where all the good stuff happens. First, we call `get_llm()` with `max_tokens=1024`, since we don't want to limit the model too much when it is extracting our terms and definitions (the default is 256 if not set). Then, we define our `ServiceContext` object, aligning `num_output` with our `max_tokens` value, as well as setting the chunk size to be no larger than the output. When documents are indexed by Llama Index, they are broken into chunks (also called nodes) if they are large, and `chunk_size_limit` sets the maximum size for these chunks.

Next, we create a temporary list index and pass in our service context. A list index will read every single piece of text in our index, which is perfect for extracting terms. Finally, we use our pre-defined query text to extract terms, using `response_mode="tree_summarize`. This response mode will generate a tree of summaries from the bottom up, where each parent summarizes its children. Finally, the top of the tree is returned, which will contain all our extracted terms and definitions.

Lastly, we do some minor post processing. We assume the model followed instructions and put a term/definition pair on each line. If a line is missing the `Term:` or `Definition:` labels, we skip it. Then, we convert this to a dictionary for easy storage!

Saving Extracted Terms

Now that we can extract terms, we need to put them somewhere so that we can query for them later. A `GPTSimpleVectorIndex` should be a perfect choice for now! But in addition, our app should also keep track of which terms are inserted into the index so that we can inspect them later. Using `st.session_state`, we can store the current list of terms in a session dict, unique to each user!

First things first though, let's add a feature to initialize a global vector index and another function to insert the extracted terms.

```
...
if 'all_terms' not in st.session_state:
    st.session_state['all_terms'] = DEFAULT_TERMS
...

def insert_terms(terms_to_definition):
    for term, definition in terms_to_definition.items():
        doc = Document(f"Term: {term}\nDefinition: {definition}")
        st.session_state['llama_index'].insert(doc)

@st.cache_resource
def initialize_index(llm_name, model_temperature, api_key):
    """Create the GPTSQLStructStoreIndex object."""
```

(continues on next page)

(continued from previous page)

```
    llm = get_llm(llm_name, model_temperature, api_key)

    service_context = ServiceContext.from_defaults(llm_predictor=LLMPredictor(llm=llm))

    index = GPTSimpleVectorIndex([], service_context=service_context)

    return index

...

with upload_tab:
    st.subheader("Extract and Query Definitions")
    if st.button("Initialize Index and Reset Terms"):
        st.session_state['llama_index'] = initialize_index(llm_name, model_temperature,␣
↪api_key)
        st.session_state['all_terms'] = {}

    if "llama_index" in st.session_state:
        st.markdown("Either upload an image/screenshot of a document, or enter the text␣
↪manually.")
        document_text = st.text_area("Or enter raw text")
        if st.button("Extract Terms and Definitions") and (uploaded_file or document_
↪text):
            st.session_state['terms'] = {}
            terms_docs = {}
            with st.spinner("Extracting..."):
                terms_docs.update(extract_terms([Document(document_text)], term_extract_
↪str, llm_name, model_temperature, api_key))
            st.session_state['terms'].update(terms_docs)

        if "terms" in st.session_state and st.session_state["terms"]::
            st.markdown("Extracted terms")
            st.json(st.session_state['terms'])

            if st.button("Insert terms?"):
                with st.spinner("Inserting terms"):
                    insert_terms(st.session_state['terms'])
                st.session_state['all_terms'].update(st.session_state['terms'])
                st.session_state['terms'] = {}
                st.experimental_rerun()
```

Now you are really starting to leverage the power of streamlit! Let's start with the code under the upload tab. We added a button to initialize the vector index, and we store it in the global streamlit state dictionary, as well as resetting the currently extracted terms. Then, after extracting terms from the input text, we store it the extracted terms in the global state again and give the user a chance to review them before inserting. If the insert button is pressed, then we call our insert terms function, update our global tracking of inserted terms, and remove the most recently extracted terms from the session state.

Querying for Extracted Terms/Definitions

With the terms and definitions extracted and saved, how can we use them? And how will the user even remember what's previously been saved?? We can simply add some more tabs to the app to handle these features.

```
...
setup_tab, terms_tab, upload_tab, query_tab = st.tabs(
    ["Setup", "All Terms", "Upload/Extract Terms", "Query Terms"]
)
...
with terms_tab:
    with terms_tab:
    st.subheader("Current Extracted Terms and Definitions")
    st.json(st.session_state["all_terms"])
...
with query_tab:
    st.subheader("Query for Terms/Definitions!")
    st.markdown(
        (
            "The LLM will attempt to answer your query, and augment it's answers using␣
↪the terms/definitions you've inserted. "
            "If a term is not in the index, it will answer using it's internal knowledge.
↪"
        )
    )
    if st.button("Initialize Index and Reset Terms", key="init_index_2"):
        st.session_state["llama_index"] = initialize_index(
            llm_name, model_temperature, api_key
        )
        st.session_state["all_terms"] = {}

    if "llama_index" in st.session_state:
        query_text = st.text_input("Ask about a term or definition:")
        if query_text:
            query_text = query_text + "\nIf you can't find the answer, answer the query␣
↪with the best of your knowledge."
            with st.spinner("Generating answer..."):
                response = st.session_state["llama_index"].query(
                    query_text, similarity_top_k=5, response_mode="compact"
                )
            st.markdown(str(response))
```

While this is mostly basic, some important things to note:

- Our initialize button has the same text as our other button. Streamlit will complain about this, so we provide a unique key instead.
- Some additional text has been added to the query! This is to try and compensate for times when the index does not have the answer.
- In our index query, we've specified two options:
  - `similarity_top_k=5` means the index will fetch the top 5 closest matching terms/definitions to the query.
  - `response_mode="compact"` means as much text as possible from the 5 matching terms/definitions will be used in each LLM call. Without this, the index would make at least 5 calls to the LLM, which can slow things down for the user.

Dry Run Test

Well, actually I hope you've been testing as we went. But now, let's try one complete test.

1. Refresh the app
2. Enter your LLM settings
3. Head over to the query tab
4. Ask the following: `What is a bunnyhug?`
5. The app should give some nonsense response. If you didn't know, a bunnyhug is another word for a hoodie, used by people from the Canadian Prairies!
6. Let's add this definition to the app. Open the upload tab and enter the following text: `A bunnyhug is a common term used to describe a hoodie. This term is used by people from the Canadian Prairies.`
7. Click the extract button. After a few moments, the app should display the correctly extracted term/definition. Click the insert term button to save it!
8. If we open the terms tab, the term and definition we just extracted should be displayed
9. Go back to the query tab and try asking what a bunnyhug is. Now, the answer should be correct!

Improvement #1 - Create a Starting Index

With our base app working, it might feel like a lot of work to build up a useful index. What if we gave the user some kind of starting point to show off the app's query capabilities? We can do just that! First, let's make a small change to our app so that we save the index to disk after every upload:

```
def insert_terms(terms_to_definition):
    for term, definition in terms_to_definition.items():
        doc = Document(f"Term: {term}\nDefinition: {definition}")
        st.session_state['llama_index'].insert(doc)
    # TEMPORARY - save to disk
    st.session_state['llama_index'].save_to_disk("index.json")
```

Now, we need some document to extract from! The repository for this project used the wikipedia page on New York City, and you can find the text here.

If you paste the text into the upload tab and run it (it may take some time), we can insert the extracted terms. Make sure to also copy the text for the extracted terms into a notepad or similar before inserting into the index! We will need them in a second.

After inserting, remove the line of code we used to save the index to disk. With a starting index now saved, we can modify our `initialize_index` function to look like this:

```
@st.cache_resource
def initialize_index(llm_name, model_temperature, api_key):
    """Create the GPTSQLStructStoreIndex object."""
    llm = get_llm(llm_name, model_temperature, api_key)

    service_context = ServiceContext.from_defaults(llm_predictor=LLMPredictor(llm=llm))

    index = GPTSimpleVectorIndex.load_from_disk(
        "./index.json", service_context=service_context
```

(continues on next page)

(continued from previous page)

```
    )

    return index
```

Did you remember to save that giant list of extracted terms in a notepad? Now when our app initializes, we want to pass in the default terms that are in the index to our global terms state:

```
...
if "all_terms" not in st.session_state:
    st.session_state["all_terms"] = DEFAULT_TERMS
...
```

Repeat the above anywhere where we were previously resetting the `all_terms` values.

Improvement #2 - (Refining) Better Prompts

If you play around with the app a bit now, you might notice that it stopped following our prompt! Remember, we added to our `query_str` variable that if the term/definition could not be found, answer to the best of its knowledge. But now if you try asking about random terms (like bunnyhug!), it may or may not follow those instructions.

This is due to the concept of "refining" answers in Llama Index. Since we are querying across the top 5 matching results, sometimes all the results do not fit in a single prompt! OpenAI models typically have a max input size of 4097 tokens. So, Llama Index accounts for this by breaking up the matching results into chunks that will fit into the prompt. After Llama Index gets an initial answer from the first API call, it sends the next chunk to the API, along with the previous answer, and asks the model to refine that answer.

So, the refine process seems to be messing with our results! Rather than appending extra instructions to the `query_str`, remove that, and Llama Index will let us provide our own custom prompts! Let's create those now, using the default prompts and chat specific prompts as a guide. Using a new file `constants.py`, let's create some new query templates:

```
from langchain.chains.prompt_selector import ConditionalPromptSelector, is_chat_model
from langchain.prompts.chat import (
    AIMessagePromptTemplate,
    ChatPromptTemplate,
    HumanMessagePromptTemplate,
)

from llama_index.prompts.prompts import QuestionAnswerPrompt, RefinePrompt

# Text QA templates
DEFAULT_TEXT_QA_PROMPT_TMPL = (
    "Context information is below. \n"
    "---------------------\n"
    "{context_str}"
    "\n---------------------\n"
    "Given the context information answer the following question "
    "(if you don't know the answer, use the best of your knowledge): {query_str}\n"
)
TEXT_QA_TEMPLATE = QuestionAnswerPrompt(DEFAULT_TEXT_QA_PROMPT_TMPL)

# Refine templates
DEFAULT_REFINE_PROMPT_TMPL = (
    "The original question is as follows: {query_str}\n"
```

(continues on next page)

(continued from previous page)

```
    "We have provided an existing answer: {existing_answer}\n"
    "We have the opportunity to refine the existing answer "
    "(only if needed) with some more context below.\n"
    "------------\n"
    "{context_msg}\n"
    "------------\n"
    "Given the new context and using the best of your knowledge, improve the existing␣
↪answer. "
    "If you can't improve the existing answer, just repeat it again."
)
DEFAULT_REFINE_PROMPT = RefinePrompt(DEFAULT_REFINE_PROMPT_TMPL)

CHAT_REFINE_PROMPT_TMPL_MSGS = [
    HumanMessagePromptTemplate.from_template("{query_str}"),
    AIMessagePromptTemplate.from_template("{existing_answer}"),
    HumanMessagePromptTemplate.from_template(
        "We have the opportunity to refine the above answer "
        "(only if needed) with some more context below.\n"
        "------------\n"
        "{context_msg}\n"
        "------------\n"
        "Given the new context and using the best of your knowledge, improve the␣
↪existing answer. "
    "If you can't improve the existing answer, just repeat it again."
    ),
]

CHAT_REFINE_PROMPT_LC = ChatPromptTemplate.from_messages(CHAT_REFINE_PROMPT_TMPL_MSGS)
CHAT_REFINE_PROMPT = RefinePrompt.from_langchain_prompt(CHAT_REFINE_PROMPT_LC)

# refine prompt selector
DEFAULT_REFINE_PROMPT_SEL_LC = ConditionalPromptSelector(
    default_prompt=DEFAULT_REFINE_PROMPT.get_langchain_prompt(),
    conditionals=[(is_chat_model, CHAT_REFINE_PROMPT.get_langchain_prompt())],
)
REFINE_TEMPLATE = RefinePrompt(
    langchain_prompt_selector=DEFAULT_REFINE_PROMPT_SEL_LC
)
```

That seems like a lot of code, but it's not too bad! If you looked at the default prompts, you might have noticed that there are default prompts, and prompts specific to chat models. Continuing that trend, we do the same for our custom prompts. Then, using a prompt selector, we can combine both prompts into a single object. If the LLM being used is a chat model (ChatGPT, GPT-4), then the chat prompts are used. Otherwise, use the normal prompt templates.

Another thing to note is that we only defined one QA template. In a chat model, this will be converted to a single "human" message.

So, now we can import these prompts into our app and use them during the query.

```
from constants import REFINE_TEMPLATE, TEXT_QA_TEMPLATE
...
    if "llama_index" in st.session_state:
        query_text = st.text_input("Ask about a term or definition:")
```

(continues on next page)

(continued from previous page)

```
        if query_text:
            query_text = query_text  # Notice we removed the old instructions
            with st.spinner("Generating answer..."):
                response = st.session_state["llama_index"].query(
                    query_text, similarity_top_k=5, response_mode="compact",
                    text_qa_template=TEXT_QA_TEMPLATE, refine_template=REFINE_TEMPLATE
                )
            st.markdown(str(response))
...
```

If you experiment a bit more with queries, hopefully you notice that the responses follow our instructions a little better now!

Improvement #3 - Image Support

Llama index also supports images! Using Llama Index, we can upload images of documents (papers, letters, etc.), and Llama Index handles extracting the text. We can leverage this to also allow users to upload images of their documents and extract terms and definitions from them.

If you get an import error about PIL, install it using `pip install Pillow` first.

```
from PIL import Image
from llama_index.readers.file.base import DEFAULT_FILE_EXTRACTOR, ImageParser

@st.cache_resource
def get_file_extractor():
    image_parser = ImageParser(keep_image=True, parse_text=True)
    file_extractor = DEFAULT_FILE_EXTRACTOR
    file_extractor.update(
        {
            ".jpg": image_parser,
            ".png": image_parser,
            ".jpeg": image_parser,
        }
    )

    return file_extractor

file_extractor = get_file_extractor()
...
with upload_tab:
    st.subheader("Extract and Query Definitions")
    if st.button("Initialize Index and Reset Terms", key="init_index_1"):
        st.session_state["llama_index"] = initialize_index(
            llm_name, model_temperature, api_key
        )
        st.session_state["all_terms"] = DEFAULT_TERMS

    if "llama_index" in st.session_state:
        st.markdown(
            "Either upload an image/screenshot of a document, or enter the text manually.
↪"
```

(continues on next page)

(continued from previous page)

```
        )
        uploaded_file = st.file_uploader(
            "Upload an image/screenshot of a document:", type=["png", "jpg", "jpeg"]
        )
        document_text = st.text_area("Or enter raw text")
        if st.button("Extract Terms and Definitions") and (
            uploaded_file or document_text
        ):
            st.session_state["terms"] = {}
            terms_docs = {}
            with st.spinner("Extracting (images may be slow)..."):
                if document_text:
                    terms_docs.update(
                        extract_terms(
                            [Document(document_text)],
                            term_extract_str,
                            llm_name,
                            model_temperature,
                            api_key,
                        )
                    )
                if uploaded_file:
                    Image.open(uploaded_file).convert("RGB").save("temp.png")
                    img_reader = SimpleDirectoryReader(
                        input_files=["temp.png"], file_extractor=file_extractor
                    )
                    img_docs = img_reader.load_data()
                    os.remove("temp.png")
                    terms_docs.update(
                        extract_terms(
                            img_docs,
                            term_extract_str,
                            llm_name,
                            model_temperature,
                            api_key,
                        )
                    )
            st.session_state["terms"].update(terms_docs)

        if "terms" in st.session_state and st.session_state["terms"]:
            st.markdown("Extracted terms")
            st.json(st.session_state["terms"])

            if st.button("Insert terms?"):
                with st.spinner("Inserting terms"):
                    insert_terms(st.session_state["terms"])
                st.session_state["all_terms"].update(st.session_state["terms"])
                st.session_state["terms"] = {}
                st.experimental_rerun()
```

Here, we added the option to upload a file using Streamlit. Then the image is opened and saved to disk (this seems hacky but it keeps things simple). Then we pass the image path to the reader, extract the documents/text, and remove our temp image file.

Now that we have the documents, we can call `extract_terms()` the same as before.

Conclusion/TLDR

In this tutorial, we covered a ton of information, while solving some common issues and problems along the way:

- Using different indexes for different use cases (List vs. Vector index)
- Storing global state values with Streamlit's `session_state` concept
- Customizing internal prompts with Llama Index
- Reading text from images with Llama Index

The final version of this tutorial can be found here and a live hosted demo is available on Huggingface Spaces.

3.4.6 A Guide to Creating a Unified Query Framework over your Indexes

LlamaIndex offers a variety of different *query use cases*.

For simple queries, we may want to use a single index data structure, such as a `GPTSimpleVectorIndex` for semantic search, or `GPTListIndex` for summarization.

For more complex queries, we may want to use a composable graph.

But how do we integrate indexes and graphs into our LLM application? Different indexes and graphs may be better suited for different types of queries that you may want to run.

In this guide, we show how you can unify the diverse use cases of different index/graph structures under a **single** query framework.

Setup

In this example, we will analyze Wikipedia articles of different cities: Boston, Seattle, San Francisco, and more.

The below code snippet downloads the relevant data into files.

```
from pathlib import Path
import requests

wiki_titles = ["Toronto", "Seattle", "Chicago", "Boston", "Houston"]

for title in wiki_titles:
    response = requests.get(
        'https://en.wikipedia.org/w/api.php',
        params={
            'action': 'query',
            'format': 'json',
            'titles': title,
            'prop': 'extracts',
            # 'exintro': True,
            'explaintext': True,
        }
    ).json()
    page = next(iter(response['query']['pages'].values()))
```

(continues on next page)

(continued from previous page)

```
    wiki_text = page['extract']

    data_path = Path('data')
    if not data_path.exists():
        Path.mkdir(data_path)

    with open(data_path / f"{title}.txt", 'w') as fp:
        fp.write(wiki_text)
```

The next snippet loads all files into Document objects.

```
# Load all wiki documents
city_docs = {}
for wiki_title in wiki_titles:
    city_docs[wiki_title] = SimpleDirectoryReader(input_files=[f"data/{wiki_title}.txt
↪"]).load_data()
```

Defining the Set of Indexes

We will now define a set of indexes and graphs over your data. You can think of each index/graph as a lightweight structure that solves a distinct use case.

We will first define a vector index over the documents of each city.

```
from gpt_index import GPTSimpleVectorIndex, ServiceContext
from langchain.llms.openai import OpenAIChat

# set service context
llm_predictor_gpt4 = LLMPredictor(llm=OpenAIChat(temperature=0, model_name="gpt-4"))
service_context = ServiceContext.from_defaults(
    llm_predictor=llm_predictor_gpt4, chunk_size_limit=1024
)

# Build city document index
vector_indices = {}
for wiki_title in wiki_titles:
    # build vector index
    vector_indices[wiki_title] = GPTSimpleVectorIndex.from_documents(
        city_docs[wiki_title], service_context=service_context
    )
    # set id for vector index
    vector_indices[wiki_title].index_struct.index_id = wiki_title
    vector_indices[wiki_title].save_to_disk(f'index_{wiki_title}.json')
```

Querying a vector index lets us easily perform semantic search over a given city's documents.

```
response = vector_indices["Toronto"].query("What are the sports teams in Toronto?")
print(str(response))
```

Example response:

```
The sports teams in Toronto are the Toronto Maple Leafs (NHL), Toronto Blue Jays (MLB),␣
↪Toronto Raptors (NBA), Toronto Argonauts (CFL), Toronto FC (MLS), Toronto Rock (NLL),␣
↪Toronto Wolfpack (RFL), and Toronto Rush (NARL).
```

Defining a Graph for Compare/Contrast Queries

We will now define a composed graph in order to run **compare/contrast** queries (see *use cases doc*). This graph contains a keyword table composed on top of existing vector indexes.

To do this, we first want to set the "summary text" for each vector index.

```
index_summaries = {}
for wiki_title in wiki_titles:
    # set summary text for city
    index_summaries[wiki_title] = (
        f"This content contains Wikipedia articles about {wiki_title}. "
        f"Use this index if you need to lookup specific facts about {wiki_title}.\n"
        "Do not use this index if you want to analyze multiple cities."
    )
```

Next, we compose a keyword table on top of these vector indexes, with these indexes and summaries, in order to build the graph.

```
from gpt_index.indices.composability import ComposableGraph

graph = ComposableGraph.from_indices(
    GPTSimpleKeywordTableIndex,
    [index for _, index in vector_indices.items()],
    [summary for _, summary in index_summaries.items()],
    max_keywords_per_chunk=50
)

# get root index
root_index = graph.get_index(graph.index_struct.root_id, GPTSimpleKeywordTableIndex)
# set id of root index
root_index.index_struct.index_id = "compare_contrast"
root_summary = (
    "This index contains Wikipedia articles about multiple cities. "
    "Use this index if you want to compare multiple cities. "
)
```

Querying this graph (with a query transform module), allows us to easily compare/contrast between different cities. An example is shown below - we define query_configs and send a query through this graph.

```
# define decompose_transform
from gpt_index.indices.query.query_transform.base import DecomposeQueryTransform
decompose_transform = DecomposeQueryTransform(
    llm_predictor_chatgpt, verbose=True
)
# set query config
query_configs = [
```

(continues on next page)

(continued from previous page)

```
    {
        "index_struct_type": "simple_dict",
        "query_mode": "default",
        "query_kwargs": {
            "similarity_top_k": 1
        },
        # NOTE: set query transform for subindices
        "query_transform": decompose_transform
    },
    {
        "index_struct_type": "keyword_table",
        "query_mode": "simple",
        "query_kwargs": {
            "response_mode": "tree_summarize",
            "verbose": True
        },
    },
]

# query the graph
query_str = (
    "Compare and contrast the arts and culture of Houston and Boston. "
)
response_chatgpt = graph.query(
    query_str,
    query_configs=query_configs,
    service_context=service_context,
)
```

Defining the Unified Query Interface

Now that we've defined the set of indexes/graphs, we want to build an **outer abstraction** layer that provides a unified query interface to our data structures. This means that during query-time, we can query this outer abstraction layer and trust that the right index/graph will be used for the job.

There are a few ways to do this, both within our framework as well as outside of it!

- Compose a "router" on top of your existing indexes/graphs (basically expanding the graph!)
  - There are a few different "router" modules we can use, such as our tree index or vector index.
- Define each index/graph as a Tool within an agent framework (e.g. LangChain).

For the purposes of this tutorial, we follow the former approach. If you want to take a look at how the latter approach works, take a look at *our example tutorial here*.

We define this graph using a tree index. The tree index serves as a "router". A router is at the core of defining a unified query interface. This allows us to "route" any query to the set of indexes/graphs that you have defined under the hood.

We compose the tree index over all the vector indexes + the graph (used for compare/contrast queries).

```
from gpt_index import GPTTreeIndex

# num children is num vector indexes + graph
```

(continues on next page)

(continued from previous page)

```
num_children = len(vector_indices) + 1
outer_graph = ComposableGraph.from_indices(
    GPTTreeIndex,
    [index for _, index in vector_indices.items()] + [root_index],
    [summary for _, summary in index_summaries.items()] + [root_summary],
    num_children=num_children
)
```

Querying our Unified Interface

The advantage of a unified query interface is that it can now handle different types of queries.

It can now handle queries about specific cities (by routing to the specific city vector index), and also compare/contrast different cities.

```
# set query config
query_configs = [
    {
        "index_struct_type": "keyword_table",
        "query_mode": "simple",
        "query_kwargs": {
            "response_mode": "tree_summarize",
            "verbose": True
        },
    },
    {
        "index_struct_type": "tree",
        "query_mode": "default",

    }
]
for wiki_title in wiki_titles:
    query_config = {
        "index_struct_id": wiki_title,
        "index_struct_type": "simple_dict",
        "query_mode": "default",
        "query_kwargs": {
            "similarity_top_k": 1
        },
        # NOTE: set query transform for subindices
        "query_transform": decompose_transform
    }
    query_configs.append(query_config)
```

Let's take a look at a few examples!

Asking a Compare/Contrast Question

```
# ask a compare/contrast question
response = outer_graph.query(
    "Compare and contrast the arts and culture of Houston and Boston.",
```

(continues on next page)

(continued from previous page)

```
    query_configs=query_configs,
    service_context=service_context
)
print(str(response)
```

Asking Questions about specific Cities

```
response = outer_graph.query("What are the sports teams in Toronto?")
print(str(response))
```

This "outer" abstraction is able to handle different queries by routing to the right underlying abstractions.

3.5 Notebooks

We offer a wide variety of example notebooks. They are referenced throughout the documentation.

Example notebooks are found here.

3.6 Queries over your Data

At a high-level, LlamaIndex gives you the ability to query your data for any downstream LLM use case, whether it's question-answering, summarization, or a component in a chatbot.

This section describes the different ways you can query your data with LlamaIndex, roughly in order of simplest (top-k semantic search), to more advanced capabilities.

3.6.1 Semantic Search

The most basic example usage of LlamaIndex is through semantic search. We provide a simple in-memory vector store for you to get started, but you can also choose to use any one of our *vector store integrations*:

```
from llama_index import GPTSimpleVectorIndex, SimpleDirectoryReader
documents = SimpleDirectoryReader('data').load_data()
index = GPTSimpleVectorIndex.from_documents(documents)
response = index.query("What did the author do growing up?")
print(response)
```

Relevant Resources:

- *Quickstart*
- Example notebook

3.6.2 Summarization

A summarization query requires the LLM to iterate through many if not most documents in order to synthesize an answer. For instance, a summarization query could look like one of the following:

- "What is a summary of this collection of text?"
- "Give me a summary of person X's experience with the company."

In general, a list index would be suited for this use case. A list index by default goes through all the data.

Empirically, setting `response_mode="tree_summarize"` also leads to better summarization results.

```
index = GPTListIndex.from_documents(documents)

response = index.query("<summarization_query>", response_mode="tree_summarize")
```

3.6.3 Queries over Structured Data

LlamaIndex supports queries over structured data, whether that's a Pandas DataFrame or a SQL Database.

Here are some relevant resources:

- *Guide on Text-to-SQL*
- SQL Demo Notebook 1
- SQL Demo Notebook 2 (Context)
- SQL Demo Notebook 3 (Big tables)
- Pandas Demo Notebook.

3.6.4 Synthesis over Heterogeneous Data

LlamaIndex supports synthesizing across heterogeneous data sources. This can be done by composing a graph over your existing data. Specifically, compose a list index over your subindices. A list index inherently combines information for each node; therefore it can synthesize information across your heterogeneous data sources.

```
from llama_index import GPTSimpleVectorIndex, GPTListIndex
from llama_index.indices.composability import ComposableGraph

index1 = GPTSimpleVectorIndex.from_documents(notion_docs)
index2 = GPTSimpleVectorIndex.from_documents(slack_docs)

graph = ComposableGraph.from_indices(GPTListIndex, [index1, index2], index_summaries=[
↪"summary1", "summary2"])
response = graph.query("<query_str>", mode="recursive", query_configs=...)
```

Here are some relevant resources:

- *Composability*
- City Analysis Demo.

3.6.5 Routing over Heterogeneous Data

LlamaIndex also supports routing over heterogeneous data sources - for instance, if you want to "route" a query to an underlying Document or a subindex. Here you have three options: `GPTTreeIndex`, `GPTKeywordTableIndex`, or a *Vector Store Index*.

A `GPTTreeIndex` uses the LLM to select the child node(s) to send the query down to. A `GPTKeywordTableIndex` uses keyword matching, and a `GPTVectorStoreIndex` uses embedding cosine similarity.

```
from llama_index import GPTTreeIndex, GPTSimpleVectorIndex
from llama_index.indices.composability import ComposableGraph

...

# subindices
index1 = GPTSimpleVectorIndex.from_documents(notion_docs)
index2 = GPTSimpleVectorIndex.from_documents(slack_docs)

# tree index for routing
tree_index = ComposableGraph.from_indices(
    GPTTreeIndex,
    [index1, index2],
    index_summaries=["summary1", "summary2"]
)

response = tree_index.query(
    "In Notion, give me a summary of the product roadmap.",
    mode="recursive",
    query_configs=...
)
```

Here are some relevant resources:

- *Composability*
- Composable Keyword Table Graph.

3.6.6 Compare/Contrast Queries

LlamaIndex can support compare/contrast queries as well. It can do this in the following fashion:

- Composing a graph over your data
- Adding in query transformations.

You can perform compare/contrast queries by just composing a graph over your data.

Here are some relevant resources:

- *Composability*
- SEC 10-k Analysis Example notebook.

You can also perform compare/contrast queries with a **query transformation** module.

```
from gpt_index.indices.query.query_transform.base import DecomposeQueryTransform
decompose_transform = DecomposeQueryTransform(
    llm_predictor_chatgpt, verbose=True
)
```

This module will help break down a complex query into a simpler one over your existing index structure.

Here are some relevant resources:

- *Query Transformations*
- City Analysis Example Notebook

3.6.7 Multi-Step Queries

LlamaIndex can also support multi-step queries. Given a complex query, break it down into subquestions.

For instance, given a question "Who was in the first batch of the accelerator program the author started?", the module will first decompose the query into a simpler initial question "What was the accelerator program the author started?", query the index, and then ask followup questions.

Here are some relevant resources:

- *Query Transformations*
- Multi-Step Query Decomposition Notebook

3.7 Integrations into LLM Applications

LlamaIndex modules provide plug and play data loaders, data structures, and query interfaces. They can be used in your downstream LLM Application. Some of these applications are described below.

3.7.1 Chatbots

Chatbots are an incredibly popular use case for LLM's. LlamaIndex gives you the tools to build Knowledge-augmented chatbots and agents.

Relevant Resources:

- *Building a Chatbot*
- *Using with a LangChain Agent*

3.7.2 Full-Stack Web Application

LlamaIndex can be integrated into a downstream full-stack web application. It can be used in a backend server (such as Flask), packaged into a Docker container, and/or directly used in a framework such as Streamlit.

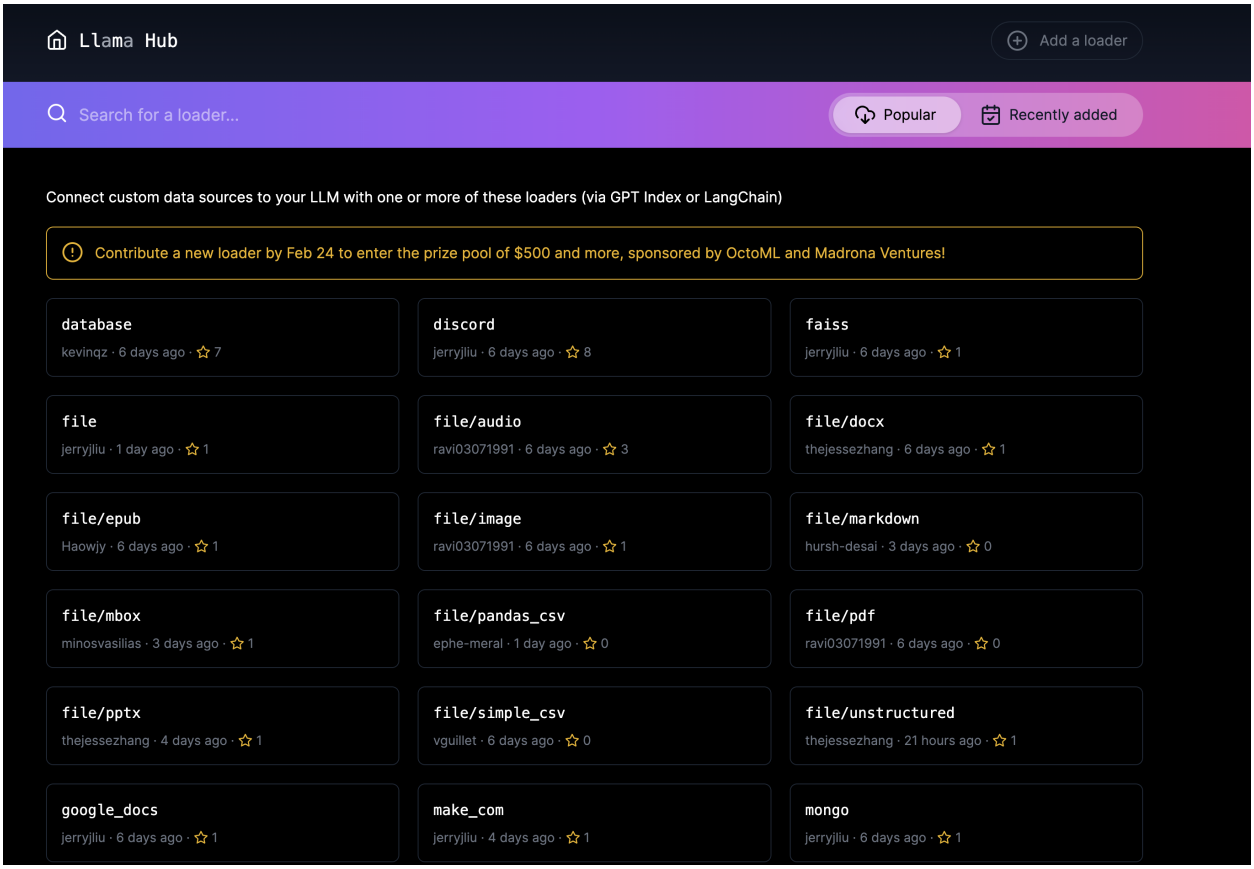
We provide tutorials and resources to help you get started in this area.

Relevant Resources:

- *Fullstack Application Guide*
- LlamaIndex Starter Pack

3.8 Data Connectors (LlamaHub)

Our data connectors are offered through LlamaHub . LlamaHub is an open-source repository containing data loaders that you can easily plug and play into any LlamaIndex application.



Some sample data connectors:

- local file directory (`SimpleDirectoryReader`). Can support parsing a wide range of file types: `.pdf`, `.jpg`, `.png`, `.docx`, etc.
- Notion (`NotionPageReader`)
- Google Docs (`GoogleDocsReader`)
- Slack (`SlackReader`)
- Discord (`DiscordReader`)

Each data loader contains a "Usage" section showing how that loader can be used. At the core of using each loader is a `download_loader` function, which downloads the loader file into a module that you can use within your application.

Example usage:

```
from llama_index import GPTSimpleVectorIndex, download_loader

GoogleDocsReader = download_loader('GoogleDocsReader')

gdoc_ids = ['1wf-y2pd9C878Oh-FmLH7Q_BQkljdm6TQal-c1pUfrec']
loader = GoogleDocsReader()
```

(continues on next page)

(continued from previous page)

```
documents = loader.load_data(document_ids=gdoc_ids)
index = GPTSimpleVectorIndex.from_documents(documents)
index.query('Where did the author go to school?')
```

3.9 Index Structures

At the core of LlamaIndex is a set of index data structures. You can choose to use them on their own, or you can choose to compose a graph over these data structures.

In the following sections, we detail how each index structure works, as well as some of the key capabilities our indices/graphs provide.

3.9.1 Updating an Index

Every LlamaIndex data structure allows **insertion**, **deletion**, and **update**.

Insertion

You can "insert" a new Document into any index data structure, after building the index initially. The underlying mechanism behind insertion depends on the index structure. For instance, for the list index, a new Document is inserted as additional node(s) in the list. For the vector store index, a new Document (and embedding) is inserted into the underlying document/embedding store.

An example notebook showcasing our insert capabilities is given here. In this notebook we showcase how to construct an empty index, manually create Document objects, and add those to our index data structures.

An example code snippet is given below:

```
index = GPTListIndex([])

embed_model = OpenAIEmbedding()
doc_chunks = []
for i, text in enumerate(text_chunks):
    doc = Document(text, doc_id=f"doc_id_{i}")
    doc_chunks.append(doc)

# insert
for doc_chunk in doc_chunks:
    index.insert(doc_chunk)
```

Deletion

You can "delete" a Document from most index data structures by specifying a document_id. (**NOTE**: the tree index currently does not support deletion). All nodes corresponding to the document will be deleted.

NOTE: In order to delete a Document, that Document must have a doc_id specified when first loaded into the index.

```
index.delete("doc_id_0")
```

Update

If a Document is already present within an index, you can "update" a Document with the same `doc_id` (for instance, if the information in the Document has changed).

```
# NOTE: the document has a `doc_id` specified
index.update(doc_chunks[0])
```

3.9.2 Composability

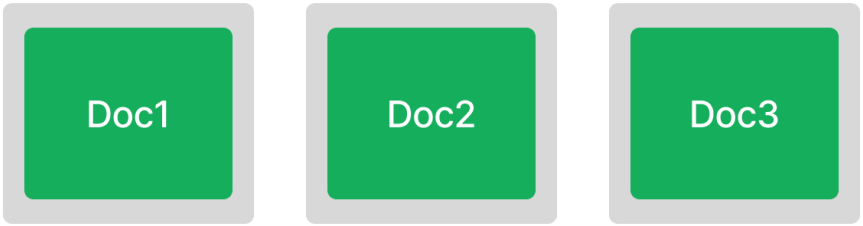
LlamaIndex offers **composability** of your indices, meaning that you can build indices on top of other indices. This allows you to more effectively index your entire document tree in order to feed custom knowledge to GPT.

Composability allows you to to define lower-level indices for each document, and higher-order indices over a collection of documents. To see how this works, imagine defining 1) a tree index for the text within each document, and 2) a list index over each tree index (one document) within your collection.

Defining Subindices

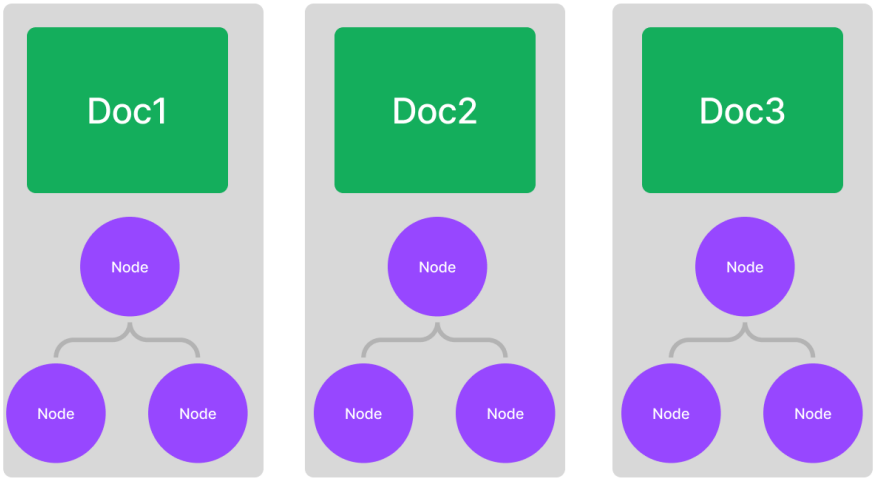
To see how this works, imagine you have 3 documents: `doc1`, `doc2`, and `doc3`.

```
doc1 = SimpleDirectoryReader('data1').load_data()
doc2 = SimpleDirectoryReader('data2').load_data()
doc3 = SimpleDirectoryReader('data3').load_data()
```



Now let's define a tree index for each document. In Python, we have:

```
index1 = GPTTreeIndex.from_documents(doc1)
index2 = GPTTreeIndex.from_documents(doc2)
index3 = GPTTreeIndex.from_documents(doc3)
```



Defining Summary Text

You then need to explicitly define *summary text* for each subindex. This allows the subindices to be used as Documents for higher-level indices.

```
index1_summary = "<summary1>"
index2_summary = "<summary2>"
index3_summary = "<summary3>"
```

You may choose to manually specify the summary text, or use LlamaIndex itself to generate a summary, for instance with the following:

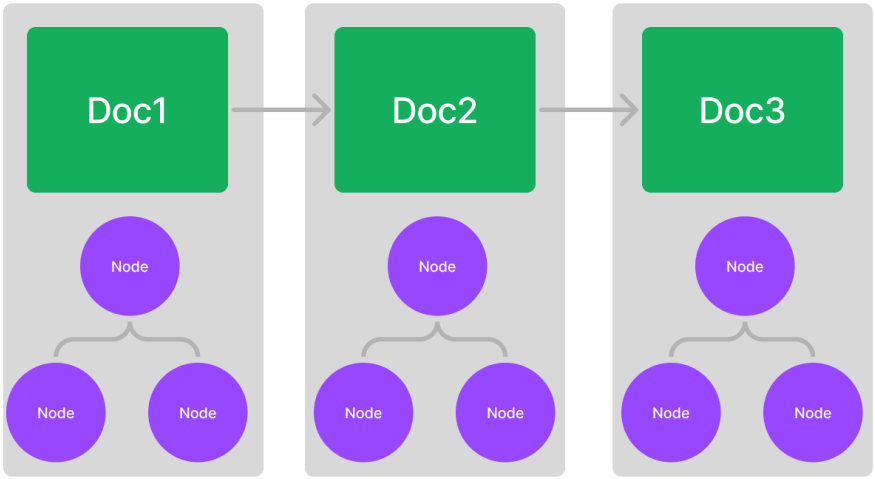
```
summary = index1.query(
    "What is a summary of this document?", mode="summarize"
)
index1_summary = str(summary)
```

If specified, this summary text for each subindex can be used to refine the answer during query-time.

Creating a Graph with a Top-Level Index

We can then create a graph with a list index on top of these 3 tree indices: We can query, save, and load the graph to/from disk as any other index.

```
from llama_index.indices.composability import ComposableGraph

graph = ComposableGraph.from_indices(
    GPTListIndex,
    [index1, index2, index3],
    index_summaries=[index1_summary, index2_summary, index3_summary],
)

# [Optional] save to disk
graph.save_to_disk("save_path.json")

# [Optional] load from disk
graph = ComposableGraph.load_from_disk("save_path.json")
```



Querying the Graph

During a query, we would start with the top-level list index. Each node in the list corresponds to an underlying tree index. We want to make sure that we define a **recursive** query, as well as a **query config** list. If the query config list is not provided, a default set will be used. Information on how to specify query configs (either as a list of JSON dicts or `QueryConfig` objects) can be found *here*.

```
# set query config. An example is provided below
query_configs = [
    {
        # NOTE: index_struct_id is optional
        "index_struct_id": "<index_id_1>",
        "index_struct_type": "tree",
        "query_mode": "default",
        "query_kwargs": {
            "child_branch_factor": 2
        }
    },
    {
        "index_struct_type": "keyword_table",
        "query_mode": "simple",
        "query_kwargs": {}
    },
    ...
]
```

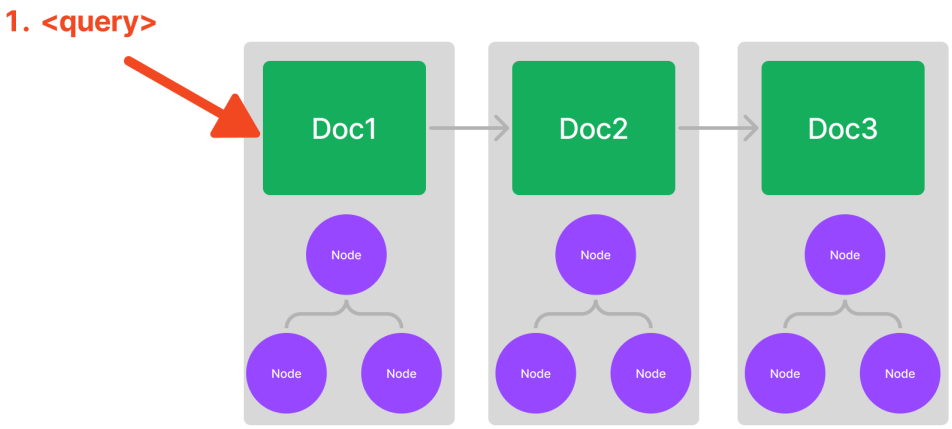
(continues on next page)

(continued from previous page)

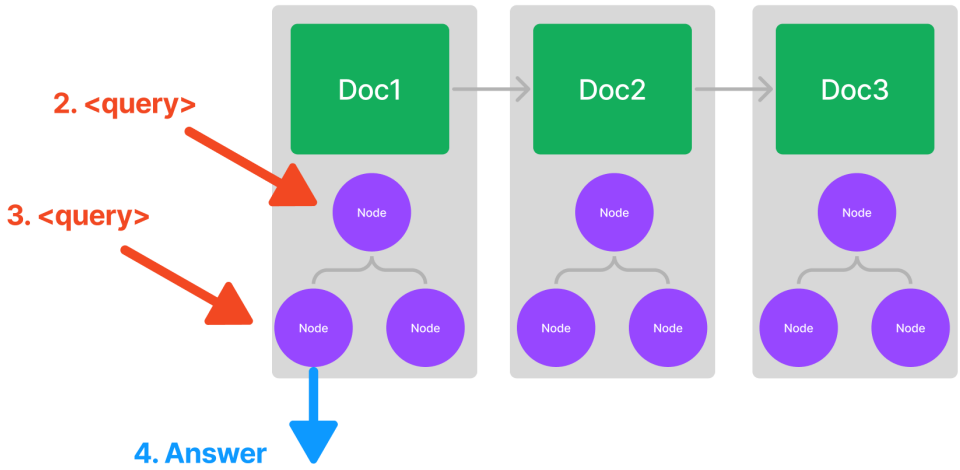
```
response = graph.query("Where did the author grow up?", query_configs=query_configs)
```

Note that specifying query config for index struct by id might require you to inspect e.g., `index1.index_struct.index_id`. Alternatively, you can explicitly set it as follows:

```
index1.index_struct.index_id = "<index_id_1>"
index2.index_struct.index_id = "<index_id_2>"
index3.index_struct.index_id = "<index_id_3>"
```



So within a node, instead of fetching the text, we would recursively query the stored tree index to retrieve our answer.



NOTE: You can stack indices as many times as you want, depending on the hierarchies of your knowledge base!

We can take a look at a code example below as well. We first build two tree indices, one over the Wikipedia NYC page, and the other over Paul Graham's essay. We then define a keyword extractor index over the two tree indices.

Here is an example notebook.

3.10 Query Interface

LlamaIndex provides a *query interface* over your index or graph structure. This query interface allows you to both retrieve the set of relevant documents, as well as synthesize a response.

- The basic query interface is found in our usage pattern guide. The guide details how to specify parameters for a basic query over a single index structure.
- A more advanced query interface is found in our composability guide. The guide describes how to specify a graph over multiple index structures.
- Finally, we provide a guide to our **Query Transformations** module.

3.10.1 Query Transformations

LlamaIndex allows you to perform *query transformations* over your index structures. Query transformations are modules that will convert a query into another query. They can be **single-step**, as in the transformation is run once before the query is executed against an index.

They can also be **multi-step**, as in:

1. The query is transformed, executed against an index,
2. The response is retrieved.
3. Subsequent queries are transformed/executed in a sequential fashion.

We list some of our query transformations in more detail below.

Use Cases

Query transformations have multiple use cases:

- Transforming an initial query into a form that can be more easily embedded (e.g. HyDE)
- Transforming an initial query into a subquestion that can be more easily answered from the data (single-step query decomposition)
- Breaking an initial query into multiple subquestions that can be more easily answered on their own. (multi-step query decomposition)

HyDE (Hypothetical Document Embeddings)

HyDE is a technique where given a natural language query, a hypothetical document/answer is generated first. This hypothetical document is then used for embedding lookup rather than the raw query.

To use HyDE, an example code snippet is shown below.

```
from llama_index import GPTSimpleVectorIndex, SimpleDirectoryReader
from llama_index.indices.query.query_transform.base import HyDEQueryTransform

# load documents, build index
documents = SimpleDirectoryReader('../paul_graham_essay/data').load_data()
index = GPTSimpleVectorIndex(documents)

# run query with HyDE query transform
query_str = "what did paul graham do after going to RISD"
hyde = HyDEQueryTransform(include_original=True)
response = index.query(query_str, query_transform=hyde)
print(response)
```

Check out our example notebook for a full walkthrough.

Single-Step Query Decomposition

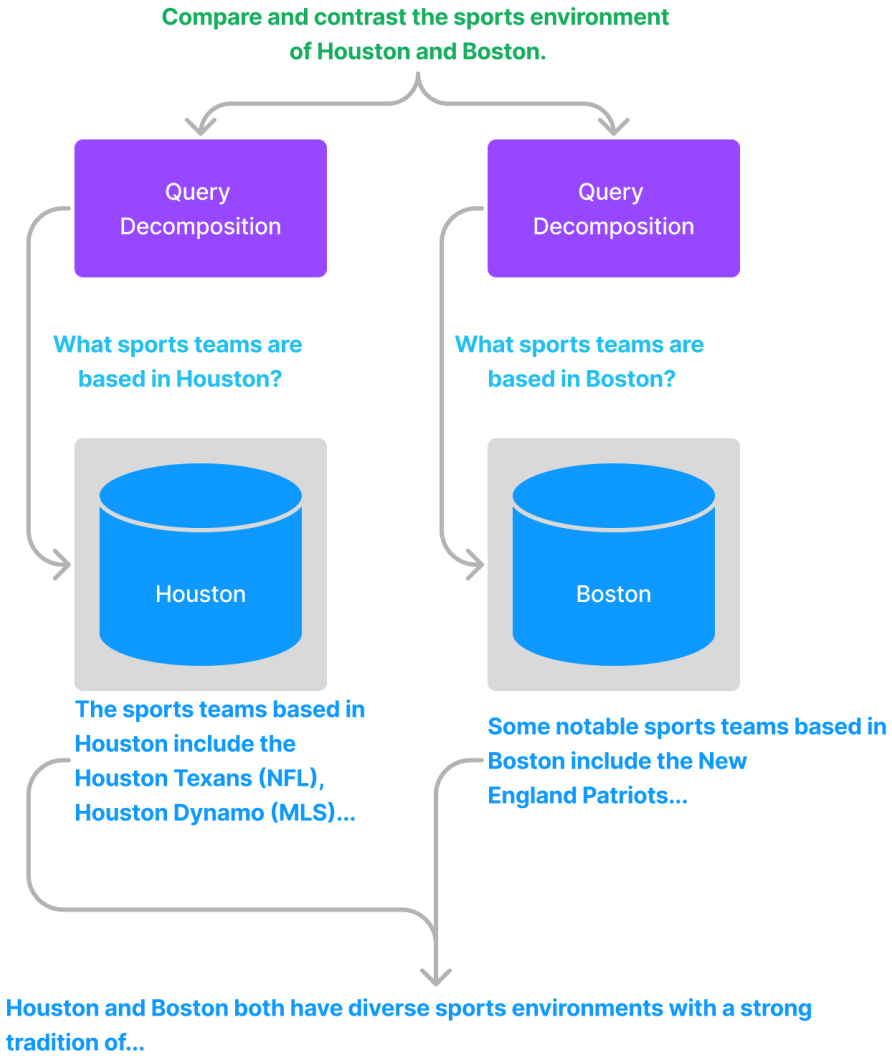
Some recent approaches (e.g. self-ask, ReAct) have suggested that LLM's perform better at answering complex questions when they break the question into smaller steps. We have found that this is true for queries that require knowledge augmentation as well.

If your query is complex, different parts of your knowledge base may answer different "subqueries" around the overall query.

Our single-step query decomposition feature transforms a **complicated** question into a simpler one over the data collection to help provide a sub-answer to the original question.

This is especially helpful over a *composed graph*. Within a composed graph, a query can be routed to multiple subindexes, each representing a subset of the overall knowledge corpus. Query decomposition allows us to transform the query into a more suitable question over any given index.

An example image is shown below.



Here's a corresponding example code snippet over a composed graph.

```
# Setting: a list index composed over multiple vector indices
# llm_predictor_chatgpt corresponds to the ChatGPT LLM interface
from llama_index.indices.query.query_transform.base import DecomposeQueryTransform
decompose_transform = DecomposeQueryTransform(
    llm_predictor_chatgpt, verbose=True
```

(continues on next page)

(continued from previous page)

```
)

# initialize indexes and graph
...

# set query config
query_configs = [
    {
        "index_struct_type": "simple_dict",
        "query_mode": "default",
        "query_kwargs": {
            "similarity_top_k": 1
        },
        # NOTE: set query transform for subindices
        "query_transform": decompose_transform
    },
    {
        "index_struct_type": "keyword_table",
        "query_mode": "simple",
        "query_kwargs": {
            "response_mode": "tree_summarize",
            "verbose": True
        },
    },
]

query_str = (
    "Compare and contrast the airports in Seattle, Houston, and Toronto. "
)
response_chatgpt = graph.query(
    query_str,
    query_configs=query_configs,
    llm_predictor=llm_predictor_chatgpt
)
```

Check out our example notebook for a full walkthrough.

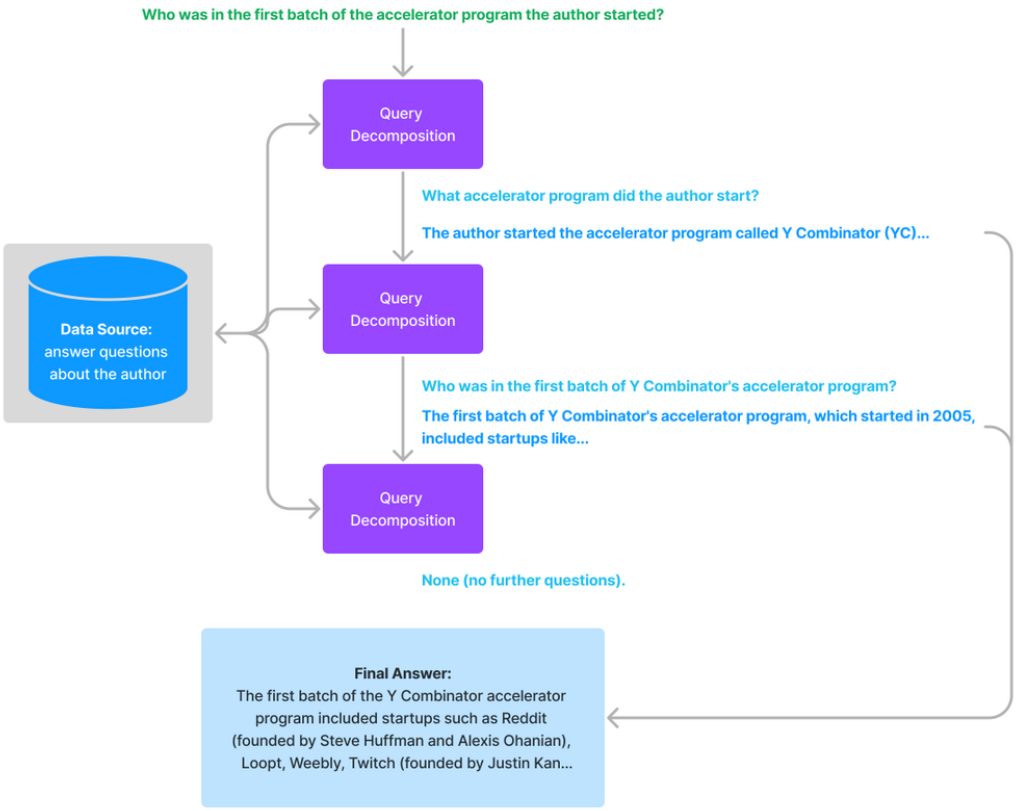
Multi-Step Query Transformations

Multi-step query transformations are a generalization on top of existing single-step query transformation approaches.

Given an initial, complex query, the query is transformed and executed against an index. The response is retrieved from the query. Given the response (along with prior responses) and the query, followup questions may be asked against the index as well. This technique allows a query to be run against a single knowledge source until that query has satisfied all questions.

We have an additional `QueryCombiner` class that runs queries against a given index in a sequential fashion, allowing subsequent queries to be "followup" questions. At the moment, the `QueryCombiner` class is not yet exposed to the user. Coming soon!

An example image is shown below.



Here's a corresponding example code snippet.

```
from llama_index.indices.query.query_transform.base import StepDecomposeQueryTransform
# gpt-4
step_decompose_transform = StepDecomposeQueryTransform(
    llm_predictor, verbose=True
)

response = index.query(
    "Who was in the first batch of the accelerator program the author started?",
    query_transform=step_decompose_transform,
)
print(str(response))
```

Check out our example notebook for a full walkthrough.

3.10.2 Node Postprocessor

By default, when a query is executed on an index or a composed graph, LlamaIndex performs the following steps:

1. **Retrieval step**: Retrieve a set of nodes from the index given the query. For instance, with a vector index, this would be top-k relevant nodes; with a list index this would be all nodes.
2. **Synthesis step**: Synthesize a response over the set of nodes.

LlamaIndex provides a set of "postprocessor" modules that can augment the retrieval process in (1). The process is very simple. After the retrieval step, we can analyze the initial set of nodes and add a "processing" step to refine this set of nodes - whether its by filtering out irrelevant nodes, adding more nodes, and more.

This is a simple but powerful step. This allows us to perform tasks like keyword filtering, as well as temporal reasoning over your data.

We first provide the high-level API interface, and provide some example modules, and finally discuss usage.

We are also very open to contributions! Take a look at our contribution guide if you are interested in contributing a Postprocessor.

API Interface

The base class is `BaseNodePostprocessor`, and the API interface is very simple:

```
class BaseNodePostprocessor(BasePostprocessor, BaseModel):
    """Node postprocessor."""

    @abstractmethod
    def postprocess_nodes(
        self, nodes: List[Node], extra_info: Optional[Dict] = None
    ) -> List[Node]:
        """Postprocess nodes."""
```

It takes in a list of Node objects, and outputs another list of Node objects.

The full API reference can be found *here*.

Example Usage

The postprocessor can be used as part of an `index.query` call, or on its own.

Index querying

```
from gpt_index.indices.postprocessor import (
    FixedRecencyPostprocessor,
)
node_postprocessor = FixedRecencyPostprocessor(service_context=service_context)

response = index.query(
    "How much did the author raise in seed funding from Idelle's husband (Julian) for␣
↪Viaweb?",
```

(continues on next page)

(continued from previous page)

```
    similarity_top_k=3,
    node_postprocessors=[node_postprocessor]
)
```

Using as Independent Module (Lower-Level Usage)

The module can also be used on its own as part of a broader flow. For instance, here's an example where you choose to manually postprocess an initial set of source nodes.

```
from gpt_index.indices.postprocessor import (
    FixedRecencyPostprocessor,
)

# get initial response from vector index
init_response = index.query(
    query_str,
    similarity_top_k=3,
    response_mode="no_text"
)
resp_nodes = [n.node for n in init_response.source_nodes]

# use node postprocessor to filter nodes
node_postprocessor = FixedRecencyPostprocessor(service_context=service_context)
new_nodes = node_postprocessor.postprocess_nodes(resp_nodes)

# use list index to synthesize answers
list_index = GPTListIndex(new_nodes)
response = list_index.query(query_str, node_postprocessors=[node_postprocessor])
```

Example Modules

Default Postprocessors

These postprocessors are simple modules that are already included by default during an `index.query` call

KeywordNodePostprocessor

A simple postprocessor module where you are able to specify `required_keywords` or `exclude_keywords`. This will filter out nodes that don't have required keywords, or contain excluded keywords.

SimilarityPostprocessor

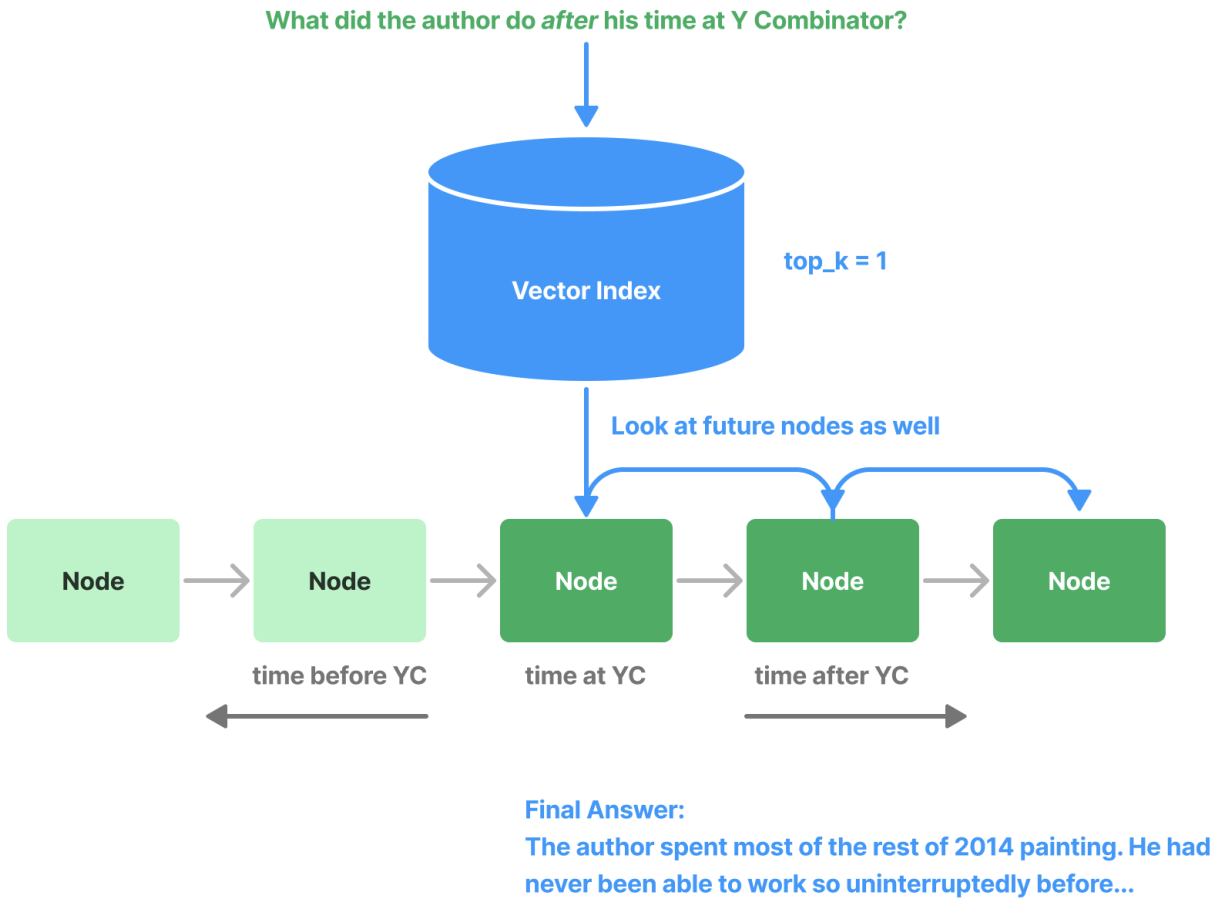
A module where you are able to specify a `similarity_cutoff`.

Previous/Next Postprocessors

These postprocessors are able to exploit temporal relationships between nodes (e.g. prev/next relationships) in order to retrieve additional context, in the event that the existing context may not directly answer the question. They augment the set of retrieved nodes with context either in the future or the past (or both).

The most basic version is `PrevNextNodePostprocessor`, which takes a fixed `num_nodes` as well as `mode` specifying "previous", "next", or "both".

We also have `AutoPrevNextNodePostprocessor`, which is able to infer the `previous`, `next` direction.



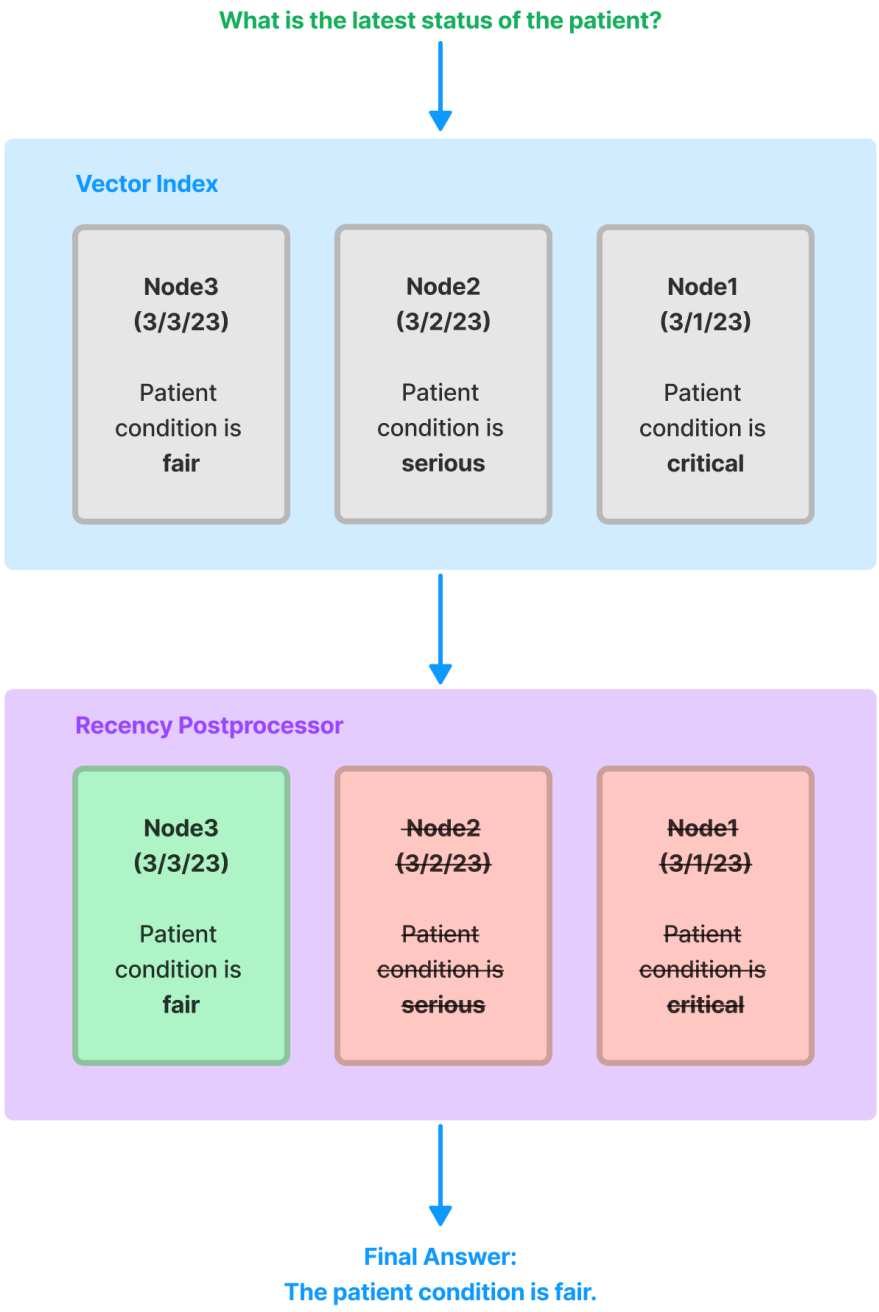
Recency Postprocessors

These postprocessors are able to ensure that only the most recent data is used as context, and that out of date context information is filtered out.

Imagine that you have three versions of a document, with slight changes between versions. For instance, this document may be describing patient history. If you ask a question over this data, you would want to make sure that you're referencing the latest document, and that out of date information is not passed in.

We support recency filtering through the following modules.

`FixedRecencyPostProcessor`: sorts retrieved nodes by date in reverse order, and takes a fixed top-k set of nodes.



`EmbeddingRecencyPostprocessor`: sorts retrieved nodes by date in reverse order, and then looks at subsequent

nodes and filters out nodes that have high embedding similarity with the current node. This allows us to maintain recent Nodes that have "distinct" context, but filter out overlapping Nodes that are outdated and overlap with more recent context.

`TimeWeightedPostprocessor`: adds time-weighting to retrieved nodes, using the formula `(1-time_decay) ** hours_passed`. The recency score is added to any score that the node already contains.

3.11 Customization

LlamaIndex provides the ability to customize the following components:

- LLM
- Prompts
- Embedding model

These are described in their respective guides below.

3.11.1 Defining LLMs

The goal of LlamaIndex is to provide a toolkit of data structures that can organize external information in a manner that is easily compatible with the prompt limitations of an LLM. Therefore LLMs are always used to construct the final answer. Depending on the *type of index* being used, LLMs may also be used during index construction, insertion, and query traversal.

LlamaIndex uses Langchain's LLM and LLMChain module to define the underlying abstraction. We introduce a wrapper class, *`LLMPredictor`*, for integration into LlamaIndex.

We also introduce a *`PromptHelper` class*, to allow the user to explicitly set certain constraint parameters, such as maximum input size (default is 4096 for davinci models), number of generated output tokens, maximum chunk overlap, and more.

By default, we use OpenAI's `text-davinci-003` model. But you may choose to customize the underlying LLM being used.

Below we show a few examples of LLM customization. This includes

- changing the underlying LLM
- changing the number of output tokens (for OpenAI, Cohere, or AI21)
- having more fine-grained control over all parameters for any LLM, from input size to chunk overlap

Example: Changing the underlying LLM

An example snippet of customizing the LLM being used is shown below. In this example, we use `text-davinci-002` instead of `text-davinci-003`. Available models include `text-davinci-003`,`text-curie-001`,`text-babbage-001`,`text-ada-001`, `code-davinci-002`,`code-cushman-001`. Note that you may plug in any LLM shown on Langchain's LLM page.

```
from llama_index import (
    GPTKeywordTableIndex,
    SimpleDirectoryReader,
```

(continues on next page)

(continued from previous page)

```
    LLMPredictor,
    ServiceContext
)
from langchain import OpenAI

documents = SimpleDirectoryReader('data').load_data()

# define LLM
llm_predictor = LLMPredictor(llm=OpenAI(temperature=0, model_name="text-davinci-002"))
service_context = ServiceContext.from_defaults(llm_predictor=llm_predictor)

# build index
index = GPTKeywordTableIndex.from_documents(documents, service_context=service_context)

# get response from query
response = index.query("What did the author do after his time at Y Combinator?")
```

Example: Changing the number of output tokens (for OpenAI, Cohere, AI21)

The number of output tokens is usually set to some low number by default (for instance, with OpenAI the default is 256).

For OpenAI, Cohere, AI21, you just need to set the `max_tokens` parameter (or maxTokens for AI21). We will handle text chunking/calculations under the hood.

```
from llama_index import (
    GPTKeywordTableIndex,
    SimpleDirectoryReader,
    LLMPredictor,
    ServiceContext
)
from langchain import OpenAI

documents = SimpleDirectoryReader('data').load_data()

# define LLM
llm_predictor = LLMPredictor(llm=OpenAI(temperature=0, model_name="text-davinci-002",␣
↪max_tokens=512))
service_context = ServiceContext.from_defaults(llm_predictor=llm_predictor)

# build index
index = GPTKeywordTableIndex.from_documents(documents, service_context=service_context)

# get response from query
response = index.query("What did the author do after his time at Y Combinator?")
```

If you are using other LLM classes from langchain, please see below.

Example: Fine-grained control over all parameters

To have fine-grained control over all parameters, you will need to define a custom PromptHelper class.

```
from llama_index import (
    GPTKeywordTableIndex,
    SimpleDirectoryReader,
    LLMPredictor,
    PromptHelper,
    ServiceContext
)
from langchain import OpenAI

documents = SimpleDirectoryReader('data').load_data()


# define prompt helper
# set maximum input size
max_input_size = 4096
# set number of output tokens
num_output = 256
# set maximum chunk overlap
max_chunk_overlap = 20
prompt_helper = PromptHelper(max_input_size, num_output, max_chunk_overlap)

# define LLM
llm_predictor = LLMPredictor(llm=OpenAI(temperature=0, model_name="text-davinci-002",␣
↪max_tokens=num_output))

service_context = ServiceContext.from_defaults(llm_predictor=llm_predictor, prompt_
↪helper=prompt_helper)

# build index
index = GPTKeywordTableIndex.from_documents(documents, service_context=service_context)

# get response from query
response = index.query("What did the author do after his time at Y Combinator?")
```

Example: Using a Custom LLM Model

To use a custom LLM model, you only need to implement the `LLM` class from Langchain. You will be responsible for passing the text to the model and returning the newly generated tokens.

Here is a small example using locally running FLAN-T5 model and Huggingface's pipeline abstraction:

```
import torch
from langchain.llms.base import LLM
from llama_index import SimpleDirectoryReader, LangchainEmbedding, GPTListIndex,␣
↪PromptHelper
from llama_index import LLMPredictor, ServiceContext
from transformers import pipeline
```

(continues on next page)

(continued from previous page)

```
from typing import Optional, List, Mapping, Any


# define prompt helper
# set maximum input size
max_input_size = 2048
# set number of output tokens
num_output = 256
# set maximum chunk overlap
max_chunk_overlap = 20
prompt_helper = PromptHelper(max_input_size, num_output, max_chunk_overlap)


class CustomLLM(LLM):
    model_name = "facebook/opt-iml-max-30b"
    pipeline = pipeline("text-generation", model=model_name, device="cuda:0", model_
↪kwargs={"torch_dtype":torch.bfloat16})

    def _call(self, prompt: str, stop: Optional[List[str]] = None) -> str:
        prompt_length = len(prompt)
        response = self.pipeline(prompt, max_new_tokens=num_output)[0]["generated_text"]

        # only return newly generated tokens
        return response[prompt_length:]

    @property
    def _identifying_params(self) -> Mapping[str, Any]:
        return {"name_of_model": self.model_name}

    @property
    def _llm_type(self) -> str:
        return "custom"

# define our LLM
llm_predictor = LLMPredictor(llm=CustomLLM())

service_context = ServiceContext.from_defaults(llm_predictor=llm_predictor, prompt_
↪helper=prompt_helper)

# Load the your data
documents = SimpleDirectoryReader('./data').load_data()
index = GPTListIndex.from_documents(documents, service_context=service_context)

# Query and print response
response = index.query("<query_text>")
print(response)
```

Using this method, you can use any LLM. Maybe you have one running locally, or running on your own server. As long as the class is implemented and the generated tokens are returned, it should work out. Note that we need to use the prompt helper to customize the prompt sizes, since every model has a slightly different context length.

Note that you may have to adjust the internal prompts to get good performance. Even then, you should be using a sufficiently large LLM to ensure it's capable of handling the complex queries that LlamaIndex uses internally, so your

mileage may vary.

A list of all default internal prompts is available here, and chat-specific prompts are listed here. You can also implement your own custom prompts, as described here.

3.11.2 Defining Prompts

Prompting is the fundamental input that gives LLMs their expressive power. LlamaIndex uses prompts to build the index, do insertion, perform traversal during querying, and to synthesize the final answer.

LlamaIndex uses a finite set of *prompt types*, described *here*. All index classes, along with their associated queries, utilize a subset of these prompts. The user may provide their own prompt. If the user does not provide their own prompt, default prompts are used.

NOTE: The majority of custom prompts are typically passed in during **query-time**, not during **index construction**. For instance, both the `QuestionAnswerPrompt` and `RefinePrompt` are used during query-time to synthesize an answer. Some indices do use prompts during index construction to build the index; for instance, `GPTTreeIndex` uses a `SummaryPrompt` to hierarchically summarize the nodes, and `GPTKeywordTableIndex` uses a `KeywordExtractPrompt` to extract keywords. Some indices do allow `QuestionAnswerPrompt` and `RefinePrompt` to be passed in during index construction, but that usage is deprecated.

An API reference of all query classes and index classes (used for index construction) are found below. The definition of each query class and index class contains optional prompts that the user may pass in.

- *Queries*
- *Indices*

Example

An example can be found in this notebook.

A corresponding snippet is below. We show how to define a custom `QuestionAnswer` prompt which requires both a `context_str` and `query_str` field. The prompt is passed in during query-time.

```
from llama_index import QuestionAnswerPrompt, GPTSimpleVectorIndex, SimpleDirectoryReader

# load documents
documents = SimpleDirectoryReader('data').load_data()

# define custom QuestionAnswerPrompt
query_str = "What did the author do growing up?"
QA_PROMPT_TMPL = (
    "We have provided context information below. \n"
    "---------------------\n"
    "{context_str}"
    "\n---------------------\n"
    "Given this information, please answer the question: {query_str}\n"
)
QA_PROMPT = QuestionAnswerPrompt(QA_PROMPT_TMPL)
# Build GPTSimpleVectorIndex
index = GPTSimpleVectorIndex.from_documents(documents)

response = index.query(query_str, text_qa_template=QA_PROMPT)
```

(continues on next page)

(continued from previous page)

```
print(response)
```

Check out the *reference documentation* for a full set of all prompts.

3.11.3 Embedding support

LlamaIndex provides support for embeddings in the following format:

- Adding embeddings to Document objects
- Using a Vector Store as an underlying index (e.g. `GPTSimpleVectorIndex`, `GPTFaissIndex`)
- Querying our list and tree indices with embeddings.

Adding embeddings to Document objects

You can pass in user-specified embeddings when constructing an index. This gives you control in specifying embeddings per Document instead of having us determine embeddings for your text (see below).

Simply specify the `embedding` field when creating a Document:

Insert into Index and Query

```
from gpt_index import GPTListIndex, SimpleDirectoryReader
from gpt_index.embeddings.openai import OpenAIEmbedding
from IPython.display import Markdown, display
```

Initialize Blank List Index

```
index = GPTListIndex([])
```

```
> [build_index_from_documents] Total token usage: 0 tokens
```

Create collection of documents to insert

```
embed_model = OpenAIEmbedding()
doc_chunks = []
for i, text in enumerate(text_chunks):
    print(f"Getting embedding for chunk {i}")
    embedding = embed_model.get_text_embedding(text)
    doc = Document(text, embedding=embedding)
    doc_chunks.append(doc)
```

Insert New Document Chunks

```
for doc_chunk in doc_chunks:
    index.insert(doc_chunk)
```

```
# query
response = index.query("What did the author do growing up?", mode="embedding")
```

```
> Starting query: What did the author do growing up?
> [query] Total token usage: 1931 tokens
```

```
display(Markdown(f"<b>{response}</b>"))
```

The author grew up writing short stories and programming on an IBM 1401. He eventually convinced his father to buy him a TRS-80, and he wrote simple games, a program to predict how high his model rockets would fly, and a word processor. He then went to college to study philosophy, but switched to AI after becoming interested in a novel by Heinlein and a PBS documentary. He reverse-engineered SHRDLU for his undergraduate thesis and wrote a book about Lisp hacking while in grad school.

Using a Vector Store as an Underlying Index

Please see the corresponding section in our *Vector Stores* guide for more details.

Using an Embedding Query Mode in List/Tree Index

LlamaIndex provides embedding support to our tree and list indices. In addition to each node storing text, each node can optionally store an embedding. During query-time, we can use embeddings to do max-similarity retrieval of nodes before calling the LLM to synthesize an answer. Since similarity lookup using embeddings (e.g. using cosine similarity) does not require a LLM call, embeddings serve as a cheaper lookup mechanism instead of using LLMs to traverse nodes.

How are Embeddings Generated?

Since we offer embedding support during *query-time* for our list and tree indices, embeddings are lazily generated and then cached (if `mode="embedding"` is specified during `index.query(...)`), and not during index construction. This design choice prevents the need to generate embeddings for all text chunks during index construction.

NOTE: Our *vector-store based indices* generate embeddings during index construction.

Embedding Lookups

For the list index (`GPTListIndex`):

- We iterate through every node in the list, and identify the top k nodes through embedding similarity. We use these nodes to synthesize an answer.
- See the *List Query API* for more details.
- NOTE: the embedding-mode usage of the list index is roughly equivalent with the usage of our `GPTSimpleVectorIndex`; the main difference is when embeddings are generated (during query-time for the list index vs. index construction for the simple vector index).

For the tree index (`GPTTreeIndex`):

- We start with the root nodes, and traverse down the tree by picking the child node through embedding similarity.
- See the *Tree Query API* for more details.

Example Notebook

An example notebook is given here.

Custom Embeddings

LlamaIndex allows you to define custom embedding modules. By default, we use `text-embedding-ada-002` from OpenAI.

You can also choose to plug in embeddings from Langchain's embeddings module. We introduce a wrapper class, *`LangchainEmbedding`*, for integration into LlamaIndex.

An example snippet is shown below (to use Hugging Face embeddings) on the GPTListIndex:

```
from llama_index import GPTListIndex, SimpleDirectoryReader
from langchain.embeddings.huggingface import HuggingFaceEmbeddings
from llama_index import LangchainEmbedding, ServiceContext
```

(continues on next page)

(continued from previous page)

```
# load in HF embedding model from langchain
embed_model = LangchainEmbedding(HuggingFaceEmbeddings())
service_context = ServiceContext.from_defaults(embed_model=embed_model)

# load index
new_index = GPTListIndex.load_from_disk('index_list_emb.json')

# query with embed_model specified
response = new_index.query(
    "<query_text>",
    mode="embedding",
    verbose=True,
    service_context=service_context
)
print(response)
```

Another example snippet is shown for GPTSimpleVectorIndex.

```
from llama_index import GPTSimpleVectorIndex, SimpleDirectoryReader
from langchain.embeddings.huggingface import HuggingFaceEmbeddings
from llama_index import LangchainEmbedding, ServiceContext

# load in HF embedding model from langchain
embed_model = LangchainEmbedding(HuggingFaceEmbeddings())
service_context = ServiceContext.from_defaults(embed_model=embed_model)

# load index
new_index = GPTSimpleVectorIndex.load_from_disk(
    'index_simple_vector.json',
    service_context=service_context
)

# query will use the same embed_model
response = new_index.query(
    "<query_text>",
    mode="default",
    verbose=True,
)
print(response)
```

3.12 Analysis and Optimization

LlamaIndex provides a variety of tools for analysis and optimization of your indices and queries. Some of our tools involve the analysis/ optimization of token usage and cost.

We also offer a Playground module, giving you a visual means of analyzing the token usage of various index structures + performance.

3.12.1 Cost Analysis

Each call to an LLM will cost some amount of money - for instance, OpenAI's Davinci costs $0.02 / 1k tokens. The cost of building an index and querying depends on

- the type of LLM used
- the type of data structure used
- parameters used during building
- parameters used during querying

The cost of building and querying each index is a TODO in the reference documentation. In the meantime, we provide the following information:

1. A high-level overview of the cost structure of the indices.
2. A token predictor that you can use directly within LlamaIndex!

Overview of Cost Structure

Indices with no LLM calls

The following indices don't require LLM calls at all during building (0 cost):

- `GPTListIndex`
- `GPTSimpleKeywordTableIndex` - uses a regex keyword extractor to extract keywords from each document
- `GPTRAKEKeywordTableIndex` - uses a RAKE keyword extractor to extract keywords from each document

Indices with LLM calls

The following indices do require LLM calls during build time:

- `GPTTreeIndex` - use LLM to hierarchically summarize the text to build the tree
- `GPTKeywordTableIndex` - use LLM to extract keywords from each document

Query Time

There will always be >= 1 LLM call during query time, in order to synthesize the final answer. Some indices contain cost tradeoffs between index building and querying. `GPTListIndex`, for instance, is free to build, but running a query over a list index (without filtering or embedding lookups), will call the LLM N times.

Here are some notes regarding each of the indices:

- `GPTListIndex`: by default requires N LLM calls, where N is the number of nodes.
  - However, can do `index.query(..., keyword="<keyword>")` to filter out nodes that don't contain the keyword
- `GPTTreeIndex`: by default requires $\log(N)$ LLM calls, where N is the number of leaf nodes.
  - Setting `child_branch_factor=2` will be more expensive than the default `child_branch_factor=1` (polynomial vs logarithmic), because we traverse 2 children instead of just 1 for each parent node.
- `GPTKeywordTableIndex`: by default requires an LLM call to extract query keywords.

- Can do `index.query(..., mode="simple")` or `index.query(..., mode="rake")` to also use regex/RAKE keyword extractors on your query text.

Token Predictor Usage

LlamaIndex offers token **predictors** to predict token usage of LLM and embedding calls. This allows you to estimate your costs during 1) index construction, and 2) index querying, before any respective LLM calls are made.

Using MockLLMPredictor

To predict token usage of LLM calls, import and instantiate the MockLLMPredictor with the following:

```
from llama_index import MockLLMPredictor, ServiceContext

llm_predictor = MockLLMPredictor(max_tokens=256)
```

You can then use this predictor during both index construction and querying. Examples are given below.

Index Construction

```
from llama_index import GPTTreeIndex, MockLLMPredictor, SimpleDirectoryReader

documents = SimpleDirectoryReader('../paul_graham_essay/data').load_data()
# the "mock" llm predictor is our token counter
llm_predictor = MockLLMPredictor(max_tokens=256)
service_context = ServiceContext.from_defaults(llm_predictor=llm_predictor)
# pass the "mock" llm_predictor into GPTTreeIndex during index construction
index = GPTTreeIndex.from_documents(documents, service_context=service_context)

# get number of tokens used
print(llm_predictor.last_token_usage)
```

Index Querying

```
response = index.query("What did the author do growing up?", service_context=service_
↪context)

# get number of tokens used
print(llm_predictor.last_token_usage)
```

Using MockEmbedding

You may also predict the token usage of embedding calls with `MockEmbedding`. You can use it in tandem with `MockLLMPredictor`.

```
from llama_index import (
    GPTSimpleVectorIndex,
    MockLLMPredictor,
    MockEmbedding,
    SimpleDirectoryReader,
    ServiceContext
```

(continues on next page)

(continued from previous page)

```
)

documents = SimpleDirectoryReader('../paul_graham_essay/data').load_data()
index = GPTSimpleVectorIndex.load_from_disk('../paul_graham_essay/index_simple_vec.json')

# specify both a MockLLMPredictor as wel as MockEmbedding
llm_predictor = MockLLMPredictor(max_tokens=256)
embed_model = MockEmbedding(embed_dim=1536)
service_context = ServiceContext.from_defaults(llm_predictor=llm_predictor, embed_
↪model=embed_model)

response = index.query(
    "What did the author do after his time at Y Combinator?",
    service_context=service_context
)
```

Here is an example notebook.

3.12.2 Playground

The Playground module in LlamaIndex is a way to automatically test your data (i.e. documents) across a diverse combination of indices, models, embeddings, modes, etc. to decide which ones are best for your purposes. More options will continue to be added.

For each combination, you'll be able to compare the results for any query and compare the answers, latency, tokens used, and so on.

You may initialize a Playground with a list of pre-built indices, or initialize one from a list of Documents using the preset indices.

Sample Code

A sample usage is given below.

```
from llama_index import download_loader
from llama_index.indices.vector_store import GPTSimpleVectorIndex
from llama_index.indices.tree.base import GPTTreeIndex
from llama_index.playground import Playground

# load data
WikipediaReader = download_loader("WikipediaReader")
loader = WikipediaReader()
documents = loader.load_data(pages=['Berlin'])

# define multiple index data structures (vector index, list index)
indices = [GPTSimpleVectorIndex(documents), GPTTreeIndex(documents)]

# initialize playground
playground = Playground(indices=indices)

# playground compare
```

(continues on next page)

(continued from previous page)

```
playground.compare("What is the population of Berlin?")
```

API Reference

API Reference here

Example Notebook

Link to Example Notebook.

3.12.3 Optimizers

NOTE: We'll be adding more to this section soon!

Our optimizers module consists of ways for users to optimize for token usage (we are currently exploring ways to expand optimization capabilities to other areas, such as performance!)

Here is a sample code snippet on comparing the outputs without optimization and with.

```
from llama_index import GPTSimpleVectorIndex
from llama_index.optimization.optimizer import SentenceEmbeddingOptimizer
# load from disk
index = GPTSimpleVectorIndex.load_from_disk('simple_vector_index.json')

print("Without optimization")
start_time = time.time()
res = index.query("What is the population of Berlin?")
end_time = time.time()
print("Total time elapsed: {}".format(end_time - start_time))
print("Answer: {}".format(res))

print("With optimization")
start_time = time.time()
res = index.query("What is the population of Berlin?",␣
↪optimizer=SentenceEmbeddingOptimizer(percentile_cutoff=0.5))
end_time = time.time()
print("Total time elapsed: {}".format(end_time - start_time))
print("Answer: {}".format(res))
```

Output:

```
Without optimization
INFO:root:> [query] Total LLM token usage: 3545 tokens
INFO:root:> [query] Total embedding token usage: 7 tokens
Total time elapsed: 2.8928110599517822
Answer:
The population of Berlin in 1949 was approximately 2.2 million inhabitants. After the␣
↪fall of the Berlin Wall in 1989, the population of Berlin increased to approximately 3.
↪7 million inhabitants.
```

(continues on next page)

(continued from previous page)

```
With optimization
INFO:root:> [optimize] Total embedding token usage: 7 tokens
INFO:root:> [query] Total LLM token usage: 1779 tokens
INFO:root:> [query] Total embedding token usage: 7 tokens
Total time elapsed: 2.346346139907837
Answer:
The population of Berlin is around 4.5 million.
```

Full example notebook here.

API Reference

An API reference can be found *here*.

3.13 Output Parsing

LLM output/validation capabilities are crucial to LlamaIndex in the following areas:

- **Document retrieval**: Many data structures within LlamaIndex rely on LLM calls with a specific schema for Document retrieval. For instance, the tree index expects LLM calls to be in the format "ANSWER: (number)".
- **Response synthesis**: Users may expect that the final response contains some degree of structure (e.g. a JSON output, a formatted SQL query, etc.)

LlamaIndex supports integrations with output parsing modules offered by other frameworks. These output parsing modules can be used in the following ways:

- To provide formatting instructions for any prompt / query (through `output_parser.format`)
- To provide "parsing" for LLM outputs (through `output_parser.parse`)

3.13.1 Guardrails

Guardrails is an open-source Python package for specification/validation/correction of output schemas. See below for a code example.

```
from llama_index import GPTSimpleVectorIndex, SimpleDirectoryReader
from llama_index.output_parsers import GuardrailsOutputParser
from llama_index.llm_predictor import StructuredLLMPredictor
from llama_index.prompts.prompts import QuestionAnswerPrompt, RefinePrompt
from llama_index.prompts.default_prompts import DEFAULT_TEXT_QA_PROMPT_TMPL, DEFAULT_
↪REFINE_PROMPT_TMPL


# load documents, build index
documents = SimpleDirectoryReader('../paul_graham_essay/data').load_data()
index = GPTSimpleVectorIndex(documents, chunk_size_limit=512)
llm_predictor = StructuredLLMPredictor()
```

(continues on next page)

(continued from previous page)

```
# specify StructuredLLMPredictor
# this is a special LLMPredictor that allows for structured outputs

# define query / output spec
rail_spec = ("""
<rail version="0.1">

<output>
    <list name="points" description="Bullet points regarding events in the author's life.
↪">
        <object>
            <string name="explanation" format="one-line" on-fail-one-line="noop" />
            <string name="explanation2" format="one-line" on-fail-one-line="noop" />
            <string name="explanation3" format="one-line" on-fail-one-line="noop" />
        </object>
    </list>
</output>

<prompt>

Query string here.

@xml_prefix_prompt

{output_schema}

@json_suffix_prompt_v2_wo_none
</prompt>
</rail>
""")

# define output parser
output_parser = GuardrailsOutputParser.from_rail_string(rail_spec, llm=llm_predictor.llm)

# format each prompt with output parser instructions
fmt_qa_tmpl = output_parser.format(DEFAULT_TEXT_QA_PROMPT_TMPL)
fmt_refine_tmpl = output_parser.format(DEFAULT_REFINE_PROMPT_TMPL)

qa_prompt = QuestionAnswerPrompt(fmt_qa_tmpl, output_parser=output_parser)
refine_prompt = RefinePrompt(fmt_refine_tmpl, output_parser=output_parser)

# obtain a structured response
response = index.query(
    "What are the three items the author did growing up?",
    text_qa_template=qa_prompt,
    refine_template=refine_prompt,
    llm_predictor=llm_predictor
)
print(response)
```

Output:

```
{'points': [{'explanation': 'Writing short stories', 'explanation2': 'Programming on an␣
↪IBM 1401', 'explanation3': 'Using microcomputers'}]}
```

3.13.2 Langchain

Langchain also offers output parsing modules that you can use within LlamaIndex.

```
from llama_index import GPTSimpleVectorIndex, SimpleDirectoryReader
from llama_index.output_parsers import LangchainOutputParser
from llama_index.llm_predictor import StructuredLLMPredictor
from llama_index.prompts.prompts import QuestionAnswerPrompt, RefinePrompt
from llama_index.prompts.default_prompts import DEFAULT_TEXT_QA_PROMPT_TMPL, DEFAULT_
↪REFINE_PROMPT_TMPL
from langchain.output_parsers import StructuredOutputParser, ResponseSchema


# load documents, build index
documents = SimpleDirectoryReader('../paul_graham_essay/data').load_data()
index = GPTSimpleVectorIndex(documents, chunk_size_limit=512)
llm_predictor = StructuredLLMPredictor()

# define output schema
response_schemas = [
    ResponseSchema(name="Education", description="Describes the author's educational␣
↪experience/background."),
    ResponseSchema(name="Work", description="Describes the author's work experience/
↪background.")
]

# define output parser
lc_output_parser = StructuredOutputParser.from_response_schemas(response_schemas)
output_parser = LangchainOutputParser(lc_output_parser)

# format each prompt with output parser instructions
fmt_qa_tmpl = output_parser.format(DEFAULT_TEXT_QA_PROMPT_TMPL)
fmt_refine_tmpl = output_parser.format(DEFAULT_REFINE_PROMPT_TMPL)
qa_prompt = QuestionAnswerPrompt(fmt_qa_tmpl, output_parser=output_parser)
refine_prompt = RefinePrompt(fmt_refine_tmpl, output_parser=output_parser)

# query index
response = index.query(
    "What are a few things the author did growing up?",
    text_qa_template=qa_prompt,
    refine_template=refine_prompt,
    llm_predictor=llm_predictor
)
print(str(response))
```

Output:

```
{'Education': 'Before college, the author wrote short stories and experimented with␣
↪programming on an IBM 1401.', 'Work': 'The author worked on writing and programming␣
```

(continues on next page)

(continued from previous page)

```
↪outside of school.'}
```

3.14 Evaluation

LlamaIndex offers a few key modules for evaluating the quality of both Document retrieval and response synthesis. Here are some key questions for each component:

- **Document retrieval**: Are the sources relevant to the query?
- **Response synthesis**: Does the response match the retrieved context? Does it also match the query?

This guide describes how the evaluation components within LlamaIndex work. Note that our current evaluation modules do *not* require ground-truth labels. Evaluation can be done with some combination of the query, context, response, and combine these with LLM calls.

3.14.1 Evaluation of the Response + Context

Each response from an `index.query` calls returns both the synthesized response as well as source documents.

We can evaluate the response against the retrieved sources - without taking into account the query!

This allows you to measure hallucination - if the response does not match the retrieved sources, this means that the model may be "hallucinating" an answer since it is not rooting the answer in the context provided to it in the prompt.

There are two sub-modes of evaluation here. We can either get a binary response "YES"/"NO" on whether response matches *any* source context, and also get a response list across sources to see which sources match.

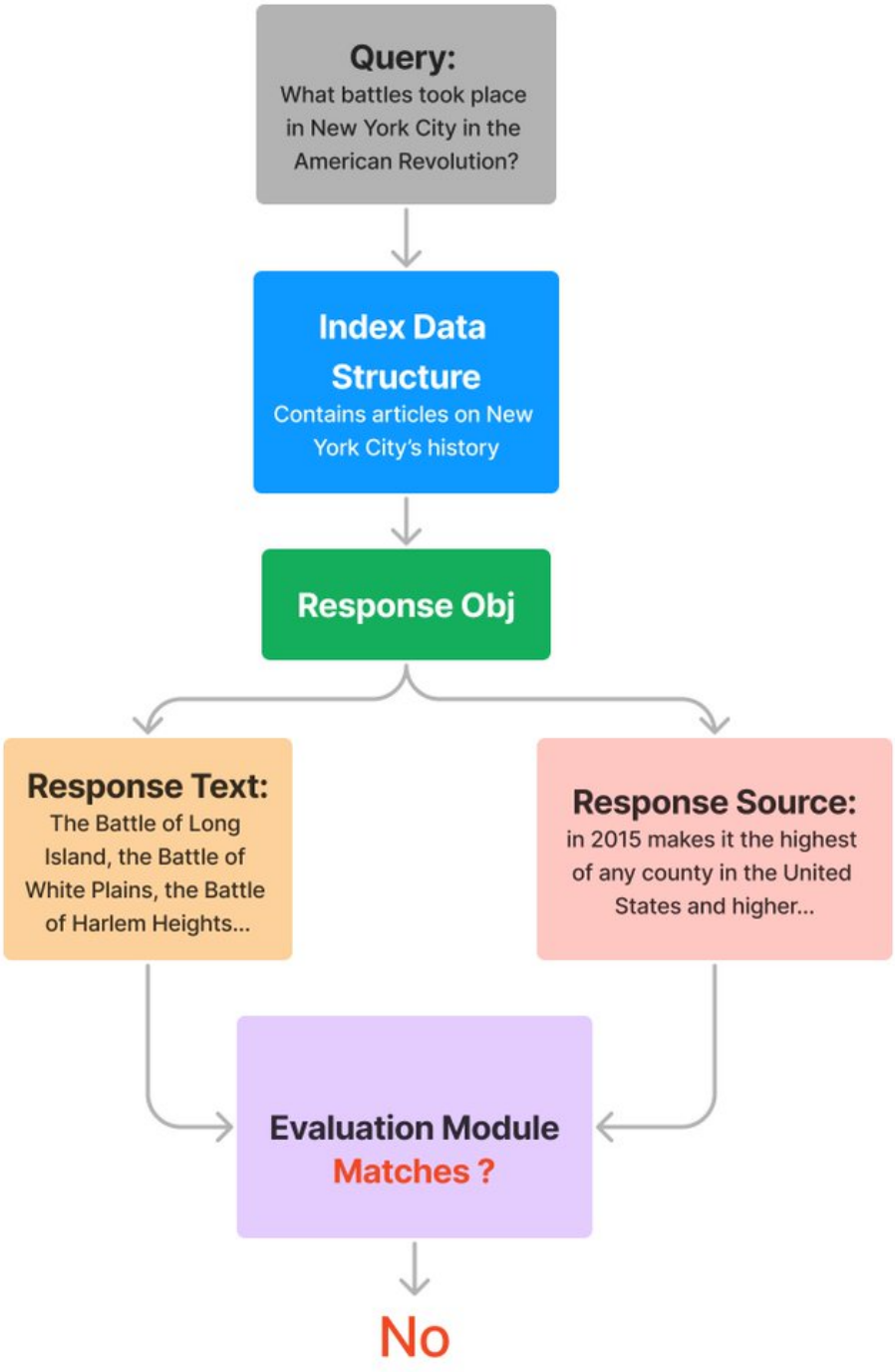
Binary Evaluation

This mode of evaluation will return "YES"/"NO" if the synthesized response matches any source context.

```
from gpt_index import GPTSimpleVectorIndex
from gpt_index.evaluation import ResponseEvaluator

# build service context
llm_predictor = LLMPredictor(llm=ChatOpenAI(temperature=0, model_name="gpt-4"))
service_context = ServiceContext.from_defaults(llm_predictor=llm_predictor)

# build index
...

# define evaluator
evaluator = ResponseEvaluator(service_context=service_context)

# query index
response = vector_index.query("What battles took place in New York City in the American␣
↪Revolution?")
eval_result = evaluator.evaluate(response)
print(str(eval_result))
```

You'll get back either a `YES` or `NO` response.

Diagram


Query:
What battles took place in New York City in the American Revolution?
Index Data Structure
Contains articles on New York City's history
Response Obj
Response Text:
The Battle of Long Island, the Battle of White Plains, the Battle of Harlem Heights...
Response Source:
in 2015 makes it the highest of any county in the United States and higher...
Evaluation Module
Matches ?
No

Sources Evaluation

This mode of evaluation will return "YES"/"NO" for every source node.

```
from gpt_index import GPTSimpleVectorIndex
from gpt_index.evaluation import ResponseEvaluator

# build service context
llm_predictor = LLMPredictor(llm=ChatOpenAI(temperature=0, model_name="gpt-4"))
service_context = ServiceContext.from_defaults(llm_predictor=llm_predictor)

# build index
...

# define evaluator
evaluator = ResponseEvaluator(service_context=service_context)

# query index
response = vector_index.query("What battles took place in New York City in the American␣
↪Revolution?")
eval_result = evaluator.evaluate_source_nodes(response)
print(str(eval_result))
```

You'll get back a list of "YES"/"NO", corresponding to each source node in `response.source_nodes`.

Notebook

Take a look at this notebook.

3.14.2 Evaluation of the Query + Response + Source Context

This is similar to the above section, except now we also take into account the query. The goal is to determine if the response + source context answers the query.

As with the above, there are two submodes of evaluation.

- We can either get a binary response "YES"/"NO" on whether the response matches the query, and whether any source node also matches the query.
- We can also ignore the synthesized response, and check every source node to see if it matches the query.

Binary Evaluation

This mode of evaluation will return "YES"/"NO" if the synthesized response matches the query + any source context.

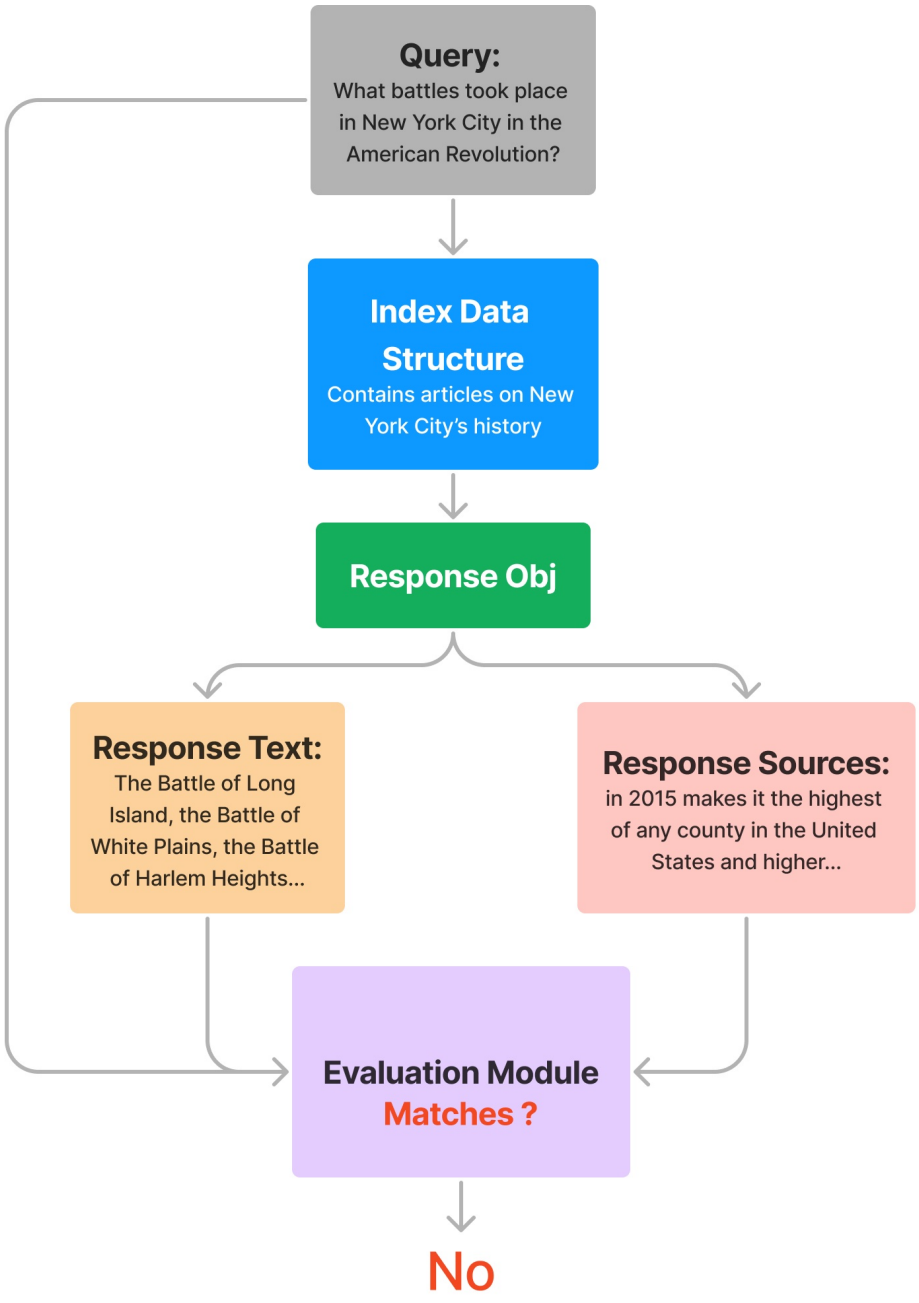
```
from gpt_index import GPTSimpleVectorIndex
from gpt_index.evaluation import QueryResponseEvaluator

# build service context
llm_predictor = LLMPredictor(llm=ChatOpenAI(temperature=0, model_name="gpt-4"))
service_context = ServiceContext.from_defaults(llm_predictor=llm_predictor)
```

(continues on next page)

(continued from previous page)

```
# build index
...

# define evaluator
evaluator = QueryResponseEvaluator(service_context=service_context)

# query index
response = vector_index.query("What battles took place in New York City in the American␣
↪Revolution?")
eval_result = evaluator.evaluate(response)
print(str(eval_result))
```

Diagram

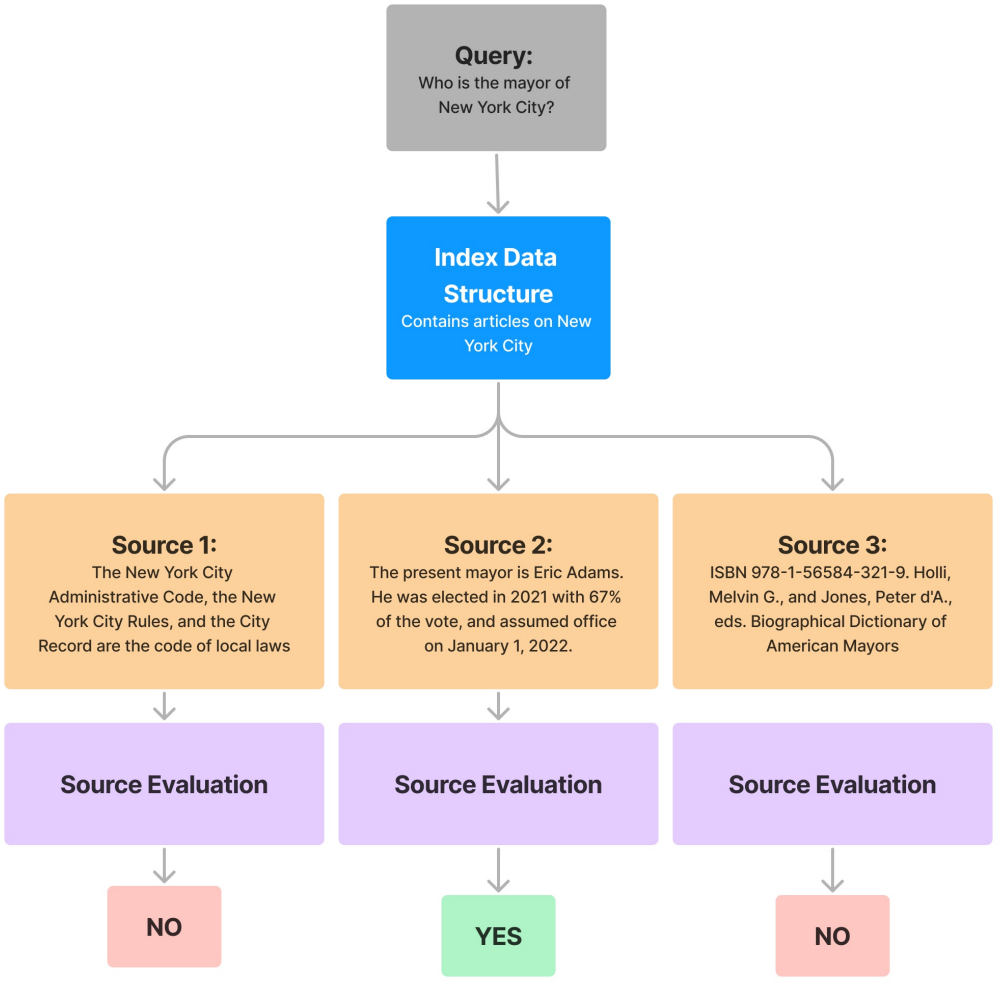

Query:
What battles took place in New York City in the American Revolution?
Index Data Structure
Contains articles on New York City's history
Response Obj
Response Text:
The Battle of Long Island, the Battle of White Plains, the Battle of Harlem Heights...
Response Sources:
in 2015 makes it the highest of any county in the United States and higher...
Evaluation Module
Matches ?
No

Sources Evaluation

This mode of evaluation will look at each source node, and see if each source node contains an answer to the query.

```
from gpt_index import GPTSimpleVectorIndex
from gpt_index.evaluation import QueryResponseEvaluator

# build service context
llm_predictor = LLMPredictor(llm=ChatOpenAI(temperature=0, model_name="gpt-4"))
service_context = ServiceContext.from_defaults(llm_predictor=llm_predictor)

# build index
...

# define evaluator
evaluator = QueryResponseEvaluator(service_context=service_context)

# query index
response = vector_index.query("What battles took place in New York City in the American␣
↪Revolution?")
eval_result = evaluator.evaluate_source_nodes(response)
print(str(eval_result))
```

Diagram


Query:
Who is the mayor of New York City?
Index Data Structure
Contains articles on New York City
Source 1:
The New York City Administrative Code, the New York City Rules, and the City Record are the code of local laws
Source 2:
The present mayor is Eric Adams. He was elected in 2021 with 67% of the vote, and assumed office on January 1, 2022.
Source 3:
ISBN 978-1-56584-321-9. Holli, Melvin G., and Jones, Peter d'A., eds. Biographical Dictionary of American Mayors
Source Evaluation
Source Evaluation
Source Evaluation
NO
YES
NO

Notebook

Take a look at this notebook.

3.15 Integrations

LlamaIndex provides a diverse range of integrations with other toolsets and storage providers.

Some of these integrations are provided in more detailed guides below.

3.15.1 Using Vector Stores

LlamaIndex offers multiple integration points with vector stores / vector databases:

1. LlamaIndex can load data from vector stores, similar to any other data connector. This data can then be used within LlamaIndex data structures.
2. LlamaIndex can use a vector store itself as an index. Like any other index, this index can store documents and be used to answer queries.

Loading Data from Vector Stores using Data Connector

LlamaIndex supports loading data from the following sources. See *Data Connectors* for more details and API documentation.

- Chroma (`ChromaReader`) Installation
- DeepLake (`DeepLakeReader`) Installation
- Qdrant (`QdrantReader`) Installation Python Client
- Weaviate (`WeaviateReader`). Installation. Python Client.
- Pinecone (`PineconeReader`). Installation/Quickstart.
- Faiss (`FaissReader`). Installation.
- Milvus (`MilvusReader`). Installation
- Zilliz (`MilvusReader`). Quickstart
- MyScale (`MyScaleReader`). Quickstart. Installation/Python Client.

Chroma stores both documents and vectors. This is an example of how to use Chroma:

```
from gpt_index.readers.chroma import ChromaReader
from gpt_index.indices import GPTListIndex

# The chroma reader loads data from a persisted Chroma collection.
# This requires a collection name and a persist directory.
reader = ChromaReader(
    collection_name="chroma_collection",
    persist_directory="examples/data_connectors/chroma_collection"
)

query_vector=[n1, n2, n3, ...]
```

(continues on next page)

(continued from previous page)

```
documents = reader.load_data(collection_name="demo", query_vector=query_vector, limit=5)
index = GPTListIndex.from_documents(documents)

response = index.query("<query_text>")
display(Markdown(f"<b>{response}</b>"))
```

Qdrant also stores both documents and vectors. This is an example of how to use Qdrant:

```
from gpt_index.readers.qdrant import QdrantReader
```

```
reader = QdrantReader(host="localhost")
```

```
# the query_vector is an embedding representation of your query_vector
# Example query vector:
#   query_vector=[0.3, 0.3, 0.3, 0.3, 0.3, 0.3, 0.3, 0.3]

query_vector=[n1, n2, n3, ...]
```

```
# NOTE: Required args are collection_name, query_vector.
# See the Python client: https://github.com/qdrant/qdrant_client
# for more details.
documents = reader.load_data(collection_name="demo", query_vector=query_vector, limit=5)
```

NOTE: Since Weaviate can store a hybrid of document and vector objects, the user may either choose to explicitly specify `class_name` and `properties` in order to query documents, or they may choose to specify a raw GraphQL query. See below for usage.

[4]:
```
# 1) load data using class_name and properties
# docs = reader.load_data(
#    class_name="Author", properties=["name", "description"], separate_documents=True
# )

documents = reader.load_data(
    class_name="<class_name>",
    properties=["property1", "property2", "..."],
    separate_documents=True
)
```

[4]:
```
# 2) example GraphQL query
# query = """
# {
#   Get {
#     Author {
#       name
#       description
#     }
#   }
# }
# """
# docs = reader.load_data(graphql_query=query, separate_documents=True)

query = """
{
  Get {
    <class_name> {
      <property1>
      <property2>
      ...
    }
  }
}
"""

documents = reader.load_data(graphql_query=query, separate_documents=True)
```

NOTE: Both Pinecone and Faiss data loaders assume that the respective data sources only store vectors; text content is stored elsewhere. Therefore, both data loaders require that the user specifies an `id_to_text_map` in the load_data call.

For instance, this is an example usage of the Pinecone data loader `PineconeReader`:

[2]:
```
from gpt_index.readers.pinecone import PineconeReader
```

[3]:
```
reader = PineconeReader(api_key=api_key, environment="us-west1-gcp")
```

[4]:
```
# the id_to_text_map specifies a mapping from the ID specified in Pinecone to your text.
id_to_text_map = {
    "id1": "text blob 1",
    "id2": "text blob 2",
}

# the query_vector is an embedding representation of your query_vector
# Example query vector:
#   query_vector=[0.3, 0.3, 0.3, 0.3, 0.3, 0.3, 0.3, 0.3]

query_vector=[n1, n2, n3, ...]
```

[ ]:
```
# NOTE: Required args are index_name, id_to_text_map, vector.
# In addition, we pass-through all kwargs that can be passed into the the `Query` operation in Pinecone.
# See the API reference: https://docs.pinecone.io/reference/query
# and also the Python client: https://github.com/pinecone-io/pinecone-python-client
# for more details.
documents = reader.load_data(index_name='quickstart', id_to_text_map=id_to_text_map, top_k=3, vector=query_vector, separate_documents=True)
```

Example notebooks can be found here.

Using a Vector Store as an Index

LlamaIndex also supports using a vector store itself as an index. These are found in the following classes:

- GPTSimpleVectorIndex
- GPTFaissIndex
- GPTWeaviateIndex
- GPTPineconeIndex
- GPTQdrantIndex
- GPTChromaIndex
- GPTMilvusIndex
- GPTDeepLakeIndex
- GPTMyScaleIndex

An API reference of each vector index is *found here*.

Similar to any other index within LlamaIndex (tree, keyword table, list), this index can be constructed upon any collection of documents. We use the vector store within the index to store embeddings for the input text chunks.

Once constructed, the index can be used for querying.

Simple Index Construction/Querying

```
from gpt_index import GPTSimpleVectorIndex, SimpleDirectoryReader

# Load documents, build the GPTSimpleVectorIndex
documents = SimpleDirectoryReader('../paul_graham_essay/data').load_data()
index = GPTSimpleVectorIndex.from_documents(documents)

# Query index
response = index.query("What did the author do growing up?")
```

DeepLake Index Construction/Querying

```
import os
import getpath

from gpt_index import GPTDeepLakeIndex, SimpleDirectoryReader


os.environ["OPENAI_API_KEY"] = getpath.getpath("OPENAI_API_KEY: ")
os.environ["ACTIVELOOP_TOKEN"] = getpath.getpath("ACTIVELOOP_TOKEN: ")

documents = SimpleDirectoryReader('../paul_graham_essay/data').load_data()
deeplake_dataset_path = "hub://adilkhan/paul_graham_essay"

# Create an index over the documnts
index = GPTDeepLakeIndex.from_documents(documents, dataset_path=dataset_path,␣
↪overwrite=True)
```

(continues on next page)

(continued from previous page)

```
# Query index
response = index.query("What did the author do growing up?")
```

Faiss Index Construction/Querying

```
from gpt_index import GPTFaissIndex, SimpleDirectoryReader
import faiss

# Creating a faiss index
d = 1536
faiss_index = faiss.IndexFlatL2(d)

# Load documents, build the GPTFaissIndex
documents = SimpleDirectoryReader('../paul_graham_essay/data').load_data()
index = GPTFaissIndex.from_documents(documents, faiss_index=faiss_index)

# Query index
response = index.query("What did the author do growing up?")
```

Weaviate Index Construction/Querying

```
from gpt_index import GPTWeaviateIndex, SimpleDirectoryReader
import weaviate

# Creating a Weaviate vector store
resource_owner_config = weaviate.AuthClientPassword(
    username="<username>",
    password="<password>",
)
client = weaviate.Client(
    "https://<cluster-id>.semi.network/", auth_client_secret=resource_owner_config
)

# Load documents, build the GPTWeaviateIndex
documents = SimpleDirectoryReader('../paul_graham_essay/data').load_data()
index = GPTWeaviateIndex.from_documents(documents, weaviate_client=client)

# Query index
response = index.query("What did the author do growing up?")
```

Pinecone Index Construction/Querying

```
from gpt_index import GPTPineconeIndex, SimpleDirectoryReader
import pinecone

# Creating a Pinecone index
api_key = "api_key"
pinecone.init(api_key=api_key, environment="us-west1-gcp")
pinecone.create_index(
    "quickstart",
    dimension=1536,
```

(continues on next page)

(continued from previous page)

```
    metric="euclidean",
    pod_type="p1"
)
index = pinecone.Index("quickstart")

# can define filters specific to this vector index (so you can
# reuse pinecone indexes)
metadata_filters = {"title": "paul_graham_essay"}


# Load documents, build the GPTPineconeIndex
documents = SimpleDirectoryReader('../paul_graham_essay/data').load_data()
index = GPTPineconeIndex.from_documents(
    documents, pinecone_index=index, metadata_filters=metadata_filters
)

# Query index
response = index.query("What did the author do growing up?")
```

Qdrant Index Construction/Querying

```
import qdrant_client
from gpt_index import GPTQdrantIndex, SimpleDirectoryReader

# Creating a Qdrant vector store
client = qdrant_client.QdrantClient(
    host="<qdrant-host>",
    api_key="<qdrant-api-key>",
    https=True
)
collection_name = "paul_graham"

# Load documents, build the GPTQdrantIndex
documents = SimpleDirectoryReader('../paul_graham_essay/data').load_data()
index = GPTQdrantIndex.from_documents(documents, collection_name=collection_name,␣
↪client=client)

# Query index
response = index.query("What did the author do growing up?")
```

Chroma Index Construction/Querying

```
import chromadb
from gpt_index import GPTChromaIndex, SimpleDirectoryReader

# Creating a Chroma vector store
# By default, Chroma will operate purely in-memory.
chroma_client = chromadb.Client()
chroma_collection = chroma_client.create_collection("quickstart")

# Load documents, build the GPTChromaIndex
documents = SimpleDirectoryReader('../paul_graham_essay/data').load_data()
```

(continues on next page)

(continued from previous page)

```
index = GPTChromaIndex.from_documents(documents, chroma_collection=chroma_collection)

# Query index
response = index.query("What did the author do growing up?")
```

Milvus Index Construction/Querying

- Milvus Index offers the ability to store both Documents and their embeddings. Documents are limited to the predefined Document attributes and does not include extra_info.

```
import pymilvus
from gpt_index import GPTMilvusIndex, SimpleDirectoryReader


# Load documents, build the GPTMilvusStore
documents = SimpleDirectoryReader('../paul_graham_essay/data').load_data()
index = GPTMilvusIndex.from_documents(documents, host='localhost', port=19530, overwrite=
↪'True')

# Query index
response = index.query("What did the author do growing up?")
```

Note: `GPTMilvusIndex` depends on the `pymilvus` library. Use `pip install pymilvus` if not already installed. If you get stuck at building wheel for `grpcio`, check if you are using python 3.11 (there's a known issue: https://github.com/milvus-io/pymilvus/issues/1308) and try downgrading.

Zilliz Index Construction/Querying

- Zilliz Cloud (hosted version of Milvus) uses the Milvus Index with some extra arguments.

```
import pymilvus
from gpt_index import GPTMilvusIndex, SimpleDirectoryReader


# Load documents, build the GPTMilvusStore
documents = SimpleDirectoryReader('../paul_graham_essay/data').load_data()
index = GPTMilvusIndex.from_documents(
    documents,
    host='foo.vectordb.zillizcloud.com',
    port=403,
    user="db_admin",
    password="foo",
    use_secure=True,
    overwrite='True'
)

# Query index
response = index.query("What did the author do growing up?")
```

Note: `GPTMilvusIndex` depends on the `pymilvus` library. Use `pip install pymilvus` if not already installed. If you get stuck at building wheel for `grpcio`, check if you are using python 3.11 (there's a known issue: https://github.com/milvus-io/pymilvus/issues/1308) and try downgrading.

MyScale Index Construction/Querying

```
import clickhouse_connect
from gpt_index import GPTMyScaleIndex, SimpleDirectoryReader

# Creating a MyScale client

client = clickhouse_connect.get_client(
    host='YOUR_CLUSTER_HOST',
    port=8443,
    username='YOUR_USERNAME',
    password='YOUR_CLUSTER_PASSWORD'
)

# Load documents, build the GPTMyScaleIndex
documents = SimpleDirectoryReader('../paul_graham_essay/data').load_data()
index = GPTMyScaleIndex.from_documents(documents, myscale_client=client)

# Query index
response = index.query("What did the author do growing up?")
```

Example notebooks can be found here.

3.15.2 ChatGPT Plugin Integrations

NOTE: This is a work-in-progress, stay tuned for more exciting updates on this front!

ChatGPT Retrieval Plugin Integrations

The OpenAI ChatGPT Retrieval Plugin offers a centralized API specification for any document storage system to interact with ChatGPT. Since this can be deployed on any service, this means that more and more document retrieval services will implement this spec; this allows them to not only interact with ChatGPT, but also interact with any LLM toolkit that may use a retrieval service.

LlamaIndex provides a variety of integrations with the ChatGPT Retrieval Plugin.

Loading Data from LlamaHub into the ChatGPT Retrieval Plugin

The ChatGPT Retrieval Plugin defines an `/upsert` endpoint for users to load documents. This offers a natural integration point with LlamaHub, which offers over 65 data loaders from various API's and document formats.

Here is a sample code snippet of showing how to load a document from LlamaHub into the JSON format that `/upsert` expects:

```
from llama_index import download_loader, Document
from typing import Dict, List
import json

# download loader, load documents
SimpleWebPageReader = download_loader("SimpleWebPageReader")
loader = SimpleWebPageReader(html_to_text=True)
url = "http://www.paulgraham.com/worked.html"
```

(continues on next page)

(continued from previous page)

```
documents = loader.load_data(urls=[url])

# Convert LlamaIndex Documents to JSON format
def dump_docs_to_json(documents: List[Document], out_path: str) -> Dict:
    """Convert LlamaIndex Documents to JSON format and save it."""
    result_json = []
    for doc in documents:
        cur_dict = {
            "text": doc.get_text(),
            "id": doc.get_doc_id(),
            # NOTE: feel free to customize the other fields as you wish
            # fields taken from https://github.com/openai/chatgpt-retrieval-plugin/tree/
↪main/scripts/process_json#usage
            # "source": ...,
            # "source_id": ...,
            # "url": url,
            # "created_at": ...,
            # "author": "Paul Graham",
        }
        result_json.append(cur_dict)

    json.dump(result_json, open(out_path, 'w'))
```

For more details, check out the full example notebook.

ChatGPT Retrieval Plugin Data Loader

The ChatGPT Retrieval Plugin data loader can be accessed on LlamaHub.

It allows you to easily load data from any docstore that implements the plugin API, into a LlamaIndex data structure.

Example code:

```
from llama_index.readers import ChatGPTRetrievalPluginReader
import os

# load documents
bearer_token = os.getenv("BEARER_TOKEN")
reader = ChatGPTRetrievalPluginReader(
    endpoint_url="http://localhost:8000",
    bearer_token=bearer_token
)
documents = reader.load_data("What did the author do growing up?")

# build and query index
from gpt_index import GPTListIndex
index = GPTListIndex(documents)
# set Logging to DEBUG for more detailed outputs
response = index.query(
    "Summarize the retrieved content and describe what the author did growing up",
    response_mode="compact"
```

(continues on next page)

(continued from previous page)

```
)
```

For more details, check out the full example notebook.

ChatGPT Retrieval Plugin Index

The ChatGPT Retrieval Plugin Index allows you to easily build a vector index over any documents, with storage backed by a document store implementing the ChatGPT endpoint.

Note: this index is a vector index, allowing top-k retrieval.

Example code:

```
from llama_index.indices.vector_store import ChatGPTRetrievalPluginIndex
from llama_index import SimpleDirectoryReader
import os

# load documents
documents = SimpleDirectoryReader('../paul_graham_essay/data').load_data()

# build index
bearer_token = os.getenv("BEARER_TOKEN")
# initialize without metadata filter
index = ChatGPTRetrievalPluginIndex(
    documents,
    endpoint_url="http://localhost:8000",
    bearer_token=bearer_token,
)

# query index
response = index.query("What did the author do growing up?", similarity_top_k=3)
```

For more details, check out the full example notebook.

3.15.3 Using with Langchain

LlamaIndex provides both Tool abstractions for a Langchain agent as well as a memory module.

The API reference of the Tool abstractions + memory modules are *here*.

Llama Tool abstractions

LlamaIndex provides Tool abstractions so that you can use LlamaIndex along with a Langchain agent.

For instance, you can choose to create a "Tool" from an index directly as follows:

```
from gpt_index.langchain_helpers.agents import IndexToolConfig, LlamaIndexTool

tool_config = IndexToolConfig(
    index=index,
```

(continues on next page)

(continued from previous page)

```
    name=f"Vector Index",
    description=f"useful for when you want to answer queries about X",
    index_query_kwargs={"similarity_top_k": 3},
    tool_kwargs={"return_direct": True}
)

tool = LlamaIndexTool.from_tool_config(tool_config)
```

Similarly, you can choose to create a "Tool" from a composed graph.

```
from gpt_index.langchain_helpers.agents import GraphToolConfig, LlamaGraphTool

graph_config = GraphToolConfig(
    graph=graph,
    name=f"Graph Index",
    description="useful for when you want to answer queries about Y",
    query_configs=query_configs,
    tool_kwargs={"return_direct": True}
)

tool = LlamaGraphTool.from_tool_config(tool_config)
```

You can also choose to provide a `LlamaToolkit`:

```
toolkit = LlamaToolkit(
    index_configs=index_configs,
    graph_configs=[graph_config]
)
```

Such a toolkit can be used to create a downstream Langchain-based chat agent through our `create_llama_agent` and `create_llama_chat_agent` commands:

```
from gpt_index.langchain_helpers.agents import create_llama_chat_agent

agent_chain = create_llama_chat_agent(
    toolkit,
    llm,
    memory=memory,
    verbose=True
)

agent_chain.run(input="Query about X")
```

You can take a look at the full tutorial notebook here.

Llama Demo Notebook: Tool + Memory module

We provide another demo notebook showing how you can build a chat agent with the following components.

- Using LlamaIndex as a generic callable tool with a Langchain agent
- Using LlamaIndex as a memory module; this allows you to insert arbitrary amounts of conversation history with a Langchain chatbot!

Please see the notebook here.

3.16 Document Store

LlamaIndex provides a high-level interface for ingesting, indexing, and querying your external data.

By default, LlamaIndex hides away the complexities and let you query your data in under 5 lines of code:

```
from llama_index import GPTSimpleVectorIndex, SimpleDirectoryReader

documents = SimpleDirectoryReader('data').load_data()
index = GPTSimpleVectorIndex.from_documents(documents)
response = index.query("Summarize the documents.")
```

Under the hood, LlamaIndex also supports a swappable **storage layer** that allows you to customize where ingested data (i.e., `Node` objects) are stored.

To do this, instead of the high-level API,

```
index = GPTSimpleVectorIndex.from_documents(documents)
```

we use a lower-level API that gives more granular control:

```
from llama_index.docstore import SimpleDocumentStore
from llama_index.node_parser import SimpleNodeParser

# create parser and parse document into nodes
parser = SimpleNodeParser()
nodes = parser.get_nodes_from_documents(documents)

# create document store and add nodes
docstore = SimpleDocumentStore()
docstore.add_documents(nodes)

# build index
index = GPTSimpleVectorIndex(nodes, docstore=docstore)
```

You can customize the underlying storage with a one-line change to instantiate a different document store.

3.16.1 Simple Document Store

By default, the `SimpleDocumentStore` stores `Node` objects in-memory. They can be persisted to (and loaded from) disk by calling `index.save_to_disk(...)` (and `Index.load_from_disk(...)` respectively).

3.16.2 MongoDB Document Store

We support MongoDB as an alternative document store backend that persists data as `Node` objects are ingested.

```
from llama_index.docstore import MongoDocumentStore
from llama_index.node_parser import SimpleNodeParser

# create parser and parse document into nodes
parser = SimpleNodeParser()
nodes = parser.get_nodes_from_documents(documents)

# create document store and add nodes
docstore = MongoDocumentStore.from_uri(uri="<mongodb+srv://...>")
docstore.add_documents(nodes)

# build index
index = GPTSimpleVectorIndex(nodes, docstore=docstore)
```

Under the hood, `MongoDocumentStore` connects to a fixed MongoDB database and initializes a new collection for your nodes.

> Note: You can configure the `db_name` and `collection_name` when instantiating `MongoDocumentStore`, otherwise they default to `db_name=db_docstore` and `collection_name=collection_<uuid>`.

When using `MongoDocumentStore`, calling `index.save_to_disk(...)` only saves the MongoDB connection config to disk (instead of the `Node` objects).

You can easily reconnect to your MongoDB collection and reload the index by calling `Index.load_from_disk(...)` (or by explicitly initializing a `MongoDocumentStore` with an exiting `db_name` and `collection_name`).

3.17 Indices

This doc shows both the overarching class used to represent an index. These classes allow for index creation, insertion, and also querying. We first show the different index subclasses. We then show the base class that all indices inherit from, which contains parameters and methods common to all indices.

3.17.1 List Index

Building the List Index

List-based data structures.

class gpt_index.indices.list.**GPTListIndex**(*nodes: Optional[Sequence[Node]] = None*, *index_struct: Optional[IndexList] = None*, *service_context: Optional[ServiceContext] = None*, *text_qa_template: Optional[QuestionAnswerPrompt] = None*, ***kwargs: Any*)

GPT List Index.

The list index is a simple data structure where nodes are stored in a sequence. During index construction, the document texts are chunked up, converted to nodes, and stored in a list.

During query time, the list index iterates through the nodes with some optional filter parameters, and synthesizes an answer from all the nodes.

Parameters
text_qa_template (*Optional[QuestionAnswerPrompt]*) – A Question-Answer Prompt (see *Prompt Templates*). NOTE: this is a deprecated field.

async aquery(*query_str: Union[str, QueryBundle]*, *mode: str = QueryMode.DEFAULT*, *query_transform: Optional[BaseQueryTransform] = None*, ***query_kwargs: Any*) → Union[*Response*, *StreamingResponse*]

Asynchronously answer a query.

When *query* is called, we query the index with the given *mode* and *query_kwargs*. The *mode* determines the type of query to run, and *query_kwargs* are parameters that are specific to the query type.

For a comprehensive documentation of available *mode* and *query_kwargs* to query a given index, please visit *Querying an Index*.

classmethod from_documents(*documents: Sequence[Document]*, *docstore: Optional[BaseDocumentStore] = None*, *service_context: Optional[ServiceContext] = None*, ***kwargs: Any*) → *BaseGPTIndex*

Create index from documents.

Parameters
documents (*Optional[Sequence[BaseDocument]]*) – List of documents to build the index from.

classmethod get_query_map() → Dict[str, Type[*BaseGPTIndexQuery*]]

Get query map.

insert(*document: Document*, ***insert_kwargs: Any*) → None

Insert a document.

classmethod load_from_dict(*result_dict: Dict[str, Any]*, ***kwargs: Any*) → *BaseGPTIndex*

Load index from dict.

classmethod load_from_disk(*save_path: str*, ***kwargs: Any*) → *BaseGPTIndex*

Load index from disk.

This method loads the index from a JSON file stored on disk. The index data structure itself is preserved completely. If the index is defined over subindices, those subindices will also be preserved (and subindices of those subindices, etc.).

NOTE: load_from_disk should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_disk* and *load_from_disk* on that instead.

Parameters
save_path (*str*) – The save_path of the file.

Returns
The loaded index.

Return type
BaseGPTIndex

classmethod load_from_string(*index_string: str*, ***kwargs: Any*) → *BaseGPTIndex*

Load index from string (in JSON-format).

This method loads the index from a JSON string. The index data structure itself is preserved completely. If the index is defined over subindices, those subindices will also be preserved (and subindices of those subindices, etc.).

NOTE: load_from_string should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_string* and *load_from_string* on that instead.

Parameters
index_string (*str*) – The index string (in JSON-format).

Returns
The loaded index.

Return type
BaseGPTIndex

query(*query_str: Union[str, QueryBundle]*, *mode: str = QueryMode.DEFAULT*, *query_transform: Optional[BaseQueryTransform] = None*, *use_async: bool = False*, ***query_kwargs: Any*) → Union[*Response*, *StreamingResponse*]

Answer a query.

When *query* is called, we query the index with the given *mode* and *query_kwargs*. The *mode* determines the type of query to run, and *query_kwargs* are parameters that are specific to the query type.

For a comprehensive documentation of available *mode* and *query_kwargs* to query a given index, please visit *Querying an Index*.

property query_context: Dict[str, Any]

Additional context necessary for making a query.

This should capture any index-specific clients, services, etc, that's not captured by index struct, docstore, and service context. For example, a vector store index would pass vector store.

refresh(*documents: Sequence[Document]*, ***update_kwargs: Any*) → List[bool]

Refresh an index with documents that have changed.

This allows users to save LLM and Embedding model calls, while only updating documents that have any changes in text or extra_info. It will also insert any documents that previously were not stored.

save_to_dict(***save_kwargs: Any*) → dict

Save to dict.

save_to_disk(*save_path: str*, *encoding: str = 'ascii'*, ***save_kwargs: Any*) → None

Save to file.

This method stores the index into a JSON file stored on disk.

NOTE: save_to_disk should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_disk* and *load_from_disk* on that instead.

Parameters

- **save_path** (*str*) – The save_path of the file.
- **encoding** (*str*) – The encoding of the file.

save_to_string(***save_kwargs: Any*) → str

Save to string.

This method stores the index into a JSON string.

NOTE: save_to_string should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_string* and *load_from_string* on that instead.

Returns
The JSON string of the index.

Return type
str

update(*document:* Document, ***update_kwargs: Any*) → None

Update a document.

This is equivalent to deleting the document and then inserting it again.

Parameters

- **document** (*Union[BaseDocument, BaseGPTIndex]*) – document to update
- **insert_kwargs** (*Dict*) – kwargs to pass to insert
- **delete_kwargs** (*Dict*) – kwargs to pass to delete

class gpt_index.indices.list.**GPTListIndexEmbeddingQuery**(*index_struct: IndexList*, *similarity_top_k: Optional[int] = 1*, ***kwargs: Any*)

GPTListIndex query.

An embedding-based query for GPTListIndex, which traverses each node in sequence and retrieves top-k nodes by embedding similarity to the query. Set when *mode="embedding"* in *query* method of *GPTListIndex*.

```
response = index.query("<query_str>", mode="embedding")
```

See BaseGPTListIndexQuery for arguments.

retrieve(*query_bundle:* QueryBundle) → List[*NodeWithScore*]

Get list of tuples of node and similarity for response.

First part of the tuple is the node. Second part of tuple is the distance from query to the node. If not applicable, it's None.

class gpt_index.indices.list.**GPTListIndexQuery**(*index_struct: IS*, *service_context:* ServiceContext, *response_synthesizer: ResponseSynthesizer*, *docstore: Optional[*BaseDocumentStore*] = None*, *node_postprocessors: Optional[List[BaseNodePostprocessor]] = None*, *include_extra_info: bool = False*, *verbose: bool = False*)

GPTListIndex query.

The default query mode for GPTListIndex, which traverses each node in sequence and synthesizes a response across all nodes (with an optional keyword filter). Set when *mode="default"* in *query* method of *GPTListIndex*.

```
response = index.query("<query_str>", mode="default")
```

See BaseGPTListIndexQuery for arguments.

retrieve(*query_bundle:* QueryBundle) → List[*NodeWithScore*]

Get list of tuples of node and similarity for response.

First part of the tuple is the node. Second part of tuple is the distance from query to the node. If not applicable, it's None.

3.17.2 Table Index

Building the Keyword Table Index

Keyword Table Index Data Structures.

class gpt_index.indices.keyword_table.**GPTKeywordTableGPTQuery**(*index_struct: KeywordTable*, *keyword_extract_template: Optional[KeywordExtractPrompt] = None*, *query_keyword_extract_template: Optional[QueryKeywordExtractPrompt] = None*, *max_keywords_per_query: int = 10*, *num_chunks_per_query: int = 10*, ***kwargs: Any*)

GPT Keyword Table Index Query.

Extracts keywords using GPT. Set when *mode="default"* in *query* method of *GPTKeywordTableIndex*.

```
response = index.query("<query_str>", mode="default")
```

See BaseGPTKeywordTableQuery for arguments.

property index_struct: IS

Get the index struct.

retrieve(*query_bundle:* QueryBundle) → List[*NodeWithScore*]

Get list of tuples of node and similarity for response.

First part of the tuple is the node. Second part of tuple is the distance from query to the node. If not applicable, it's None.

class gpt_index.indices.keyword_table.**GPTKeywordTableIndex**(*nodes: Optional[Sequence[Node]] = None*, *index_struct: Optional[KeywordTable] = None*, *service_context: Optional[ServiceContext] = None*, *keyword_extract_template: Optional[KeywordExtractPrompt] = None*, *max_keywords_per_chunk: int = 10*, *use_async: bool = False*, ***kwargs: Any*)

GPT Keyword Table Index.

This index uses a GPT model to extract keywords from the text.

async aquery(*query_str: Union[str,* QueryBundle*]*, *mode: str = QueryMode.DEFAULT*, *query_transform: Optional[BaseQueryTransform] = None*, ***query_kwargs: Any*) → Union[*Response*, *StreamingResponse*]

Asynchronously answer a query.

When *query* is called, we query the index with the given *mode* and *query_kwargs*. The *mode* determines the type of query to run, and *query_kwargs* are parameters that are specific to the query type.

For a comprehensive documentation of available *mode* and *query_kwargs* to query a given index, please visit *Querying an Index*.

delete(*doc_id: str*, ***delete_kwargs: Any*) → None

Delete a document from the index.

All nodes in the index related to the index will be deleted.

Parameters
doc_id (*str*) – document id

property docstore: *BaseDocumentStore*

Get the docstore corresponding to the index.

classmethod from_documents(*documents: Sequence[*Document*]*, *docstore: Optional[*BaseDocumentStore*] = None*, *service_context: Optional[*ServiceContext*] = None*, ***kwargs: Any*) → *BaseGPTIndex*

Create index from documents.

Parameters
documents (*Optional[Sequence[BaseDocument]]*) – List of documents to build the index from.

classmethod get_query_map() → Dict[str, Type[*BaseGPTIndexQuery*]]

Get query map.

property index_struct: *IS*

Get the index struct.

index_struct_cls

alias of KeywordTable

insert(*document:* Document, ***insert_kwargs: Any*) → None

Insert a document.

classmethod load_from_dict(*result_dict: Dict[str, Any]*, ***kwargs: Any*) → *BaseGPTIndex*

Load index from dict.

classmethod load_from_disk(*save_path: str*, ***kwargs: Any*) → *BaseGPTIndex*

Load index from disk.

This method loads the index from a JSON file stored on disk. The index data structure itself is preserved completely. If the index is defined over subindices, those subindices will also be preserved (and subindices of those subindices, etc.).

NOTE: load_from_disk should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_disk* and *load_from_disk* on that instead.

Parameters
save_path (*str*) – The save_path of the file.

Returns
The loaded index.

Return type
BaseGPTIndex

classmethod load_from_string(*index_string: str*, ***kwargs: Any*) → *BaseGPTIndex*

Load index from string (in JSON-format).

This method loads the index from a JSON string. The index data structure itself is preserved completely. If the index is defined over subindices, those subindices will also be preserved (and subindices of those subindices, etc.).

NOTE: load_from_string should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_string* and *load_from_string* on that instead.

Parameters
index_string (*str*) – The index string (in JSON-format).

Returns
The loaded index.

Return type
BaseGPTIndex

query(*query_str: Union[str, QueryBundle]*, *mode: str = QueryMode.DEFAULT*, *query_transform: Optional[BaseQueryTransform] = None*, *use_async: bool = False*, ***query_kwargs: Any*) → Union[*Response*, *StreamingResponse*]

Answer a query.

When *query* is called, we query the index with the given *mode* and *query_kwargs*. The *mode* determines the type of query to run, and *query_kwargs* are parameters that are specific to the query type.

For a comprehensive documentation of available *mode* and *query_kwargs* to query a given index, please visit *Querying an Index*.

property query_context: Dict[str, Any]

Additional context necessary for making a query.

This should capture any index-specific clients, services, etc, that's not captured by index struct, docstore, and service context. For example, a vector store index would pass vector store.

refresh(*documents: Sequence[Document]*, ***update_kwargs: Any*) → List[bool]

Refresh an index with documents that have changed.

This allows users to save LLM and Embedding model calls, while only updating documents that have any changes in text or extra_info. It will also insert any documents that previously were not stored.

save_to_dict(***save_kwargs: Any*) → dict

Save to dict.

save_to_disk(*save_path: str*, *encoding: str = 'ascii'*, ***save_kwargs: Any*) → None

Save to file.

This method stores the index into a JSON file stored on disk.

NOTE: save_to_disk should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_disk* and *load_from_disk* on that instead.

Parameters

- **save_path** (*str*) – The save_path of the file.
- **encoding** (*str*) – The encoding of the file.

save_to_string(***save_kwargs: Any*) → str

Save to string.

This method stores the index into a JSON string.

NOTE: save_to_string should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_string* and *load_from_string* on that instead.

Returns
The JSON string of the index.

Return type
str

update(*document:* Document, ***update_kwargs: Any*) → None

Update a document.

This is equivalent to deleting the document and then inserting it again.

Parameters

- **document** (*Union[BaseDocument, BaseGPTIndex]*) – document to update
- **insert_kwargs** (*Dict*) – kwargs to pass to insert
- **delete_kwargs** (*Dict*) – kwargs to pass to delete

class gpt_index.indices.keyword_table.**GPTKeywordTableRAKEQuery**(*index_struct: KeywordTable*, *keyword_extract_template: Optional[KeywordExtractPrompt] = None*, *query_keyword_extract_template: Optional[QueryKeywordExtractPrompt] = None*, *max_keywords_per_query: int = 10*, *num_chunks_per_query: int = 10*, ***kwargs: Any*)

GPT Keyword Table Index RAKE Query.

Extracts keywords using RAKE keyword extractor. Set when *mode="rake"* in *query* method of *GPTKeywordTableIndex*.

```
response = index.query("<query_str>", mode="rake")
```

See BaseGPTKeywordTableQuery for arguments.

property index_struct: IS

Get the index struct.

retrieve(*query_bundle:* QueryBundle) → List[*NodeWithScore*]

Get list of tuples of node and similarity for response.

First part of the tuple is the node. Second part of tuple is the distance from query to the node. If not applicable, it's None.

class gpt_index.indices.keyword_table.**GPTKeywordTableSimpleQuery**(*index_struct: KeywordTable*, *keyword_extract_template: Optional[KeywordExtractPrompt] = None*, *query_keyword_extract_template: Optional[QueryKeywordExtractPrompt] = None*, *max_keywords_per_query: int = 10*, *num_chunks_per_query: int = 10*, ***kwargs: Any*)

GPT Keyword Table Index Simple Query.

Extracts keywords using simple regex-based keyword extractor. Set when *mode="simple"* in *query* method of *GPTKeywordTableIndex*.

```
response = index.query("<query_str>", mode="simple")
```

See BaseGPTKeywordTableQuery for arguments.

property index_struct: IS

Get the index struct.

retrieve(*query_bundle:* QueryBundle) → List[*NodeWithScore*]

Get list of tuples of node and similarity for response.

First part of the tuple is the node. Second part of tuple is the distance from query to the node. If not applicable, it's None.

class gpt_index.indices.keyword_table.**GPTRAKEKeywordTableIndex**(*nodes: Optional[Sequence[Node]] = None*, *index_struct: Optional[KeywordTable] = None*, *service_context: Optional[ServiceContext] = None*, *keyword_extract_template: Optional[KeywordExtractPrompt] = None*, *max_keywords_per_chunk: int = 10*, *use_async: bool = False*, ***kwargs: Any*)

GPT RAKE Keyword Table Index.

This index uses a RAKE keyword extractor to extract keywords from the text.

async aquery(*query_str: Union[str,* QueryBundle*]*, *mode: str = QueryMode.DEFAULT*, *query_transform: Optional[BaseQueryTransform] = None*, ***query_kwargs: Any*) → Union[*Response*, *StreamingResponse*]

Asynchronously answer a query.

When *query* is called, we query the index with the given *mode* and *query_kwargs*. The *mode* determines the type of query to run, and *query_kwargs* are parameters that are specific to the query type.

For a comprehensive documentation of available *mode* and *query_kwargs* to query a given index, please visit *Querying an Index*.

delete(*doc_id: str*, ***delete_kwargs: Any*) → None

Delete a document from the index.

All nodes in the index related to the index will be deleted.

Parameters
doc_id (*str*) – document id

property docstore: *BaseDocumentStore*

Get the docstore corresponding to the index.

classmethod from_documents(*documents: Sequence[*Document*]*, *docstore: Optional[*BaseDocumentStore*] = None*, *service_context: Optional[*ServiceContext*] = None*, ***kwargs: Any*) → *BaseGPTIndex*

Create index from documents.

Parameters
documents (*Optional[Sequence[BaseDocument]]*) – List of documents to build the index from.

classmethod get_query_map() → Dict[str, Type[*BaseGPTIndexQuery*]]

Get query map.

property index_struct: IS

Get the index struct.

index_struct_cls

alias of KeywordTable

insert(*document:* Document, ***insert_kwargs: Any*) → None

Insert a document.

classmethod load_from_dict(*result_dict: Dict[str, Any]*, ***kwargs: Any*) → *BaseGPTIndex*

Load index from dict.

classmethod load_from_disk(*save_path: str*, ***kwargs: Any*) → *BaseGPTIndex*

Load index from disk.

This method loads the index from a JSON file stored on disk. The index data structure itself is preserved completely. If the index is defined over subindices, those subindices will also be preserved (and subindices of those subindices, etc.).

NOTE: load_from_disk should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_disk* and *load_from_disk* on that instead.

Parameters
save_path (*str*) – The save_path of the file.

Returns
The loaded index.

Return type
BaseGPTIndex

classmethod load_from_string(*index_string: str*, ***kwargs: Any*) → *BaseGPTIndex*

Load index from string (in JSON-format).

This method loads the index from a JSON string. The index data structure itself is preserved completely. If the index is defined over subindices, those subindices will also be preserved (and subindices of those subindices, etc.).

NOTE: load_from_string should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_string* and *load_from_string* on that instead.

Parameters
index_string (*str*) – The index string (in JSON-format).

Returns
The loaded index.

Return type
BaseGPTIndex

query(*query_str: Union[str, QueryBundle]*, *mode: str = QueryMode.DEFAULT*, *query_transform: Optional[BaseQueryTransform] = None*, *use_async: bool = False*, ***query_kwargs: Any*) → Union[*Response*, *StreamingResponse*]

Answer a query.

When *query* is called, we query the index with the given *mode* and *query_kwargs*. The *mode* determines the type of query to run, and *query_kwargs* are parameters that are specific to the query type.

For a comprehensive documentation of available *mode* and *query_kwargs* to query a given index, please visit *Querying an Index*.

property query_context: Dict[str, Any]

Additional context necessary for making a query.

This should capture any index-specific clients, services, etc, that's not captured by index struct, docstore, and service context. For example, a vector store index would pass vector store.

refresh(*documents: Sequence[Document]*, ***update_kwargs: Any*) → List[bool]

Refresh an index with documents that have changed.

This allows users to save LLM and Embedding model calls, while only updating documents that have any changes in text or extra_info. It will also insert any documents that previously were not stored.

save_to_dict(***save_kwargs: Any*) → dict

Save to dict.

save_to_disk(*save_path: str*, *encoding: str = 'ascii'*, ***save_kwargs: Any*) → None

Save to file.

This method stores the index into a JSON file stored on disk.

NOTE: save_to_disk should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_disk* and *load_from_disk* on that instead.

Parameters

- **save_path** (*str*) – The save_path of the file.
- **encoding** (*str*) – The encoding of the file.

save_to_string(***save_kwargs: Any*) → str

Save to string.

This method stores the index into a JSON string.

NOTE: save_to_string should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_string* and *load_from_string* on that instead.

Returns
The JSON string of the index.

Return type
str

update(*document: Document*, ***update_kwargs: Any*) → None

Update a document.

This is equivalent to deleting the document and then inserting it again.

Parameters

- **document** (*Union[BaseDocument, BaseGPTIndex]*) – document to update
- **insert_kwargs** (*Dict*) – kwargs to pass to insert
- **delete_kwargs** (*Dict*) – kwargs to pass to delete

class gpt_index.indices.keyword_table.**GPTSimpleKeywordTableIndex**(*nodes: Optional[Sequence[Node]] = None*, *index_struct: Optional[KeywordTable] = None*, *service_context: Optional[ServiceContext] = None*, *keyword_extract_template: Optional[KeywordExtractPrompt] = None*, *max_keywords_per_chunk: int = 10*, *use_async: bool = False*, ***kwargs: Any*)

GPT Simple Keyword Table Index.

This index uses a simple regex extractor to extract keywords from the text.

async aquery(*query_str: Union[str, QueryBundle]*, *mode: str = QueryMode.DEFAULT*, *query_transform: Optional[BaseQueryTransform] = None*, ***query_kwargs: Any*) → Union[*Response*, *StreamingResponse*]

Asynchronously answer a query.

When *query* is called, we query the index with the given *mode* and *query_kwargs*. The *mode* determines the type of query to run, and *query_kwargs* are parameters that are specific to the query type.

For a comprehensive documentation of available *mode* and *query_kwargs* to query a given index, please visit *Querying an Index*.

delete(*doc_id: str*, ***delete_kwargs: Any*) → None

Delete a document from the index.

All nodes in the index related to the index will be deleted.

Parameters
doc_id (*str*) – document id

property docstore: *BaseDocumentStore*

Get the docstore corresponding to the index.

classmethod from_documents(*documents: Sequence[Document]*, *docstore: Optional[BaseDocumentStore] = None*, *service_context: Optional[ServiceContext] = None*, ***kwargs: Any*) → *BaseGPTIndex*

Create index from documents.

Parameters
documents (*Optional[Sequence[BaseDocument]]*) – List of documents to build the index from.

classmethod get_query_map() → Dict[str, Type[*BaseGPTIndexQuery*]]

Get query map.

property index_struct: IS

Get the index struct.

index_struct_cls

alias of KeywordTable

insert(*document:* Document, ***insert_kwargs: Any*) → None

Insert a document.

classmethod load_from_dict(*result_dict: Dict[str, Any]*, ***kwargs: Any*) → *BaseGPTIndex*

Load index from dict.

classmethod load_from_disk(*save_path: str*, ***kwargs: Any*) → *BaseGPTIndex*

Load index from disk.

This method loads the index from a JSON file stored on disk. The index data structure itself is preserved completely. If the index is defined over subindices, those subindices will also be preserved (and subindices of those subindices, etc.).

NOTE: load_from_disk should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_disk* and *load_from_disk* on that instead.

Parameters
save_path (*str*) – The save_path of the file.

Returns
The loaded index.

Return type
BaseGPTIndex

classmethod load_from_string(*index_string: str*, ***kwargs: Any*) → *BaseGPTIndex*

Load index from string (in JSON-format).

This method loads the index from a JSON string. The index data structure itself is preserved completely. If the index is defined over subindices, those subindices will also be preserved (and subindices of those subindices, etc.).

NOTE: load_from_string should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_string* and *load_from_string* on that instead.

Parameters
index_string (*str*) – The index string (in JSON-format).

Returns
The loaded index.

Return type
BaseGPTIndex

query(*query_str: Union[str,* QueryBundle*]*, *mode: str = QueryMode.DEFAULT*, *query_transform: Optional[BaseQueryTransform] = None*, *use_async: bool = False*, ***query_kwargs: Any*) → Union[*Response*, *StreamingResponse*]

Answer a query.

When *query* is called, we query the index with the given *mode* and *query_kwargs*. The *mode* determines the type of query to run, and *query_kwargs* are parameters that are specific to the query type.

For a comprehensive documentation of available *mode* and *query_kwargs* to query a given index, please visit *Querying an Index*.

property query_context: Dict[str, Any]

Additional context necessary for making a query.

This should capture any index-specific clients, services, etc, that's not captured by index struct, docstore, and service context. For example, a vector store index would pass vector store.

refresh(*documents: Sequence[*Document*]*, ***update_kwargs: Any*) → List[bool]

Refresh an index with documents that have changed.

This allows users to save LLM and Embedding model calls, while only updating documents that have any changes in text or extra_info. It will also insert any documents that previously were not stored.

save_to_dict(***save_kwargs: Any*) → dict

Save to dict.

save_to_disk(*save_path: str*, *encoding: str = 'ascii'*, ***save_kwargs: Any*) → None

Save to file.

This method stores the index into a JSON file stored on disk.

NOTE: save_to_disk should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_disk* and *load_from_disk* on that instead.

Parameters

- **save_path** (*str*) – The save_path of the file.
- **encoding** (*str*) – The encoding of the file.

save_to_string(***save_kwargs: Any*) → str

Save to string.

This method stores the index into a JSON string.

NOTE: save_to_string should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_string* and *load_from_string* on that instead.

Returns
The JSON string of the index.

Return type
str

update(*document:* Document, ***update_kwargs: Any*) → None

Update a document.

This is equivalent to deleting the document and then inserting it again.

Parameters

- **document** (*Union[BaseDocument,* BaseGPTIndex*]*) – document to update
- **insert_kwargs** (*Dict*) – kwargs to pass to insert
- **delete_kwargs** (*Dict*) – kwargs to pass to delete

3.17.3 Tree Index

Building the Tree Index

Tree-structured Index Data Structures.

class gpt_index.indices.tree.**GPTTreeIndex**(*nodes: Optional[Sequence[Node]] = None*, *index_struct: Optional[IndexGraph] = None*, *service_context: Optional[ServiceContext] = None*, *summary_template: Optional[SummaryPrompt] = None*, *insert_prompt: Optional[TreeInsertPrompt] = None*, *num_children: int = 10*, *build_tree: bool = True*, *use_async: bool = False*, ***kwargs: Any*)

GPT Tree Index.

The tree index is a tree-structured index, where each node is a summary of the children nodes. During index construction, the tree is constructed in a bottoms-up fashion until we end up with a set of root_nodes.

There are a few different options during query time (see *Querying an Index*). The main option is to traverse down the tree from the root nodes. A secondary answer is to directly synthesize the answer from the root nodes.

Parameters

- **summary_template** (*Optional[SummaryPrompt]*) – A Summarization Prompt (see *Prompt Templates*).
- **insert_prompt** (*Optional[TreeInsertPrompt]*) – An Tree Insertion Prompt (see *Prompt Templates*).
- **num_children** (*int*) – The number of children each node should have.
- **build_tree** (*bool*) – Whether to build the tree during index construction.

async aquery(*query_str: Union[str, QueryBundle]*, *mode: str = QueryMode.DEFAULT*, *query_transform: Optional[BaseQueryTransform] = None*, ***query_kwargs: Any*) → Union[*Response*, *StreamingResponse*]

Asynchronously answer a query.

When *query* is called, we query the index with the given *mode* and *query_kwargs*. The *mode* determines the type of query to run, and *query_kwargs* are parameters that are specific to the query type.

For a comprehensive documentation of available *mode* and *query_kwargs* to query a given index, please visit *Querying an Index*.

delete(*doc_id: str*, ***delete_kwargs: Any*) → None

Delete a document from the index.

All nodes in the index related to the index will be deleted.

Parameters

doc_id (*str*) – document id

property docstore: *BaseDocumentStore*

Get the docstore corresponding to the index.

classmethod from_documents(*documents: Sequence[Document]*, *docstore: Optional[BaseDocumentStore] = None*, *service_context: Optional[ServiceContext] = None*, ***kwargs: Any*) → *BaseGPTIndex*

Create index from documents.

Parameters
documents (*Optional[Sequence[BaseDocument]]*) – List of documents to build the index from.

classmethod get_query_map() → Dict[str, Type[*BaseGPTIndexQuery*]]

Get query map.

property index_struct: IS

Get the index struct.

index_struct_cls

alias of IndexGraph

insert(*document:* Document, ***insert_kwargs: Any*) → None

Insert a document.

classmethod load_from_dict(*result_dict: Dict[str, Any]*, ***kwargs: Any*) → *BaseGPTIndex*

Load index from dict.

classmethod load_from_disk(*save_path: str*, ***kwargs: Any*) → *BaseGPTIndex*

Load index from disk.

This method loads the index from a JSON file stored on disk. The index data structure itself is preserved completely. If the index is defined over subindices, those subindices will also be preserved (and subindices of those subindices, etc.).

NOTE: load_from_disk should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_disk* and *load_from_disk* on that instead.

Parameters
save_path (*str*) – The save_path of the file.

Returns
The loaded index.

Return type
BaseGPTIndex

classmethod load_from_string(*index_string: str*, ***kwargs: Any*) → *BaseGPTIndex*

Load index from string (in JSON-format).

This method loads the index from a JSON string. The index data structure itself is preserved completely. If the index is defined over subindices, those subindices will also be preserved (and subindices of those subindices, etc.).

NOTE: load_from_string should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_string* and *load_from_string* on that instead.

Parameters
index_string (*str*) – The index string (in JSON-format).

Returns
The loaded index.

Return type
BaseGPTIndex

query(*query_str: Union[str,* QueryBundle*]*, *mode: str = QueryMode.DEFAULT*, *query_transform: Optional[BaseQueryTransform] = None*, *use_async: bool = False*, ***query_kwargs: Any*) → Union[*Response*, *StreamingResponse*]

Answer a query.

When *query* is called, we query the index with the given *mode* and *query_kwargs*. The *mode* determines the type of query to run, and *query_kwargs* are parameters that are specific to the query type.

For a comprehensive documentation of available *mode* and *query_kwargs* to query a given index, please visit *Querying an Index*.

property query_context: Dict[str, Any]

Additional context necessary for making a query.

This should capture any index-specific clients, services, etc, that's not captured by index struct, docstore, and service context. For example, a vector store index would pass vector store.

refresh(*documents: Sequence[Document]*, ***update_kwargs: Any*) → List[bool]

Refresh an index with documents that have changed.

This allows users to save LLM and Embedding model calls, while only updating documents that have any changes in text or extra_info. It will also insert any documents that previously were not stored.

save_to_dict(***save_kwargs: Any*) → dict

Save to dict.

save_to_disk(*save_path: str*, *encoding: str = 'ascii'*, ***save_kwargs: Any*) → None

Save to file.

This method stores the index into a JSON file stored on disk.

NOTE: save_to_disk should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_disk* and *load_from_disk* on that instead.

Parameters

- **save_path** (*str*) – The save_path of the file.
- **encoding** (*str*) – The encoding of the file.

save_to_string(***save_kwargs: Any*) → str

Save to string.

This method stores the index into a JSON string.

NOTE: save_to_string should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_string* and *load_from_string* on that instead.

Returns
The JSON string of the index.

Return type
str

update(*document: Document*, ***update_kwargs: Any*) → None

Update a document.

This is equivalent to deleting the document and then inserting it again.

Parameters

- **document** (*Union[BaseDocument, BaseGPTIndex]*) – document to update
- **insert_kwargs** (*Dict*) – kwargs to pass to insert
- **delete_kwargs** (*Dict*) – kwargs to pass to delete

class gpt_index.indices.tree.**GPTTreeIndexEmbeddingQuery**(*index_struct: IndexGraph*, *query_template: Optional[TreeSelectPrompt] = None*, *query_template_multiple: Optional[TreeSelectMultiplePrompt] = None*, *child_branch_factor: int = 1*, ***kwargs: Any*)

GPT Tree Index embedding query.

This class traverses the index graph using the embedding similarity between the query and the node text.

```
response = index.query("<query_str>", mode="embedding")
```

Parameters

- **query_template** (*Optional[TreeSelectPrompt]*) – Tree Select Query Prompt (see *Prompt Templates*).
- **query_template_multiple** (*Optional[TreeSelectMultiplePrompt]*) – Tree Select Query Prompt (Multiple) (see *Prompt Templates*).
- **text_qa_template** (*Optional[QuestionAnswerPrompt]*) – Question-Answer Prompt (see *Prompt Templates*).
- **refine_template** (*Optional[RefinePrompt]*) – Refinement Prompt (see *Prompt Templates*).
- **child_branch_factor** (*int*) – Number of child nodes to consider at each level. If child_branch_factor is 1, then the query will only choose one child node to traverse for any given parent node. If child_branch_factor is 2, then the query will choose two child nodes.
- **embed_model** (*Optional[BaseEmbedding]*) – Embedding model to use for embedding similarity.

property index_struct: IS

Get the index struct.

retrieve(*query_bundle:* QueryBundle) → List[*NodeWithScore*]

Get list of tuples of node and similarity for response.

First part of the tuple is the node. Second part of tuple is the distance from query to the node. If not applicable, it's None.

class gpt_index.indices.tree.**GPTTreeIndexLeafQuery**(*index_struct: IndexGraph*, *query_template: Optional[TreeSelectPrompt] = None*, *text_qa_template: Optional[QuestionAnswerPrompt] = None*, *refine_template: Optional[RefinePrompt] = None*, *query_template_multiple: Optional[TreeSelectMultiplePrompt] = None*, *child_branch_factor: int = 1*, ***kwargs: Any*)

GPT Tree Index leaf query.

This class traverses the index graph and searches for a leaf node that can best answer the query.

```
response = index.query("<query_str>", mode="default")
```

Parameters

- **query_template** (*Optional[TreeSelectPrompt]*) – Tree Select Query Prompt (see *Prompt Templates*).
- **query_template_multiple** (*Optional[TreeSelectMultiplePrompt]*) – Tree Select Query Prompt (Multiple) (see *Prompt Templates*).
- **child_branch_factor** (*int*) – Number of child nodes to consider at each level. If child_branch_factor is 1, then the query will only choose one child node to traverse for any given parent node. If child_branch_factor is 2, then the query will choose two child nodes.

property index_struct: IS

Get the index struct.

retrieve(*query_bundle:* QueryBundle) → List[*NodeWithScore*]

Get list of tuples of node and similarity for response.

First part of the tuple is the node. Second part of tuple is the distance from query to the node. If not applicable, it's None.

class gpt_index.indices.tree.**GPTTreeIndexRetQuery**(*index_struct: IS*, *service_context:* ServiceContext, *response_synthesizer: ResponseSynthesizer*, *docstore: Optional[*BaseDocumentStore*] = None*, *node_postprocessors: Optional[List[BaseNodePostprocessor]] = None*, *include_extra_info: bool = False*, *verbose: bool = False*)

GPT Tree Index retrieve query.

This class directly retrieves the answer from the root nodes.

Unlike GPTTreeIndexLeafQuery, this class assumes the graph already stores the answer (because it was constructed with a query_str), so it does not attempt to parse information down the graph in order to synthesize an answer.

```
response = index.query("<query_str>", mode="retrieve")
```

Parameters

text_qa_template (*Optional[QuestionAnswerPrompt]*) – Question-Answer Prompt (see *Prompt Templates*).

property index_struct: IS

Get the index struct.

retrieve(*query_bundle:* QueryBundle) → List[*NodeWithScore*]

Get list of tuples of node and similarity for response.

First part of the tuple is the node. Second part of tuple is the distance from query to the node. If not applicable, it's None.

3.17.4 Vector Store Index

Below we show the vector store index classes.

Each vector store index class is a combination of a base vector store index class and a vector store, shown below.

NOTE: the vector store is currently not user-facing but will be soon!

Vector Stores

Vector stores.

class gpt_index.vector_stores.**ChatGPTRetrievalPluginClient**(*endpoint_url: str*, *bearer_token: Optional[str] = None*, *retries: Optional[Retry] = None*, *batch_size: int = 100*, ***kwargs: Any*)

ChatGPT Retrieval Plugin Client.

In this client, we make use of the endpoints defined by ChatGPT.

Parameters

- **endpoint_url** (*str*) – URL of the ChatGPT Retrieval Plugin.
- **bearer_token** (*Optional[str]*) – Bearer token for the ChatGPT Retrieval Plugin.
- **retries** (*Optional[Retry]*) – Retry object for the ChatGPT Retrieval Plugin.
- **batch_size** (*int*) – Batch size for the ChatGPT Retrieval Plugin.

add(*embedding_results: List[NodeEmbeddingResult]*) → List[str]

Add embedding_results to index.

property client: None

Get client.

property config_dict: dict

Get config dict.

query(*query: VectorStoreQuery*) → VectorStoreQueryResult

Get nodes for response.

class gpt_index.vector_stores.**ChromaVectorStore**(*chroma_collection: Any*, ***kwargs: Any*)

Chroma vector store.

In this vector store, embeddings are stored within a ChromaDB collection.

During query time, the index uses ChromaDB to query for the top k most similar nodes.

Parameters

chroma_collection (*chromadb.api.models.Collection.Collection*) – ChromaDB collection instance

add(*embedding_results: List[NodeEmbeddingResult]*) → List[str]

Add embedding results to index.

Args

embedding_results: List[NodeEmbeddingResult]: list of embedding results

property client: Any

Return client.

property config_dict: dict

Return config dict.

query(*query: VectorStoreQuery*) → VectorStoreQueryResult

Query index for top k most similar nodes.

Parameters

- **query_embedding** (*List[float]*) – query embedding
- **similarity_top_k** (*int*) – top k most similar nodes

class gpt_index.vector_stores.**DeepLakeVectorStore**(*dataset_path: str = 'llama_index'*, *token: Optional[str] = None*, *read_only: Optional[bool] = False*, *ingestion_batch_size: int = 1024*, *ingestion_num_workers: int = 4*, *overwrite: bool = False*)

The DeepLake Vector Store.

In this vector store we store the text, its embedding and a few pieces of its metadata in a deeplake dataset. This implemnetation allows the use of an already existing deeplake dataset if it is one that was created this vector store. It also supports creating a new one if the dataset doesnt exist or if *overwrite* is set to True.

Parameters

- **deeplake_path** (*str, optional*) – Path to the deeplake dataset, where data will be
- **"llama_index".** (*stored. Defaults to*) –
- **overwrite** (*bool, optional*) – Whether to overwrite existing dataset with same name. Defaults to False.
- **token** (*str, optional*) – the deeplake token that allows you to access the dataset with proper access. Defaults to None.
- **read_only** (*bool, optional*) – Whether to open the dataset with read only mode.
- **ingestion_batch_size** (*bool, 1024*) – used for controlling batched data injestion to deeplake dataset. Defaults to 1024.
- **injestion_num_workers** (*int, 1*) – number of workers to use during data injestion. Defaults to 4.
- **overwrite** – Whether to overwrite existing dataset with the new dataset with the same name.

Raises

- **ImportError** – Unable to import *deeplake*.
- **UserNotLoggedinException** – When user is not logged in with credentials or token.
- **TokenPermissionError** – When dataset does not exist or user doesn't have enough permissions to modify the dataset.
- **InvalidTokenException** – If the specified token is invalid

Returns

Vectorstore that supports add, delete, and query.

Return type

DeepLakeVectorstore

add(*embedding_results: List[NodeEmbeddingResult]*) → List[str]

Add the embeddings and their nodes into DeepLake.

Parameters
embedding_results (*List[NodeEmbeddingResult]*) – The embeddings and their data to insert.

Raises

- **UserNotLoggedinException** – When user is not logged in with credentials or token.
- **TokenPermissionError** – When dataset does not exist or user doesn't have enough permissions to modify the dataset.
- **InvalidTokenException** – If the specified token is invalid

Returns
List of ids inserted.

Return type
List[str]

property client: None

Get client.

property config_dict: dict

Return config dict.

classmethod from_dict(*config_dict: Dict[str, Any]*) → VectorStore

Initialize a VectorStore from a dictionary

Parameters
config_dict (*Dict[str, Any]*) – dictionary of configuration

Returns
loaded DeepLakeVectorStore

Return type
VectorStore

query(*query: VectorStoreQuery*) → VectorStoreQueryResult

Query index for top k most similar nodes.

Parameters

- **query_embedding** (*List[float]*) – query embedding
- **similarity_top_k** (*int*) – top k most similar nodes

class gpt_index.vector_stores.**FaissVectorStore**(*faiss_index: Any*, *save_path: Optional[str] = None*)

Faiss Vector Store.

Embeddings are stored within a Faiss index.

During query time, the index uses Faiss to query for the top k embeddings, and returns the corresponding indices.

Parameters
faiss_index (*faiss.Index*) – Faiss index instance

add(*embedding_results: List[NodeEmbeddingResult]*) → List[str]

Add embedding results to index.

NOTE: in the Faiss vector store, we do not store text in Faiss.

Args
embedding_results: List[NodeEmbeddingResult]: list of embedding results

property client: Any
Return the faiss index.

property config_dict: dict
Return config dict.

classmethod load(*save_path: str*) → *FaissVectorStore*
Load vector store from disk.

Parameters
save_path (*str*) – The save_path of the file.

Returns
The loaded vector store.

Return type
FaissVectorStore

query(*query: VectorStoreQuery*) → VectorStoreQueryResult
Query index for top k most similar nodes.

Parameters
- **query_embedding** (*List[float]*) – query embedding
- **similarity_top_k** (*int*) – top k most similar nodes

save(*save_path: str*) → None
Save to file.

This method saves the vector store to disk.

Parameters
save_path (*str*) – The save_path of the file.

class gpt_index.vector_stores.**MilvusVectorStore**(*collection_name: str = 'llamalection', index_params: Optional[dict] = None, search_params: Optional[dict] = None, dim: Optional[int] = None, host: str = 'localhost', port: int = 19530, user: str = '', password: str = '', use_secure: bool = False, overwrite: bool = False, **kwargs: Any*)
The Milvus Vector Store.

In this vector store we store the text, its embedding and a few pieces of its metadata in a Milvus collection. This implemnetation allows the use of an already existing collection if it is one that was created this vector store. It also supports creating a new one if the collection doesnt exist or if *overwrite* is set to True.

Parameters
- **collection_name** (*str, optional*) – The name of the collection where data will be stored. Defaults to "llamalection".
- **index_params** (*dict, optional*) – The index parameters for Milvus, if none are provided an HNSW index will be used. Defaults to None.
- **search_params** (*dict, optional*) – The search parameters for a Milvus query. If none are provided, default params will be generated. Defaults to None.
- **dim** (*int, optional*) – The dimension of the embeddings. If it is not provided, collection creation will be done on first insert. Defaults to None.

- **host** (*str, optional*) – The host address of Milvus. Defaults to "localhost".
- **port** (*int, optional*) – The port of Milvus. Defaults to 19530.
- **user** (*str, optional*) – The username for RBAC. Defaults to "".
- **password** (*str, optional*) – The password for RBAC. Defaults to "".
- **use_secure** (*bool, optional*) – Use https. Required for Zilliz Cloud. Defaults to False.
- **overwrite** (*bool, optional*) – Whether to overwrite existing collection with same name. Defaults to False.

Raises

- **ImportError** – Unable to import *pymilvus*.
- **MilvusException** – Error communicating with Milvus, more can be found in logging under Debug.

Returns
Vectorstore that supports add, delete, and query.

Return type
MilvusVectorstore

add(*embedding_results: List[NodeEmbeddingResult]*) → List[str]

Add the embeddings and their nodes into Milvus.

Parameters
embedding_results (*List[NodeEmbeddingResult]*) – The embeddings and their data to insert.

Raises
MilvusException – Failed to insert data.

Returns
List of ids inserted.

Return type
List[str]

property client: Any

Get client.

property config_dict: dict

Return config dict.

query(*query: VectorStoreQuery*) → VectorStoreQueryResult

Query index for top k most similar nodes.

Parameters

- **query_embedding** (*List[float]*) – query embedding
- **similarity_top_k** (*int*) – top k most similar nodes
- **doc_ids** (*Optional[List[str]]*) – list of doc_ids to filter by

class gpt_index.vector_stores.**MyScaleVectorStore**(*myscale_client: Optional[Any] = None*, *table: str = 'llama_index'*, *database: str = 'default'*, *index_type: str = 'IVFFLAT'*, *metric: str = 'cosine'*, *batch_size: int = 32*, *index_params: Optional[dict] = None*, *search_params: Optional[dict] = None*, *service_context: Optional[ServiceContext] = None*, ***kwargs: Any*)

MyScale Vector Store.

In this vector store, embeddings and docs are stored within an existing MyScale cluster.

During query time, the index uses MyScale to query for the top k most similar nodes.

Parameters

- **myscale_client** (*httpclient*) – clickhouse-connect httpclient of an existing MyScale cluster.
- **table** (*str, optional*) – The name of the MyScale table where data will be stored. Defaults to "llama_index".
- **database** (*str, optional*) – The name of the MyScale database where data will be stored. Defaults to "default".
- **index_type** (*str, optional*) – The type of the MyScale vector index. Defaults to "IVFFLAT".
- **metric** (*str, optional*) – The metric type of the MyScale vector index. Defaults to "cosine".
- **batch_size** (*int, optional*) – the size of documents to insert. Defaults to 32.
- **index_params** (*dict, optional*) – The index parameters for MyScale. Defaults to None.
- **search_params** (*dict, optional*) – The search parameters for a MyScale query. Defaults to None.
- **service_context** (ServiceContext, *optional*) – Vector store service context. Defaults to None

add(*embedding_results: List[NodeEmbeddingResult]*) → List[str]

Add embedding results to index.

Args

embedding_results: List[NodeEmbeddingResult]: list of embedding results

property client: Any

Get client.

property config_dict: dict

Return config dict.

drop() → None

Drop MyScale Index and table

query(*query: VectorStoreQuery*) → VectorStoreQueryResult

Query index for top k most similar nodes.

Parameters

query (*VectorStoreQuery*) – query

class gpt_index.vector_stores.**OpensearchVectorClient**(*endpoint: str*, *index: str*, *dim: int*, *embedding_field: str = 'embedding'*, *text_field: str = 'content'*, *method: Optional[dict] = None*, *auth: Optional[dict] = None*)

Object encapsulating an Opensearch index that has vector search enabled.

If the index does not yet exist, it is created during init. Therefore, the underlying index is assumed to either: 1) not exist yet or 2) be created due to previous usage of this class.

Parameters

- **endpoint** (*str*) – URL (http/https) of elasticsearch endpoint
- **index** (*str*) – Name of the elasticsearch index
- **dim** (*int*) – Dimension of the vector
- **embedding_field** (*str*) – Name of the field in the index to store embedding array in.
- **text_field** (*str*) – Name of the field to grab text from
- **method** (*Optional[dict]*) – Opensearch "method" JSON obj for configuring the KNN index. This includes engine, metric, and other config params. Defaults to: {"name": "hnsw", "space_type": "l2", "engine": "faiss", "parameters": {"ef_construction": 256, "m": 48}}

delete_doc_id(*doc_id: str*) → None

Delete a document.

Parameters
doc_id (*str*) – document id

do_approx_knn(*query_embedding: List[float]*, *k: int*) → VectorStoreQueryResult

Do approximate knn.

index_results(*results: List[NodeEmbeddingResult]*) → List[str]

Store results in the index.

class gpt_index.vector_stores.**OpensearchVectorStore**(*client: OpensearchVectorClient*)

Elasticsearch/Opensearch vector store.

Parameters
client (OpensearchVectorClient) – Vector index client to use for data insertion/querying.

add(*embedding_results: List[NodeEmbeddingResult]*) → List[str]

Add embedding results to index.

Args
embedding_results: List[NodeEmbeddingResult]: list of embedding results

property client: Any

Get client.

property config_dict: dict

Get config dict.

query(*query: VectorStoreQuery*) → VectorStoreQueryResult

Query index for top k most similar nodes.

Parameters

- **query_embedding** (*List[float]*) – query embedding
- **similarity_top_k** (*int*) – top k most similar nodes

class gpt_index.vector_stores.**PineconeVectorStore**(*pinecone_index: Optional[Any] = None*, *index_name: Optional[str] = None*, *environment: Optional[str] = None*, *namespace: Optional[str] = None*, *metadata_filters: Optional[Dict[str, Any]] = None*, *pinecone_kwargs: Optional[Dict] = None*, *insert_kwargs: Optional[Dict] = None*, *query_kwargs: Optional[Dict] = None*, *delete_kwargs: Optional[Dict] = None*, *add_sparse_vector: bool = False*, *tokenizer: Optional[Callable] = None*, ***kwargs: Any*)

Pinecone Vector Store.

In this vector store, embeddings and docs are stored within a Pinecone index.

During query time, the index uses Pinecone to query for the top k most similar nodes.

Parameters

- **pinecone_index** (*Optional[pinecone.Index]*) – Pinecone index instance
- **pinecone_kwargs** (*Optional[Dict]*) – kwargs to pass to Pinecone index. NOTE: deprecated. If specified, then insert_kwargs, query_kwargs, and delete_kwargs cannot be specified.
- **insert_kwargs** (*Optional[Dict]*) – insert kwargs during *upsert* call.
- **query_kwargs** (*Optional[Dict]*) – query kwargs during *query* call.
- **delete_kwargs** (*Optional[Dict]*) – delete kwargs during *delete* call.
- **add_sparse_vector** (*bool*) – whether to add sparse vector to index.
- **tokenizer** (*Optional[Callable]*) – tokenizer to use to generate sparse

add(*embedding_results: List[NodeEmbeddingResult]*) → List[str]

Add embedding results to index.

Args

embedding_results: List[NodeEmbeddingResult]: list of embedding results

property client: Any

Return Pinecone client.

property config_dict: dict

Return config dict.

query(*query: VectorStoreQuery*) → VectorStoreQueryResult

Query index for top k most similar nodes.

Parameters

- **query_embedding** (*List[float]*) – query embedding
- **similarity_top_k** (*int*) – top k most similar nodes

class gpt_index.vector_stores.**QdrantVectorStore**(*collection_name: str*, *client: Optional[Any] = None*, ***kwargs: Any*)

Qdrant Vector Store.

In this vector store, embeddings and docs are stored within a Qdrant collection.

During query time, the index uses Qdrant to query for the top k most similar nodes.

Parameters

- **collection_name** – (str): name of the Qdrant collection
- **client** (*Optional[Any]*) – QdrantClient instance from *qdrant-client* package

add(*embedding_results: List[NodeEmbeddingResult]*) → List[str]

Add embedding results to index.

Args
embedding_results: List[NodeEmbeddingResult]: list of embedding results

property client: Any

Return the Qdrant client.

property config_dict: dict

Return config dict.

query(*query: VectorStoreQuery*) → VectorStoreQueryResult

Query index for top k most similar nodes.

Parameters
query (*VectorStoreQuery*) – query

class gpt_index.vector_stores.**SimpleVectorStore**(*simple_vector_store_data_dict: Optional[dict] = None*, ***kwargs: Any*)

Simple Vector Store.

In this vector store, embeddings are stored within a simple, in-memory dictionary.

Parameters
simple_vector_store_data_dict (*Optional[dict]*) – data dict containing the embeddings and doc_ids. See SimpleVectorStoreData for more details.

add(*embedding_results: List[NodeEmbeddingResult]*) → List[str]

Add embedding_results to index.

property client: None

Get client.

property config_dict: dict

Get config dict.

get(*text_id: str*) → List[float]

Get embedding.

query(*query: VectorStoreQuery*) → VectorStoreQueryResult

Get nodes for response.

class gpt_index.vector_stores.**WeaviateVectorStore**(*weaviate_client: Optional[Any] = None*, *class_prefix: Optional[str] = None*, ***kwargs: Any*)

Weaviate vector store.

In this vector store, embeddings and docs are stored within a Weaviate collection.

During query time, the index uses Weaviate to query for the top k most similar nodes.

Parameters

- **weaviate_client** (*weaviate.Client*) – WeaviateClient instance from *weaviate-client* package

- **class_prefix** (*Optional[str]*) – prefix for Weaviate classes

add(*embedding_results: List[NodeEmbeddingResult]*) → List[str]

Add embedding results to index.

Args
embedding_results: List[NodeEmbeddingResult]: list of embedding results

property client: Any

Get client.

property config_dict: dict

Get config dict.

query(*query: VectorStoreQuery*) → VectorStoreQueryResult

Query index for top k most similar nodes.

Base Vector Index class

Base vector store index.

An index that that is built on top of an existing vector store.

class gpt_index.indices.vector_store.base.**GPTVectorStoreIndex**(*nodes: Optional[Sequence[Node]] = None*, *index_struct: Optional[IndexDict] = None*, *service_context: Optional[ServiceContext] = None*, *vector_store: Optional[VectorStore] = None*, *use_async: bool = False*, ***kwargs: Any*)

Base GPT Vector Store Index.

Parameters

- **embed_model** (*Optional[BaseEmbedding]*) – Embedding model to use for embedding similarity.
- **vector_store** (*Optional[VectorStore]*) – Vector store to use for embedding similarity. See *Vector Stores* for more details.
- **use_async** (*bool*) – Whether to use asynchronous calls. Defaults to False.

async aquery(*query_str: Union[str, QueryBundle]*, *mode: str = QueryMode.DEFAULT*, *query_transform: Optional[BaseQueryTransform] = None*, ***query_kwargs: Any*) → Union[*Response*, *StreamingResponse*]

Asynchronously answer a query.

When *query* is called, we query the index with the given *mode* and *query_kwargs*. The *mode* determines the type of query to run, and *query_kwargs* are parameters that are specific to the query type.

For a comprehensive documentation of available *mode* and *query_kwargs* to query a given index, please visit *Querying an Index*.

classmethod from_documents(*documents: Sequence[Document]*, *docstore: Optional[BaseDocumentStore] = None*, *service_context: Optional[ServiceContext] = None*, ***kwargs: Any*) → *BaseGPTIndex*

Create index from documents.

Parameters
documents (*Optional[Sequence[BaseDocument]]*) – List of documents to build the index from.

classmethod get_query_map() → Dict[str, Type[*BaseGPTIndexQuery*]]

Get query map.

insert(*document:* Document, ***insert_kwargs: Any*) → None

Insert a document.

classmethod load_from_dict(*result_dict: Dict[str, Any]*, ***kwargs: Any*) → *BaseGPTIndex*

Load index from string (in JSON-format).

This method loads the index from a JSON string. The index data structure itself is preserved completely. If the index is defined over subindices, those subindices will also be preserved (and subindices of those subindices, etc.).

NOTE: load_from_string should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_string* and *load_from_string* on that instead.

Parameters
index_string (*str*) – The index string (in JSON-format).

Returns
The loaded index.

Return type
BaseGPTIndex

classmethod load_from_disk(*save_path: str*, ***kwargs: Any*) → *BaseGPTIndex*

Load index from disk.

This method loads the index from a JSON file stored on disk. The index data structure itself is preserved completely. If the index is defined over subindices, those subindices will also be preserved (and subindices of those subindices, etc.).

NOTE: load_from_disk should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_disk* and *load_from_disk* on that instead.

Parameters
save_path (*str*) – The save_path of the file.

Returns
The loaded index.

Return type
BaseGPTIndex

classmethod load_from_string(*index_string: str*, ***kwargs: Any*) → *BaseGPTIndex*

Load index from string (in JSON-format).

This method loads the index from a JSON string. The index data structure itself is preserved completely. If the index is defined over subindices, those subindices will also be preserved (and subindices of those subindices, etc.).

NOTE: load_from_string should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_string* and *load_from_string* on that instead.

Parameters
index_string (*str*) – The index string (in JSON-format).

Returns
The loaded index.

Return type
BaseGPTIndex

query(*query_str: Union[str, QueryBundle]*, *mode: str = QueryMode.DEFAULT*, *query_transform: Optional[BaseQueryTransform] = None*, *use_async: bool = False*, ***query_kwargs: Any*) → Union[*Response*, *StreamingResponse*]

Answer a query.

When *query* is called, we query the index with the given *mode* and *query_kwargs*. The *mode* determines the type of query to run, and *query_kwargs* are parameters that are specific to the query type.

For a comprehensive documentation of available *mode* and *query_kwargs* to query a given index, please visit *Querying an Index*.

property query_context: Dict[str, Any]

Additional context necessary for making a query.

This should capture any index-specific clients, services, etc, that's not captured by index struct, docstore, and service context. For example, a vector store index would pass vector store.

refresh(*documents: Sequence[Document]*, ***update_kwargs: Any*) → List[bool]

Refresh an index with documents that have changed.

This allows users to save LLM and Embedding model calls, while only updating documents that have any changes in text or extra_info. It will also insert any documents that previously were not stored.

save_to_dict(***save_kwargs: Any*) → dict

Save to string.

This method stores the index into a JSON string.

NOTE: save_to_string should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_string* and *load_from_string* on that instead.

Returns
The JSON dict of the index.

Return type
dict

save_to_disk(*save_path: str*, *encoding: str = 'ascii'*, ***save_kwargs: Any*) → None

Save to file.

This method stores the index into a JSON file stored on disk.

NOTE: save_to_disk should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_disk* and *load_from_disk* on that instead.

Parameters

- **save_path** (*str*) – The save_path of the file.
- **encoding** (*str*) – The encoding of the file.

save_to_string(***save_kwargs: Any*) → str

Save to string.

This method stores the index into a JSON string.

NOTE: save_to_string should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_string* and *load_from_string* on that instead.

Returns
The JSON string of the index.

Return type
str

update(*document:* Document, ***update_kwargs: Any*) → None

Update a document.

This is equivalent to deleting the document and then inserting it again.

Parameters

- **document** (*Union[BaseDocument, BaseGPTIndex]*) – document to update
- **insert_kwargs** (*Dict*) – kwargs to pass to insert
- **delete_kwargs** (*Dict*) – kwargs to pass to delete

Deprecated vector store indices.

class gpt_index.indices.vector_store.vector_indices.**ChatGPTRetrievalPluginIndex**(*nodes: Optional[Sequence[Node]] = None*, *index_struct: Optional[ChatGPTRetrievalPlugi = None*, *service_context: Optional[ServiceContext] = None*, *endpoint_url: Optional[str] = None*, *bearer_token: Optional[str] = None*, *retries: Optional[Retry] = None*, *batch_size: int = 100*, *vector_store: Optional[ChatGPTRetrievalPlugi = None*, ***kwargs: Any*)

ChatGPTRetrievalPlugin index.

This index directly interfaces with any server that hosts the ChatGPT Retrieval Plugin interface: https://github.com/openai/chatgpt-retrieval-plugin.

Parameters

- **client** (*Optional[OpensearchVectorClient]*) – The client which encapsulates logic for using Opensearch as a vector store (that is, it holds stuff like endpoint, index_name and performs operations like initializing the index and adding new doc/embeddings to said index).
- **service_context** (ServiceContext) – Service context container (contains components like LLMPredictor, PromptHelper, etc.).

async aquery(*query_str: Union[str, QueryBundle]*, *mode: str = QueryMode.DEFAULT*, *query_transform: Optional[BaseQueryTransform] = None*, ***query_kwargs: Any*) → Union[*Response*, *StreamingResponse*]

Asynchronously answer a query.

When *query* is called, we query the index with the given *mode* and *query_kwargs*. The *mode* determines the type of query to run, and *query_kwargs* are parameters that are specific to the query type.

For a comprehensive documentation of available *mode* and *query_kwargs* to query a given index, please visit *Querying an Index*.

classmethod from_documents(*documents: Sequence[Document]*, *docstore: Optional[BaseDocumentStore] = None*, *service_context: Optional[ServiceContext] = None*, ***kwargs: Any*) → *BaseGPTIndex*

Create index from documents.

Parameters
documents (*Optional[Sequence[BaseDocument]]*) – List of documents to build the index from.

classmethod get_query_map() → Dict[str, Type[*BaseGPTIndexQuery*]]

Get query map.

insert(*document: Document*, ***insert_kwargs: Any*) → None

Insert a document.

classmethod load_from_dict(*result_dict: Dict[str, Any]*, ***kwargs: Any*) → *BaseGPTIndex*

Load index from string (in JSON-format).

This method loads the index from a JSON string. The index data structure itself is preserved completely. If the index is defined over subindices, those subindices will also be preserved (and subindices of those subindices, etc.).

NOTE: load_from_string should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_string* and *load_from_string* on that instead.

Parameters
index_string (*str*) – The index string (in JSON-format).

Returns
The loaded index.

Return type
BaseGPTIndex

classmethod load_from_disk(*save_path: str*, ***kwargs: Any*) → *BaseGPTIndex*

Load index from disk.

This method loads the index from a JSON file stored on disk. The index data structure itself is preserved completely. If the index is defined over subindices, those subindices will also be preserved (and subindices of those subindices, etc.).

NOTE: load_from_disk should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_disk* and *load_from_disk* on that instead.

Parameters
save_path (*str*) – The save_path of the file.

Returns
The loaded index.

Return type
BaseGPTIndex

classmethod load_from_string(*index_string: str*, ***kwargs: Any*) → *BaseGPTIndex*

Load index from string (in JSON-format).

This method loads the index from a JSON string. The index data structure itself is preserved completely. If the index is defined over subindices, those subindices will also be preserved (and subindices of those subindices, etc.).

NOTE: load_from_string should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_string* and *load_from_string* on that instead.

Parameters
index_string (*str*) – The index string (in JSON-format).

Returns
The loaded index.

Return type
BaseGPTIndex

query(*query_str: Union[str,* QueryBundle*]*, *mode: str = QueryMode.DEFAULT*, *query_transform: Optional[BaseQueryTransform] = None*, *use_async: bool = False*, ***query_kwargs: Any*) → Union[*Response*, *StreamingResponse*]

Answer a query.

When *query* is called, we query the index with the given *mode* and *query_kwargs*. The *mode* determines the type of query to run, and *query_kwargs* are parameters that are specific to the query type.

For a comprehensive documentation of available *mode* and *query_kwargs* to query a given index, please visit *Querying an Index*.

property query_context: Dict[str, Any]

Additional context necessary for making a query.

This should capture any index-specific clients, services, etc, that's not captured by index struct, docstore, and service context. For example, a vector store index would pass vector store.

refresh(*documents: Sequence[*Document*]*, ***update_kwargs: Any*) → List[bool]

Refresh an index with documents that have changed.

This allows users to save LLM and Embedding model calls, while only updating documents that have any changes in text or extra_info. It will also insert any documents that previously were not stored.

save_to_dict(***save_kwargs: Any*) → dict

Save to string.

This method stores the index into a JSON string.

NOTE: save_to_string should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_string* and *load_from_string* on that instead.

Returns
The JSON dict of the index.

Return type
dict

save_to_disk(*save_path: str*, *encoding: str = 'ascii'*, ***save_kwargs: Any*) → None

Save to file.

This method stores the index into a JSON file stored on disk.

NOTE: save_to_disk should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_disk* and *load_from_disk* on that instead.

Parameters

- **save_path** (*str*) – The save_path of the file.
- **encoding** (*str*) – The encoding of the file.

save_to_string(***save_kwargs: Any*) → str

Save to string.

This method stores the index into a JSON string.

NOTE: save_to_string should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_string* and *load_from_string* on that instead.

Returns
The JSON string of the index.

Return type
str

update(*document: Document*, ***update_kwargs: Any*) → None

Update a document.

This is equivalent to deleting the document and then inserting it again.

Parameters

- **document** (*Union[BaseDocument, BaseGPTIndex]*) – document to update
- **insert_kwargs** (*Dict*) – kwargs to pass to insert
- **delete_kwargs** (*Dict*) – kwargs to pass to delete

class gpt_index.indices.vector_store.vector_indices.**GPTChromaIndex**(*nodes: Optional[Sequence[Node]] = None*, *index_struct: Optional[IndexDict] = None*, *service_context: Optional[ServiceContext] = None*, *chroma_collection: Optional[Any] = None*, *vector_store: Optional[ChromaVectorStore] = None*, ***kwargs: Any*)

GPT Chroma Index.

The GPTChromaIndex is a data structure where nodes are keyed by embeddings, and those embeddings are stored within a Chroma collection. During index construction, the document texts are chunked up, converted to nodes with text; they are then encoded in document embeddings stored within Chroma.

During query time, the index uses Chroma to query for the top k most similar nodes, and synthesizes an answer from the retrieved nodes.

Parameters

- **service_context** (ServiceContext) – Service context container (contains components like LLMPredictor, PromptHelper, etc.).
- **chroma_collection** (*Optional[Any]*) – Collection instance from *chromadb* package.

async aquery(*query_str: Union[str, QueryBundle]*, *mode: str = QueryMode.DEFAULT*, *query_transform: Optional[BaseQueryTransform] = None*, ***query_kwargs: Any*) → Union[*Response*, *StreamingResponse*]

Asynchronously answer a query.

When *query* is called, we query the index with the given *mode* and *query_kwargs*. The *mode* determines the type of query to run, and *query_kwargs* are parameters that are specific to the query type.

For a comprehensive documentation of available *mode* and *query_kwargs* to query a given index, please visit *Querying an Index*.

classmethod from_documents(*documents: Sequence[Document]*, *docstore: Optional[BaseDocumentStore] = None*, *service_context: Optional[ServiceContext] = None*, ***kwargs: Any*) → *BaseGPTIndex*

Create index from documents.

Parameters

documents (*Optional[Sequence[BaseDocument]]*) – List of documents to build the index from.

classmethod get_query_map() → Dict[str, Type[*BaseGPTIndexQuery*]]

Get query map.

insert(*document: Document*, ***insert_kwargs: Any*) → None

Insert a document.

classmethod load_from_dict(*result_dict: Dict[str, Any]*, ***kwargs: Any*) → *BaseGPTIndex*

Load index from string (in JSON-format).

This method loads the index from a JSON string. The index data structure itself is preserved completely. If the index is defined over subindices, those subindices will also be preserved (and subindices of those subindices, etc.).

NOTE: load_from_string should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_string* and *load_from_string* on that instead.

Parameters

index_string (*str*) – The index string (in JSON-format).

Returns

The loaded index.

Return type

BaseGPTIndex

classmethod load_from_disk(*save_path: str*, ***kwargs: Any*) → *BaseGPTIndex*

Load index from disk.

This method loads the index from a JSON file stored on disk. The index data structure itself is preserved completely. If the index is defined over subindices, those subindices will also be preserved (and subindices of those subindices, etc.).

NOTE: load_from_disk should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_disk* and *load_from_disk* on that instead.

Parameters
save_path (*str*) – The save_path of the file.

Returns
The loaded index.

Return type
BaseGPTIndex

classmethod load_from_string(*index_string: str*, ***kwargs: Any*) → *BaseGPTIndex*

Load index from string (in JSON-format).

This method loads the index from a JSON string. The index data structure itself is preserved completely. If the index is defined over subindices, those subindices will also be preserved (and subindices of those subindices, etc.).

NOTE: load_from_string should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_string* and *load_from_string* on that instead.

Parameters
index_string (*str*) – The index string (in JSON-format).

Returns
The loaded index.

Return type
BaseGPTIndex

query(*query_str: Union[str,* QueryBundle*]*, *mode: str = QueryMode.DEFAULT*, *query_transform: Optional[BaseQueryTransform] = None*, *use_async: bool = False*, ***query_kwargs: Any*) → Union[*Response*, *StreamingResponse*]

Answer a query.

When *query* is called, we query the index with the given *mode* and *query_kwargs*. The *mode* determines the type of query to run, and *query_kwargs* are parameters that are specific to the query type.

For a comprehensive documentation of available *mode* and *query_kwargs* to query a given index, please visit *Querying an Index*.

property query_context: Dict[str, Any]

Additional context necessary for making a query.

This should capture any index-specific clients, services, etc, that's not captured by index struct, docstore, and service context. For example, a vector store index would pass vector store.

refresh(*documents: Sequence[*Document*]*, ***update_kwargs: Any*) → List[bool]

Refresh an index with documents that have changed.

This allows users to save LLM and Embedding model calls, while only updating documents that have any changes in text or extra_info. It will also insert any documents that previously were not stored.

save_to_dict(***save_kwargs: Any*) → dict

Save to string.

This method stores the index into a JSON string.

NOTE: save_to_string should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_string* and *load_from_string* on that instead.

Returns
The JSON dict of the index.

Return type
dict

save_to_disk(*save_path: str*, *encoding: str = 'ascii'*, ***save_kwargs: Any*) → None

Save to file.

This method stores the index into a JSON file stored on disk.

NOTE: save_to_disk should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_disk* and *load_from_disk* on that instead.

Parameters

- **save_path** (*str*) – The save_path of the file.
- **encoding** (*str*) – The encoding of the file.

save_to_string(***save_kwargs: Any*) → str

Save to string.

This method stores the index into a JSON string.

NOTE: save_to_string should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_string* and *load_from_string* on that instead.

Returns
The JSON string of the index.

Return type
str

update(*document: Document*, ***update_kwargs: Any*) → None

Update a document.

This is equivalent to deleting the document and then inserting it again.

Parameters

- **document** (*Union[BaseDocument, BaseGPTIndex]*) – document to update
- **insert_kwargs** (*Dict*) – kwargs to pass to insert
- **delete_kwargs** (*Dict*) – kwargs to pass to delete

class gpt_index.indices.vector_store.vector_indices.**GPTDeepLakeIndex**(*nodes: Optional[Sequence[Node]] = None*, *index_struct: Optional[IndexDict] = None*, *service_context: Optional[ServiceContext] = None*, *vector_store: Optional[DeepLakeVectorStore] = None*, *dataset_path: str = 'llama_index'*, *overwrite: bool = False*, *read_only: bool = False*, *ingestion_batch_size: int = 1024*, *ingestion_num_workers: int = 4*, *token: Optional[str] = None*, ***kwargs: Any*)

GPT DeepLake Vector Store.

In this vector store we store the text, its embedding and a few pieces of its metadata in a deeplake dataset. This implemnetation allows the use of an already existing deeplake dataset if it is one that was created this vector store. It also supports creating a new one if the dataset doesnt exist or if *overwrite* is set to True.

Parameters

- **deeplake_path** (*str, optional*) – Path to the deeplake dataset, where data will be
- **"llama_index".** (*stored. Defaults to*) –
- **overwrite** (*bool, optional*) – Whether to overwrite existing dataset with same name. Defaults to False.
- **token** (*str, optional*) – the deeplake token that allows you to access the dataset with proper access. Defaults to None.
- **read_only** (*bool, optional*) – Whether to open the dataset with read only mode.
- **ingestion_batch_size** (*bool*) – used for controlling batched data injestion to deeplake dataset. Defaults to 1024.
- **ingestion_num_workers** (*int*) – number of workers to use during data injestion. Defaults to 4.
- **overwrite** – Whether to overwrite existing dataset with the new dataset with the same name.

Raises

- **ImportError** – Unable to import *deeplake*.
- **UserNotLoggedinException** – When user is not logged in with credentials or token.
- **TokenPermissionError** – When dataset does not exist or user doesn't have enough permissions to modify the dataset.
- **InvalidTokenException** – If the specified token is invalid

Returns

Vectorstore that supports add, delete, and query.

Return type

DeepLakeVectorstore

async aquery(*query_str: Union[str, QueryBundle]*, *mode: str = QueryMode.DEFAULT*, *query_transform: Optional[BaseQueryTransform] = None*, ***query_kwargs: Any*) → Union[*Response*, *StreamingResponse*]

Asynchronously answer a query.

When *query* is called, we query the index with the given *mode* and *query_kwargs*. The *mode* determines the type of query to run, and *query_kwargs* are parameters that are specific to the query type.

For a comprehensive documentation of available *mode* and *query_kwargs* to query a given index, please visit *Querying an Index*.

classmethod from_documents(*documents: Sequence[Document]*, *docstore: Optional[BaseDocumentStore] = None*, *service_context: Optional[ServiceContext] = None*, ***kwargs: Any*) → *BaseGPTIndex*

Create index from documents.

Parameters
documents (*Optional[Sequence[BaseDocument]]*) – List of documents to build the index from.

classmethod get_query_map() → Dict[str, Type[*BaseGPTIndexQuery*]]

Get query map.

insert(*document: Document*, ***insert_kwargs: Any*) → None

Insert a document.

classmethod load_from_dict(*result_dict: Dict[str, Any]*, ***kwargs: Any*) → *BaseGPTIndex*

Load index from string (in JSON-format).

This method loads the index from a JSON string. The index data structure itself is preserved completely. If the index is defined over subindices, those subindices will also be preserved (and subindices of those subindices, etc.).

NOTE: load_from_string should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_string* and *load_from_string* on that instead.

Parameters
index_string (*str*) – The index string (in JSON-format).

Returns
The loaded index.

Return type
BaseGPTIndex

classmethod load_from_disk(*save_path: str*, ***kwargs: Any*) → *BaseGPTIndex*

Load index from disk.

This method loads the index from a JSON file stored on disk. The index data structure itself is preserved completely. If the index is defined over subindices, those subindices will also be preserved (and subindices of those subindices, etc.).

NOTE: load_from_disk should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_disk* and *load_from_disk* on that instead.

Parameters
save_path (*str*) – The save_path of the file.

Returns
The loaded index.

Return type
BaseGPTIndex

classmethod load_from_string(*index_string: str*, ***kwargs: Any*) → *BaseGPTIndex*

Load index from string (in JSON-format).

This method loads the index from a JSON string. The index data structure itself is preserved completely. If the index is defined over subindices, those subindices will also be preserved (and subindices of those subindices, etc.).

NOTE: load_from_string should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_string* and *load_from_string* on that instead.

Parameters
index_string (*str*) – The index string (in JSON-format).

Returns
The loaded index.

Return type
BaseGPTIndex

query(*query_str: Union[str,* QueryBundle*]*, *mode: str = QueryMode.DEFAULT*, *query_transform: Optional[BaseQueryTransform] = None*, *use_async: bool = False*, ***query_kwargs: Any*) → Union[*Response*, *StreamingResponse*]

Answer a query.

When *query* is called, we query the index with the given *mode* and *query_kwargs*. The *mode* determines the type of query to run, and *query_kwargs* are parameters that are specific to the query type.

For a comprehensive documentation of available *mode* and *query_kwargs* to query a given index, please visit *Querying an Index*.

property query_context: Dict[str, Any]

Additional context necessary for making a query.

This should capture any index-specific clients, services, etc, that's not captured by index struct, docstore, and service context. For example, a vector store index would pass vector store.

refresh(*documents: Sequence[*Document*]*, ***update_kwargs: Any*) → List[bool]

Refresh an index with documents that have changed.

This allows users to save LLM and Embedding model calls, while only updating documents that have any changes in text or extra_info. It will also insert any documents that previously were not stored.

save_to_dict(***save_kwargs: Any*) → dict

Save to string.

This method stores the index into a JSON string.

NOTE: save_to_string should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_string* and *load_from_string* on that instead.

Returns
The JSON dict of the index.

Return type
dict

save_to_disk(*save_path: str*, *encoding: str = 'ascii'*, ***save_kwargs: Any*) → None

Save to file.

This method stores the index into a JSON file stored on disk.

NOTE: save_to_disk should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_disk* and *load_from_disk* on that instead.

Parameters

- **save_path** (*str*) – The save_path of the file.
- **encoding** (*str*) – The encoding of the file.

save_to_string(***save_kwargs: Any*) → str

Save to string.

This method stores the index into a JSON string.

NOTE: save_to_string should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_string* and *load_from_string* on that instead.

Returns
The JSON string of the index.

Return type
str

update(*document: Document*, ***update_kwargs: Any*) → None

Update a document.

This is equivalent to deleting the document and then inserting it again.

Parameters

- **document** (*Union[BaseDocument, BaseGPTIndex]*) – document to update
- **insert_kwargs** (*Dict*) – kwargs to pass to insert
- **delete_kwargs** (*Dict*) – kwargs to pass to delete

class gpt_index.indices.vector_store.vector_indices.**GPTFaissIndex**(*nodes: Optional[Sequence[Node]] = None*, *service_context: Optional[ServiceContext] = None*, *faiss_index: Optional[Any] = None*, *index_struct: Optional[IndexDict] = None*, *vector_store: Optional[FaissVectorStore] = None*, ***kwargs: Any*)

GPT Faiss Index.

The GPTFaissIndex is a data structure where nodes are keyed by embeddings, and those embeddings are stored within a Faiss index. During index construction, the document texts are chunked up, converted to nodes with text; they are then encoded in document embeddings stored within Faiss.

During query time, the index uses Faiss to query for the top k most similar nodes, and synthesizes an answer from the retrieved nodes.

Parameters

- **faiss_index** (*faiss.Index*) – A Faiss Index object (required). Note: the index will be reset during index construction.
- **service_context** (ServiceContext) – Service context container (contains components like LLMPredictor, PromptHelper, etc.).

async aquery(*query_str: Union[str, QueryBundle]*, *mode: str = QueryMode.DEFAULT*, *query_transform: Optional[BaseQueryTransform] = None*, ***query_kwargs: Any*) → Union[*Response*, *StreamingResponse*]

Asynchronously answer a query.

When *query* is called, we query the index with the given *mode* and *query_kwargs*. The *mode* determines the type of query to run, and *query_kwargs* are parameters that are specific to the query type.

For a comprehensive documentation of available *mode* and *query_kwargs* to query a given index, please visit *Querying an Index*.

classmethod from_documents(*documents: Sequence[Document]*, *docstore: Optional[BaseDocumentStore] = None*, *service_context: Optional[ServiceContext] = None*, ***kwargs: Any*) → *BaseGPTIndex*

Create index from documents.

Parameters
documents (*Optional[Sequence[BaseDocument]]*) – List of documents to build the index from.

classmethod get_query_map() → Dict[str, Type[*BaseGPTIndexQuery*]]

Get query map.

insert(*document: Document*, ***insert_kwargs: Any*) → None

Insert a document.

classmethod load_from_dict(*result_dict: Dict[str, Any]*, ***kwargs: Any*) → *BaseGPTIndex*

Load index from string (in JSON-format).

This method loads the index from a JSON string. The index data structure itself is preserved completely. If the index is defined over subindices, those subindices will also be preserved (and subindices of those subindices, etc.).

NOTE: load_from_string should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_string* and *load_from_string* on that instead.

Parameters
index_string (*str*) – The index string (in JSON-format).

Returns
The loaded index.

Return type
BaseGPTIndex

classmethod load_from_disk(*save_path: str*, *faiss_index_save_path: Optional[str] = None*, ***kwargs: Any*) → *BaseGPTIndex*

Load index from disk.

This method loads the index from a JSON file stored on disk. The index data structure itself is preserved completely. If the index is defined over subindices, those subindices will also be preserved (and subindices of those subindices, etc.). In GPTFaissIndex, we allow user to specify an additional *faiss_index_save_path* to load faiss index from a file - that way, the user does not have to recreate the faiss index outside of this class.

Parameters

- **save_path** (*str*) – The save_path of the file.
- **faiss_index_save_path** (*Optional[str]*) – The save_path of the Faiss index file. If not specified, the Faiss index will not be saved to disk.
- ****kwargs** – Additional kwargs to pass to the index constructor.

Returns
The loaded index.

Return type
BaseGPTIndex

classmethod load_from_string(*index_string: str*, ***kwargs: Any*) → *BaseGPTIndex*

Load index from string (in JSON-format).

This method loads the index from a JSON string. The index data structure itself is preserved completely. If the index is defined over subindices, those subindices will also be preserved (and subindices of those subindices, etc.).

NOTE: load_from_string should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_string* and *load_from_string* on that instead.

Parameters
index_string (*str*) – The index string (in JSON-format).

Returns
The loaded index.

Return type
BaseGPTIndex

query(*query_str: Union[str,* QueryBundle*]*, *mode: str = QueryMode.DEFAULT*, *query_transform: Optional[BaseQueryTransform] = None*, *use_async: bool = False*, ***query_kwargs: Any*) → Union[*Response*, *StreamingResponse*]

Answer a query.

When *query* is called, we query the index with the given *mode* and *query_kwargs*. The *mode* determines the type of query to run, and *query_kwargs* are parameters that are specific to the query type.

For a comprehensive documentation of available *mode* and *query_kwargs* to query a given index, please visit *Querying an Index*.

property query_context: Dict[str, Any]

Additional context necessary for making a query.

This should capture any index-specific clients, services, etc, that's not captured by index struct, docstore, and service context. For example, a vector store index would pass vector store.

refresh(*documents: Sequence[*Document*]*, ***update_kwargs: Any*) → List[bool]

Refresh an index with documents that have changed.

This allows users to save LLM and Embedding model calls, while only updating documents that have any changes in text or extra_info. It will also insert any documents that previously were not stored.

save_to_dict(***save_kwargs: Any*) → dict

Save to string.

This method stores the index into a JSON string.

NOTE: save_to_string should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_string* and *load_from_string* on that instead.

Returns
The JSON dict of the index.

Return type
dict

save_to_disk(*save_path: str*, *encoding: str = 'ascii'*, *faiss_index_save_path: Optional[str] = None*, ***save_kwargs: Any*) → None

Save to file.

This method stores the index into a JSON file stored on disk. In GPTFaissIndex, we allow user to specify an additional *faiss_index_save_path* to save the faiss index to a file - that way, the user can pass in the same argument in *GPTFaissIndex.load_from_disk* without having to recreate the Faiss index outside of this class.

Parameters

- **save_path** (*str*) – The save_path of the file.
- **encoding** (*str*) – The encoding to use when saving the file.
- **faiss_index_save_path** (*Optional[str]*) – The save_path of the Faiss index file. If not specified, the Faiss index will not be saved to disk.

save_to_string(***save_kwargs: Any*) → str

Save to string.

This method stores the index into a JSON string.

NOTE: save_to_string should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_string* and *load_from_string* on that instead.

Returns
The JSON string of the index.

Return type
str

update(*document: Document*, ***update_kwargs: Any*) → None

Update a document.

This is equivalent to deleting the document and then inserting it again.

Parameters

- **document** (*Union[BaseDocument, BaseGPTIndex]*) – document to update
- **insert_kwargs** (*Dict*) – kwargs to pass to insert
- **delete_kwargs** (*Dict*) – kwargs to pass to delete

class gpt_index.indices.vector_store.vector_indices.**GPTMilvusIndex**(*nodes: Optional[Sequence[Node]] = None*, *collection_name: str = 'llamalection'*, *index_params: Optional[dict] = None*, *search_params: Optional[dict] = None*, *dim: Optional[int] = None*, *host: str = 'localhost'*, *port: int = 19530*, *user: str = ''*, *password: str = ''*, *use_secure: bool = False*, *overwrite: bool = False*, *service_context: Optional[ServiceContext] = None*, *index_struct: Optional[IndexDict] = None*, *vector_store: Optional[MilvusVectorStore] = None*, ***kwargs: Any*)

GPT Milvus Index.

In this GPT index we store the text, its embedding and a few pieces of its metadata in a Milvus collection. This implementation allows the use of an already existing collection if it is one that was created this vector store. It also supports creating a new one if the collection doesnt exist or if *overwrite* is set to True.

Parameters

- **service_context** (ServiceContext) – Service context container (contains components like LLMPredictor, PromptHelper, etc.).
- **collection_name** (*str, optional*) – The name of the collection where data will be stored. Defaults to "llamalection".
- **index_params** (*dict, optional*) – The index parameters for Milvus, if none are provided an HNSW index will be used. Defaults to None.
- **search_params** (*dict, optional*) – The search parameters for a Milvus query. If none are provided, default params will be generated. Defaults to None.
- **dim** (*int, optional*) – The dimension of the embeddings. If it is not provided, collection creation will be done on first insert. Defaults to None.
- **host** (*str, optional*) – The host address of Milvus. Defaults to "localhost".
- **port** (*int, optional*) – The port of Milvus. Defaults to 19530.
- **user** (*str, optional*) – The username for RBAC. Defaults to "".
- **password** (*str, optional*) – The password for RBAC. Defaults to "".
- **use_secure** (*bool, optional*) – Use https. Defaults to False.
- **overwrite** (*bool, optional*) – Whether to overwrite existing collection with same name. Defaults to False.

Raises

- **ImportError** – Unable to import *pymilvus*.

- **MilvusException** – Error communicating with Milvus, more can be found in logging under Debug.

Returns
Vectorstore that supports add, delete, and query.

Return type
MilvusVectorstore

async aquery(*query_str: Union[str, QueryBundle]*, *mode: str = QueryMode.DEFAULT*, *query_transform: Optional[BaseQueryTransform] = None*, ***query_kwargs: Any*) → Union[*Response*, *StreamingResponse*]

Asynchronously answer a query.

When *query* is called, we query the index with the given *mode* and *query_kwargs*. The *mode* determines the type of query to run, and *query_kwargs* are parameters that are specific to the query type.

For a comprehensive documentation of available *mode* and *query_kwargs* to query a given index, please visit *Querying an Index*.

classmethod from_documents(*documents: Sequence[Document]*, *docstore: Optional[BaseDocumentStore] = None*, *service_context: Optional[ServiceContext] = None*, ***kwargs: Any*) → *BaseGPTIndex*

Create index from documents.

Parameters
documents (*Optional[Sequence[BaseDocument]]*) – List of documents to build the index from.

classmethod get_query_map() → Dict[str, Type[*BaseGPTIndexQuery*]]

Get query map.

insert(*document: Document*, ***insert_kwargs: Any*) → None

Insert a document.

classmethod load_from_dict(*result_dict: Dict[str, Any]*, ***kwargs: Any*) → *BaseGPTIndex*

Load index from string (in JSON-format).

This method loads the index from a JSON string. The index data structure itself is preserved completely. If the index is defined over subindices, those subindices will also be preserved (and subindices of those subindices, etc.).

NOTE: load_from_string should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_string* and *load_from_string* on that instead.

Parameters
index_string (*str*) – The index string (in JSON-format).

Returns
The loaded index.

Return type
BaseGPTIndex

classmethod load_from_disk(*save_path: str*, ***kwargs: Any*) → *BaseGPTIndex*

Load index from disk.

This method loads the index from a JSON file stored on disk. The index data structure itself is preserved completely. If the index is defined over subindices, those subindices will also be preserved (and subindices of those subindices, etc.).

NOTE: load_from_disk should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_disk* and *load_from_disk* on that instead.

Parameters
save_path (*str*) – The save_path of the file.

Returns
The loaded index.

Return type
BaseGPTIndex

classmethod load_from_string(*index_string: str*, ***kwargs: Any*) → *BaseGPTIndex*

Load index from string (in JSON-format).

This method loads the index from a JSON string. The index data structure itself is preserved completely. If the index is defined over subindices, those subindices will also be preserved (and subindices of those subindices, etc.).

NOTE: load_from_string should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_string* and *load_from_string* on that instead.

Parameters
index_string (*str*) – The index string (in JSON-format).

Returns
The loaded index.

Return type
BaseGPTIndex

query(*query_str: Union[str,* QueryBundle*]*, *mode: str = QueryMode.DEFAULT*, *query_transform: Optional[BaseQueryTransform] = None*, *use_async: bool = False*, ***query_kwargs: Any*) → Union[*Response*, *StreamingResponse*]

Answer a query.

When *query* is called, we query the index with the given *mode* and *query_kwargs*. The *mode* determines the type of query to run, and *query_kwargs* are parameters that are specific to the query type.

For a comprehensive documentation of available *mode* and *query_kwargs* to query a given index, please visit *Querying an Index*.

property query_context: Dict[str, Any]

Additional context necessary for making a query.

This should capture any index-specific clients, services, etc, that's not captured by index struct, docstore, and service context. For example, a vector store index would pass vector store.

refresh(*documents: Sequence[*Document*]*, ***update_kwargs: Any*) → List[bool]

Refresh an index with documents that have changed.

This allows users to save LLM and Embedding model calls, while only updating documents that have any changes in text or extra_info. It will also insert any documents that previously were not stored.

save_to_dict(***save_kwargs: Any*) → dict

Save to string.

This method stores the index into a JSON string.

NOTE: save_to_string should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_string* and *load_from_string* on that instead.

Returns
The JSON dict of the index.

Return type
dict

save_to_disk(*save_path: str*, *encoding: str = 'ascii'*, ***save_kwargs: Any*) → None

Save to file.

This method stores the index into a JSON file stored on disk.

NOTE: save_to_disk should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_disk* and *load_from_disk* on that instead.

Parameters

- **save_path** (*str*) – The save_path of the file.
- **encoding** (*str*) – The encoding of the file.

save_to_string(***save_kwargs: Any*) → str

Save to string.

This method stores the index into a JSON string.

NOTE: save_to_string should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_string* and *load_from_string* on that instead.

Returns
The JSON string of the index.

Return type
str

update(*document:* Document, ***update_kwargs: Any*) → None

Update a document.

This is equivalent to deleting the document and then inserting it again.

Parameters

- **document** (*Union[BaseDocument,* BaseGPTIndex*]*) – document to update
- **insert_kwargs** (*Dict*) – kwargs to pass to insert
- **delete_kwargs** (*Dict*) – kwargs to pass to delete

class gpt_index.indices.vector_store.vector_indices.**GPTMyScaleIndex**(*myscale_client: Optional[Any] = None*, *table_name: str = 'llama_index'*, *database_name: str = 'default'*, *index_type: str = 'IVFFLAT'*, *metric: str = 'cosine'*, *batch_size: int = 32*, *index_params: Optional[dict] = None*, *search_params: Optional[dict] = None*, *nodes: Optional[Sequence[Node]] = None*, *service_context: Optional[ServiceContext] = None*, *index_struct: Optional[IndexDict] = None*, *vector_store: Optional[MyScaleVectorStore] = None*, ***kwargs: Any*)

GPT MyScale Index.

In this GPT index we store the text, its embedding and a few pieces of its metadata in a MyScale table. There will be a vector index build for the embedding column. This implementation allows the use of an already existing table if it is one that was created this vector store. It also supports creating a new table if the table doesn't exist

Parameters

- **myscale_client** (*httpclient*) – clickhouse-connect httpclient of an existing MyScale cluster.
- **table_name** (*str, optional*) – The name of the MyScale table where data will be stored. Defaults to "llama_index".
- **database_name** (*str, optional*) – The name of the MyScale database where data will be stored. Defaults to "default".
- **index_type** (*str, optional*) – The type of the MyScale vector index. Defaults to "IVFFLAT".
- **metric** (*str, optional*) – The metric type of the MyScale vector index. Defaults to "cosine".
- **batch_size** (*int, optional*) – the size of documents to insert. Defaults to 32.
- **index_params** (*dict, optional*) – The index parameters for MyScale. Defaults to None.
- **search_params** (*dict, optional*) – The search parameters for a MyScale query. Defaults to None.

Returns

Vectorstore that supports add, delete, and query.

Return type

MyScaleVectorStore

async **aquery**(*query_str: Union[str, QueryBundle]*, *mode: str = QueryMode.DEFAULT*, *query_transform: Optional[BaseQueryTransform] = None*, ***query_kwargs: Any*) → Union[*Response*, *StreamingResponse*]

Asynchronously answer a query.

When *query* is called, we query the index with the given *mode* and *query_kwargs*. The *mode* determines the type of query to run, and *query_kwargs* are parameters that are specific to the query type.

For a comprehensive documentation of available *mode* and *query_kwargs* to query a given index, please visit *Querying an Index*.

classmethod from_documents(*documents: Sequence[*Document*]*, *docstore: Optional[*BaseDocumentStore*] = None*, *service_context: Optional[*ServiceContext*] = None*, ***kwargs: Any*) → *BaseGPTIndex*

Create index from documents.

Parameters
documents (*Optional[Sequence[BaseDocument]]*) – List of documents to build the index from.

classmethod get_query_map() → Dict[str, Type[*BaseGPTIndexQuery*]]

Get query map.

insert(*document:* Document, ***insert_kwargs: Any*) → None

Insert a document.

classmethod load_from_dict(*result_dict: Dict[str, Any]*, ***kwargs: Any*) → *BaseGPTIndex*

Load index from string (in JSON-format).

This method loads the index from a JSON string. The index data structure itself is preserved completely. If the index is defined over subindices, those subindices will also be preserved (and subindices of those subindices, etc.).

NOTE: load_from_string should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_string* and *load_from_string* on that instead.

Parameters
index_string (*str*) – The index string (in JSON-format).

Returns
The loaded index.

Return type
BaseGPTIndex

classmethod load_from_disk(*save_path: str*, ***kwargs: Any*) → *BaseGPTIndex*

Load index from disk.

This method loads the index from a JSON file stored on disk. The index data structure itself is preserved completely. If the index is defined over subindices, those subindices will also be preserved (and subindices of those subindices, etc.).

NOTE: load_from_disk should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_disk* and *load_from_disk* on that instead.

Parameters
save_path (*str*) – The save_path of the file.

Returns
The loaded index.

Return type
BaseGPTIndex

classmethod load_from_string(*index_string: str*, ***kwargs: Any*) → *BaseGPTIndex*

Load index from string (in JSON-format).

This method loads the index from a JSON string. The index data structure itself is preserved completely. If the index is defined over subindices, those subindices will also be preserved (and subindices of those subindices, etc.).

NOTE: load_from_string should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_string* and *load_from_string* on that instead.

Parameters
index_string (*str*) – The index string (in JSON-format).

Returns
The loaded index.

Return type
BaseGPTIndex

query(*query_str: Union[str, QueryBundle]*, *mode: str = QueryMode.DEFAULT*, *query_transform: Optional[BaseQueryTransform] = None*, *use_async: bool = False*, ***query_kwargs: Any*) → Union[*Response*, *StreamingResponse*]

Answer a query.

When *query* is called, we query the index with the given *mode* and *query_kwargs*. The *mode* determines the type of query to run, and *query_kwargs* are parameters that are specific to the query type.

For a comprehensive documentation of available *mode* and *query_kwargs* to query a given index, please visit *Querying an Index*.

property query_context: Dict[str, Any]

Additional context necessary for making a query.

This should capture any index-specific clients, services, etc, that's not captured by index struct, docstore, and service context. For example, a vector store index would pass vector store.

refresh(*documents: Sequence[Document]*, ***update_kwargs: Any*) → List[bool]

Refresh an index with documents that have changed.

This allows users to save LLM and Embedding model calls, while only updating documents that have any changes in text or extra_info. It will also insert any documents that previously were not stored.

save_to_dict(***save_kwargs: Any*) → dict

Save to string.

This method stores the index into a JSON string.

NOTE: save_to_string should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_string* and *load_from_string* on that instead.

Returns
The JSON dict of the index.

Return type
dict

save_to_disk(*save_path: str*, *encoding: str = 'ascii'*, ***save_kwargs: Any*) → None

Save to file.

This method stores the index into a JSON file stored on disk.

NOTE: save_to_disk should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_disk* and *load_from_disk* on that instead.

Parameters

- **save_path** (*str*) – The save_path of the file.
- **encoding** (*str*) – The encoding of the file.

save_to_string(***save_kwargs: Any*) → str

Save to string.

This method stores the index into a JSON string.

NOTE: save_to_string should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_string* and *load_from_string* on that instead.

Returns
The JSON string of the index.

Return type
str

update(*document:* Document, ***update_kwargs: Any*) → None

Update a document.

This is equivalent to deleting the document and then inserting it again.

Parameters

- **document** (*Union[BaseDocument, BaseGPTIndex]*) – document to update
- **insert_kwargs** (*Dict*) – kwargs to pass to insert
- **delete_kwargs** (*Dict*) – kwargs to pass to delete

class gpt_index.indices.vector_store.vector_indices.**GPTOpensearchIndex**(*nodes: Optional[Sequence[*Node*]] = None*, *service_context: Optional[*ServiceContext*] = None*, *client: Optional[*OpensearchVectorClient*] = None*, *index_struct: Optional[IndexDict] = None*, *vector_store: Optional[*OpensearchVectorStore*] = None*, ***kwargs: Any*)

GPT Opensearch Index.

The GPTOpensearchIndex is a data structure where nodes are keyed by embeddings, and those embeddings are stored in a document that is indexed with its embedding as well as its textual data (text field is defined in the OpensearchVectorClient). During index construction, the document texts are chunked up, converted to nodes with text; each node's embedding is computed, and then the node's text, along with the embedding, is converted into JSON document that is indexed in Opensearch. The embedding data is put into a field with type "knn_vector" and the text is put into a standard Opensearch text field.

During query time, the index performs approximate KNN search using the "knn_vector" field that the embeddings were mapped to.

Parameters

- **client** (*Optional[OpensearchVectorClient]*) – The client which encapsulates logic for using Opensearch as a vector store (that is, it holds stuff like endpoint, index_name and

performs operations like initializing the index and adding new doc/embeddings to said index).

- **service_context** (ServiceContext) – Service context container (contains components like LLMPredictor, PromptHelper, etc.).

async aquery(*query_str: Union[str, QueryBundle]*, *mode: str = QueryMode.DEFAULT*, *query_transform: Optional[BaseQueryTransform] = None*, ***query_kwargs: Any*) → Union[*Response*, *StreamingResponse*]

Asynchronously answer a query.

When *query* is called, we query the index with the given *mode* and *query_kwargs*. The *mode* determines the type of query to run, and *query_kwargs* are parameters that are specific to the query type.

For a comprehensive documentation of available *mode* and *query_kwargs* to query a given index, please visit *Querying an Index*.

classmethod from_documents(*documents: Sequence[Document]*, *docstore: Optional[BaseDocumentStore] = None*, *service_context: Optional[ServiceContext] = None*, ***kwargs: Any*) → *BaseGPTIndex*

Create index from documents.

Parameters
documents (*Optional[Sequence[BaseDocument]]*) – List of documents to build the index from.

classmethod get_query_map() → Dict[str, Type[*BaseGPTIndexQuery*]]

Get query map.

insert(*document: Document*, ***insert_kwargs: Any*) → None

Insert a document.

classmethod load_from_dict(*result_dict: Dict[str, Any]*, ***kwargs: Any*) → *BaseGPTIndex*

Load index from string (in JSON-format).

This method loads the index from a JSON string. The index data structure itself is preserved completely. If the index is defined over subindices, those subindices will also be preserved (and subindices of those subindices, etc.).

NOTE: load_from_string should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_string* and *load_from_string* on that instead.

Parameters
index_string (*str*) – The index string (in JSON-format).

Returns
The loaded index.

Return type
BaseGPTIndex

classmethod load_from_disk(*save_path: str*, ***kwargs: Any*) → *BaseGPTIndex*

Load index from disk.

This method loads the index from a JSON file stored on disk. The index data structure itself is preserved completely. If the index is defined over subindices, those subindices will also be preserved (and subindices of those subindices, etc.).

NOTE: load_from_disk should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_disk* and *load_from_disk* on that instead.

Parameters
save_path (*str*) – The save_path of the file.

Returns
The loaded index.

Return type
BaseGPTIndex

classmethod load_from_string(*index_string: str*, ***kwargs: Any*) → *BaseGPTIndex*

Load index from string (in JSON-format).

This method loads the index from a JSON string. The index data structure itself is preserved completely. If the index is defined over subindices, those subindices will also be preserved (and subindices of those subindices, etc.).

NOTE: load_from_string should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_string* and *load_from_string* on that instead.

Parameters
index_string (*str*) – The index string (in JSON-format).

Returns
The loaded index.

Return type
BaseGPTIndex

query(*query_str: Union[str,* QueryBundle*]*, *mode: str = QueryMode.DEFAULT*, *query_transform: Optional[BaseQueryTransform] = None*, *use_async: bool = False*, ***query_kwargs: Any*) → Union[*Response*, *StreamingResponse*]

Answer a query.

When *query* is called, we query the index with the given *mode* and *query_kwargs*. The *mode* determines the type of query to run, and *query_kwargs* are parameters that are specific to the query type.

For a comprehensive documentation of available *mode* and *query_kwargs* to query a given index, please visit *Querying an Index*.

property query_context: Dict[str, Any]

Additional context necessary for making a query.

This should capture any index-specific clients, services, etc, that's not captured by index struct, docstore, and service context. For example, a vector store index would pass vector store.

refresh(*documents: Sequence[*Document*]*, ***update_kwargs: Any*) → List[bool]

Refresh an index with documents that have changed.

This allows users to save LLM and Embedding model calls, while only updating documents that have any changes in text or extra_info. It will also insert any documents that previously were not stored.

save_to_dict(***save_kwargs: Any*) → dict

Save to string.

This method stores the index into a JSON string.

NOTE: save_to_string should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_string* and *load_from_string* on that instead.

Returns
The JSON dict of the index.

Return type
dict

save_to_disk(*save_path: str*, *encoding: str = 'ascii'*, ***save_kwargs: Any*) → None

Save to file.

This method stores the index into a JSON file stored on disk.

NOTE: save_to_disk should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_disk* and *load_from_disk* on that instead.

Parameters

- **save_path** (*str*) – The save_path of the file.
- **encoding** (*str*) – The encoding of the file.

save_to_string(***save_kwargs: Any*) → str

Save to string.

This method stores the index into a JSON string.

NOTE: save_to_string should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_string* and *load_from_string* on that instead.

Returns
The JSON string of the index.

Return type
str

update(*document:* Document, ***update_kwargs: Any*) → None

Update a document.

This is equivalent to deleting the document and then inserting it again.

Parameters

- **document** (*Union[BaseDocument,* BaseGPTIndex*]*) – document to update
- **insert_kwargs** (*Dict*) – kwargs to pass to insert
- **delete_kwargs** (*Dict*) – kwargs to pass to delete

class gpt_index.indices.vector_store.vector_indices.**GPTPineconeIndex**(*nodes: Optional[Sequence[Node]] = None*, *pinecone_index: Optional[Any] = None*, *index_name: Optional[str] = None*, *environment: Optional[str] = None*, *namespace: Optional[str] = None*, *metadata_filters: Optional[Dict[str, Any]] = None*, *pinecone_kwargs: Optional[Dict] = None*, *insert_kwargs: Optional[Dict] = None*, *query_kwargs: Optional[Dict] = None*, *delete_kwargs: Optional[Dict] = None*, *index_struct: Optional[IndexDict] = None*, *service_context: Optional[ServiceContext] = None*, *vector_store: Optional[PineconeVectorStore] = None*, *add_sparse_vector: bool = False*, *tokenizer: Optional[Callable] = None*, ***kwargs: Any*)

GPT Pinecone Index.

The GPTPineconeIndex is a data structure where nodes are keyed by embeddings, and those embeddings are stored within a Pinecone index. During index construction, the document texts are chunked up, converted to nodes with text; they are then encoded in document embeddings stored within Pinecone.

During query time, the index uses Pinecone to query for the top k most similar nodes, and synthesizes an answer from the retrieved nodes.

Parameters

service_context (ServiceContext) – Service context container (contains components like LLMPredictor, PromptHelper, etc.).

async **aquery**(*query_str: Union[str, QueryBundle]*, *mode: str = QueryMode.DEFAULT*, *query_transform: Optional[BaseQueryTransform] = None*, ***query_kwargs: Any*) → Union[*Response*, *StreamingResponse*]

Asynchronously answer a query.

When *query* is called, we query the index with the given *mode* and *query_kwargs*. The *mode* determines the type of query to run, and *query_kwargs* are parameters that are specific to the query type.

For a comprehensive documentation of available *mode* and *query_kwargs* to query a given index, please visit *Querying an Index*.

classmethod **from_documents**(*documents: Sequence[Document]*, *docstore: Optional[BaseDocumentStore] = None*, *service_context: Optional[ServiceContext] = None*, ***kwargs: Any*) → *BaseGPTIndex*

Create index from documents.

Parameters
documents (*Optional[Sequence[BaseDocument]]*) – List of documents to build the index from.

classmethod get_query_map() → Dict[str, Type[*BaseGPTIndexQuery*]]

Get query map.

insert(*document:* Document, ***insert_kwargs: Any*) → None

Insert a document.

classmethod load_from_dict(*result_dict: Dict[str, Any]*, ***kwargs: Any*) → *BaseGPTIndex*

Load index from string (in JSON-format).

This method loads the index from a JSON string. The index data structure itself is preserved completely. If the index is defined over subindices, those subindices will also be preserved (and subindices of those subindices, etc.).

NOTE: load_from_string should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_string* and *load_from_string* on that instead.

Parameters
index_string (*str*) – The index string (in JSON-format).

Returns
The loaded index.

Return type
BaseGPTIndex

classmethod load_from_disk(*save_path: str*, ***kwargs: Any*) → *BaseGPTIndex*

Load index from disk.

This method loads the index from a JSON file stored on disk. The index data structure itself is preserved completely. If the index is defined over subindices, those subindices will also be preserved (and subindices of those subindices, etc.).

NOTE: load_from_disk should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_disk* and *load_from_disk* on that instead.

Parameters
save_path (*str*) – The save_path of the file.

Returns
The loaded index.

Return type
BaseGPTIndex

classmethod load_from_string(*index_string: str*, ***kwargs: Any*) → *BaseGPTIndex*

Load index from string (in JSON-format).

This method loads the index from a JSON string. The index data structure itself is preserved completely. If the index is defined over subindices, those subindices will also be preserved (and subindices of those subindices, etc.).

NOTE: load_from_string should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_string* and *load_from_string* on that instead.

Parameters
index_string (*str*) – The index string (in JSON-format).

Returns
The loaded index.

Return type
BaseGPTIndex

query(*query_str: Union[str, QueryBundle]*, *mode: str = QueryMode.DEFAULT*, *query_transform: Optional[BaseQueryTransform] = None*, *use_async: bool = False*, ***query_kwargs: Any*) → Union[*Response*, *StreamingResponse*]

Answer a query.

When *query* is called, we query the index with the given *mode* and *query_kwargs*. The *mode* determines the type of query to run, and *query_kwargs* are parameters that are specific to the query type.

For a comprehensive documentation of available *mode* and *query_kwargs* to query a given index, please visit *Querying an Index*.

property query_context: Dict[str, Any]

Additional context necessary for making a query.

This should capture any index-specific clients, services, etc, that's not captured by index struct, docstore, and service context. For example, a vector store index would pass vector store.

refresh(*documents: Sequence[Document]*, ***update_kwargs: Any*) → List[bool]

Refresh an index with documents that have changed.

This allows users to save LLM and Embedding model calls, while only updating documents that have any changes in text or extra_info. It will also insert any documents that previously were not stored.

save_to_dict(***save_kwargs: Any*) → dict

Save to string.

This method stores the index into a JSON string.

NOTE: save_to_string should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_string* and *load_from_string* on that instead.

Returns
The JSON dict of the index.

Return type
dict

save_to_disk(*save_path: str*, *encoding: str = 'ascii'*, ***save_kwargs: Any*) → None

Save to file.

This method stores the index into a JSON file stored on disk.

NOTE: save_to_disk should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_disk* and *load_from_disk* on that instead.

Parameters

- **save_path** (*str*) – The save_path of the file.
- **encoding** (*str*) – The encoding of the file.

save_to_string(***save_kwargs: Any*) → str

Save to string.

This method stores the index into a JSON string.

NOTE: save_to_string should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_string* and *load_from_string* on that instead.

Returns
The JSON string of the index.

Return type
str

update(*document:* Document, ***update_kwargs: Any*) → None

Update a document.

This is equivalent to deleting the document and then inserting it again.

Parameters

- **document** (*Union[BaseDocument, BaseGPTIndex]*) – document to update
- **insert_kwargs** (*Dict*) – kwargs to pass to insert
- **delete_kwargs** (*Dict*) – kwargs to pass to delete

class gpt_index.indices.vector_store.vector_indices.**GPTQdrantIndex**(*nodes: Optional[Sequence[Node]] = None*, *service_context: Optional[ServiceContext] = None*, *client: Optional[Any] = None*, *collection_name: Optional[str] = None*, *index_struct: Optional[IndexDict] = None*, *vector_store: Optional[QdrantVectorStore] = None*, ***kwargs: Any*)

GPT Qdrant Index.

The GPTQdrantIndex is a data structure where nodes are keyed by embeddings, and those embeddings are stored within a Qdrant collection. During index construction, the document texts are chunked up, converted to nodes with text; they are then encoded in document embeddings stored within Qdrant.

During query time, the index uses Qdrant to query for the top k most similar nodes, and synthesizes an answer from the retrieved nodes.

Parameters

- **service_context** (ServiceContext) – Service context container (contains components like LLMPredictor, PromptHelper, etc.).
- **client** (*Optional[Any]*) – QdrantClient instance from *qdrant-client* package
- **collection_name** – (Optional[str]): name of the Qdrant collection

async aquery(*query_str: Union[str,* QueryBundle*]*, *mode: str = QueryMode.DEFAULT*, *query_transform: Optional[BaseQueryTransform] = None*, ***query_kwargs: Any*) → Union[*Response*, *StreamingResponse*]

Asynchronously answer a query.

When *query* is called, we query the index with the given *mode* and *query_kwargs*. The *mode* determines the type of query to run, and *query_kwargs* are parameters that are specific to the query type.

For a comprehensive documentation of available *mode* and *query_kwargs* to query a given index, please visit *Querying an Index*.

classmethod from_documents(*documents: Sequence[*Document*]*, *docstore: Optional[*BaseDocumentStore*] = None*, *service_context: Optional[*ServiceContext*] = None*, ***kwargs: Any*) → *BaseGPTIndex*

Create index from documents.

Parameters
documents (*Optional[Sequence[BaseDocument]]*) – List of documents to build the index from.

classmethod get_query_map() → Dict[str, Type[*BaseGPTIndexQuery*]]

Get query map.

insert(*document:* Document, ***insert_kwargs: Any*) → None

Insert a document.

classmethod load_from_dict(*result_dict: Dict[str, Any]*, ***kwargs: Any*) → *BaseGPTIndex*

Load index from string (in JSON-format).

This method loads the index from a JSON string. The index data structure itself is preserved completely. If the index is defined over subindices, those subindices will also be preserved (and subindices of those subindices, etc.).

NOTE: load_from_string should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_string* and *load_from_string* on that instead.

Parameters
index_string (*str*) – The index string (in JSON-format).

Returns
The loaded index.

Return type
BaseGPTIndex

classmethod load_from_disk(*save_path: str*, ***kwargs: Any*) → *BaseGPTIndex*

Load index from disk.

This method loads the index from a JSON file stored on disk. The index data structure itself is preserved completely. If the index is defined over subindices, those subindices will also be preserved (and subindices of those subindices, etc.).

NOTE: load_from_disk should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_disk* and *load_from_disk* on that instead.

Parameters
save_path (*str*) – The save_path of the file.

Returns
The loaded index.

Return type
BaseGPTIndex

classmethod load_from_string(*index_string: str*, ***kwargs: Any*) → *BaseGPTIndex*

Load index from string (in JSON-format).

This method loads the index from a JSON string. The index data structure itself is preserved completely. If the index is defined over subindices, those subindices will also be preserved (and subindices of those subindices, etc.).

NOTE: load_from_string should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_string* and *load_from_string* on that instead.

Parameters
index_string (*str*) – The index string (in JSON-format).

Returns
The loaded index.

Return type
BaseGPTIndex

query(*query_str: Union[str, QueryBundle]*, *mode: str = QueryMode.DEFAULT*, *query_transform: Optional[BaseQueryTransform] = None*, *use_async: bool = False*, ***query_kwargs: Any*) → Union[*Response*, *StreamingResponse*]

Answer a query.

When *query* is called, we query the index with the given *mode* and *query_kwargs*. The *mode* determines the type of query to run, and *query_kwargs* are parameters that are specific to the query type.

For a comprehensive documentation of available *mode* and *query_kwargs* to query a given index, please visit *Querying an Index*.

property query_context: Dict[str, Any]

Additional context necessary for making a query.

This should capture any index-specific clients, services, etc, that's not captured by index struct, docstore, and service context. For example, a vector store index would pass vector store.

refresh(*documents: Sequence[Document]*, ***update_kwargs: Any*) → List[bool]

Refresh an index with documents that have changed.

This allows users to save LLM and Embedding model calls, while only updating documents that have any changes in text or extra_info. It will also insert any documents that previously were not stored.

save_to_dict(***save_kwargs: Any*) → dict

Save to string.

This method stores the index into a JSON string.

NOTE: save_to_string should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_string* and *load_from_string* on that instead.

Returns
The JSON dict of the index.

Return type
dict

save_to_disk(*save_path: str*, *encoding: str = 'ascii'*, ***save_kwargs: Any*) → None

Save to file.

This method stores the index into a JSON file stored on disk.

NOTE: save_to_disk should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_disk* and *load_from_disk* on that instead.

Parameters

- **save_path** (*str*) – The save_path of the file.
- **encoding** (*str*) – The encoding of the file.

save_to_string(***save_kwargs: Any*) → str

Save to string.

This method stores the index into a JSON string.

NOTE: save_to_string should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_string* and *load_from_string* on that instead.

Returns
The JSON string of the index.

Return type
str

update(*document:* Document, ***update_kwargs: Any*) → None

Update a document.

This is equivalent to deleting the document and then inserting it again.

Parameters

- **document** (*Union[BaseDocument, BaseGPTIndex]*) – document to update
- **insert_kwargs** (*Dict*) – kwargs to pass to insert
- **delete_kwargs** (*Dict*) – kwargs to pass to delete

class gpt_index.indices.vector_store.vector_indices.**GPTSimpleVectorIndex**(*nodes: Optional[Sequence[*Node*]] = None*, *index_struct: Optional[IndexDict] = None*, *service_context: Optional[*ServiceContext*] = None*, *vector_store: Optional[*SimpleVectorStore*] = None*, ***kwargs: Any*)

GPT Simple Vector Index.

The GPTSimpleVectorIndex is a data structure where nodes are keyed by embeddings, and those embeddings are stored within a simple dictionary. During index construction, the document texts are chunked up, converted to nodes with text; they are then encoded in document embeddings stored within the dict.

During query time, the index uses the dict to query for the top k most similar nodes, and synthesizes an answer from the retrieved nodes.

Parameters
service_context (ServiceContext) – Service context container (contains components like LLMPredictor, PromptHelper, etc.).

async aquery(*query_str: Union[str,* QueryBundle*]*, *mode: str = QueryMode.DEFAULT*, *query_transform: Optional[BaseQueryTransform] = None*, ***query_kwargs: Any*) → Union[*Response*, *StreamingResponse*]

Asynchronously answer a query.

When *query* is called, we query the index with the given *mode* and *query_kwargs*. The *mode* determines the type of query to run, and *query_kwargs* are parameters that are specific to the query type.

For a comprehensive documentation of available *mode* and *query_kwargs* to query a given index, please visit *Querying an Index*.

classmethod from_documents(*documents: Sequence[*Document*]*, *docstore: Optional[*BaseDocumentStore*] = None*, *service_context: Optional[*ServiceContext*] = None*, ***kwargs: Any*) → *BaseGPTIndex*

Create index from documents.

Parameters
documents (*Optional[Sequence[BaseDocument]]*) – List of documents to build the index from.

classmethod get_query_map() → Dict[str, Type[*BaseGPTIndexQuery*]]

Get query map.

insert(*document:* Document, ***insert_kwargs: Any*) → None

Insert a document.

classmethod load_from_dict(*result_dict: Dict[str, Any]*, ***kwargs: Any*) → *BaseGPTIndex*

Load index from string (in JSON-format).

This method loads the index from a JSON string. The index data structure itself is preserved completely. If the index is defined over subindices, those subindices will also be preserved (and subindices of those subindices, etc.).

NOTE: load_from_string should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_string* and *load_from_string* on that instead.

Parameters
index_string (*str*) – The index string (in JSON-format).

Returns
The loaded index.

Return type
BaseGPTIndex

classmethod load_from_disk(*save_path: str*, ***kwargs: Any*) → *BaseGPTIndex*

Load index from disk.

This method loads the index from a JSON file stored on disk. The index data structure itself is preserved completely. If the index is defined over subindices, those subindices will also be preserved (and subindices of those subindices, etc.).

NOTE: load_from_disk should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_disk* and *load_from_disk* on that instead.

Parameters
save_path (*str*) – The save_path of the file.

Returns
The loaded index.

Return type
BaseGPTIndex

classmethod load_from_string(*index_string: str*, ***kwargs: Any*) → *BaseGPTIndex*

Load index from string (in JSON-format).

This method loads the index from a JSON string. The index data structure itself is preserved completely. If the index is defined over subindices, those subindices will also be preserved (and subindices of those subindices, etc.).

NOTE: load_from_string should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_string* and *load_from_string* on that instead.

Parameters
index_string (*str*) – The index string (in JSON-format).

Returns
The loaded index.

Return type
BaseGPTIndex

query(*query_str: Union[str, QueryBundle]*, *mode: str = QueryMode.DEFAULT*, *query_transform: Optional[BaseQueryTransform] = None*, *use_async: bool = False*, ***query_kwargs: Any*) → Union[*Response*, *StreamingResponse*]

Answer a query.

When *query* is called, we query the index with the given *mode* and *query_kwargs*. The *mode* determines the type of query to run, and *query_kwargs* are parameters that are specific to the query type.

For a comprehensive documentation of available *mode* and *query_kwargs* to query a given index, please visit *Querying an Index*.

property query_context: Dict[str, Any]

Additional context necessary for making a query.

This should capture any index-specific clients, services, etc, that's not captured by index struct, docstore, and service context. For example, a vector store index would pass vector store.

refresh(*documents: Sequence[Document]*, ***update_kwargs: Any*) → List[bool]

Refresh an index with documents that have changed.

This allows users to save LLM and Embedding model calls, while only updating documents that have any changes in text or extra_info. It will also insert any documents that previously were not stored.

save_to_dict(***save_kwargs: Any*) → dict

Save to string.

This method stores the index into a JSON string.

NOTE: save_to_string should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_string* and *load_from_string* on that instead.

Returns
The JSON dict of the index.

Return type
dict

save_to_disk(*save_path: str*, *encoding: str = 'ascii'*, ***save_kwargs: Any*) → None

Save to file.

This method stores the index into a JSON file stored on disk.

NOTE: save_to_disk should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_disk* and *load_from_disk* on that instead.

Parameters

- **save_path** (*str*) – The save_path of the file.
- **encoding** (*str*) – The encoding of the file.

save_to_string(***save_kwargs: Any*) → str

Save to string.

This method stores the index into a JSON string.

NOTE: save_to_string should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_string* and *load_from_string* on that instead.

Returns
The JSON string of the index.

Return type
str

update(*document:* Document, ***update_kwargs: Any*) → None

Update a document.

This is equivalent to deleting the document and then inserting it again.

Parameters

- **document** (*Union[BaseDocument, BaseGPTIndex]*) – document to update
- **insert_kwargs** (*Dict*) – kwargs to pass to insert
- **delete_kwargs** (*Dict*) – kwargs to pass to delete

class gpt_index.indices.vector_store.vector_indices.**GPTWeaviateIndex**(*nodes: Optional[Sequence[*Node*]] = None*, *service_context: Optional[*ServiceContext*] = None*, *weaviate_client: Optional[Any] = None*, *class_prefix: Optional[str] = None*, *index_struct: Optional[IndexDict] = None*, *vector_store: Optional[*WeaviateVectorStore*] = None*, ***kwargs: Any*)

GPT Weaviate Index.

The GPTWeaviateIndex is a data structure where nodes are keyed by embeddings, and those embeddings are stored within a Weaviate index. During index construction, the document texts are chunked up, converted to nodes with text; they are then encoded in document embeddings stored within Weaviate.

During query time, the index uses Weaviate to query for the top k most similar nodes, and synthesizes an answer from the retrieved nodes.

Parameters
service_context (ServiceContext) – Service context container (contains components like LLMPredictor, PromptHelper, etc.).

async aquery(*query_str: Union[str,* QueryBundle*]*, *mode: str = QueryMode.DEFAULT*, *query_transform: Optional[BaseQueryTransform] = None*, ***query_kwargs: Any*) → Union[*Response*, *StreamingResponse*]

Asynchronously answer a query.

When *query* is called, we query the index with the given *mode* and *query_kwargs*. The *mode* determines the type of query to run, and *query_kwargs* are parameters that are specific to the query type.

For a comprehensive documentation of available *mode* and *query_kwargs* to query a given index, please visit *Querying an Index*.

classmethod from_documents(*documents: Sequence[*Document*]*, *docstore: Optional[*BaseDocumentStore*] = None*, *service_context: Optional[*ServiceContext*] = None*, ***kwargs: Any*) → *BaseGPTIndex*

Create index from documents.

Parameters
documents (*Optional[Sequence[BaseDocument]]*) – List of documents to build the index from.

classmethod get_query_map() → Dict[str, Type[*BaseGPTIndexQuery*]]

Get query map.

insert(*document:* Document, ***insert_kwargs: Any*) → None

Insert a document.

classmethod load_from_dict(*result_dict: Dict[str, Any]*, ***kwargs: Any*) → *BaseGPTIndex*

Load index from string (in JSON-format).

This method loads the index from a JSON string. The index data structure itself is preserved completely. If the index is defined over subindices, those subindices will also be preserved (and subindices of those subindices, etc.).

NOTE: load_from_string should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_string* and *load_from_string* on that instead.

Parameters
index_string (*str*) – The index string (in JSON-format).

Returns
The loaded index.

Return type
BaseGPTIndex

classmethod load_from_disk(*save_path: str*, ***kwargs: Any*) → *BaseGPTIndex*

Load index from disk.

This method loads the index from a JSON file stored on disk. The index data structure itself is preserved completely. If the index is defined over subindices, those subindices will also be preserved (and subindices of those subindices, etc.).

NOTE: load_from_disk should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_disk* and *load_from_disk* on that instead.

Parameters
save_path (*str*) – The save_path of the file.

Returns
The loaded index.

Return type
BaseGPTIndex

classmethod load_from_string(*index_string: str*, ***kwargs: Any*) → *BaseGPTIndex*

Load index from string (in JSON-format).

This method loads the index from a JSON string. The index data structure itself is preserved completely. If the index is defined over subindices, those subindices will also be preserved (and subindices of those subindices, etc.).

NOTE: load_from_string should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_string* and *load_from_string* on that instead.

Parameters
index_string (*str*) – The index string (in JSON-format).

Returns
The loaded index.

Return type
BaseGPTIndex

query(*query_str: Union[str, QueryBundle]*, *mode: str = QueryMode.DEFAULT*, *query_transform: Optional[BaseQueryTransform] = None*, *use_async: bool = False*, ***query_kwargs: Any*) → Union[*Response*, *StreamingResponse*]

Answer a query.

When *query* is called, we query the index with the given *mode* and *query_kwargs*. The *mode* determines the type of query to run, and *query_kwargs* are parameters that are specific to the query type.

For a comprehensive documentation of available *mode* and *query_kwargs* to query a given index, please visit *Querying an Index*.

property query_context: Dict[str, Any]

Additional context necessary for making a query.

This should capture any index-specific clients, services, etc, that's not captured by index struct, docstore, and service context. For example, a vector store index would pass vector store.

refresh(*documents: Sequence[Document]*, ***update_kwargs: Any*) → List[bool]

Refresh an index with documents that have changed.

This allows users to save LLM and Embedding model calls, while only updating documents that have any changes in text or extra_info. It will also insert any documents that previously were not stored.

save_to_dict(***save_kwargs: Any*) → dict

Save to string.

This method stores the index into a JSON string.

NOTE: save_to_string should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_string* and *load_from_string* on that instead.

Returns
The JSON dict of the index.

Return type
dict

save_to_disk(*save_path: str*, *encoding: str = 'ascii'*, ***save_kwargs: Any*) → None

Save to file.

This method stores the index into a JSON file stored on disk.

NOTE: save_to_disk should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_disk* and *load_from_disk* on that instead.

Parameters

- **save_path** (*str*) – The save_path of the file.
- **encoding** (*str*) – The encoding of the file.

save_to_string(***save_kwargs: Any*) → str

Save to string.

This method stores the index into a JSON string.

NOTE: save_to_string should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_string* and *load_from_string* on that instead.

Returns
The JSON string of the index.

Return type
str

update(*document:* Document, ***update_kwargs: Any*) → None

Update a document.

This is equivalent to deleting the document and then inserting it again.

Parameters

- **document** (*Union[BaseDocument, BaseGPTIndex]*) – document to update
- **insert_kwargs** (*Dict*) – kwargs to pass to insert
- **delete_kwargs** (*Dict*) – kwargs to pass to delete

3.17.5 Structured Store Index

Structured store indices.

class gpt_index.indices.struct_store.**GPTNLPandasIndexQuery**(*index_struct: PandasStructTable*, *df: Optional[DataFrame] = None*, *instruction_str: Optional[str] = None*, *output_processor: Optional[Callable] = None*, *pandas_prompt: Optional[PandasPrompt] = None*, *output_kwargs: Optional[dict] = None*, *head: int = 5*, ***kwargs: Any*)

GPT Pandas query.

Convert natural language to Pandas python code.

```
response = index.query("<query_str>", mode="default")
```

Parameters

- **df** (*pd.DataFrame*) – Pandas dataframe to use.
- **instruction_str** (*Optional[str]*) – Instruction string to use.
- **output_processor** (*Optional[Callable[[str], str]]*) – Output processor. A callable that takes in the output string, pandas DataFrame, and any output kwargs and returns a string.

- **pandas_prompt** (*Optional[PandasPrompt]*) – Pandas prompt to use.
- **head** (*int*) – Number of rows to show in the table context.

query(*query_bundle:* QueryBundle) → *Response*

Answer a query.

retrieve(*query_bundle:* QueryBundle) → List[*NodeWithScore*]

Get list of tuples of node and similarity for response.

First part of the tuple is the node. Second part of tuple is the distance from query to the node. If not applicable, it's None.

class gpt_index.indices.struct_store.**GPTNLStructStoreIndexQuery**(*index_struct: SQLStructTable*, *sql_database: Optional[SQLDatabase] = None*, *sql_context_container: Optional[SQLContextContainer] = None*, *ref_doc_id_column: Optional[str] = None*, *text_to_sql_prompt: Optional[TextToSQLPrompt] = None*, *context_query_mode:* QueryMode *= QueryMode.DEFAULT*, *context_query_kwargs: Optional[dict] = None*, ***kwargs: Any*)

GPT natural language query over a structured database.

Given a natural language query, we will extract the query to SQL. Runs raw SQL over a GPTSQLStructStoreIndex. No LLM calls are made during the SQL execution. NOTE: this query cannot work with composed indices - if the index contains subindices, those subindices will not be queried.

```
response = index.query("<query_str>", mode="default")
```

async aquery(*query_bundle:* QueryBundle) → *Response*

Answer a query.

query(*query_bundle:* QueryBundle) → *Response*

Answer a query.

retrieve(*query_bundle:* QueryBundle) → List[*NodeWithScore*]

Get list of tuples of node and similarity for response.

First part of the tuple is the node. Second part of tuple is the distance from query to the node. If not applicable, it's None.

class gpt_index.indices.struct_store.**GPTPandasIndex**(*nodes: Optional[Sequence[Node]] = None*, *df: Optional[DataFrame] = None*, *index_struct: Optional[PandasStructTable] = None*, ***kwargs: Any*)

Base GPT Pandas Index.

The GPTPandasStructStoreIndex is an index that stores a Pandas dataframe under the hood. Currently index "construction" is not supported.

During query time, the user can either specify a raw SQL query or a natural language query to retrieve their data.

Parameters
pandas_df (*Optional[pd.DataFrame]*) – Pandas dataframe to use. See *Structured Index Configuration* for more details.

async aquery(*query_str: Union[str, QueryBundle]*, *mode: str = QueryMode.DEFAULT*, *query_transform: Optional[BaseQueryTransform] = None*, ***query_kwargs: Any*) → Union[*Response*, *StreamingResponse*]

Asynchronously answer a query.

When *query* is called, we query the index with the given *mode* and *query_kwargs*. The *mode* determines the type of query to run, and *query_kwargs* are parameters that are specific to the query type.

For a comprehensive documentation of available *mode* and *query_kwargs* to query a given index, please visit *Querying an Index*.

classmethod from_documents(*documents: Sequence[Document]*, *docstore: Optional[BaseDocumentStore] = None*, *service_context: Optional[ServiceContext] = None*, ***kwargs: Any*) → *BaseGPTIndex*

Create index from documents.

Parameters
documents (*Optional[Sequence[BaseDocument]]*) – List of documents to build the index from.

classmethod get_query_map() → Dict[str, Type[*BaseGPTIndexQuery*]]

Get query map.

insert(*document: Document*, ***insert_kwargs: Any*) → None

Insert a document.

classmethod load_from_dict(*result_dict: Dict[str, Any]*, ***kwargs: Any*) → *BaseGPTIndex*

Load index from dict.

classmethod load_from_disk(*save_path: str*, ***kwargs: Any*) → *BaseGPTIndex*

Load index from disk.

This method loads the index from a JSON file stored on disk. The index data structure itself is preserved completely. If the index is defined over subindices, those subindices will also be preserved (and subindices of those subindices, etc.).

NOTE: load_from_disk should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_disk* and *load_from_disk* on that instead.

Parameters
save_path (*str*) – The save_path of the file.

Returns
The loaded index.

Return type
BaseGPTIndex

classmethod load_from_string(*index_string: str*, ***kwargs: Any*) → *BaseGPTIndex*

Load index from string (in JSON-format).

This method loads the index from a JSON string. The index data structure itself is preserved completely. If the index is defined over subindices, those subindices will also be preserved (and subindices of those subindices, etc.).

NOTE: load_from_string should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_string* and *load_from_string* on that instead.

Parameters
index_string (*str*) – The index string (in JSON-format).

Returns
The loaded index.

Return type
BaseGPTIndex

query(*query_str: Union[str, QueryBundle]*, *mode: str = QueryMode.DEFAULT*, *query_transform: Optional[BaseQueryTransform] = None*, *use_async: bool = False*, ***query_kwargs: Any*) → Union[*Response*, *StreamingResponse*]

Answer a query.

When *query* is called, we query the index with the given *mode* and *query_kwargs*. The *mode* determines the type of query to run, and *query_kwargs* are parameters that are specific to the query type.

For a comprehensive documentation of available *mode* and *query_kwargs* to query a given index, please visit *Querying an Index*.

property query_context: Dict[str, Any]

Additional context necessary for making a query.

This should capture any index-specific clients, services, etc, that's not captured by index struct, docstore, and service context. For example, a vector store index would pass vector store.

refresh(*documents: Sequence[Document]*, ***update_kwargs: Any*) → List[bool]

Refresh an index with documents that have changed.

This allows users to save LLM and Embedding model calls, while only updating documents that have any changes in text or extra_info. It will also insert any documents that previously were not stored.

save_to_dict(***save_kwargs: Any*) → dict

Save to dict.

save_to_disk(*save_path: str*, *encoding: str = 'ascii'*, ***save_kwargs: Any*) → None

Save to file.

This method stores the index into a JSON file stored on disk.

NOTE: save_to_disk should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_disk* and *load_from_disk* on that instead.

Parameters

- **save_path** (*str*) – The save_path of the file.
- **encoding** (*str*) – The encoding of the file.

save_to_string(***save_kwargs: Any*) → str

Save to string.

This method stores the index into a JSON string.

NOTE: save_to_string should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_string* and *load_from_string* on that instead.

Returns
The JSON string of the index.

Return type
str

update(*document:* Document, ***update_kwargs: Any*) → None

Update a document.

This is equivalent to deleting the document and then inserting it again.

Parameters

- **document** (*Union[BaseDocument, BaseGPTIndex]*) – document to update
- **insert_kwargs** (*Dict*) – kwargs to pass to insert
- **delete_kwargs** (*Dict*) – kwargs to pass to delete

class gpt_index.indices.struct_store.**GPTSQLStructStoreIndex**(*nodes: Optional[Sequence[*Node*]] = None*, *index_struct: Optional[SQLStructTable] = None*, *service_context: Optional[*ServiceContext*] = None*, *sql_database: Optional[*SQLDatabase*] = None*, *table_name: Optional[str] = None*, *table: Optional[Table] = None*, *ref_doc_id_column: Optional[str] = None*, *sql_context_container: Optional[SQLContextContainer] = None*, ***kwargs: Any*)

Base GPT SQL Struct Store Index.

The GPTSQLStructStoreIndex is an index that uses a SQL database under the hood. During index construction, the data can be inferred from unstructured documents given a schema extract prompt, or it can be pre-loaded in the database.

During query time, the user can either specify a raw SQL query or a natural language query to retrieve their data.

Parameters

- **documents** (*Optional[Sequence[DOCUMENTS_INPUT]]*) – Documents to index. NOTE: in the SQL index, this is an optional field.
- **sql_database** (*Optional[*SQLDatabase*]*) – SQL database to use, including table names to specify. See *Structured Index Configuration* for more details.
- **table_name** (*Optional[str]*) – Name of the table to use for extracting data. Either table_name or table must be specified.
- **table** (*Optional[Table]*) – SQLAlchemy Table object to use. Specifying the Table object explicitly, instead of the table name, allows you to pass in a view. Either table_name or table must be specified.
- **sql_context_container** (*Optional[SQLContextContainer]*) – SQL context container. an be generated from a SQLContextContainerBuilder. See *Structured Index Configuration* for more details.

async aquery(*query_str: Union[str,* QueryBundle*]*, *mode: str = QueryMode.DEFAULT*, *query_transform: Optional[BaseQueryTransform] = None*, ***query_kwargs: Any*) → Union[*Response*, *StreamingResponse*]

Asynchronously answer a query.

When *query* is called, we query the index with the given *mode* and *query_kwargs*. The *mode* determines the type of query to run, and *query_kwargs* are parameters that are specific to the query type.

For a comprehensive documentation of available *mode* and *query_kwargs* to query a given index, please visit *Querying an Index*.

classmethod from_documents(*documents: Sequence[*Document*]*, *docstore: Optional[*BaseDocumentStore*] = None*, *service_context: Optional[*ServiceContext*] = None*, ***kwargs: Any*) → *BaseGPTIndex*

Create index from documents.

Parameters
documents (*Optional[Sequence[BaseDocument]]*) – List of documents to build the index from.

classmethod get_query_map() → Dict[str, Type[*BaseGPTIndexQuery*]]

Get query map.

insert(*document:* Document, ***insert_kwargs: Any*) → None

Insert a document.

classmethod load_from_dict(*result_dict: Dict[str, Any]*, ***kwargs: Any*) → *BaseGPTIndex*

Load index from dict.

classmethod load_from_disk(*save_path: str*, ***kwargs: Any*) → *BaseGPTIndex*

Load index from disk.

This method loads the index from a JSON file stored on disk. The index data structure itself is preserved completely. If the index is defined over subindices, those subindices will also be preserved (and subindices of those subindices, etc.).

NOTE: load_from_disk should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_disk* and *load_from_disk* on that instead.

Parameters
save_path (*str*) – The save_path of the file.

Returns
The loaded index.

Return type
BaseGPTIndex

classmethod load_from_string(*index_string: str*, ***kwargs: Any*) → *BaseGPTIndex*

Load index from string (in JSON-format).

This method loads the index from a JSON string. The index data structure itself is preserved completely. If the index is defined over subindices, those subindices will also be preserved (and subindices of those subindices, etc.).

NOTE: load_from_string should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_string* and *load_from_string* on that instead.

Parameters
index_string (*str*) – The index string (in JSON-format).

Returns
The loaded index.

Return type
BaseGPTIndex

query(*query_str: Union[str,* QueryBundle*]*, *mode: str = QueryMode.DEFAULT*, *query_transform: Optional[BaseQueryTransform] = None*, *use_async: bool = False*, ***query_kwargs: Any*) → Union[*Response*, *StreamingResponse*]

Answer a query.

When *query* is called, we query the index with the given *mode* and *query_kwargs*. The *mode* determines the type of query to run, and *query_kwargs* are parameters that are specific to the query type.

For a comprehensive documentation of available *mode* and *query_kwargs* to query a given index, please visit *Querying an Index*.

property query_context: Dict[str, Any]

Additional context necessary for making a query.

This should capture any index-specific clients, services, etc, that's not captured by index struct, docstore, and service context. For example, a vector store index would pass vector store.

refresh(*documents: Sequence[*Document*]*, ***update_kwargs: Any*) → List[bool]

Refresh an index with documents that have changed.

This allows users to save LLM and Embedding model calls, while only updating documents that have any changes in text or extra_info. It will also insert any documents that previously were not stored.

save_to_dict(***save_kwargs: Any*) → dict

Save to dict.

save_to_disk(*save_path: str*, *encoding: str = 'ascii'*, ***save_kwargs: Any*) → None

Save to file.

This method stores the index into a JSON file stored on disk.

NOTE: save_to_disk should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_disk* and *load_from_disk* on that instead.

Parameters

- **save_path** (*str*) – The save_path of the file.
- **encoding** (*str*) – The encoding of the file.

save_to_string(***save_kwargs: Any*) → str

Save to string.

This method stores the index into a JSON string.

NOTE: save_to_string should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_string* and *load_from_string* on that instead.

Returns
The JSON string of the index.

Return type
str

update(*document:* Document, ***update_kwargs: Any*) → None

Update a document.

This is equivalent to deleting the document and then inserting it again.

Parameters

- **document** (*Union[BaseDocument, BaseGPTIndex]*) – document to update
- **insert_kwargs** (*Dict*) – kwargs to pass to insert
- **delete_kwargs** (*Dict*) – kwargs to pass to delete

class gpt_index.indices.struct_store.**GPTSQLStructStoreIndexQuery**(*index_struct: SQLStructTable*, *sql_database: Optional[SQLDatabase] = None*, *sql_context_container: Optional[SQLContextContainer] = None*, ***kwargs: Any*)

GPT SQL query over a structured database.

Runs raw SQL over a GPTSQLStructStoreIndex. No LLM calls are made here. NOTE: this query cannot work with composed indices - if the index contains subindices, those subindices will not be queried.

```
response = index.query("<query_str>", mode="sql")
```

retrieve(*query_bundle:* QueryBundle) → List[*NodeWithScore*]

Get list of tuples of node and similarity for response.

First part of the tuple is the node. Second part of tuple is the distance from query to the node. If not applicable, it's None.

class gpt_index.indices.struct_store.**SQLContextContainerBuilder**(*sql_database:* SQLDatabase, *context_dict: Optional[Dict[str, str]] = None*, *context_str: Optional[str] = None*)

SQLContextContainerBuilder.

Build a SQLContextContainer that can be passed to the SQL index during index construction or during queryt-time.

NOTE: if context_str is specified, that will be used as context instead of context_dict

Parameters

- **sql_database** (SQLDatabase) – SQL database
- **context_dict** (*Optional[Dict[str, str]]*) – context dict

build_context_container(*ignore_db_schema: bool = False*) → SQLContextContainer

Build index structure.

derive_index_from_context(*index_cls: Type[*BaseGPTIndex*]*, *ignore_db_schema: bool = False*, ***index_kwargs: Any*) → *BaseGPTIndex*

Derive index from context.

classmethod from_documents(*documents_dict: Dict[str, List[BaseDocument]]*, *sql_database:* SQLDatabase, ***context_builder_kwargs: Any*) → *SQLContextContainerBuilder*

Build context from documents.

query_index_for_context(*index:* BaseGPTIndex, *query_str: Union[str,* QueryBundle*]*, *query_tmpl: Optional[str] = 'Please return the relevant tables (including the full schema) for the following query: {orig_query_str}'*, *store_context_str: bool = True*, ***index_kwargs: Any*) → str

Query index for context.

A simple wrapper around the index.query call which injects a query template to specifically fetch table information, and can store a context_str.

Parameters

- **index** (BaseGPTIndex) – index data structure
- **query_str** (*Union[str, QueryBundle]*) – query string
- **query_tmpl** (*Optional[str]*) – query template
- **store_context_str** (*bool*) – store context_str

3.17.6 Knowledge Graph Index

Building the Knowledge Graph Index

KG-based data structures.

class gpt_index.indices.knowledge_graph.**GPTKGTableQuery**(*index_struct: KG*, *query_keyword_extract_template: Optional[QueryKeywordExtractPrompt] = None*, *max_keywords_per_query: int = 10*, *num_chunks_per_query: int = 10*, *include_text: bool = True*, *embedding_mode: Optional[KGQueryMode] = KGQueryMode.KEYWORD*, *similarity_top_k: int = 2*, ***kwargs: Any*)

Base GPT KG Table Index Query.

Arguments are shared among subclasses.

Parameters

- **query_keyword_extract_template** (*Optional[QueryKGExtractPrompt]*) – A Query KG Extraction Prompt (see *Prompt Templates*).
- **refine_template** (*Optional[RefinePrompt]*) – A Refinement Prompt (see *Prompt Templates*).
- **text_qa_template** (*Optional[QuestionAnswerPrompt]*) – A Question Answering Prompt (see *Prompt Templates*).
- **max_keywords_per_query** (*int*) – Maximum number of keywords to extract from query.
- **num_chunks_per_query** (*int*) – Maximum number of text chunks to query.
- **include_text** (*bool*) – Use the document text source from each relevant triplet during queries.
- **embedding_mode** (KGQueryMode) – Specifies whether to use keyowrds, embeddings, or both to find relevant triplets. Should be one of "keyword", "embedding", or "hybrid".
- **similarity_top_k** (*int*) – The number of top embeddings to use (if embeddings are used).

retrieve(*query_bundle:* QueryBundle) → List[*NodeWithScore*]

Get list of tuples of node and similarity for response.

First part of the tuple is the node. Second part of tuple is the distance from query to the node. If not applicable, it's None.

class gpt_index.indices.knowledge_graph.**GPTKnowledgeGraphIndex**(*nodes: Optional[Sequence[Node]] = None*, *index_struct: Optional[KG] = None*, *kg_triple_extract_template: Optional[KnowledgeGraphPrompt] = None*, *max_triplets_per_chunk: int = 10*, *include_embeddings: bool = False*, ***kwargs: Any*)

GPT Knowledge Graph Index.

Build a KG by extracting triplets, and leveraging the KG during query-time.

Parameters

- **kg_triple_extract_template** (KnowledgeGraphPrompt) – The prompt to use for extracting triplets.
- **max_triplets_per_chunk** (*int*) – The maximum number of triplets to extract.

add_node(*keywords: List[str]*, *node: Node*) → None

Add node.

Used for manual insertion of nodes (keyed by keywords).

Parameters

- **keywords** (*List[str]*) – Keywords to index the node.
- **node** (Node) – Node to be indexed.

async aquery(*query_str: Union[str, QueryBundle]*, *mode: str = QueryMode.DEFAULT*, *query_transform: Optional[BaseQueryTransform] = None*, ***query_kwargs: Any*) → Union[*Response*, *StreamingResponse*]

Asynchronously answer a query.

When *query* is called, we query the index with the given *mode* and *query_kwargs*. The *mode* determines the type of query to run, and *query_kwargs* are parameters that are specific to the query type.

For a comprehensive documentation of available *mode* and *query_kwargs* to query a given index, please visit *Querying an Index*.

classmethod from_documents(*documents: Sequence[Document]*, *docstore: Optional[BaseDocumentStore] = None*, *service_context: Optional[ServiceContext] = None*, ***kwargs: Any*) → *BaseGPTIndex*

Create index from documents.

Parameters

documents (*Optional[Sequence[BaseDocument]]*) – List of documents to build the index from.

get_networkx_graph() → Any

Get networkx representation of the graph structure.

NOTE: This function requires networkx to be installed. NOTE: This is a beta feature.

classmethod get_query_map() → Dict[str, Type[*BaseGPTIndexQuery*]]

Get query map.

insert(*document: Document*, ***insert_kwargs: Any*) → None

Insert a document.

classmethod load_from_dict(*result_dict: Dict[str, Any]*, ***kwargs: Any*) → *BaseGPTIndex*

Load index from dict.

classmethod load_from_disk(*save_path: str*, ***kwargs: Any*) → *BaseGPTIndex*

Load index from disk.

This method loads the index from a JSON file stored on disk. The index data structure itself is preserved completely. If the index is defined over subindices, those subindices will also be preserved (and subindices of those subindices, etc.).

NOTE: load_from_disk should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_disk* and *load_from_disk* on that instead.

Parameters
save_path (*str*) – The save_path of the file.

Returns
The loaded index.

Return type
BaseGPTIndex

classmethod load_from_string(*index_string: str*, ***kwargs: Any*) → *BaseGPTIndex*

Load index from string (in JSON-format).

This method loads the index from a JSON string. The index data structure itself is preserved completely. If the index is defined over subindices, those subindices will also be preserved (and subindices of those subindices, etc.).

NOTE: load_from_string should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_string* and *load_from_string* on that instead.

Parameters
index_string (*str*) – The index string (in JSON-format).

Returns
The loaded index.

Return type
BaseGPTIndex

query(*query_str: Union[str,* QueryBundle*]*, *mode: str = QueryMode.DEFAULT*, *query_transform: Optional[BaseQueryTransform] = None*, *use_async: bool = False*, ***query_kwargs: Any*) → Union[*Response*, *StreamingResponse*]

Answer a query.

When *query* is called, we query the index with the given *mode* and *query_kwargs*. The *mode* determines the type of query to run, and *query_kwargs* are parameters that are specific to the query type.

For a comprehensive documentation of available *mode* and *query_kwargs* to query a given index, please visit *Querying an Index*.

property query_context: Dict[str, Any]

Additional context necessary for making a query.

This should capture any index-specific clients, services, etc, that's not captured by index struct, docstore, and service context. For example, a vector store index would pass vector store.

refresh(*documents: Sequence[*Document*]*, ***update_kwargs: Any*) → List[bool]

Refresh an index with documents that have changed.

This allows users to save LLM and Embedding model calls, while only updating documents that have any changes in text or extra_info. It will also insert any documents that previously were not stored.

save_to_dict(***save_kwargs: Any*) → dict

Save to dict.

save_to_disk(*save_path: str*, *encoding: str = 'ascii'*, ***save_kwargs: Any*) → None

Save to file.

This method stores the index into a JSON file stored on disk.

NOTE: save_to_disk should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_disk* and *load_from_disk* on that instead.

Parameters

- **save_path** (*str*) – The save_path of the file.
- **encoding** (*str*) – The encoding of the file.

save_to_string(***save_kwargs: Any*) → str

Save to string.

This method stores the index into a JSON string.

NOTE: save_to_string should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_string* and *load_from_string* on that instead.

Returns
The JSON string of the index.

Return type
str

update(*document:* Document, ***update_kwargs: Any*) → None

Update a document.

This is equivalent to deleting the document and then inserting it again.

Parameters

- **document** (*Union[BaseDocument,* BaseGPTIndex*]*) – document to update
- **insert_kwargs** (*Dict*) – kwargs to pass to insert
- **delete_kwargs** (*Dict*) – kwargs to pass to delete

upsert_triplet(*triplet: Tuple[str, str, str]*) → None

Insert triplets.

Used for manual insertion of KG triplets (in the form of (subject, relationship, object)).

Args
triplet (str): Knowledge triplet

upsert_triplet_and_node(*triplet: Tuple[str, str, str]*, *node:* Node) → None

Upsert KG triplet and node.

Calls both upsert_triplet and add_node. Behavior is idempotent; if Node already exists, only triplet will be added.

Parameters

- **keywords** (*List[str]*) – Keywords to index the node.
- **node** (Node) – Node to be indexed.

class gpt_index.indices.knowledge_graph.**KGQueryMode**(*value*)

Query mode enum for Knowledge Graphs.

Can be passed as the enum struct, or as the underlying string.

KEYWORD

Default query mode, using keywords to find triplets.

Type
"keyword"

EMBEDDING

Embedding mode, using embeddings to find similar triplets.

Type
"embedding"

HYBRID

Hyrbid mode, combining both keywords and embeddings to find relevant triplets.

Type
"hybrid"

3.17.7 Empty Index

Building the Empty Index

Empty Index.

class gpt_index.indices.empty.**GPTEmptyIndex**(*index_struct: Optional[EmptyIndex] = None*, *service_context: Optional[ServiceContext] = None*, ***kwargs: Any*)

GPT Empty Index.

An index that doesn't contain any documents. Used for pure LLM calls. NOTE: this exists because an empty index it allows certain properties, such as the ability to be composed with other indices + token counting + others.

async aquery(*query_str: Union[str, QueryBundle]*, *mode: str = QueryMode.DEFAULT*, *query_transform: Optional[BaseQueryTransform] = None*, ***query_kwargs: Any*) → Union[*Response*, *StreamingResponse*]

Asynchronously answer a query.

When *query* is called, we query the index with the given *mode* and *query_kwargs*. The *mode* determines the type of query to run, and *query_kwargs* are parameters that are specific to the query type.

For a comprehensive documentation of available *mode* and *query_kwargs* to query a given index, please visit *Querying an Index*.

classmethod from_documents(*documents: Sequence[Document]*, *docstore: Optional[BaseDocumentStore] = None*, *service_context: Optional[ServiceContext] = None*, ***kwargs: Any*) → *BaseGPTIndex*

Create index from documents.

Parameters
documents (*Optional[Sequence[BaseDocument]]*) – List of documents to build the index from.

classmethod get_query_map() → Dict[str, Type[*BaseGPTIndexQuery*]]

Get query map.

insert(*document:* Document, ***insert_kwargs: Any*) → None

Insert a document.

classmethod load_from_dict(*result_dict: Dict[str, Any]*, ***kwargs: Any*) → *BaseGPTIndex*

Load index from dict.

classmethod load_from_disk(*save_path: str*, ***kwargs: Any*) → *BaseGPTIndex*

Load index from disk.

This method loads the index from a JSON file stored on disk. The index data structure itself is preserved completely. If the index is defined over subindices, those subindices will also be preserved (and subindices of those subindices, etc.).

NOTE: load_from_disk should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_disk* and *load_from_disk* on that instead.

Parameters
save_path (*str*) – The save_path of the file.

Returns
The loaded index.

Return type
BaseGPTIndex

classmethod load_from_string(*index_string: str*, ***kwargs: Any*) → *BaseGPTIndex*

Load index from string (in JSON-format).

This method loads the index from a JSON string. The index data structure itself is preserved completely. If the index is defined over subindices, those subindices will also be preserved (and subindices of those subindices, etc.).

NOTE: load_from_string should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_string* and *load_from_string* on that instead.

Parameters
index_string (*str*) – The index string (in JSON-format).

Returns
The loaded index.

Return type
BaseGPTIndex

query(*query_str: Union[str,* QueryBundle*]*, *mode: str = QueryMode.DEFAULT*, *query_transform: Optional[BaseQueryTransform] = None*, *use_async: bool = False*, ***query_kwargs: Any*) → Union[*Response*, *StreamingResponse*]

Answer a query.

When *query* is called, we query the index with the given *mode* and *query_kwargs*. The *mode* determines the type of query to run, and *query_kwargs* are parameters that are specific to the query type.

For a comprehensive documentation of available *mode* and *query_kwargs* to query a given index, please visit *Querying an Index*.

property query_context: Dict[str, Any]

Additional context necessary for making a query.

This should capture any index-specific clients, services, etc, that's not captured by index struct, docstore, and service context. For example, a vector store index would pass vector store.

refresh(*documents: Sequence[*Document*]*, ***update_kwargs: Any*) → List[bool]

Refresh an index with documents that have changed.

This allows users to save LLM and Embedding model calls, while only updating documents that have any changes in text or extra_info. It will also insert any documents that previously were not stored.

save_to_dict(***save_kwargs: Any*) → dict

Save to dict.

save_to_disk(*save_path: str*, *encoding: str = 'ascii'*, ***save_kwargs: Any*) → None

Save to file.

This method stores the index into a JSON file stored on disk.

NOTE: save_to_disk should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_disk* and *load_from_disk* on that instead.

Parameters

- **save_path** (*str*) – The save_path of the file.
- **encoding** (*str*) – The encoding of the file.

save_to_string(***save_kwargs: Any*) → str

Save to string.

This method stores the index into a JSON string.

NOTE: save_to_string should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_string* and *load_from_string* on that instead.

Returns
The JSON string of the index.

Return type
str

update(*document:* Document, ***update_kwargs: Any*) → None

Update a document.

This is equivalent to deleting the document and then inserting it again.

Parameters

- **document** (*Union[BaseDocument,* BaseGPTIndex*]*) – document to update
- **insert_kwargs** (*Dict*) – kwargs to pass to insert
- **delete_kwargs** (*Dict*) – kwargs to pass to delete

class gpt_index.indices.empty.**GPTEmptyIndexQuery**(*input_prompt: Optional[*SimpleInputPrompt*] = None*, ***kwargs: Any*)

GPTEmptyIndex query.

Passes the raw LLM call to the underlying LLM model.

```
response = index.query("<query_str>", mode="default")
```

Parameters
input_prompt (*Optional[SimpleInputPrompt]*) – A Simple Input Prompt (see *Prompt Templates*).

retrieve(*query_bundle:* QueryBundle) → List[*NodeWithScore*]

Retrieve relevant nodes.

3.17.8 Base Index Class

Base index classes.

class gpt_index.indices.base.**BaseGPTIndex**(*nodes: Optional[Sequence[*Node*]] = None*, *index_struct: Optional[IS] = None*, *docstore: Optional[*BaseDocumentStore*] = None*, *service_context: Optional[*ServiceContext*] = None*)

Base LlamaIndex.

Parameters

- **nodes** (*List[Node]*) – List of nodes to index
- **service_context** (ServiceContext) – Service context container (contains components like LLMPredictor, PromptHelper, etc.).

async aquery(*query_str: Union[str,* QueryBundle*]*, *mode: str = QueryMode.DEFAULT*, *query_transform: Optional[BaseQueryTransform] = None*, ***query_kwargs: Any*) → Union[*Response*, *StreamingResponse*]

Asynchronously answer a query.

When *query* is called, we query the index with the given *mode* and *query_kwargs*. The *mode* determines the type of query to run, and *query_kwargs* are parameters that are specific to the query type.

For a comprehensive documentation of available *mode* and *query_kwargs* to query a given index, please visit *Querying an Index*.

delete(*doc_id: str*, ***delete_kwargs: Any*) → None

Delete a document from the index.

All nodes in the index related to the index will be deleted.

Parameters
doc_id (*str*) – document id

property docstore: *BaseDocumentStore*

Get the docstore corresponding to the index.

classmethod from_documents(*documents: Sequence[*Document*]*, *docstore: Optional[*BaseDocumentStore*] = None*, *service_context: Optional[*ServiceContext*] = None*, ***kwargs: Any*) → *BaseGPTIndex*

Create index from documents.

Parameters
documents (*Optional[Sequence[BaseDocument]]*) – List of documents to build the index from.

abstract classmethod get_query_map() → Dict[str, Type[*BaseGPTIndexQuery*]]

Get query map.

property index_struct: IS

Get the index struct.

insert(*document:* Document, ***insert_kwargs: Any*) → None

Insert a document.

classmethod load_from_dict(*result_dict: Dict[str, Any]*, ***kwargs: Any*) → *BaseGPTIndex*

Load index from dict.

classmethod load_from_disk(*save_path: str*, ***kwargs: Any*) → *BaseGPTIndex*

Load index from disk.

This method loads the index from a JSON file stored on disk. The index data structure itself is preserved completely. If the index is defined over subindices, those subindices will also be preserved (and subindices of those subindices, etc.).

NOTE: load_from_disk should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_disk* and *load_from_disk* on that instead.

Parameters
save_path (*str*) – The save_path of the file.

Returns
The loaded index.

Return type
BaseGPTIndex

classmethod load_from_string(*index_string: str*, ***kwargs: Any*) → *BaseGPTIndex*

Load index from string (in JSON-format).

This method loads the index from a JSON string. The index data structure itself is preserved completely. If the index is defined over subindices, those subindices will also be preserved (and subindices of those subindices, etc.).

NOTE: load_from_string should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_string* and *load_from_string* on that instead.

Parameters
index_string (*str*) – The index string (in JSON-format).

Returns
The loaded index.

Return type
BaseGPTIndex

query(*query_str: Union[str,* QueryBundle*]*, *mode: str = QueryMode.DEFAULT*, *query_transform: Optional[BaseQueryTransform] = None*, *use_async: bool = False*, ***query_kwargs: Any*) → Union[*Response*, *StreamingResponse*]

Answer a query.

When *query* is called, we query the index with the given *mode* and *query_kwargs*. The *mode* determines the type of query to run, and *query_kwargs* are parameters that are specific to the query type.

For a comprehensive documentation of available *mode* and *query_kwargs* to query a given index, please visit *Querying an Index*.

property query_context: Dict[str, Any]

Additional context necessary for making a query.

This should capture any index-specific clients, services, etc, that's not captured by index struct, docstore, and service context. For example, a vector store index would pass vector store.

refresh(*documents: Sequence[*Document*]*, ***update_kwargs: Any*) → List[bool]

Refresh an index with documents that have changed.

This allows users to save LLM and Embedding model calls, while only updating documents that have any changes in text or extra_info. It will also insert any documents that previously were not stored.

save_to_dict(***save_kwargs: Any*) → dict

Save to dict.

save_to_disk(*save_path: str*, *encoding: str = 'ascii'*, ***save_kwargs: Any*) → None

Save to file.

This method stores the index into a JSON file stored on disk.

NOTE: save_to_disk should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_disk* and *load_from_disk* on that instead.

Parameters

- **save_path** (*str*) – The save_path of the file.
- **encoding** (*str*) – The encoding of the file.

save_to_string(***save_kwargs: Any*) → str

Save to string.

This method stores the index into a JSON string.

NOTE: save_to_string should not be used for indices composed on top of other indices. Please define a *ComposableGraph* and use *save_to_string* and *load_from_string* on that instead.

Returns
The JSON string of the index.

Return type
str

update(*document:* Document, ***update_kwargs: Any*) → None

Update a document.

This is equivalent to deleting the document and then inserting it again.

Parameters

- **document** (*Union[BaseDocument,* BaseGPTIndex*]*) – document to update
- **insert_kwargs** (*Dict*) – kwargs to pass to insert
- **delete_kwargs** (*Dict*) – kwargs to pass to delete

3.18 Querying an Index

This doc shows the classes that are used to query indices. We first show index-specific query subclasses. We then show how to define a query config in order to recursively query multiple indices that are composed together. We then show the base query class, which contains parameters that are shared among all queries. Lastly, we show how to customize the string(s) used for an embedding-based query.

3.18.1 Querying a List Index

Default query for GPTListIndex.

class gpt_index.indices.list.query.**BaseGPTListIndexQuery**(*index_struct: IS*, *service_context: ServiceContext*, *response_synthesizer: ResponseSynthesizer*, *docstore: Optional[BaseDocumentStore] = None*, *node_postprocessors: Optional[List[BaseNodePostprocessor]] = None*, *include_extra_info: bool = False*, *verbose: bool = False*)

GPTListIndex query.

Arguments are shared among subclasses.

Parameters

- **text_qa_template** (*Optional[QuestionAnswerPrompt]*) – A Question Answering Prompt (see *Prompt Templates*).
- **refine_template** (*Optional[RefinePrompt]*) – A Refinement Prompt (see *Prompt Templates*).

retrieve(*query_bundle: QueryBundle*) → List[*NodeWithScore*]

Get list of tuples of node and similarity for response.

First part of the tuple is the node. Second part of tuple is the distance from query to the node. If not applicable, it's None.

class gpt_index.indices.list.query.**GPTListIndexQuery**(*index_struct: IS*, *service_context: ServiceContext*, *response_synthesizer: ResponseSynthesizer*, *docstore: Optional[BaseDocumentStore] = None*, *node_postprocessors: Optional[List[BaseNodePostprocessor]] = None*, *include_extra_info: bool = False*, *verbose: bool = False*)

GPTListIndex query.

The default query mode for GPTListIndex, which traverses each node in sequence and synthesizes a response across all nodes (with an optional keyword filter). Set when *mode="default"* in *query* method of *GPTListIndex*.

```
response = index.query("<query_str>", mode="default")
```

See BaseGPTListIndexQuery for arguments.

retrieve(*query_bundle:* QueryBundle) → List[*NodeWithScore*]

Get list of tuples of node and similarity for response.

First part of the tuple is the node. Second part of tuple is the distance from query to the node. If not applicable, it's None.

3.18.2 Querying a Keyword Table Index

Query for GPTKeywordTableIndex.

class gpt_index.indices.keyword_table.query.**BaseGPTKeywordTableQuery**(*index_struct: KeywordTable*, *keyword_extract_template: Optional[*KeywordExtractPrompt*]* = *None*, *query_keyword_extract_template: Optional[*QueryKeywordExtractPrompt*]* = *None*, *max_keywords_per_query: int* = *10*, *num_chunks_per_query: int* = *10*, ***kwargs: Any*)

Base GPT Keyword Table Index Query.

Arguments are shared among subclasses.

Parameters

- **keyword_extract_template** (*Optional[*KeywordExtractPrompt*]*) – A Keyword Extraction Prompt (see *Prompt Templates*).
- **query_keyword_extract_template** (*Optional[*QueryKeywordExtractPrompt*]*) – A Query Keyword Extraction Prompt (see *Prompt Templates*).
- **refine_template** (*Optional[*RefinePrompt*]*) – A Refinement Prompt (see *Prompt Templates*).
- **text_qa_template** (*Optional[*QuestionAnswerPrompt*]*) – A Question Answering Prompt (see *Prompt Templates*).
- **max_keywords_per_query** (*int*) – Maximum number of keywords to extract from query.
- **num_chunks_per_query** (*int*) – Maximum number of text chunks to query.

retrieve(*query_bundle:* QueryBundle) → List[*NodeWithScore*]

Get list of tuples of node and similarity for response.

First part of the tuple is the node. Second part of tuple is the distance from query to the node. If not applicable, it's None.

class gpt_index.indices.keyword_table.query.**GPTKeywordTableGPTQuery**(*index_struct: KeywordTable*, *keyword_extract_template: Optional[KeywordExtractPrompt] = None*, *query_keyword_extract_template: Optional[QueryKeywordExtractPrompt] = None*, *max_keywords_per_query: int = 10*, *num_chunks_per_query: int = 10*, ***kwargs: Any*)

GPT Keyword Table Index Query.

Extracts keywords using GPT. Set when *mode="default"* in *query* method of *GPTKeywordTableIndex*.

```
response = index.query("<query_str>", mode="default")
```

See BaseGPTKeywordTableQuery for arguments.

retrieve(*query_bundle:* QueryBundle) → List[*NodeWithScore*]

Get list of tuples of node and similarity for response.

First part of the tuple is the node. Second part of tuple is the distance from query to the node. If not applicable, it's None.

class gpt_index.indices.keyword_table.query.**GPTKeywordTableRAKEQuery**(*index_struct: KeywordTable*, *keyword_extract_template: Optional[KeywordExtractPrompt] = None*, *query_keyword_extract_template: Optional[QueryKeywordExtractPrompt] = None*, *max_keywords_per_query: int = 10*, *num_chunks_per_query: int = 10*, ***kwargs: Any*)

GPT Keyword Table Index RAKE Query.

Extracts keywords using RAKE keyword extractor. Set when *mode="rake"* in *query* method of *GPTKeywordTableIndex*.

```
response = index.query("<query_str>", mode="rake")
```

See BaseGPTKeywordTableQuery for arguments.

retrieve(*query_bundle:* QueryBundle) → List[*NodeWithScore*]

Get list of tuples of node and similarity for response.

First part of the tuple is the node. Second part of tuple is the distance from query to the node. If not applicable, it's None.

class gpt_index.indices.keyword_table.query.**GPTKeywordTableSimpleQuery**(*index_struct: KeywordTable*, *keyword_extract_template: Optional[KeywordExtractPrompt] = None*, *query_keyword_extract_template: Optional[QueryKeywordExtractPrompt] = None*, *max_keywords_per_query: int = 10*, *num_chunks_per_query: int = 10*, ***kwargs: Any*)

GPT Keyword Table Index Simple Query.

Extracts keywords using simple regex-based keyword extractor. Set when *mode="simple"* in *query* method of *GPTKeywordTableIndex*.

```
response = index.query("<query_str>", mode="simple")
```

See BaseGPTKeywordTableQuery for arguments.

retrieve(*query_bundle: QueryBundle*) → List[*NodeWithScore*]

Get list of tuples of node and similarity for response.

First part of the tuple is the node. Second part of tuple is the distance from query to the node. If not applicable, it's None.

3.18.3 Querying a Tree Index

Leaf query mechanism.

class gpt_index.indices.tree.leaf_query.**GPTTreeIndexLeafQuery**(*index_struct: IndexGraph*, *query_template: Optional[TreeSelectPrompt] = None*, *text_qa_template: Optional[QuestionAnswerPrompt] = None*, *refine_template: Optional[RefinePrompt] = None*, *query_template_multiple: Optional[TreeSelectMultiplePrompt] = None*, *child_branch_factor: int = 1*, ***kwargs: Any*)

GPT Tree Index leaf query.

This class traverses the index graph and searches for a leaf node that can best answer the query.

```
response = index.query("<query_str>", mode="default")
```

Parameters

- **query_template** (*Optional[TreeSelectPrompt]*) – Tree Select Query Prompt (see *Prompt Templates*).

- **query_template_multiple** (*Optional[TreeSelectMultiplePrompt]*) – Tree Select Query Prompt (Multiple) (see *Prompt Templates*).
- **child_branch_factor** (*int*) – Number of child nodes to consider at each level. If child_branch_factor is 1, then the query will only choose one child node to traverse for any given parent node. If child_branch_factor is 2, then the query will choose two child nodes.

retrieve(*query_bundle:* QueryBundle) → List[*NodeWithScore*]

Get list of tuples of node and similarity for response.

First part of the tuple is the node. Second part of tuple is the distance from query to the node. If not applicable, it's None.

gpt_index.indices.tree.leaf_query.**get_text_from_node**(*node:* Node, *level: Optional[int] = None*, *verbose: bool = False*) → str

Get text from node.

Query Tree using embedding similarity between query and node text.

class gpt_index.indices.tree.embedding_query.**GPTTreeIndexEmbeddingQuery**(*index_struct: IndexGraph*, *query_template: Optional[TreeSelectPrompt] = None*, *query_template_multiple: Optional[TreeSelectMultiplePrompt] = None*, *child_branch_factor: int = 1*, ***kwargs: Any*)

GPT Tree Index embedding query.

This class traverses the index graph using the embedding similarity between the query and the node text.

```
response = index.query("<query_str>", mode="embedding")
```

Parameters

- **query_template** (*Optional[TreeSelectPrompt]*) – Tree Select Query Prompt (see *Prompt Templates*).
- **query_template_multiple** (*Optional[TreeSelectMultiplePrompt]*) – Tree Select Query Prompt (Multiple) (see *Prompt Templates*).
- **text_qa_template** (*Optional[QuestionAnswerPrompt]*) – Question-Answer Prompt (see *Prompt Templates*).
- **refine_template** (*Optional[RefinePrompt]*) – Refinement Prompt (see *Prompt Templates*).
- **child_branch_factor** (*int*) – Number of child nodes to consider at each level. If child_branch_factor is 1, then the query will only choose one child node to traverse for any given parent node. If child_branch_factor is 2, then the query will choose two child nodes.
- **embed_model** (*Optional[BaseEmbedding]*) – Embedding model to use for embedding similarity.

retrieve(*query_bundle:* QueryBundle) → List[*NodeWithScore*]

Get list of tuples of node and similarity for response.

First part of the tuple is the node. Second part of tuple is the distance from query to the node. If not applicable, it's None.

Retrieve query.

class gpt_index.indices.tree.retrieve_query.**GPTTreeIndexRetQuery**(*index_struct: IS*, *service_context: ServiceContext*, *response_synthesizer: ResponseSynthesizer*, *docstore: Optional[BaseDocumentStore] = None*, *node_postprocessors: Optional[List[BaseNodePostprocessor]] = None*, *include_extra_info: bool = False*, *verbose: bool = False*)

GPT Tree Index retrieve query.

This class directly retrieves the answer from the root nodes.

Unlike GPTTreeIndexLeafQuery, this class assumes the graph already stores the answer (because it was constructed with a query_str), so it does not attempt to parse information down the graph in order to synthesize an answer.

```
response = index.query("<query_str>", mode="retrieve")
```

Parameters

text_qa_template (*Optional[QuestionAnswerPrompt]*) – Question-Answer Prompt (see *Prompt Templates*).

retrieve(*query_bundle:* QueryBundle) → List[*NodeWithScore*]

Get list of tuples of node and similarity for response.

First part of the tuple is the node. Second part of tuple is the distance from query to the node. If not applicable, it's None.

Summarize query.

class gpt_index.indices.tree.summarize_query.**GPTTreeIndexSummarizeQuery**(*index_struct: IndexGraph*, ***kwargs: Any*)

GPT Tree Index summarize query.

This class builds a query-specific tree from leaf nodes to return a response. Using this query mode means that the tree index doesn't need to be built when initialized, since we rebuild the tree for each query.

```
response = index.query("<query_str>", mode="summarize")
```

Parameters

text_qa_template (*Optional[QuestionAnswerPrompt]*) – Question-Answer Prompt (see *Prompt Templates*).

retrieve(*query_bundle:* QueryBundle) → List[*NodeWithScore*]

Get list of tuples of node and similarity for response.

First part of the tuple is the node. Second part of tuple is the distance from query to the node. If not applicable, it's None.

3.18.4 Querying a Vector Store Index

We first show the base vector store query class. We then show the query classes specific to each vector store.

Base Vector Store Query Class

Base vector store index query.

class gpt_index.indices.vector_store.base_query.**GPTVectorStoreIndexQuery**(*index_struct: IndexDict*, *service_context:* ServiceContext, *vector_store: Optional[VectorStore] = None*, *similarity_top_k: int = 1*, *vector_store_query_mode: str = VectorStoreQueryMode.DEFAULT*, *alpha: Optional[float] = None*, *doc_ids: Optional[List[str]] = None*, ***kwargs: Any*)

Base vector store query.

Parameters

- **embed_model** (*Optional[BaseEmbedding]*) – embedding model
- **similarity_top_k** (*int*) – number of top k results to return
- **vector_store** (*Optional[VectorStore]*) – vector store

retrieve(*query_bundle:* QueryBundle) → List[*NodeWithScore*]

Get list of tuples of node and similarity for response.

First part of the tuple is the node. Second part of tuple is the distance from query to the node. If not applicable, it's None.

Vector Store-specific Query Classes

3.18.5 Querying a Structured Store Index

Default query for GPTSQLStructStoreIndex.

class gpt_index.indices.struct_store.sql_query.**GPTNLStructStoreIndexQuery**(*index_struct: SQLStructTable*, *sql_database: Optional[SQLDatabase] = None*, *sql_context_container: Optional[SQLContextContainer] = None*, *ref_doc_id_column: Optional[str] = None*, *text_to_sql_prompt: Optional[TextToSQLPrompt] = None*, *context_query_mode: QueryMode = QueryMode.DEFAULT*, *context_query_kwargs: Optional[dict] = None*, ***kwargs: Any*)

GPT natural language query over a structured database.

Given a natural language query, we will extract the query to SQL. Runs raw SQL over a GPTSQLStructStoreIndex. No LLM calls are made during the SQL execution. NOTE: this query cannot work with composed indices - if the index contains subindices, those subindices will not be queried.

```
response = index.query("<query_str>", mode="default")
```

async aquery(*query_bundle:* QueryBundle) → *Response*

Answer a query.

retrieve(*query_bundle:* QueryBundle) → List[*NodeWithScore*]

Get list of tuples of node and similarity for response.

First part of the tuple is the node. Second part of tuple is the distance from query to the node. If not applicable, it's None.

class gpt_index.indices.struct_store.sql_query.**GPTSQLStructStoreIndexQuery**(*index_struct: SQLStructTable*, *sql_database: Optional[SQLDatabase] = None*, *sql_context_container: Optional[SQLContextContainer] = None*, ***kwargs: Any*)

GPT SQL query over a structured database.

Runs raw SQL over a GPTSQLStructStoreIndex. No LLM calls are made here. NOTE: this query cannot work with composed indices - if the index contains subindices, those subindices will not be queried.

```
response = index.query("<query_str>", mode="sql")
```

retrieve(*query_bundle:* QueryBundle) → List[*NodeWithScore*]

Get list of tuples of node and similarity for response.

First part of the tuple is the node. Second part of tuple is the distance from query to the node. If not applicable, it's None.

Default query for GPTPandasIndex.

class gpt_index.indices.struct_store.pandas_query.**GPTNLPandasIndexQuery**(*index_struct: PandasStructTable*, *df: Optional[DataFrame] = None*, *instruction_str: Optional[str] = None*, *output_processor: Optional[Callable] = None*, *pandas_prompt: Optional[PandasPrompt] = None*, *output_kwargs: Optional[dict] = None*, *head: int = 5*, ***kwargs: Any*)

GPT Pandas query.

Convert natural language to Pandas python code.

```
response = index.query("<query_str>", mode="default")
```

Parameters

- **df** (*pd.DataFrame*) – Pandas dataframe to use.
- **instruction_str** (*Optional[str]*) – Instruction string to use.
- **output_processor** (*Optional[Callable[[str], str]]*) – Output processor. A callable that takes in the output string, pandas DataFrame, and any output kwargs and returns a string.
- **pandas_prompt** (*Optional[PandasPrompt]*) – Pandas prompt to use.

- **head** (*int*) – Number of rows to show in the table context.

retrieve(*query_bundle:* QueryBundle) → List[*NodeWithScore*]

Get list of tuples of node and similarity for response.

First part of the tuple is the node. Second part of tuple is the distance from query to the node. If not applicable, it's None.

gpt_index.indices.struct_store.pandas_query.**default_output_processor**(*output: str*, *df: DataFrame*, ***output_kwargs: Any*) → str

Process outputs in a default manner.

3.18.6 Querying a Knowledge Graph Index

Query for GPTKGTableIndex.

class gpt_index.indices.knowledge_graph.query.**GPTKGTableQuery**(*index_struct: KG*, *query_keyword_extract_template: Optional[QueryKeywordExtractPrompt] = None*, *max_keywords_per_query: int = 10*, *num_chunks_per_query: int = 10*, *include_text: bool = True*, *embedding_mode: Optional[KGQueryMode] = KGQueryMode.KEYWORD*, *similarity_top_k: int = 2*, ***kwargs: Any*)

Base GPT KG Table Index Query.

Arguments are shared among subclasses.

Parameters

- **query_keyword_extract_template** (*Optional[QueryKGExtractPrompt]*) – A Query KG Extraction Prompt (see *Prompt Templates*).
- **refine_template** (*Optional[RefinePrompt]*) – A Refinement Prompt (see *Prompt Templates*).
- **text_qa_template** (*Optional[QuestionAnswerPrompt]*) – A Question Answering Prompt (see *Prompt Templates*).
- **max_keywords_per_query** (*int*) – Maximum number of keywords to extract from query.
- **num_chunks_per_query** (*int*) – Maximum number of text chunks to query.
- **include_text** (*bool*) – Use the document text source from each relevant triplet during queries.
- **embedding_mode** (KGQueryMode) – Specifies whether to use keyowrds, embeddings, or both to find relevant triplets. Should be one of "keyword", "embedding", or "hybrid".
- **similarity_top_k** (*int*) – The number of top embeddings to use (if embeddings are used).

retrieve(*query_bundle:* QueryBundle) → List[*NodeWithScore*]

Get list of tuples of node and similarity for response.

First part of the tuple is the node. Second part of tuple is the distance from query to the node. If not applicable, it's None.

class gpt_index.indices.knowledge_graph.query.**KGQueryMode**(*value*)

Query mode enum for Knowledge Graphs.

Can be passed as the enum struct, or as the underlying string.

KEYWORD

Default query mode, using keywords to find triplets.

Type
"keyword"

EMBEDDING

Embedding mode, using embeddings to find similar triplets.

Type
"embedding"

HYBRID

Hyrbid mode, combining both keywords and embeddings to find relevant triplets.

Type
"hybrid"

3.18.7 Querying an Empty Index

Default query for GPTEmptyIndex.

class gpt_index.indices.empty.query.**GPTEmptyIndexQuery**(*input_prompt: Optional[SimpleInputPrompt] = None*, ***kwargs: Any*)

GPTEmptyIndex query.

Passes the raw LLM call to the underlying LLM model.

```
response = index.query("<query_str>", mode="default")
```

Parameters
input_prompt (*Optional[SimpleInputPrompt]*) – A Simple Input Prompt (see *Prompt Templates*).

retrieve(*query_bundle:* QueryBundle) → List[*NodeWithScore*]

Retrieve relevant nodes.

This section shows how to define a query config in order to recursively query multiple indices that are composed together.

3.18.8 Composable Queries

Query Configuration Schema.

This schema is used under the hood for all queries, but is primarily exposed for recursive queries over composable indices.

class gpt_index.indices.query.schema.**QueryConfig**(*index_struct_type: str*, *query_mode: ~gpt_index.indices.query.schema.QueryMode*, *query_kwargs: ~typing.Dict[str, ~typing.Any] = <factory>*, *index_struct_id: ~typing.Optional[str] = None*, *query_transform: ~typing.Optional[~typing.Any] = None*, *query_combiner: ~typing.Optional[~typing.Any] = None*)

Query config.

Used under the hood for all queries. The user must explicitly specify a list of query config objects is passed during a query call to define configurations for each individual subindex within an overall composed index.

The user may choose to specify either the query config objects directly, or as a list of JSON dictionaries. For instance, the following are equivalent:

```
# using JSON dictionaries
query_configs = [
    {
        # index_struct_id is optional
        "index_struct_id": "<index_struct_id>",
        "index_struct_type": "tree",
        "query_mode": "default",
        "query_kwargs": {
            "child_branch_factor": 2
        }
    },
    ...
]
response = index.query(
    "<query_str>", mode="recursive", query_configs=query_configs
)
```

```
query_configs = [
    QueryConfig(
        index_struct_id="<index_struct_id>",
        index_struct_type=IndexStructType.TREE,
        query_mode=QueryMode.DEFAULT,
        query_kwargs={
            "child_branch_factor": 2
        }
    )
    ...
]
response = index.query(
    "<query_str>", mode="recursive", query_configs=query_configs
)
```

Parameters

- **index_struct_id** (*Optional[str]*) – The index struct id. This can be obtained by calling "get_doc_id" on the original index class. This can be set by calling "set_doc_id" on the original index class.
- **index_struct_type** (IndexStructType) – The type of index struct.
- **query_mode** (QueryMode) – The query mode.
- **query_kwargs** (*Dict[str, Any], optional*) – The query kwargs. Defaults to {}.

class gpt_index.indices.query.schema.**QueryMode**(*value*)

Query mode enum.

Can be passed as the enum struct, or as the underlying string.

DEFAULT

Default query mode.

Type
"default"

RETRIEVE

Retrieve mode.

Type
"retrieve"

EMBEDDING

Embedding mode.

Type
"embedding"

SUMMARIZE

Summarize mode. Used for hierarchical summarization in the tree index.

Type
"summarize"

SIMPLE

Simple mode. Used for keyword extraction.

Type
"simple"

RAKE

RAKE mode. Used for keyword extraction.

Type
"rake"

RECURSIVE

Recursive mode. Used to recursively query over composed indices.

Type
"recursive"

IndexStructType class.

class gpt_index.data_structs.struct_type.**IndexStructType**(*value*)

Index struct type. Identifier for a "type" of index.

TREE

Tree index. See *Tree Index* for tree indices.

Type
"tree"

LIST

List index. See *List Index* for list indices.

Type
"list"

KEYWORD_TABLE

Keyword table index. See *Table Index* for keyword table indices.

Type
"keyword_table"

DICT

Faiss Vector Store Index. See *Vector Store Index* for more information on the faiss vector store index.

Type
"dict"

SIMPLE_DICT

Simple Vector Store Index. See *Vector Store Index* for more information on the simple vector store index.

Type
"simple_dict"

WEAVIATE

Weaviate Vector Store Index. See *Vector Store Index* for more information on the Weaviate vector store index.

Type
"weaviate"

PINECONE

Pinecone Vector Store Index. See *Vector Store Index* for more information on the Pinecone vector store index.

Type
"pinecone"

DEEPLAKE

DeepLake Vector Store Index. See *Vector Store Index* for more information on the Pinecone vector store index.

Type
"deeplake"

QDRANT

Qdrant Vector Store Index. See *Vector Store Index* for more information on the Qdrant vector store index.

Type
"qdrant"

MILVUS

Milvus Vector Store Index. See *Vector Store Index* for more information on the Milvus vector store index.

Type
"milvus"

CHROMA

Chroma Vector Store Index. See *Vector Store Index* for more information on the Chroma vector store index.

Type
"chroma"

OPENSEARCH

Opensearch Vector Store Index. See *Vector Store Index* for more information on the Opensearch vector store index.

Type
"opensearch"

MYSCALE

MyScale Vector Store Index. See *Vector Store Index* for more information on the MyScale vector store index.

Type
"myscale"

CHATGPT_RETRIEVAL_PLUGIN

ChatGPT retrieval plugin index.

Type
"chatgpt_retrieval_plugin"

SQL

SQL Structured Store Index. See *Structured Store Index* for more information on the SQL vector store index.

Type
"SQL"

KG

Knowledge Graph index. See *Knowledge Graph Index* for KG indices.

Type
"kg"

3.18.9 Base Query Class

Base query classes.

class gpt_index.indices.query.base.**BaseGPTIndexQuery**(*index_struct: IS*, *service_context:* ServiceContext, *response_synthesizer: ResponseSynthesizer*, *docstore: Optional[*BaseDocumentStore*] = None*, *node_postprocessors: Optional[List[BaseNodePostprocessor]] = None*, *include_extra_info: bool = False*, *verbose: bool = False*)

Base LlamaIndex Query.

Helper class that is used to query an index. Can be called within *query* method of a BaseGPTIndex object, or instantiated independently.

Parameters

- **service_context** (ServiceContext) – service context container (contains components like LLMPredictor, PromptHelper).
- **response_mode** (*ResponseMode*) – Optional ResponseMode. If not provided, will use the default ResponseMode.
- **text_qa_template** (QuestionAnswerPrompt) – Optional QuestionAnswerPrompt object. If not provided, will use the default QuestionAnswerPrompt.
- **refine_template** (RefinePrompt) – Optional RefinePrompt object. If not provided, will use the default RefinePrompt.
- **streaming** (*bool*) – Optional bool. If True, will return a StreamingResponse object. If False, will return a Response object.

property index_struct: IS

Get the index struct.

retrieve(*query_bundle:* QueryBundle) → List[*NodeWithScore*]

Get list of tuples of node and similarity for response.

First part of the tuple is the node. Second part of tuple is the distance from query to the node. If not applicable, it's None.

3.18.10 Query Bundle

Query bundle enables user to customize the string(s) used for embedding-based query.

Query Configuration Schema.

This schema is used under the hood for all queries, but is primarily exposed for recursive queries over composable indices.

class gpt_index.indices.query.schema.**QueryBundle**(*query_str: str*, *custom_embedding_strs: Optional[List[str]] = None*, *embedding: Optional[List[float]] = None*)

Query bundle.

This dataclass contains the original query string and associated transformations.

Parameters

- **query_str** (*str*) – the original user-specified query string. This is currently used by all non embedding-based queries.
- **embedding_strs** (*list[str]*) – list of strings used for embedding the query. This is currently used by all embedding-based queries.
- **embedding** (*list[float]*) – the stored embedding for the query.

property embedding_strs: List[str]

Use custom embedding strs if specified, otherwise use query str.

class gpt_index.indices.query.schema.**QueryConfig**(*index_struct_type: str*, *query_mode: ~gpt_index.indices.query.schema.QueryMode*, *query_kwargs: ~typing.Dict[str, ~typing.Any] = <factory>*, *index_struct_id: ~typing.Optional[str] = None*, *query_transform: ~typing.Optional[~typing.Any] = None*, *query_combiner: ~typing.Optional[~typing.Any] = None*)

Query config.

Used under the hood for all queries. The user must explicitly specify a list of query config objects is passed during a query call to define configurations for each individual subindex within an overall composed index.

The user may choose to specify either the query config objects directly, or as a list of JSON dictionaries. For instance, the following are equivalent:

```
# using JSON dictionaries
query_configs = [
    {
        # index_struct_id is optional
        "index_struct_id": "<index_struct_id>",
        "index_struct_type": "tree",
        "query_mode": "default",
        "query_kwargs": {
            "child_branch_factor": 2
        }
    },
    ...
]
response = index.query(
    "<query_str>", mode="recursive", query_configs=query_configs
)
```

```
query_configs = [
    QueryConfig(
        index_struct_id="<index_struct_id>",
        index_struct_type=IndexStructType.TREE,
        query_mode=QueryMode.DEFAULT,
        query_kwargs={
            "child_branch_factor": 2
        }
    )
    ...
]
response = index.query(
    "<query_str>", mode="recursive", query_configs=query_configs
)
```

Parameters

- **index_struct_id** (*Optional[str]*) – The index struct id. This can be obtained by calling "get_doc_id" on the original index class. This can be set by calling "set_doc_id" on the original index class.
- **index_struct_type** (IndexStructType) – The type of index struct.

- **query_mode** (QueryMode) – The query mode.
- **query_kwargs** (*Dict[str, Any], optional*) – The query kwargs. Defaults to {}.

class gpt_index.indices.query.schema.**QueryMode**(*value*)

Query mode enum.

Can be passed as the enum struct, or as the underlying string.

DEFAULT

Default query mode.

Type
"default"

RETRIEVE

Retrieve mode.

Type
"retrieve"

EMBEDDING

Embedding mode.

Type
"embedding"

SUMMARIZE

Summarize mode. Used for hierarchical summarization in the tree index.

Type
"summarize"

SIMPLE

Simple mode. Used for keyword extraction.

Type
"simple"

RAKE

RAKE mode. Used for keyword extraction.

Type
"rake"

RECURSIVE

Recursive mode. Used to recursively query over composed indices.

Type
"recursive"

Query Transform

Query transform augments a raw query string with associated transformations to improve index querying.

Query Transforms.

class gpt_index.indices.query.query_transform.**DecomposeQueryTransform**(*llm_predictor: Optional[LLMPredictor] = None*, *decompose_query_prompt: Optional[DecomposeQueryTransformPrompt] = None*, *verbose: bool = False*)

Decompose query transform.

Decomposes query into a subquery given the current index struct. Performs a single step transformation.

Parameters
llm_predictor (*Optional[LLMPredictor]*) – LLM for generating hypothetical documents

run(*query_bundle_or_str: Union[str, QueryBundle]*, *extra_info: Optional[Dict] = None*) → *QueryBundle*

Run query transform.

class gpt_index.indices.query.query_transform.**HyDEQueryTransform**(*llm_predictor: Optional[LLMPredictor] = None*, *hyde_prompt: Optional[Prompt] = None*, *include_original: bool = True*)

Hypothetical Document Embeddings (HyDE) query transform.

It uses an LLM to generate hypothetical answer(s) to a given query, and use the resulting documents as embedding strings.

As described in *[Precise Zero-Shot Dense Retrieval without Relevance Labels] (https://arxiv.org/abs/2212.10496)*

run(*query_bundle_or_str: Union[str, QueryBundle]*, *extra_info: Optional[Dict] = None*) → *QueryBundle*

Run query transform.

class gpt_index.indices.query.query_transform.**StepDecomposeQueryTransform**(*llm_predictor: Optional[LLMPredictor] = None*, *step_decompose_query_prompt: Optional[StepDecomposeQueryTransform = None*, *verbose: bool = False*)

Step decompose query transform.

Decomposes query into a subquery given the current index struct and previous reasoning.

NOTE: doesn't work yet.

Parameters
llm_predictor (*Optional[LLMPredictor]*) – LLM for generating hypothetical documents

run(*query_bundle_or_str: Union[str, QueryBundle]*, *extra_info: Optional[Dict] = None*) → *QueryBundle*

Run query transform.

3.19 Node

Node data structure.

Node is a generic data container that contains a piece of data (e.g. chunk of text, an image, a table, etc).

In comparison to a raw *Document*, it contains additional metadata about its relationship to other *Node* objects (and *Document* objects).

It is often used as an atomic unit of data in various indices.

class gpt_index.data_structs.node_v2.**DocumentRelationship**(*value*)

Document relationships used in *Node* class.

SOURCE

The node is the source document.

PREVIOUS

The node is the previous node in the document.

NEXT

The node is the next node in the document.

PARENT

The node is the parent node in the document.

CHILD

The node is a child node in the document.

class gpt_index.data_structs.node_v2.**Node**(*text: ~typing.Optional[str] = None*, *doc_id: ~typing.Optional[str] = None*, *embedding: ~typing.Optional[~typing.List[float]] = None*, *doc_hash: ~typing.Optional[str] = None*, *extra_info: ~typing.Optional[~typing.Dict[str, ~typing.Any]] = None*, *node_info: ~typing.Optional[~typing.Dict[str, ~typing.Any]] = None*, *relationships: ~typing.Dict[~gpt_index.data_structs.node_v2.DocumentRelationship, ~typing.Any] = <factory>*)

A generic node of data.

Parameters

- **text** (*str*) – The text of the node.
- **doc_id** (*Optional[str]*) – The document id of the node.
- **embeddings** (*Optional[List[float]]*) – The embeddings of the node.
- **relationships** (*Dict[DocumentRelationship, Any]*) – The relationships of the node.

property child_node_ids: List[str]

Child node ids.

property extra_info_str: Optional[str]

Extra info string.

get_doc_hash() → str

Get doc_hash.

get_doc_id() → str
Get doc_id.

get_embedding() → List[float]
Get embedding.

Errors if embedding is None.

get_node_info() → Dict[str, Any]
Get node info.

get_text() → str
Get text.

classmethod get_type() → str
Get type.

classmethod get_types() → List[str]
Get Document type.

property is_doc_id_none: bool
Check if doc_id is None.

property is_text_none: bool
Check if text is None.

property next_node_id: str
Next node id.

property parent_node_id: str
Parent node id.

property prev_node_id: str
Prev node id.

property ref_doc_id: Optional[str]
Source document id.

Extracted from the relationships field.

class gpt_index.data_structs.node_v2.**NodeWithScore**(*node:* gpt_index.data_structs.node_v2.Node, *score: Optional[float] = None*)

3.20 Node Postprocessor

Node PostProcessor module.

class gpt_index.indices.postprocessor.**AutoPrevNextNodePostprocessor**(*, *docstore:* BaseDocumentStore, *service_context:* ServiceContext, *num_nodes: int = 1*, *infer_prev_next_tmpl: str = "The current context information is provided. \nA question is also provided. \nYou are a retrieval agent deciding whether to search the document store for additional prior context or future context. \nGiven the context and question, return PREVIOUS or NEXT or NONE. \nExamples: \n\nContext: Describes the author's experience at Y Combinator.Question: What did the author do after his time at Y Combinator? \nAnswer: NEXT \n\nContext: Describes the author's experience at Y Combinator.Question: What did the author do before his time at Y Combinator? \nAnswer: PREVIOUS \n\nContext: Describe the author's experience at Y Combinator.Question: What did the author do at Y Combinator? \nAnswer: NONE \n\nContext: {context_str}\nQuestion: {query_str}\nAnswer: "*, *refine_prev_next_tmpl: str = 'The current context information is provided. \nA question is also provided. \nAn existing answer is also provided.\nYou are a retrieval agent deciding whether to search the document store for additional prior context or future context. \nGiven the context, question, and previous answer, return PREVIOUS or NEXT or NONE.\nExamples: \n\nContext: {context_str}\nQuestion: {query_str}\nExisting Answer: {existing_answer}\nAnswer: '*,

Previous/Next Node post-processor.

Allows users to fetch additional nodes from the document store, based on the prev/next relationships of the nodes.

NOTE: difference with PrevNextPostprocessor is that this infers forward/backwards direction.

NOTE: this is a beta feature.

Parameters

- **docstore** (BaseDocumentStore) – The document store.
- **llm_predictor** (*LLMPredictor*) – The LLM predictor.
- **num_nodes** (*int*) – The number of nodes to return (default: 1)
- **infer_prev_next_tmpl** (*str*) – The template to use for inference. Required fields are {context_str} and {query_str}.

class Config

Configuration for this pydantic object.

classmethod construct(*_fields_set: Optional[SetStr] = None*, ***values: Any*) → Model

Creates a new model setting __dict__ and __fields_set__ from trusted or pre-validated data. Default values are respected, but no other validation is performed. Behaves as if *Config.extra = 'allow'* was set since it adds all passed values

copy(***, *include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *update: Optional[DictStrAny] = None*, *deep: bool = False*) → Model

Duplicate a model, optionally choose which fields to include, exclude and change.

Parameters

- **include** – fields to include in new model
- **exclude** – fields to exclude from new model, as with values this takes precedence over include
- **update** – values to change/add in the new model. Note: the data is not validated before creating the new model: you should trust this data
- **deep** – set to *True* to make a deep copy of the model

Returns

new model instance

dict(***, *include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *by_alias: bool = False*, *skip_defaults: Optional[bool] = None*, *exclude_unset: bool = False*, *exclude_defaults: bool = False*, *exclude_none: bool = False*) → DictStrAny

Generate a dictionary representation of the model, optionally specifying which fields to include or exclude.

json(***, *include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *by_alias: bool = False*, *skip_defaults: Optional[bool] = None*, *exclude_unset: bool = False*, *exclude_defaults: bool = False*, *exclude_none: bool = False*, *encoder: Optional[Callable[[Any], Any]] = None*, *models_as_dict: bool = True*, ***dumps_kwargs: Any*) → unicode

Generate a JSON representation of the model, *include* and *exclude* arguments as per *dict()*.

encoder is an optional function to supply as *default* to json.dumps(), other arguments as per *json.dumps()*.

postprocess_nodes(*nodes: List[Node]*, *extra_info: Optional[Dict] = None*) → List[*Node*]

Postprocess nodes.

classmethod update_forward_refs(***localns: Any*) → None

Try to update ForwardRefs on fields based on this Model, globalns and localns.

class gpt_index.indices.postprocessor.**BasePostprocessor**

Base Postprocessor.

class gpt_index.indices.postprocessor.**EmbeddingRecencyPostprocessor**(**, service_context: ServiceContext, date_key: str = 'date', in_extra_info: bool = True, similarity_cutoff: float = 0.7, query_embedding_tmpl: str = 'The current document is provided.\n---------- \n{context_str}\n---------- ---\nGiven the document, we wish to find documents that contain \nsimilar context. Note that these documents are older than the current document, meaning that certain details may be changed. \nHowever, the high-level context should be similar.\n'*)

Recency post-processor.

This post-processor does the following steps: - Decides if we need to use the post-processor given the query

(is it temporal-related?)

- If yes, sorts nodes by date.
- **For each node, look at subsequent nodes and filter out nodes**
  that have high embedding similarity with the current node. (because this means)

classmethod construct(*_fields_set: Optional[SetStr] = None*, ***values: Any*) → Model

Creates a new model setting __dict__ and __fields_set__ from trusted or pre-validated data. Default values are respected, but no other validation is performed. Behaves as if *Config.extra = 'allow'* was set since it adds all passed values

copy(**, include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *update: Optional[DictStrAny] = None*, *deep: bool = False*) → Model

Duplicate a model, optionally choose which fields to include, exclude and change.

Parameters

- **include** – fields to include in new model
- **exclude** – fields to exclude from new model, as with values this takes precedence over include

- **update** – values to change/add in the new model. Note: the data is not validated before creating the new model: you should trust this data
- **deep** – set to *True* to make a deep copy of the model

Returns
new model instance

dict(*, *include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *by_alias: bool = False*, *skip_defaults: Optional[bool] = None*, *exclude_unset: bool = False*, *exclude_defaults: bool = False*, *exclude_none: bool = False*) → DictStrAny

Generate a dictionary representation of the model, optionally specifying which fields to include or exclude.

json(*, *include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *by_alias: bool = False*, *skip_defaults: Optional[bool] = None*, *exclude_unset: bool = False*, *exclude_defaults: bool = False*, *exclude_none: bool = False*, *encoder: Optional[Callable[[Any], Any]] = None*, *models_as_dict: bool = True*, ***dumps_kwargs: Any*) → unicode

Generate a JSON representation of the model, *include* and *exclude* arguments as per *dict()*.

encoder is an optional function to supply as *default* to json.dumps(), other arguments as per *json.dumps()*.

postprocess_nodes(*nodes: List[Node]*, *extra_info: Optional[Dict] = None*) → List[*Node*]

Postprocess nodes.

classmethod update_forward_refs(***localns: Any*) → None

Try to update ForwardRefs on fields based on this Model, globalns and localns.

class gpt_index.indices.postprocessor.**FixedRecencyPostprocessor**(*, *service_context: ServiceContext*, *top_k: int = 1*, *date_key: str = 'date'*, *in_extra_info: bool = True*)

Recency post-processor.

This post-processor does the following steps: - Decides if we need to use the post-processor given the query

(is it temporal-related?)

- If yes, sorts nodes by date.
- Take the first k nodes (by default 1), and use that to synthesize an answer.

classmethod construct(*_fields_set: Optional[SetStr] = None*, ***values: Any*) → Model

Creates a new model setting __dict__ and __fields_set__ from trusted or pre-validated data. Default values are respected, but no other validation is performed. Behaves as if *Config.extra = 'allow'* was set since it adds all passed values

copy(*, *include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *update: Optional[DictStrAny] = None*, *deep: bool = False*) → Model

Duplicate a model, optionally choose which fields to include, exclude and change.

Parameters

- **include** – fields to include in new model
- **exclude** – fields to exclude from new model, as with values this takes precedence over include

- **update** – values to change/add in the new model. Note: the data is not validated before creating the new model: you should trust this data
- **deep** – set to *True* to make a deep copy of the model

Returns
new model instance

dict(*, *include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *by_alias: bool = False*, *skip_defaults: Optional[bool] = None*, *exclude_unset: bool = False*, *exclude_defaults: bool = False*, *exclude_none: bool = False*) → DictStrAny

Generate a dictionary representation of the model, optionally specifying which fields to include or exclude.

json(*, *include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *by_alias: bool = False*, *skip_defaults: Optional[bool] = None*, *exclude_unset: bool = False*, *exclude_defaults: bool = False*, *exclude_none: bool = False*, *encoder: Optional[Callable[[Any], Any]] = None*, *models_as_dict: bool = True*, ***dumps_kwargs: Any*) → unicode

Generate a JSON representation of the model, *include* and *exclude* arguments as per *dict()*.

encoder is an optional function to supply as *default* to json.dumps(), other arguments as per *json.dumps()*.

postprocess_nodes(*nodes: List[*Node*]*, *extra_info: Optional[Dict] = None*) → List[*Node*]

Postprocess nodes.

classmethod update_forward_refs(***localns: Any*) → None

Try to update ForwardRefs on fields based on this Model, globalns and localns.

class gpt_index.indices.postprocessor.**KeywordNodePostprocessor**(*, *required_keywords: List[str] = None*, *exclude_keywords: List[str] = None*)

Keyword-based Node processor.

classmethod construct(*_fields_set: Optional[SetStr] = None*, ***values: Any*) → Model

Creates a new model setting __dict__ and __fields_set__ from trusted or pre-validated data. Default values are respected, but no other validation is performed. Behaves as if *Config.extra = 'allow'* was set since it adds all passed values

copy(*, *include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *update: Optional[DictStrAny] = None*, *deep: bool = False*) → Model

Duplicate a model, optionally choose which fields to include, exclude and change.

Parameters

- **include** – fields to include in new model
- **exclude** – fields to exclude from new model, as with values this takes precedence over include
- **update** – values to change/add in the new model. Note: the data is not validated before creating the new model: you should trust this data
- **deep** – set to *True* to make a deep copy of the model

Returns
new model instance

dict(*, *include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *by_alias: bool = False*, *skip_defaults: Optional[bool] = None*, *exclude_unset: bool = False*, *exclude_defaults: bool = False*, *exclude_none: bool = False*) → DictStrAny

Generate a dictionary representation of the model, optionally specifying which fields to include or exclude.

json(*, *include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *by_alias: bool = False*, *skip_defaults: Optional[bool] = None*, *exclude_unset: bool = False*, *exclude_defaults: bool = False*, *exclude_none: bool = False*, *encoder: Optional[Callable[[Any], Any]] = None*, *models_as_dict: bool = True*, ***dumps_kwargs: Any*) → unicode

Generate a JSON representation of the model, *include* and *exclude* arguments as per *dict()*.

encoder is an optional function to supply as *default* to json.dumps(), other arguments as per *json.dumps()*.

postprocess_nodes(*nodes: List[*Node*]*, *extra_info: Optional[Dict] = None*) → List[*Node*]

Postprocess nodes.

classmethod update_forward_refs(***localns: Any*) → None

Try to update ForwardRefs on fields based on this Model, globalns and localns.

class gpt_index.indices.postprocessor.**NERPIINodePostprocessor**(*, *pii_node_info_key: str = '__pii_node_info__'*)

NER PII Node processor.

Uses a HF transformers model.

classmethod construct(*_fields_set: Optional[SetStr] = None*, ***values: Any*) → Model

Creates a new model setting __dict__ and __fields_set__ from trusted or pre-validated data. Default values are respected, but no other validation is performed. Behaves as if *Config.extra = 'allow'* was set since it adds all passed values

copy(*, *include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *update: Optional[DictStrAny] = None*, *deep: bool = False*) → Model

Duplicate a model, optionally choose which fields to include, exclude and change.

Parameters

- **include** – fields to include in new model
- **exclude** – fields to exclude from new model, as with values this takes precedence over include
- **update** – values to change/add in the new model. Note: the data is not validated before creating the new model: you should trust this data
- **deep** – set to *True* to make a deep copy of the model

Returns

new model instance

dict(*, *include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *by_alias: bool = False*, *skip_defaults: Optional[bool] = None*, *exclude_unset: bool = False*, *exclude_defaults: bool = False*, *exclude_none: bool = False*) → DictStrAny

Generate a dictionary representation of the model, optionally specifying which fields to include or exclude.

json(*, *include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *by_alias: bool = False*, *skip_defaults: Optional[bool] = None*, *exclude_unset: bool = False*, *exclude_defaults: bool = False*, *exclude_none: bool = False*, *encoder: Optional[Callable[[Any], Any]] = None*, *models_as_dict: bool = True*, ***dumps_kwargs: Any*) → unicode

Generate a JSON representation of the model, *include* and *exclude* arguments as per *dict()*.

encoder is an optional function to supply as *default* to json.dumps(), other arguments as per *json.dumps()*.

mask_pii(*ner: Callable*, *text: str*) → Tuple[str, Dict]

Mask PII in text.

postprocess_nodes(*nodes: List[Node]*, *extra_info: Optional[Dict] = None*) → List[*Node*]

Postprocess nodes.

classmethod update_forward_refs(***localns: Any*) → None

Try to update ForwardRefs on fields based on this Model, globalns and localns.

class gpt_index.indices.postprocessor.**PIINodePostprocessor**(*, *service_context: ServiceContext*, *pii_str_tmpl: str = 'The current context information is provided. \nA task is also provided to mask the PII within the context. \nReturn the text, with all PII masked out, and a mapping of the original PII to the masked PII. \nReturn the output of the task in JSON. \nContext:\nHello Zhang Wei, I am John. Your AnyCompany Financial Services, LLC credit card account 1111-0000-1111-0008 has a minimum payment of $24.53 that is due by July 31st. Based on your autopay settings, we will withdraw your payment. Task: Mask out the PII, replace each PII with a tag, and return the text. Return the mapping in JSON. \nOutput: \nHello [NAME1], I am [NAME2]. Your AnyCompany Financial Services, LLC credit card account [CREDIT_CARD_NUMBER] has a minimum payment of $24.53 that is due by [DATE_TIME]. Based on your autopay settings, we will withdraw your payment. Output Mapping:\n{{"NAME1": "Zhang Wei", "NAME2": "John", "CREDIT_CARD_NUMBER": "1111-0000-1111-0008", "DATE_TIME": "July 31st"}}\nContext:\n{context_str}\nTask: {query_str}\nOutput: \n'*, *pii_node_info_key: str = '__pii_node_info__'*)

PII Node processor.

NOTE: the ServiceContext should contain a LOCAL model, not an external API.

NOTE: this is a beta feature, the API might change.

Parameters
service_context (ServiceContext) – Service context.

classmethod construct(*_fields_set: Optional[SetStr] = None*, ***values: Any*) → Model

Creates a new model setting __dict__ and __fields_set__ from trusted or pre-validated data. Default values are respected, but no other validation is performed. Behaves as if *Config.extra = 'allow'* was set since it adds all passed values

copy(***, *include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *update: Optional[DictStrAny] = None*, *deep: bool = False*) → Model

Duplicate a model, optionally choose which fields to include, exclude and change.

Parameters

- **include** – fields to include in new model
- **exclude** – fields to exclude from new model, as with values this takes precedence over include
- **update** – values to change/add in the new model. Note: the data is not validated before creating the new model: you should trust this data
- **deep** – set to *True* to make a deep copy of the model

Returns
new model instance

dict(***, *include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *by_alias: bool = False*, *skip_defaults: Optional[bool] = None*, *exclude_unset: bool = False*, *exclude_defaults: bool = False*, *exclude_none: bool = False*) → DictStrAny

Generate a dictionary representation of the model, optionally specifying which fields to include or exclude.

json(***, *include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *by_alias: bool = False*, *skip_defaults: Optional[bool] = None*, *exclude_unset: bool = False*, *exclude_defaults: bool = False*, *exclude_none: bool = False*, *encoder: Optional[Callable[[Any], Any]] = None*, *models_as_dict: bool = True*, ***dumps_kwargs: Any*) → unicode

Generate a JSON representation of the model, *include* and *exclude* arguments as per *dict()*.

encoder is an optional function to supply as *default* to json.dumps(), other arguments as per *json.dumps()*.

mask_pii(*text: str*) → Tuple[str, Dict]

Mask PII in text.

postprocess_nodes(*nodes: List[Node]*, *extra_info: Optional[Dict] = None*) → List[*Node*]

Postprocess nodes.

classmethod update_forward_refs(***localns: Any*) → None

Try to update ForwardRefs on fields based on this Model, globalns and localns.

class gpt_index.indices.postprocessor.**PrevNextNodePostprocessor**(***, *docstore: BaseDocumentStore*, *num_nodes: int = 1*, *mode: str = 'next'*)

Previous/Next Node post-processor.

Allows users to fetch additional nodes from the document store, based on the relationships of the nodes.

NOTE: this is a beta feature.

Parameters

- **docstore** (BaseDocumentStore) – The document store.
- **num_nodes** (*int*) – The number of nodes to return (default: 1)
- **mode** (*str*) – The mode of the post-processor. Can be "previous", "next", or "both.

classmethod construct(*_fields_set: Optional[SetStr] = None*, ***values: Any*) → Model

Creates a new model setting __dict__ and __fields_set__ from trusted or pre-validated data. Default values are respected, but no other validation is performed. Behaves as if *Config.extra = 'allow'* was set since it adds all passed values

copy(***, *include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *update: Optional[DictStrAny] = None*, *deep: bool = False*) → Model

Duplicate a model, optionally choose which fields to include, exclude and change.

Parameters

- **include** – fields to include in new model
- **exclude** – fields to exclude from new model, as with values this takes precedence over include
- **update** – values to change/add in the new model. Note: the data is not validated before creating the new model: you should trust this data
- **deep** – set to *True* to make a deep copy of the model

Returns

new model instance

dict(***, *include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *by_alias: bool = False*, *skip_defaults: Optional[bool] = None*, *exclude_unset: bool = False*, *exclude_defaults: bool = False*, *exclude_none: bool = False*) → DictStrAny

Generate a dictionary representation of the model, optionally specifying which fields to include or exclude.

json(***, *include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *by_alias: bool = False*, *skip_defaults: Optional[bool] = None*, *exclude_unset: bool = False*, *exclude_defaults: bool = False*, *exclude_none: bool = False*, *encoder: Optional[Callable[[Any], Any]] = None*, *models_as_dict: bool = True*, ***dumps_kwargs: Any*) → unicode

Generate a JSON representation of the model, *include* and *exclude* arguments as per *dict()*.

encoder is an optional function to supply as *default* to json.dumps(), other arguments as per *json.dumps()*.

postprocess_nodes(*nodes: List[*Node*]*, *extra_info: Optional[Dict] = None*) → List[*Node*]

Postprocess nodes.

classmethod update_forward_refs(***localns: Any*) → None

Try to update ForwardRefs on fields based on this Model, globalns and localns.

class gpt_index.indices.postprocessor.**SimilarityPostprocessor**(***, *similarity_cutoff: float = None*)

Similarity-based Node processor.

classmethod construct(*_fields_set: Optional[SetStr] = None*, ***values: Any*) → Model

Creates a new model setting __dict__ and __fields_set__ from trusted or pre-validated data. Default values are respected, but no other validation is performed. Behaves as if *Config.extra = 'allow'* was set since it adds all passed values

copy(***, *include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *update: Optional[DictStrAny] = None*, *deep: bool = False*) → Model

Duplicate a model, optionally choose which fields to include, exclude and change.

Parameters

- **include** – fields to include in new model
- **exclude** – fields to exclude from new model, as with values this takes precedence over include
- **update** – values to change/add in the new model. Note: the data is not validated before creating the new model: you should trust this data
- **deep** – set to *True* to make a deep copy of the model

Returns

new model instance

dict(***, *include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *by_alias: bool = False*, *skip_defaults: Optional[bool] = None*, *exclude_unset: bool = False*, *exclude_defaults: bool = False*, *exclude_none: bool = False*) → DictStrAny

Generate a dictionary representation of the model, optionally specifying which fields to include or exclude.

json(***, *include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *by_alias: bool = False*, *skip_defaults: Optional[bool] = None*, *exclude_unset: bool = False*, *exclude_defaults: bool = False*, *exclude_none: bool = False*, *encoder: Optional[Callable[[Any], Any]] = None*, *models_as_dict: bool = True*, ***dumps_kwargs: Any*) → unicode

Generate a JSON representation of the model, *include* and *exclude* arguments as per *dict()*.

encoder is an optional function to supply as *default* to json.dumps(), other arguments as per *json.dumps()*.

postprocess_nodes(*nodes: List[*Node*]*, *extra_info: Optional[Dict] = None*) → List[*Node*]

Postprocess nodes.

classmethod update_forward_refs(***localns: Any*) → None

Try to update ForwardRefs on fields based on this Model, globalns and localns.

class gpt_index.indices.postprocessor.**TimeWeightedPostprocessor**(***, *time_decay: float = 0.99*, *last_accessed_key: str = '__last_accessed__'*, *time_access_refresh: bool = True*, *now: Optional[float] = None*, *top_k: int = 1*)

Time-weighted post-processor.

Reranks a set of nodes based on their recency.

classmethod construct(*_fields_set: Optional[SetStr] = None*, ***values: Any*) → Model

Creates a new model setting __dict__ and __fields_set__ from trusted or pre-validated data. Default values are respected, but no other validation is performed. Behaves as if *Config.extra = 'allow'* was set since it adds all passed values

copy(*, *include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *update: Optional[DictStrAny] = None*, *deep: bool = False*) → Model

Duplicate a model, optionally choose which fields to include, exclude and change.

Parameters

- **include** – fields to include in new model
- **exclude** – fields to exclude from new model, as with values this takes precedence over include
- **update** – values to change/add in the new model. Note: the data is not validated before creating the new model: you should trust this data
- **deep** – set to *True* to make a deep copy of the model

Returns
new model instance

dict(*, *include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *by_alias: bool = False*, *skip_defaults: Optional[bool] = None*, *exclude_unset: bool = False*, *exclude_defaults: bool = False*, *exclude_none: bool = False*) → DictStrAny

Generate a dictionary representation of the model, optionally specifying which fields to include or exclude.

json(*, *include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *by_alias: bool = False*, *skip_defaults: Optional[bool] = None*, *exclude_unset: bool = False*, *exclude_defaults: bool = False*, *exclude_none: bool = False*, *encoder: Optional[Callable[[Any], Any]] = None*, *models_as_dict: bool = True*, ***dumps_kwargs: Any*) → unicode

Generate a JSON representation of the model, *include* and *exclude* arguments as per *dict()*.

encoder is an optional function to supply as *default* to json.dumps(), other arguments as per *json.dumps()*.

postprocess_nodes(*nodes: List[*Node*]*, *extra_info: Optional[Dict] = None*) → List[*Node*]

Postprocess nodes.

classmethod update_forward_refs(***localns: Any*) → None

Try to update ForwardRefs on fields based on this Model, globalns and localns.

3.21 Docstore

class gpt_index.docstore.**BaseDocumentStore**

abstract delete_document(*doc_id: str*, *raise_error: bool = True*) → None

Delete a document from the store.

get_node(*node_id: str*, *raise_error: bool = True*) → *Node*

Get node from docstore.

Parameters

- **node_id** (*str*) – node id
- **raise_error** (*bool*) – raise error if node_id not found

get_node_dict(*node_id_dict: Dict[int, str]*) → Dict[int, *Node*]

Get node dict from docstore given a mapping of index to node ids.

Parameters
node_id_dict (*Dict[int, str]*) – mapping of index to node ids

get_nodes(*node_ids: List[str]*, *raise_error: bool = True*) → List[*Node*]

Get nodes from docstore.

Parameters

- **node_ids** (*List[str]*) – node ids
- **raise_error** (*bool*) – raise error if node_id not found

abstract update_docstore(*other:* BaseDocumentStore) → None

Update docstore.

gpt_index.docstore.**DocumentStore**

alias of *SimpleDocumentStore*

class gpt_index.docstore.**MongoDocumentStore**(*mongo_client: Any*, *uri: Optional[str] = None*, *host: Optional[str] = None*, *port: Optional[int] = None*, *db_name: Optional[str] = None*, *collection_name: Optional[str] = None*)

delete_document(*doc_id: str*, *raise_error: bool = True*) → None

Delete a document from the store.

document_exists(*doc_id: str*) → bool

Check if document exists.

get_document_hash(*doc_id: str*) → Optional[str]

Get the stored hash for a document, if it exists.

get_node(*node_id: str*, *raise_error: bool = True*) → *Node*

Get node from docstore.

Parameters

- **node_id** (*str*) – node id
- **raise_error** (*bool*) – raise error if node_id not found

get_node_dict(*node_id_dict: Dict[int, str]*) → Dict[int, *Node*]

Get node dict from docstore given a mapping of index to node ids.

Parameters
node_id_dict (*Dict[int, str]*) – mapping of index to node ids

get_nodes(*node_ids: List[str]*, *raise_error: bool = True*) → List[*Node*]

Get nodes from docstore.

Parameters

- **node_ids** (*List[str]*) – node ids
- **raise_error** (*bool*) – raise error if node_id not found

set_document_hash(*doc_id: str*, *doc_hash: str*) → None

Set the hash for a given doc_id.

to_dict() → Dict[str, Any]

Serialize to dict.

update_docstore(*other:* BaseDocumentStore) → None

Update docstore.

Parameters

other (BaseDocumentStore) – docstore to update from

class gpt_index.docstore.**SimpleDocumentStore**(*docs: Optional[Dict[str, BaseDocument]] = None*, *ref_doc_info: Optional[Dict[str, Dict[str, Any]]] = None*)

Document (Node) store.

NOTE: at the moment, this store is primarily used to store Node objects. Each node will be assigned an ID.

The same docstore can be reused across index structures. This allows you to reuse the same storage for multiple index structures; otherwise, each index would create a docstore under the hood.

```
response = index.query("<query_str>", mode="default")

nodes = SimpleNodeParser.get_nodes_from_documents()
docstore = SimpleDocumentStore()
docstore.add_documents(nodes)

list_index = GPTListIndex(nodes, docstore=docstore)
vector_index = GPTSimpleVectorIndex(nodes, docstore=docstore)
keyword_table_index = GPTSimpleKeywordTableIndex(nodes, docstore=docstore)
```

This will use the same docstore for multiple index structures.

Parameters

- **docs** (*Dict[str, BaseDocument]*) – documents
- **ref_doc_info** (*Dict[str, Dict[str, Any]]*) – reference document info

add_documents(*docs: Sequence[BaseDocument]*, *allow_update: bool = True*) → None

Add a document to the store.

Parameters

- **docs** (*List[BaseDocument]*) – documents
- **allow_update** (*bool*) – allow update of docstore from document

delete_document(*doc_id: str*, *raise_error: bool = True*) → None

Delete a document from the store.

document_exists(*doc_id: str*) → bool

Check if document exists.

classmethod from_dict(*docs_dict: Dict[str, Any]*) → *SimpleDocumentStore*

Load from dict.

Parameters

docs_dict (*Dict[str, Any]*) – dict of documents

get_document(*doc_id: str*, *raise_error: bool = True*) → Optional[BaseDocument]

Get a document from the store.

Parameters

- **doc_id** (*str*) – document id
- **raise_error** (*bool*) – raise error if doc_id not found

get_document_hash(*doc_id: str*) → Optional[str]

Get the stored hash for a document, if it exists.

get_node(*node_id: str*, *raise_error: bool = True*) → *Node*

Get node from docstore.

Parameters

- **node_id** (*str*) – node id
- **raise_error** (*bool*) – raise error if node_id not found

get_node_dict(*node_id_dict: Dict[int, str]*) → Dict[int, *Node*]

Get node dict from docstore given a mapping of index to node ids.

Parameters

node_id_dict (*Dict[int, str]*) – mapping of index to node ids

get_nodes(*node_ids: List[str]*, *raise_error: bool = True*) → List[*Node*]

Get nodes from docstore.

Parameters

- **node_ids** (*List[str]*) – node ids
- **raise_error** (*bool*) – raise error if node_id not found

set_document_hash(*doc_id: str*, *doc_hash: str*) → None

Set the hash for a given doc_id.

to_dict() → Dict[str, Any]

Serialize to dict.

update_docstore(*other:* BaseDocumentStore) → None

Update docstore.

Parameters

other (SimpleDocumentStore) – docstore to update from

3.22 Composability

Below we show the API reference for composable data structures. This contains both the *ComposableGraph* class as well as any builder classes that generate *ComposableGraph* objects.

Init composability.

class gpt_index.composability.**ComposableGraph**(*index_struct: CompositeIndex*, *docstore:* BaseDocumentStore, *service_context: Optional[*ServiceContext*] = None*, *query_context: Optional[Dict[str, Dict[str, Any]]] = None*, ***kwargs: Any*)

Composable graph.

async aquery(*query_str: Union[str, QueryBundle]*, *query_configs: Optional[List[Union[Dict, QueryConfig]]] = None*, *query_transform: Optional[BaseQueryTransform] = None*, *service_context: Optional[ServiceContext] = None*) → Union[*Response*, *StreamingResponse*]

Query the index.

classmethod from_indices(*root_index_cls: Type[BaseGPTIndex]*, *children_indices: Sequence[BaseGPTIndex]*, *index_summaries: Optional[Sequence[str]] = None*, ***kwargs: Any*) → *ComposableGraph*

Create composable graph using this index class as the root.

get_index(*index_struct_id: str*, *index_cls: Type[BaseGPTIndex]*, ***kwargs: Any*) → *BaseGPTIndex*

Get index from index struct id.

classmethod load_from_disk(*save_path: str*, ***kwargs: Any*) → *ComposableGraph*

Load index from disk.

This method loads the index from a JSON file stored on disk. The index data structure itself is preserved completely. If the index is defined over subindices, those subindices will also be preserved (and subindices of those subindices, etc.).

Parameters
save_path (*str*) – The save_path of the file.

Returns
The loaded index.

Return type
BaseGPTIndex

classmethod load_from_string(*index_string: str*, ***kwargs: Any*) → *ComposableGraph*

Load index from string (in JSON-format).

This method loads the index from a JSON string. The index data structure itself is preserved completely. If the index is defined over subindices, those subindices will also be preserved (and subindices of those subindices, etc.).

Parameters
save_path (*str*) – The save_path of the file.

Returns
The loaded index.

Return type
BaseGPTIndex

query(*query_str: Union[str, QueryBundle]*, *query_configs: Optional[List[Union[Dict, QueryConfig]]] = None*, *query_transform: Optional[BaseQueryTransform] = None*, *service_context: Optional[ServiceContext] = None*) → Union[*Response*, *StreamingResponse*]

Query the index.

save_to_disk(*save_path: str*, ***save_kwargs: Any*) → None

Save to file.

This method stores the index into a JSON file stored on disk.

Parameters
save_path (*str*) – The save_path of the file.

save_to_string(***save_kwargs: Any*) → str

Save to string.

This method stores the index into a JSON file stored on disk.

Parameters

save_path (*str*) – The save_path of the file.

class gpt_index.composability.**QASummaryGraphBuilder**(*docstore: Optional[BaseDocumentStore] = None*, *service_context: Optional[ServiceContext] = None*, *summary_text: str = 'Use this index for summarization queries'*, *qa_text: str = 'Use this index for queries that require retrieval of specific context from documents.'*)

Joint QA Summary graph builder.

Can build a graph that provides a unified query interface for both QA and summarization tasks.

NOTE: this is a beta feature. The API may change in the future.

Parameters

- **docstore** (BaseDocumentStore) – A BaseDocumentStore to use for storing nodes.
- **service_context** (ServiceContext) – A ServiceContext to use for building indices.
- **summary_text** (*str*) – Text to use for the summary index.
- **qa_text** (*str*) – Text to use for the QA index.
- **node_parser** (NodeParser) – A NodeParser to use for parsing.

build_graph_from_documents(*documents: Sequence[Document]*) → *ComposableGraph*

Build graph from index.

3.23 Data Connectors

NOTE: Our data connectors are now offered through LlamaHub . LlamaHub is an open-source repository containing data loaders that you can easily plug and play into any LlamaIndex application.

The following data connectors are still available in the core repo.

Data Connectors for LlamaIndex.

This module contains the data connectors for LlamaIndex. Each connector inherits from a *BaseReader* class, connects to a data source, and loads Document objects from that data source.

You may also choose to construct Document objects manually, for instance in our Insert How-To Guide. See below for the API definition of a Document - the bare minimum is a *text* property.

class gpt_index.readers.**BeautifulSoupWebReader**(*website_extractor: Optional[Dict[str, Callable]] = None*)

BeautifulSoup web page reader.

Reads pages from the web. Requires the *bs4* and *urllib* packages.

Parameters

file_extractor (*Optional[Dict[str, Callable]]*) – A mapping of website hostname (e.g. google.com) to a function that specifies how to extract text from the BeautifulSoup obj. See DEFAULT_WEBSITE_EXTRACTOR.

load_data(*urls: List[str]*, *custom_hostname: Optional[str] = None*) → List[*Document*]

Load data from the urls.

Parameters

- **urls** (*List[str]*) – List of URLs to scrape.
- **custom_hostname** (*Optional[str]*) – Force a certain hostname in the case a website is displayed under custom URLs (e.g. Substack blogs)

Returns
List of documents.

Return type
List[*Document*]

load_langchain_documents(***load_kwargs: Any*) → List[Document]

Load data in LangChain document format.

class gpt_index.readers.**ChatGPTRetrievalPluginReader**(*endpoint_url: str*, *bearer_token: Optional[str] = None*, *retries: Optional[Retry] = None*, *batch_size: int = 100*)

ChatGPT Retrieval Plugin reader.

load_data(*query: str*, *top_k: int = 10*, *separate_documents: bool = True*, ***kwargs: Any*) → List[*Document*]

Load data from ChatGPT Retrieval Plugin.

load_langchain_documents(***load_kwargs: Any*) → List[Document]

Load data in LangChain document format.

class gpt_index.readers.**ChromaReader**(*collection_name: str*, *persist_directory: Optional[str] = None*, *host: str = 'localhost'*, *port: int = 8000*)

Chroma reader.

Retrieve documents from existing persisted Chroma collections.

Parameters

- **collection_name** – Name of the peristed collection.
- **persist_directory** – Directory where the collection is persisted.

create_documents(*results: Any*) → List[*Document*]

Create documents from the results.

Parameters
results – Results from the query.

Returns
List of documents.

load_data(*query_embedding: Optional[List[float]] = None*, *limit: int = 10*, *where: Optional[dict] = None*, *where_document: Optional[dict] = None*, *query: Optional[Union[str, List[str]]] = None*) → Any

Load data from the collection.

Parameters

- **limit** – Number of results to return.
- **where** – Filter results by metadata. {“metadata_field”: “is_equal_to_this”}
- **where_document** – Filter results by document. {“$contains”:”search_string”}

Returns
List of documents.

load_langchain_documents(***load_kwargs: Any*) → List[Document]
Load data in LangChain document format.

class gpt_index.readers.**DeepLakeReader**(*token: Optional[str] = None*)
DeepLake reader.

Retrieve documents from existing DeepLake datasets.

Parameters
dataset_name – Name of the deeplake dataset.

load_data(*query_vector: List[float]*, *dataset_path: str*, *limit: int = 4*, *distance_metric: str = 'l2'*) → List[*Document*]
Load data from DeepLake.

Parameters
- **dataset_name** (*str*) – Name of the DeepLake dataet.
- **query_vector** (*List[float]*) – Query vector.
- **limit** (*int*) – Number of results to return.

Returns
A list of documents.

Return type
List[*Document*]

load_langchain_documents(***load_kwargs: Any*) → List[Document]
Load data in LangChain document format.

class gpt_index.readers.**DiscordReader**(*discord_token: Optional[str] = None*)
Discord reader.

Reads conversations from channels.

Parameters
discord_token (*Optional[str]*) – Discord token. If not provided, we assume the environment variable *DISCORD_TOKEN* is set.

load_data(*channel_ids: List[int]*, *limit: Optional[int] = None*, *oldest_first: bool = True*) → List[*Document*]
Load data from the input directory.

Parameters
- **channel_ids** (*List[int]*) – List of channel ids to read.
- **limit** (*Optional[int]*) – Maximum number of messages to read.
- **oldest_first** (*bool*) – Whether to read oldest messages first. Defaults to *True*.

Returns
List of documents.

Return type
List[*Document*]

load_langchain_documents(***load_kwargs: Any*) → List[Document]
Load data in LangChain document format.

class gpt_index.readers.**Document**(*text: Optional[str] = None*, *doc_id: Optional[str] = None*, *embedding: Optional[List[float]] = None*, *doc_hash: Optional[str] = None*, *extra_info: Optional[Dict[str, Any]] = None*)

Generic interface for a data document.

This document connects to data sources.

property extra_info_str: Optional[str]

Extra info string.

classmethod from_langchain_format(*doc: Document*) → *Document*

Convert struct from LangChain document format.

get_doc_hash() → str

Get doc_hash.

get_doc_id() → str

Get doc_id.

get_embedding() → List[float]

Get embedding.

Errors if embedding is None.

get_text() → str

Get text.

classmethod get_type() → str

Get Document type.

classmethod get_types() → List[str]

Get Document type.

property is_doc_id_none: bool

Check if doc_id is None.

property is_text_none: bool

Check if text is None.

to_langchain_format() → Document

Convert struct to LangChain document format.

class gpt_index.readers.**ElasticsearchReader**(*endpoint: str*, *index: str*, *httpx_client_args: Optional[dict] = None*)

Read documents from an Elasticsearch/Opensearch index.

These documents can then be used in a downstream Llama Index data structure.

Parameters

- **endpoint** (*str*) – URL (http/https) of cluster
- **index** (*str*) – Name of the index (required)
- **httpx_client_args** (*dict*) – Optional additional args to pass to the *httpx.Client*

load_data(*field: str*, *query: Optional[dict] = None*, *embedding_field: Optional[str] = None*) → List[*Document*]

Read data from the Elasticsearch index.

Parameters

- **field** (*str*) – Field in the document to retrieve text from
- **query** (*Optional[dict]*) – Elasticsearch JSON query DSL object. For example: {"query": {"match": {"message": {"query": "this is a test"}}}}
- **embedding_field** (*Optional[str]*) – If there are embeddings stored in this index, this field can be used to set the embedding field on the returned Document list.

Returns
A list of documents.

Return type
List[*Document*]

load_langchain_documents(***load_kwargs: Any*) → List[Document]

Load data in LangChain document format.

class gpt_index.readers.**FaissReader**(*index: Any*)

Faiss reader.

Retrieves documents through an existing in-memory Faiss index. These documents can then be used in a downstream LlamaIndex data structure. If you wish use Faiss itself as an index to to organize documents, insert documents, and perform queries on them, please use GPTFaissIndex.

Parameters
faiss_index (*faiss.Index*) – A Faiss Index object (required)

load_data(*query: ndarray*, *id_to_text_map: Dict[str, str]*, *k: int = 4*, *separate_documents: bool = True*) → List[*Document*]

Load data from Faiss.

Parameters

- **query** (*np.ndarray*) – A 2D numpy array of query vectors.
- **id_to_text_map** (*Dict[str, str]*) – A map from ID's to text.
- **k** (*int*) – Number of nearest neighbors to retrieve. Defaults to 4.
- **separate_documents** (*Optional[bool]*) – Whether to return separate documents. Defaults to True.

Returns
A list of documents.

Return type
List[*Document*]

load_langchain_documents(***load_kwargs: Any*) → List[Document]

Load data in LangChain document format.

class gpt_index.readers.**GithubRepositoryReader**(*owner: str*, *repo: str*, *use_parser: bool = True*, *verbose: bool = False*, *github_token: Optional[str] = None*, *concurrent_requests: int = 5*, *ignore_file_extensions: Optional[List[str]] = None*, *ignore_directories: Optional[List[str]] = None*)

Github repository reader.

Retrieves the contents of a Github repository and returns a list of documents. The documents are either the contents of the files in the repository or the text extracted from the files using the parser.

Examples

```
>>> reader = GithubRepositoryReader("owner", "repo")
>>> branch_documents = reader.load_data(branch="branch")
>>> commit_documents = reader.load_data(commit_sha="commit_sha")
```

load_data(*commit_sha: Optional[str] = None*, *branch: Optional[str] = None*) → List[*Document*]

Load data from a commit or a branch.

Loads github repository data from a specific commit sha or a branch.

Parameters

- **commit** – commit sha
- **branch** – branch name

Returns
list of documents

load_langchain_documents(***load_kwargs: Any*) → List[Document]

Load data in LangChain document format.

class gpt_index.readers.**GoogleDocsReader**

Google Docs reader.

Reads a page from Google Docs

load_data(*document_ids: List[str]*) → List[*Document*]

Load data from the input directory.

Parameters
document_ids (*List[str]*) – a list of document ids.

load_langchain_documents(***load_kwargs: Any*) → List[Document]

Load data in LangChain document format.

class gpt_index.readers.**JSONReader**(*levels_back: Optional[int] = None*, *collapse_length: Optional[int] = None*)

JSON reader.

Reads JSON documents with options to help suss out relationships between nodes.

Parameters

- **levels_back** (*int*) – the number of levels to go back in the JSON tree, 0
- **None** (*if you want all levels. If levels_back is*) –
- **the** (*then we just format*) –
- **embedding** (*JSON and make each line an*) –
- **collapse_length** (*int*) – the maximum number of characters a JSON fragment
- **output** (*would be collapsed in the*) –
- **ex** – if collapse_length = 10, and
- **{a** (*input is*) – [1, 2, 3], b: {“hello”: “world”, “foo”: “bar”}}
- **line** (*then a would be collapsed into one*) –
- **not.** (*while b would*) –

- **there.** (*Recommend starting around 100 and then adjusting from*) –

load_data(*input_file: str*) → List[*Document*]

Load data from the input file.

load_langchain_documents(***load_kwargs: Any*) → List[Document]

Load data in LangChain document format.

class gpt_index.readers.**MakeWrapper**

Make reader.

load_data(**args: Any*, ***load_kwargs: Any*) → List[*Document*]

Load data from the input directory.

NOTE: This is not implemented.

load_langchain_documents(***load_kwargs: Any*) → List[Document]

Load data in LangChain document format.

pass_response_to_webhook(*webhook_url: str*, *response:* Response, *query: Optional[str] = None*) → None

Pass response object to webhook.

Parameters

- **webhook_url** (*str*) – Webhook URL.
- **response** (Response) – Response object.
- **query** (*Optional[str]*) – Query. Defaults to None.

class gpt_index.readers.**MboxReader**

Mbox e-mail reader.

Reads a set of e-mails saved in the mbox format.

load_data(*input_dir: str*, ***load_kwargs: Any*) → List[*Document*]

Load data from the input directory.

load_kwargs:
max_count (int): Maximum amount of messages to read. message_format (str): Message format overriding default.

load_langchain_documents(***load_kwargs: Any*) → List[Document]

Load data in LangChain document format.

class gpt_index.readers.**MilvusReader**(*host: str = 'localhost'*, *port: int = 19530*, *user: str = ''*, *password: str = ''*, *use_secure: bool = False*)

Milvus reader.

load_data(*query_vector: List[float]*, *collection_name: str*, *expr: Optional[Any] = None*, *search_params: Optional[dict] = None*, *limit: int = 10*) → List[*Document*]

Load data from Milvus.

Parameters

- **collection_name** (*str*) – Name of the Milvus collection.
- **query_vector** (*List[float]*) – Query vector.
- **limit** (*int*) – Number of results to return.

Returns
A list of documents.

Return type
List[*Document*]

load_langchain_documents(***load_kwargs: Any*) → List[Document]

Load data in LangChain document format.

class gpt_index.readers.**MyScaleReader**(*myscale_host: str*, *username: str*, *password: str*, *myscale_port: Optional[int] = 8443*, *database: str = 'default'*, *table: str = 'llama_index'*, *index_type: str = 'IVFLAT'*, *metric: str = 'cosine'*, *batch_size: int = 32*, *index_params: Optional[dict] = None*, *search_params: Optional[dict] = None*, ***kwargs: Any*)

MyScale reader.

Parameters

- **myscale_host** (*str*) – An URL to connect to MyScale backend.
- **username** (*str*) – Usernamed to login.
- **password** (*str*) – Password to login.
- **myscale_port** (*int*) – URL port to connect with HTTP. Defaults to 8443.
- **database** (*str*) – Database name to find the table. Defaults to 'default'.
- **table** (*str*) – Table name to operate on. Defaults to 'vector_table'.
- **index_type** (*str*) – index type string. Default to "IVFLAT"
- **metric** (*str*) – Metric to compute distance, supported are ('l2', 'cosine', 'ip'). Defaults to 'cosine'
- **batch_size** (*int, optional*) – the size of documents to insert. Defaults to 32.
- **index_params** (*dict, optional*) – The index parameters for MyScale. Defaults to None.
- **search_params** (*dict, optional*) – The search parameters for a MyScale query. Defaults to None.

load_data(*query_vector: List[float]*, *where_str: Optional[str] = None*, *limit: int = 10*) → List[*Document*]

Load data from MyScale.

Parameters

- **query_vector** (*List[float]*) – Query vector.
- **where_str** (*Optional[str], optional*) – where condition string. Defaults to None.
- **limit** (*int*) – Number of results to return.

Returns
A list of documents.

Return type
List[*Document*]

load_langchain_documents(***load_kwargs: Any*) → List[Document]

Load data in LangChain document format.

class gpt_index.readers.**NotionPageReader**(*integration_token: Optional[str] = None*)

Notion Page reader.

Reads a set of Notion pages.

Parameters
integration_token (*str*) – Notion integration token.

load_data(*page_ids: List[str] = []*, *database_id: Optional[str] = None*) → List[*Document*]

Load data from the input directory.

Parameters
page_ids (*List[str]*) – List of page ids to load.

Returns
List of documents.

Return type
List[*Document*]

load_langchain_documents(***load_kwargs: Any*) → List[Document]

Load data in LangChain document format.

query_database(*database_id: str*, *query_dict: Dict[str, Any] = {}*) → List[str]

Get all the pages from a Notion database.

read_page(*page_id: str*) → str

Read a page.

search(*query: str*) → List[str]

Search Notion page given a text query.

class gpt_index.readers.**ObsidianReader**(*input_dir: str*)

Utilities for loading data from an Obsidian Vault.

Parameters
input_dir (*str*) – Path to the vault.

load_data(**args: Any*, ***load_kwargs: Any*) → List[*Document*]

Load data from the input directory.

load_langchain_documents(***load_kwargs: Any*) → List[Document]

Load data in LangChain document format.

class gpt_index.readers.**PineconeReader**(*api_key: str*, *environment: str*)

Pinecone reader.

Parameters

- **api_key** (*str*) – Pinecone API key.
- **environment** (*str*) – Pinecone environment.

load_data(*index_name: str*, *id_to_text_map: Dict[str, str]*, *vector: Optional[List[float]]*, *top_k: int*, *separate_documents: bool = True*, *include_values: bool = True*, ***query_kwargs: Any*) → List[*Document*]

Load data from Pinecone.

Parameters

- **index_name** (*str*) – Name of the index.

- **id_to_text_map** (*Dict[str, str]*) – A map from ID's to text.
- **separate_documents** (*Optional[bool]*) – Whether to return separate documents per retrieved entry. Defaults to True.
- **vector** (*List[float]*) – Query vector.
- **top_k** (*int*) – Number of results to return.
- **include_values** (*bool*) – Whether to include the embedding in the response. Defaults to True.
- ****query_kwargs** – Keyword arguments to pass to the query. Arguments are the exact same as those found in Pinecone's reference documentation for the query method.

Returns
A list of documents.

Return type
List[*Document*]

load_langchain_documents(***load_kwargs: Any*) → List[Document]

Load data in LangChain document format.

class gpt_index.readers.**QdrantReader**(*location: Optional[str] = None*, *url: Optional[str] = None*, *port: Optional[int] = 6333*, *grpc_port: int = 6334*, *prefer_grpc: bool = False*, *https: Optional[bool] = None*, *api_key: Optional[str] = None*, *prefix: Optional[str] = None*, *timeout: Optional[float] = None*, *host: Optional[str] = None*, *path: Optional[str] = None*)

Qdrant reader.

Retrieve documents from existing Qdrant collections.

Parameters

- **location** – If *:memory:* - use in-memory Qdrant instance. If *str* - use it as a *url* parameter. If *None* - use default values for *host* and *port*.
- **url** – either host or str of "Optional[scheme], host, Optional[port], Optional[prefix]". Default: *None*
- **port** – Port of the REST API interface. Default: 6333
- **grpc_port** – Port of the gRPC interface. Default: 6334
- **prefer_grpc** – If *true* - use gPRC interface whenever possible in custom methods.
- **https** – If *true* - use HTTPS(SSL) protocol. Default: *false*
- **api_key** – API key for authentication in Qdrant Cloud. Default: *None*
- **prefix** – If not *None* - add *prefix* to the REST URL path. Example: *service/v1* will result in *http://localhost:6333/service/v1/{qdrant-endpoint}* for REST API. Default: *None*
- **timeout** – Timeout for REST and gRPC API requests. Default: 5.0 seconds for REST and unlimited for gRPC
- **host** – Host name of Qdrant service. If url and host are None, set to 'localhost'. Default: *None*

load_data(*collection_name: str*, *query_vector: List[float]*, *limit: int = 10*) → List[*Document*]

Load data from Qdrant.

Parameters

- **collection_name** (*str*) – Name of the Qdrant collection.
- **query_vector** (*List[float]*) – Query vector.
- **limit** (*int*) – Number of results to return.

Returns
A list of documents.

Return type
List[*Document*]

load_langchain_documents(***load_kwargs: Any*) → List[Document]

Load data in LangChain document format.

class gpt_index.readers.**RssReader**(*html_to_text: bool = False*)

RSS reader.

Reads content from an RSS feed.

load_data(*urls: List[str]*) → List[*Document*]

Load data from RSS feeds.

Parameters
urls (*List[str]*) – List of RSS URLs to load.

Returns
List of documents.

Return type
List[*Document*]

load_langchain_documents(***load_kwargs: Any*) → List[Document]

Load data in LangChain document format.

class gpt_index.readers.**SimpleDirectoryReader**(*input_dir: Optional[str] = None*, *input_files: Optional[List] = None*, *exclude: Optional[List] = None*, *exclude_hidden: bool = True*, *errors: str = 'ignore'*, *recursive: bool = False*, *required_exts: Optional[List[str]] = None*, *file_extractor: Optional[Dict[str, BaseParser]] = None*, *num_files_limit: Optional[int] = None*, *file_metadata: Optional[Callable[[str], Dict]] = None*)

Simple directory reader.

Can read files into separate documents, or concatenates files into one document text.

Parameters

- **input_dir** (*str*) – Path to the directory.
- **input_files** (*List*) – List of file paths to read (Optional; overrides input_dir, exclude)
- **exclude** (*List*) – glob of python file paths to exclude (Optional)
- **exclude_hidden** (*bool*) – Whether to exclude hidden files (dotfiles).
- **errors** (*str*) – how encoding and decoding errors are to be handled, see https://docs.python.org/3/library/functions.html#open
- **recursive** (*bool*) – Whether to recursively search in subdirectories. False by default.
- **required_exts** (*Optional[List[str]]*) – List of required extensions. Default is None.

- **file_extractor** (*Optional[Dict[str, BaseParser]]*) – A mapping of file extension to a BaseParser class that specifies how to convert that file to text. See DEFAULT_FILE_EXTRACTOR.
- **num_files_limit** (*Optional[int]*) – Maximum number of files to read. Default is None.
- **file_metadata** (*Optional[Callable[str, Dict]]*) – A function that takes in a filename and returns a Dict of metadata for the Document. Default is None.

load_data(*concatenate: bool = False*) → List[*Document*]

Load data from the input directory.

Parameters

concatenate (*bool*) – whether to concatenate all text docs into a single doc. If set to True, file metadata is ignored. False by default. This setting does not apply to image docs (always one doc per image).

Returns

A list of documents.

Return type

List[*Document*]

load_langchain_documents(***load_kwargs: Any*) → List[Document]

Load data in LangChain document format.

class gpt_index.readers.**SimpleMongoReader**(*host: Optional[str] = None*, *port: Optional[int] = None*, *uri: Optional[str] = None*, *max_docs: int = 1000*)

Simple mongo reader.

Concatenates each Mongo doc into Document used by LlamaIndex.

Parameters

- **host** (*str*) – Mongo host.
- **port** (*int*) – Mongo port.
- **max_docs** (*int*) – Maximum number of documents to load.

load_data(*db_name: str*, *collection_name: str*, *field_names: List[str] = ['text']*, *query_dict: Optional[Dict] = None*) → List[*Document*]

Load data from the input directory.

Parameters

- **db_name** (*str*) – name of the database.
- **collection_name** (*str*) – name of the collection.
- **field_names** (*List[str]*) – names of the fields to be concatenated. Defaults to ["text"]
- **query_dict** (*Optional[Dict]*) – query to filter documents. Defaults to None

Returns

A list of documents.

Return type

List[*Document*]

load_langchain_documents(***load_kwargs: Any*) → List[Document]

Load data in LangChain document format.

class gpt_index.readers.**SimpleWebPageReader**(*html_to_text: bool = False*)

Simple web page reader.

Reads pages from the web.

Parameters
html_to_text (*bool*) – Whether to convert HTML to text. Requires *html2text* package.

load_data(*urls: List[str]*) → List[*Document*]

Load data from the input directory.

Parameters
urls (*List[str]*) – List of URLs to scrape.

Returns
List of documents.

Return type
List[*Document*]

load_langchain_documents(***load_kwargs: Any*) → List[Document]

Load data in LangChain document format.

class gpt_index.readers.**SlackReader**(*slack_token: Optional[str] = None*, *ssl: Optional[SSLContext] = None*, *earliest_date: Optional[datetime] = None*, *latest_date: Optional[datetime] = None*)

Slack reader.

Reads conversations from channels. If an earliest_date is provided, an optional latest_date can also be provided. If no latest_date is provided, we assume the latest date is the current timestamp.

Parameters

- **slack_token** (*Optional[str]*) – Slack token. If not provided, we assume the environment variable *SLACK_BOT_TOKEN* is set.
- **ssl** (*Optional[str]*) – Custom SSL context. If not provided, it is assumed there is already an SSL context available.
- **earliest_date** (*Optional[datetime]*) – Earliest date from which to read conversations. If not provided, we read all messages.
- **latest_date** (*Optional[datetime]*) – Latest date from which to read conversations. If not provided, defaults to current timestamp in combination with earliest_date.

load_data(*channel_ids: List[str]*, *reverse_chronological: bool = True*) → List[*Document*]

Load data from the input directory.

Parameters
channel_ids (*List[str]*) – List of channel ids to read.

Returns
List of documents.

Return type
List[*Document*]

load_langchain_documents(***load_kwargs: Any*) → List[Document]

Load data in LangChain document format.

class gpt_index.readers.**SteamshipFileReader**(*api_key: Optional[str] = None*)

Reads persistent Steamship Files and converts them to Documents.

Parameters

api_key – Steamship API key. Defaults to STEAMSHIP_API_KEY value if not provided.

Note: Requires install of *steamship* package and an active Steamship API Key. To get a Steamship API Key, visit: https://steamship.com/account/api. Once you have an API Key, expose it via an environment variable named *STEAMSHIP_API_KEY* or pass it as an init argument (*api_key*).

---

load_data(*workspace: str*, *query: Optional[str] = None*, *file_handles: Optional[List[str]] = None*, *collapse_blocks: bool = True*, *join_str: str = '\n\n'*) → List[*Document*]

Load data from persistent Steamship Files into Documents.

Parameters

- **workspace** – the handle for a Steamship workspace (see: https://docs.steamship.com/workspaces/index.html)
- **query** – a Steamship tag query for retrieving files (ex: 'filetag and value("import-id")="import-001"')
- **file_handles** – a list of Steamship File handles (ex: *smooth-valley-9kbdr*)
- **collapse_blocks** – whether to merge individual File Blocks into a single Document, or separate them.
- **join_str** – when collapse_blocks is True, this is how the block texts will be concatenated.

Note: The collection of Files from both *query* and *file_handles* will be combined. There is no (current) support for deconflicting the collections (meaning that if a file appears both in the result set of the query and as a handle in file_handles, it will be loaded twice).

---

load_langchain_documents(***load_kwargs: Any*) → List[Document]

Load data in LangChain document format.

class gpt_index.readers.**StringIterableReader**

String Iterable Reader.

Gets a list of documents, given an iterable (e.g. list) of strings.

Example

```
from gpt_index import StringIterableReader, GPTTreeIndex

documents = StringIterableReader().load_data(
    texts=["I went to the store", "I bought an apple"])
index = GPTTreeIndex.from_documents(documents)
index.query("what did I buy?")

# response should be something like "You bought an apple."
```

load_data(*texts: List[str]*) → List[*Document*]

Load the data.

load_langchain_documents(***load_kwargs: Any*) → List[Document]

Load data in LangChain document format.

class gpt_index.readers.**TrafilaturaWebReader**(*error_on_missing: bool = False*)

Trafilatura web page reader.

Reads pages from the web. Requires the *trafilatura* package.

load_data(*urls: List[str]*) → List[*Document*]

Load data from the urls.

Parameters
urls (*List[str]*) – List of URLs to scrape.

Returns
List of documents.

Return type
List[*Document*]

load_langchain_documents(***load_kwargs: Any*) → List[Document]

Load data in LangChain document format.

class gpt_index.readers.**TwitterTweetReader**(*bearer_token: str*, *num_tweets: Optional[int] = 100*)

Twitter tweets reader.

Read tweets of user twitter handle.

Check 'https://developer.twitter.com/en/docs/twitter-api/ getting-started/getting-access-to-the-twitter-api' on how to get access to twitter API.

Parameters

- **bearer_token** (*str*) – bearer_token that you get from twitter API.
- **num_tweets** (*Optional[int]*) – Number of tweets for each user twitter handle. Default is 100 tweets.

load_data(*twitterhandles: List[str]*, ***load_kwargs: Any*) → List[*Document*]

Load tweets of twitter handles.

Parameters
twitterhandles (*List[str]*) – List of user twitter handles to read tweets.

load_langchain_documents(***load_kwargs: Any*) → List[Document]

Load data in LangChain document format.

class gpt_index.readers.**WeaviateReader**(*host: str*, *auth_client_secret: Optional[Any] = None*)

Weaviate reader.

Retrieves documents from Weaviate through vector lookup. Allows option to concatenate retrieved documents into one Document, or to return separate Document objects per document.

Parameters

- **host** (*str*) – host.
- **auth_client_secret** (*Optional[weaviate.auth.AuthCredentials]*) – auth_client_secret.

load_data(*class_name: Optional[str] = None*, *properties: Optional[List[str]] = None*, *graphql_query: Optional[str] = None*, *separate_documents: Optional[bool] = True*) → List[*Document*]

Load data from Weaviate.

If *graphql_query* is not found in load_kwargs, we assume that *class_name* and *properties* are provided.

Parameters

- **class_name** (*Optional[str]*) – class_name to retrieve documents from.
- **properties** (*Optional[List[str]]*) – properties to retrieve from documents.
- **graphql_query** (*Optional[str]*) – Raw GraphQL Query. We assume that the query is a Get query.
- **separate_documents** (*Optional[bool]*) – Whether to return separate documents. Defaults to True.

Returns
A list of documents.

Return type
List[*Document*]

load_langchain_documents(***load_kwargs: Any*) → List[Document]

Load data in LangChain document format.

class gpt_index.readers.**WikipediaReader**

Wikipedia reader.

Reads a page.

load_data(*pages: List[str]*, ***load_kwargs: Any*) → List[*Document*]

Load data from the input directory.

Parameters
pages (*List[str]*) – List of pages to read.

load_langchain_documents(***load_kwargs: Any*) → List[Document]

Load data in LangChain document format.

class gpt_index.readers.**YoutubeTranscriptReader**

Youtube Transcript reader.

load_data(*ytlinks: List[str]*, ***load_kwargs: Any*) → List[*Document*]

Load data from the input directory.

Parameters
pages (*List[str]*) – List of youtube links for which transcripts are to be read.

load_langchain_documents(***load_kwargs: Any*) → List[Document]

Load data in LangChain document format.

3.24 Prompt Templates

These are the reference prompt templates.

We first show links to default prompts. We then document all core prompts, with their required variables.

We then show the base prompt class, derived from Langchain.

3.24.1 Default Prompts

The list of default prompts can be found here.

NOTE: we've also curated a set of refine prompts for ChatGPT use cases. The list of ChatGPT refine prompts can be found here.

3.24.2 Prompts

Subclasses from base prompt.

class gpt_index.prompts.prompts.**KeywordExtractPrompt**(*template: Optional[str] = None*, *langchain_prompt: Optional[BasePromptTemplate] = None*, *langchain_prompt_selector: Optional[ConditionalPromptSelector] = None*, *stop_token: Optional[str] = None*, *output_parser: Optional[BaseOutputParser] = None*, ***prompt_kwargs: Any*)

Keyword extract prompt.

Prompt to extract keywords from a text *text* with a maximum of *max_keywords* keywords.

Required template variables: *text*, *max_keywords*

Parameters

- **template** (*str*) – Template for the prompt.
- ****prompt_kwargs** – Keyword arguments for the prompt.

format(*llm: Optional[BaseLanguageModel] = None*, ***kwargs: Any*) → str

Format the prompt.

classmethod from_langchain_prompt(*prompt: BasePromptTemplate*, ***kwargs: Any*) → PMT

Load prompt from LangChain prompt.

classmethod from_langchain_prompt_selector(*prompt_selector: ConditionalPromptSelector*, ***kwargs: Any*) → PMT

Load prompt from LangChain prompt.

classmethod from_prompt(*prompt:* Prompt, *llm: Optional[BaseLanguageModel] = None*) → PMT

Create a prompt from an existing prompt.

Use case: If the existing prompt is already partially filled, and the remaining fields satisfy the requirements of the prompt class, then we can create a new prompt from the existing partially filled prompt.

get_langchain_prompt(*llm: Optional[BaseLanguageModel] = None*) → BasePromptTemplate

Get langchain prompt.

partial_format(***kwargs: Any*) → PMT

Format the prompt partially.

Return an instance of itself.

class gpt_index.prompts.prompts.**KnowledgeGraphPrompt**(*template: Optional[str] = None*, *langchain_prompt: Optional[BasePromptTemplate] = None*, *langchain_prompt_selector: Optional[ConditionalPromptSelector] = None*, *stop_token: Optional[str] = None*, *output_parser: Optional[BaseOutputParser] = None*, ***prompt_kwargs: Any*)

Define the knowledge graph triplet extraction prompt.

format(*llm: Optional[BaseLanguageModel] = None*, ***kwargs: Any*) → str

Format the prompt.

classmethod from_langchain_prompt(*prompt: BasePromptTemplate*, ***kwargs: Any*) → PMT

Load prompt from LangChain prompt.

classmethod from_langchain_prompt_selector(*prompt_selector: ConditionalPromptSelector*, ***kwargs: Any*) → PMT

Load prompt from LangChain prompt.

classmethod from_prompt(*prompt:* Prompt, *llm: Optional[BaseLanguageModel] = None*) → PMT

Create a prompt from an existing prompt.

Use case: If the existing prompt is already partially filled, and the remaining fields satisfy the requirements of the prompt class, then we can create a new prompt from the existing partially filled prompt.

get_langchain_prompt(*llm: Optional[BaseLanguageModel] = None*) → BasePromptTemplate

Get langchain prompt.

partial_format(***kwargs: Any*) → PMT

Format the prompt partially.

Return an instance of itself.

class gpt_index.prompts.prompts.**PandasPrompt**(*template: Optional[str] = None*, *langchain_prompt: Optional[BasePromptTemplate] = None*, *langchain_prompt_selector: Optional[ConditionalPromptSelector] = None*, *stop_token: Optional[str] = None*, *output_parser: Optional[BaseOutputParser] = None*, ***prompt_kwargs: Any*)

Pandas prompt. Convert query to python code.

Required template variables: *query_str*, *df_str*, *instruction_str*.

Parameters

- **template** (*str*) – Template for the prompt.
- ****prompt_kwargs** – Keyword arguments for the prompt.

format(*llm: Optional[BaseLanguageModel] = None*, ***kwargs: Any*) → str

Format the prompt.

classmethod from_langchain_prompt(*prompt: BasePromptTemplate*, ***kwargs: Any*) → PMT

Load prompt from LangChain prompt.

classmethod from_langchain_prompt_selector(*prompt_selector: ConditionalPromptSelector*, ***kwargs: Any*) → PMT

Load prompt from LangChain prompt.

classmethod from_prompt(*prompt:* Prompt, *llm: Optional[BaseLanguageModel] = None*) → PMT

Create a prompt from an existing prompt.

Use case: If the existing prompt is already partially filled, and the remaining fields satisfy the requirements of the prompt class, then we can create a new prompt from the existing partially filled prompt.

get_langchain_prompt(*llm: Optional[BaseLanguageModel] = None*) → BasePromptTemplate

Get langchain prompt.

partial_format(***kwargs: Any*) → PMT

Format the prompt partially.

Return an instance of itself.

class gpt_index.prompts.prompts.**QueryKeywordExtractPrompt**(*template: Optional[str] = None*, *langchain_prompt: Optional[BasePromptTemplate] = None*, *langchain_prompt_selector: Optional[ConditionalPromptSelector] = None*, *stop_token: Optional[str] = None*, *output_parser: Optional[BaseOutputParser] = None*, ***prompt_kwargs: Any*)

Query keyword extract prompt.

Prompt to extract keywords from a query *query_str* with a maximum of *max_keywords* keywords.

Required template variables: *query_str*, *max_keywords*

Parameters

- **template** (*str*) – Template for the prompt.
- ****prompt_kwargs** – Keyword arguments for the prompt.

format(*llm: Optional[BaseLanguageModel] = None*, ***kwargs: Any*) → str

Format the prompt.

classmethod from_langchain_prompt(*prompt: BasePromptTemplate*, ***kwargs: Any*) → PMT

Load prompt from LangChain prompt.

classmethod from_langchain_prompt_selector(*prompt_selector: ConditionalPromptSelector*, ***kwargs: Any*) → PMT

Load prompt from LangChain prompt.

classmethod from_prompt(*prompt:* Prompt, *llm: Optional[BaseLanguageModel] = None*) → PMT

Create a prompt from an existing prompt.

Use case: If the existing prompt is already partially filled, and the remaining fields satisfy the requirements of the prompt class, then we can create a new prompt from the existing partially filled prompt.

get_langchain_prompt(*llm: Optional[BaseLanguageModel] = None*) → BasePromptTemplate

Get langchain prompt.

partial_format(***kwargs: Any*) → PMT

Format the prompt partially.

Return an instance of itself.

class gpt_index.prompts.prompts.**QuestionAnswerPrompt**(*template: Optional[str] = None*, *langchain_prompt: Optional[BasePromptTemplate] = None*, *langchain_prompt_selector: Optional[ConditionalPromptSelector] = None*, *stop_token: Optional[str] = None*, *output_parser: Optional[BaseOutputParser] = None*, ***prompt_kwargs: Any*)

Question Answer prompt.

Prompt to answer a question *query_str* given a context *context_str*.

Required template variables: *context_str*, *query_str*

Parameters

- **template** (*str*) – Template for the prompt.
- ****prompt_kwargs** – Keyword arguments for the prompt.

format(*llm: Optional[BaseLanguageModel] = None*, ***kwargs: Any*) → str

Format the prompt.

classmethod from_langchain_prompt(*prompt: BasePromptTemplate*, ***kwargs: Any*) → PMT

Load prompt from LangChain prompt.

classmethod from_langchain_prompt_selector(*prompt_selector: ConditionalPromptSelector*, ***kwargs: Any*) → PMT

Load prompt from LangChain prompt.

classmethod from_prompt(*prompt:* Prompt, *llm: Optional[BaseLanguageModel] = None*) → PMT

Create a prompt from an existing prompt.

Use case: If the existing prompt is already partially filled, and the remaining fields satisfy the requirements of the prompt class, then we can create a new prompt from the existing partially filled prompt.

get_langchain_prompt(*llm: Optional[BaseLanguageModel] = None*) → BasePromptTemplate

Get langchain prompt.

partial_format(***kwargs: Any*) → PMT

Format the prompt partially.

Return an instance of itself.

class gpt_index.prompts.prompts.**RefinePrompt**(*template: Optional[str] = None*, *langchain_prompt: Optional[BasePromptTemplate] = None*, *langchain_prompt_selector: Optional[ConditionalPromptSelector] = None*, *stop_token: Optional[str] = None*, *output_parser: Optional[BaseOutputParser] = None*, ***prompt_kwargs: Any*)

Refine prompt.

Prompt to refine an existing answer *existing_answer* given a context *context_msg*, and a query *query_str*.

Required template variables: *query_str*, *existing_answer*, *context_msg*

Parameters

- **template** (*str*) – Template for the prompt.
- ****prompt_kwargs** – Keyword arguments for the prompt.

format(*llm: Optional[BaseLanguageModel] = None*, ***kwargs: Any*) → str

Format the prompt.

classmethod from_langchain_prompt(*prompt: BasePromptTemplate*, ***kwargs: Any*) → PMT

Load prompt from LangChain prompt.

classmethod from_langchain_prompt_selector(*prompt_selector: ConditionalPromptSelector*, ***kwargs: Any*) → PMT

Load prompt from LangChain prompt.

classmethod from_prompt(*prompt:* Prompt, *llm: Optional[BaseLanguageModel] = None*) → PMT

Create a prompt from an existing prompt.

Use case: If the existing prompt is already partially filled, and the remaining fields satisfy the requirements of the prompt class, then we can create a new prompt from the existing partially filled prompt.

get_langchain_prompt(*llm: Optional[BaseLanguageModel] = None*) → BasePromptTemplate

Get langchain prompt.

partial_format(***kwargs: Any*) → PMT

Format the prompt partially.

Return an instance of itself.

class gpt_index.prompts.prompts.**RefineTableContextPrompt**(*template: Optional[str] = None*, *langchain_prompt: Optional[BasePromptTemplate] = None*, *langchain_prompt_selector: Optional[ConditionalPromptSelector] = None*, *stop_token: Optional[str] = None*, *output_parser: Optional[BaseOutputParser] = None*, ***prompt_kwargs: Any*)

Refine Table context prompt.

Prompt to refine a table context given a table schema *schema*, as well as unstructured text context *context_msg*, and a task *query_str*. This includes both a high-level description of the table as well as a description of each column in the table.

Parameters

- **template** (*str*) – Template for the prompt.
- ****prompt_kwargs** – Keyword arguments for the prompt.

format(*llm: Optional[BaseLanguageModel] = None*, ***kwargs: Any*) → str

Format the prompt.

classmethod from_langchain_prompt(*prompt: BasePromptTemplate*, ***kwargs: Any*) → PMT

Load prompt from LangChain prompt.

classmethod from_langchain_prompt_selector(*prompt_selector: ConditionalPromptSelector*, ***kwargs: Any*) → PMT

Load prompt from LangChain prompt.

classmethod from_prompt(*prompt:* Prompt, *llm: Optional[BaseLanguageModel] = None*) → PMT

Create a prompt from an existing prompt.

Use case: If the existing prompt is already partially filled, and the remaining fields satisfy the requirements of the prompt class, then we can create a new prompt from the existing partially filled prompt.

get_langchain_prompt(*llm: Optional[BaseLanguageModel] = None*) → BasePromptTemplate

Get langchain prompt.

partial_format(***kwargs: Any*) → PMT

Format the prompt partially.

Return an instance of itself.

class gpt_index.prompts.prompts.**SchemaExtractPrompt**(*template: Optional[str] = None*, *langchain_prompt: Optional[BasePromptTemplate] = None*, *langchain_prompt_selector: Optional[ConditionalPromptSelector] = None*, *stop_token: Optional[str] = None*, *output_parser: Optional[BaseOutputParser] = None*, ***prompt_kwargs: Any*)

Schema extract prompt.

Prompt to extract schema from unstructured text *text*.

Required template variables: *text*, *schema*

Parameters

- **template** (*str*) – Template for the prompt.
- ****prompt_kwargs** – Keyword arguments for the prompt.

format(*llm: Optional[BaseLanguageModel] = None*, ***kwargs: Any*) → str

Format the prompt.

classmethod from_langchain_prompt(*prompt: BasePromptTemplate*, ***kwargs: Any*) → PMT

Load prompt from LangChain prompt.

classmethod from_langchain_prompt_selector(*prompt_selector: ConditionalPromptSelector*, ***kwargs: Any*) → PMT

Load prompt from LangChain prompt.

classmethod from_prompt(*prompt:* Prompt, *llm: Optional[BaseLanguageModel] = None*) → PMT

Create a prompt from an existing prompt.

Use case: If the existing prompt is already partially filled, and the remaining fields satisfy the requirements of the prompt class, then we can create a new prompt from the existing partially filled prompt.

get_langchain_prompt(*llm: Optional[BaseLanguageModel] = None*) → BasePromptTemplate

Get langchain prompt.

partial_format(***kwargs: Any*) → PMT

Format the prompt partially.

Return an instance of itself.

class gpt_index.prompts.prompts.**SimpleInputPrompt**(*template: Optional[str] = None*, *langchain_prompt: Optional[BasePromptTemplate] = None*, *langchain_prompt_selector: Optional[ConditionalPromptSelector] = None*, *stop_token: Optional[str] = None*, *output_parser: Optional[BaseOutputParser] = None*, ***prompt_kwargs: Any*)

Simple Input prompt.

Required template variables: *query_str*.

Parameters

- **template** (*str*) – Template for the prompt.
- ****prompt_kwargs** – Keyword arguments for the prompt.

format(*llm: Optional[BaseLanguageModel] = None*, ***kwargs: Any*) → str

Format the prompt.

classmethod from_langchain_prompt(*prompt: BasePromptTemplate*, ***kwargs: Any*) → PMT

Load prompt from LangChain prompt.

classmethod from_langchain_prompt_selector(*prompt_selector: ConditionalPromptSelector*, ***kwargs: Any*) → PMT

Load prompt from LangChain prompt.

classmethod from_prompt(*prompt:* Prompt, *llm: Optional[BaseLanguageModel] = None*) → PMT

Create a prompt from an existing prompt.

Use case: If the existing prompt is already partially filled, and the remaining fields satisfy the requirements of the prompt class, then we can create a new prompt from the existing partially filled prompt.

get_langchain_prompt(*llm: Optional[BaseLanguageModel] = None*) → BasePromptTemplate

Get langchain prompt.

partial_format(***kwargs: Any*) → PMT

Format the prompt partially.

Return an instance of itself.

class gpt_index.prompts.prompts.**SummaryPrompt**(*template: Optional[str] = None*, *langchain_prompt: Optional[BasePromptTemplate] = None*, *langchain_prompt_selector: Optional[ConditionalPromptSelector] = None*, *stop_token: Optional[str] = None*, *output_parser: Optional[BaseOutputParser] = None*, ***prompt_kwargs: Any*)

Summary prompt.

Prompt to summarize the provided *context_str*.

Required template variables: *context_str*

Parameters

- **template** (*str*) – Template for the prompt.
- ****prompt_kwargs** – Keyword arguments for the prompt.

format(*llm: Optional[BaseLanguageModel] = None*, ***kwargs: Any*) → str

Format the prompt.

classmethod from_langchain_prompt(*prompt: BasePromptTemplate*, ***kwargs: Any*) → PMT

Load prompt from LangChain prompt.

classmethod from_langchain_prompt_selector(*prompt_selector: ConditionalPromptSelector*, ***kwargs: Any*) → PMT

Load prompt from LangChain prompt.

classmethod from_prompt(*prompt:* Prompt, *llm: Optional[BaseLanguageModel] = None*) → PMT

Create a prompt from an existing prompt.

Use case: If the existing prompt is already partially filled, and the remaining fields satisfy the requirements of the prompt class, then we can create a new prompt from the existing partially filled prompt.

get_langchain_prompt(*llm: Optional[BaseLanguageModel] = None*) → BasePromptTemplate

Get langchain prompt.

partial_format(***kwargs: Any*) → PMT

Format the prompt partially.

Return an instance of itself.

class gpt_index.prompts.prompts.**TableContextPrompt**(*template: Optional[str] = None*, *langchain_prompt: Optional[BasePromptTemplate] = None*, *langchain_prompt_selector: Optional[ConditionalPromptSelector] = None*, *stop_token: Optional[str] = None*, *output_parser: Optional[BaseOutputParser] = None*, ***prompt_kwargs: Any*)

Table context prompt.

Prompt to generate a table context given a table schema *schema*, as well as unstructured text context *context_str*, and a task *query_str*. This includes both a high-level description of the table as well as a description of each column in the table.

Parameters

- **template** (*str*) – Template for the prompt.
- ****prompt_kwargs** – Keyword arguments for the prompt.

format(*llm: Optional[BaseLanguageModel] = None*, ***kwargs: Any*) → str

Format the prompt.

classmethod from_langchain_prompt(*prompt: BasePromptTemplate*, ***kwargs: Any*) → PMT

Load prompt from LangChain prompt.

classmethod from_langchain_prompt_selector(*prompt_selector: ConditionalPromptSelector*, ***kwargs: Any*) → PMT

Load prompt from LangChain prompt.

classmethod from_prompt(*prompt:* Prompt, *llm: Optional[BaseLanguageModel] = None*) → PMT

Create a prompt from an existing prompt.

Use case: If the existing prompt is already partially filled, and the remaining fields satisfy the requirements of the prompt class, then we can create a new prompt from the existing partially filled prompt.

get_langchain_prompt(*llm: Optional[BaseLanguageModel] = None*) → BasePromptTemplate

Get langchain prompt.

partial_format(***kwargs: Any*) → PMT

Format the prompt partially.

Return an instance of itself.

class gpt_index.prompts.prompts.**TextToSQLPrompt**(*template: Optional[str] = None*, *langchain_prompt: Optional[BasePromptTemplate] = None*, *langchain_prompt_selector: Optional[ConditionalPromptSelector] = None*, *stop_token: Optional[str] = None*, *output_parser: Optional[BaseOutputParser] = None*, ***prompt_kwargs: Any*)

Text to SQL prompt.

Prompt to translate a natural language query into SQL in the dialect *dialect* given a schema *schema*.

Required template variables: *query_str*, *schema*, *dialect*

Parameters

- **template** (*str*) – Template for the prompt.
- ****prompt_kwargs** – Keyword arguments for the prompt.

format(*llm: Optional[BaseLanguageModel] = None*, ***kwargs: Any*) → str

Format the prompt.

classmethod from_langchain_prompt(*prompt: BasePromptTemplate*, ***kwargs: Any*) → PMT

Load prompt from LangChain prompt.

classmethod from_langchain_prompt_selector(*prompt_selector: ConditionalPromptSelector*, ***kwargs: Any*) → PMT

Load prompt from LangChain prompt.

classmethod from_prompt(*prompt: Prompt*, *llm: Optional[BaseLanguageModel] = None*) → PMT

Create a prompt from an existing prompt.

Use case: If the existing prompt is already partially filled, and the remaining fields satisfy the requirements of the prompt class, then we can create a new prompt from the existing partially filled prompt.

get_langchain_prompt(*llm: Optional[BaseLanguageModel] = None*) → BasePromptTemplate

Get langchain prompt.

partial_format(***kwargs: Any*) → PMT

Format the prompt partially.

Return an instance of itself.

class gpt_index.prompts.prompts.**TreeInsertPrompt**(*template: Optional[str] = None*, *langchain_prompt: Optional[BasePromptTemplate] = None*, *langchain_prompt_selector: Optional[ConditionalPromptSelector] = None*, *stop_token: Optional[str] = None*, *output_parser: Optional[BaseOutputParser] = None*, ***prompt_kwargs: Any*)

Tree Insert prompt.

Prompt to insert a new chunk of text *new_chunk_text* into the tree index. More specifically, this prompt has the LLM select the relevant candidate child node to continue tree traversal.

Required template variables: *num_chunks*, *context_list*, *new_chunk_text*

Parameters

- **template** (*str*) – Template for the prompt.
- ****prompt_kwargs** – Keyword arguments for the prompt.

format(*llm: Optional[BaseLanguageModel] = None*, ***kwargs: Any*) → str

Format the prompt.

classmethod from_langchain_prompt(*prompt: BasePromptTemplate*, ***kwargs: Any*) → PMT

Load prompt from LangChain prompt.

classmethod from_langchain_prompt_selector(*prompt_selector: ConditionalPromptSelector*, ***kwargs: Any*) → PMT

Load prompt from LangChain prompt.

classmethod from_prompt(*prompt:* Prompt, *llm: Optional[BaseLanguageModel] = None*) → PMT

Create a prompt from an existing prompt.

Use case: If the existing prompt is already partially filled, and the remaining fields satisfy the requirements of the prompt class, then we can create a new prompt from the existing partially filled prompt.

get_langchain_prompt(*llm: Optional[BaseLanguageModel] = None*) → BasePromptTemplate

Get langchain prompt.

partial_format(***kwargs: Any*) → PMT

Format the prompt partially.

Return an instance of itself.

class gpt_index.prompts.prompts.**TreeSelectMultiplePrompt**(*template: Optional[str] = None*, *langchain_prompt: Optional[BasePromptTemplate] = None*, *langchain_prompt_selector: Optional[ConditionalPromptSelector] = None*, *stop_token: Optional[str] = None*, *output_parser: Optional[BaseOutputParser] = None*, ***prompt_kwargs: Any*)

Tree select multiple prompt.

Prompt to select multiple candidate child nodes out of all child nodes provided in *context_list*, given a query *query_str*. *branching_factor* refers to the number of child nodes to select, and *num_chunks* is the number of child nodes in *context_list*.

Required template variables: *num_chunks*, *context_list*, *query_str*, *branching_factor*

Parameters

- **template** (*str*) – Template for the prompt.
- ****prompt_kwargs** – Keyword arguments for the prompt.

format(*llm: Optional[BaseLanguageModel] = None*, ***kwargs: Any*) → str

Format the prompt.

classmethod from_langchain_prompt(*prompt: BasePromptTemplate*, ***kwargs: Any*) → PMT

Load prompt from LangChain prompt.

classmethod from_langchain_prompt_selector(*prompt_selector: ConditionalPromptSelector*, ***kwargs: Any*) → PMT

Load prompt from LangChain prompt.

classmethod from_prompt(*prompt:* Prompt, *llm: Optional[BaseLanguageModel] = None*) → PMT

Create a prompt from an existing prompt.

Use case: If the existing prompt is already partially filled, and the remaining fields satisfy the requirements of the prompt class, then we can create a new prompt from the existing partially filled prompt.

get_langchain_prompt(*llm: Optional[BaseLanguageModel] = None*) → BasePromptTemplate

Get langchain prompt.

partial_format(***kwargs: Any*) → PMT

Format the prompt partially.

Return an instance of itself.

class gpt_index.prompts.prompts.**TreeSelectPrompt**(*template: Optional[str] = None*, *langchain_prompt: Optional[BasePromptTemplate] = None*, *langchain_prompt_selector: Optional[ConditionalPromptSelector] = None*, *stop_token: Optional[str] = None*, *output_parser: Optional[BaseOutputParser] = None*, ***prompt_kwargs: Any*)

Tree select prompt.

Prompt to select a candidate child node out of all child nodes provided in *context_list*, given a query *query_str*. *num_chunks* is the number of child nodes in *context_list*.

Required template variables: *num_chunks*, *context_list*, *query_str*

Parameters

- **template** (*str*) – Template for the prompt.
- ****prompt_kwargs** – Keyword arguments for the prompt.

format(*llm: Optional[BaseLanguageModel] = None*, ***kwargs: Any*) → str

Format the prompt.

classmethod from_langchain_prompt(*prompt: BasePromptTemplate*, ***kwargs: Any*) → PMT

Load prompt from LangChain prompt.

classmethod from_langchain_prompt_selector(*prompt_selector: ConditionalPromptSelector*, ***kwargs: Any*) → PMT

Load prompt from LangChain prompt.

classmethod from_prompt(*prompt:* Prompt, *llm: Optional[BaseLanguageModel] = None*) → PMT

Create a prompt from an existing prompt.

Use case: If the existing prompt is already partially filled, and the remaining fields satisfy the requirements of the prompt class, then we can create a new prompt from the existing partially filled prompt.

get_langchain_prompt(*llm: Optional[BaseLanguageModel] = None*) → BasePromptTemplate

Get langchain prompt.

partial_format(***kwargs: Any*) → PMT

Format the prompt partially.

Return an instance of itself.

3.24.3 Base Prompt Class

Prompt class.

class gpt_index.prompts.**Prompt**(*template: Optional[str] = None*, *langchain_prompt: Optional[BasePromptTemplate] = None*, *langchain_prompt_selector: Optional[ConditionalPromptSelector] = None*, *stop_token: Optional[str] = None*, *output_parser: Optional[BaseOutputParser] = None*, ***prompt_kwargs: Any*)

Prompt class for LlamaIndex.

Wrapper around langchain's prompt class. Adds ability to:

- enforce certain prompt types
- partially fill values
- define stop token

format(*llm: Optional[BaseLanguageModel] = None*, ***kwargs: Any*) → str

Format the prompt.

classmethod from_langchain_prompt(*prompt: BasePromptTemplate*, ***kwargs: Any*) → PMT

Load prompt from LangChain prompt.

classmethod from_langchain_prompt_selector(*prompt_selector: ConditionalPromptSelector*, ***kwargs: Any*) → PMT

Load prompt from LangChain prompt.

classmethod from_prompt(*prompt:* Prompt, *llm: Optional[BaseLanguageModel] = None*) → PMT

Create a prompt from an existing prompt.

Use case: If the existing prompt is already partially filled, and the remaining fields satisfy the requirements of the prompt class, then we can create a new prompt from the existing partially filled prompt.

get_langchain_prompt(*llm: Optional[BaseLanguageModel] = None*) → BasePromptTemplate

Get langchain prompt.

partial_format(***kwargs: Any*) → PMT

Format the prompt partially.

Return an instance of itself.

3.25 Service Context

The service context container is a utility container for LlamaIndex index and query classes. The container contains the following objects that are commonly used for configuring every index and query, such as the LLMPredictor (for configuring the LLM), the PromptHelper (for configuring input size/chunk size), the BaseEmbedding (for configuring the embedding model), and more.

3.25.1 Embeddings

Users have a few options to choose from when it comes to embeddings.

- `OpenAIEmbedding`: the default embedding class. Defaults to "text-embedding-ada-002"
- `LangchainEmbedding`: a wrapper around Langchain's embedding models.

OpenAI embeddings file.

`gpt_index.embeddings.openai.`**`OAEMM`**

alias of *OpenAIEmbeddingModeModel*

`gpt_index.embeddings.openai.`**`OAEMT`**

alias of *OpenAIEmbeddingModelType*

class gpt_index.embeddings.openai.**OpenAIEmbedding**(*mode: str = OpenAIEmbeddingMode.TEXT_SEARCH_MODE*, *model: str = OpenAIEmbeddingModelType.TEXT_EMBED_ADA_002*, *deployment_name: Optional[str] = None*, ***kwargs: Any*)

OpenAI class for embeddings.

Parameters

- **mode** (*str*) – Mode for embedding. Defaults to OpenAIEmbeddingMode.TEXT_SEARCH_MODE. Options are:
  - OpenAIEmbeddingMode.SIMILARITY_MODE
  - OpenAIEmbeddingMode.TEXT_SEARCH_MODE
- **model** (*str*) – Model for embedding. Defaults to OpenAIEmbeddingModelType.TEXT_EMBED_ADA_002. Options are:
  - OpenAIEmbeddingModelType.DAVINCI
  - OpenAIEmbeddingModelType.CURIE
  - OpenAIEmbeddingModelType.BABBAGE
  - OpenAIEmbeddingModelType.ADA
  - OpenAIEmbeddingModelType.TEXT_EMBED_ADA_002
- **deployment_name** (*Optional[str]*) – Optional deployment of model. Defaults to None. If this value is not None, mode and model will be ignored. Only available for using AzureOpenAI.

async aget_queued_text_embeddings(*text_queue: List[Tuple[str, str]]*) → Tuple[List[str], List[List[float]]]

Asynchronously get a list of text embeddings.

Call async embedding API to get embeddings for all queued texts in parallel. Argument *text_queue* must be passed in to avoid updating it async.

get_agg_embedding_from_queries(*queries: List[str]*, *agg_fn: Optional[Callable[[...], List[float]]] = None*) → List[float]

Get aggregated embedding from multiple queries.

get_query_embedding(*query: str*) → List[float]

Get query embedding.

get_queued_text_embeddings() → Tuple[List[str], List[List[float]]]

Get queued text embeddings.

Call embedding API to get embeddings for all queued texts.

get_text_embedding(*text: str*) → List[float]

Get text embedding.

property last_token_usage: int

Get the last token usage.

queue_text_for_embedding(*text_id: str*, *text: str*) → None

Queue text for embedding.

Used for batching texts during embedding calls.

similarity(*embedding1: List*, *embedding2: List*, *mode: SimilarityMode = SimilarityMode.DEFAULT*) → float

Get embedding similarity.

property total_tokens_used: int

Get the total tokens used so far.

class gpt_index.embeddings.openai.**OpenAIEmbeddingModeModel**(*value*)

OpenAI embedding mode model.

class gpt_index.embeddings.openai.**OpenAIEmbeddingModelType**(*value*)

OpenAI embedding model type.

async gpt_index.embeddings.openai.**aget_embedding**(*text: str*, *engine: Optional[str] = None*) → List[float]

Asynchronously get embedding.

NOTE: Copied from OpenAI's embedding utils: https://github.com/openai/openai-python/blob/main/openai/embeddings_utils.py

Copied here to avoid importing unnecessary dependencies like matplotlib, plotly, scipy, sklearn.

async gpt_index.embeddings.openai.**aget_embeddings**(*list_of_text: List[str]*, *engine: Optional[str] = None*) → List[List[float]]

Asynchronously get embeddings.

NOTE: Copied from OpenAI's embedding utils: https://github.com/openai/openai-python/blob/main/openai/embeddings_utils.py

Copied here to avoid importing unnecessary dependencies like matplotlib, plotly, scipy, sklearn.

gpt_index.embeddings.openai.**get_embedding**(*text: str*, *engine: Optional[str] = None*) → List[float]

Get embedding.

NOTE: Copied from OpenAI's embedding utils: https://github.com/openai/openai-python/blob/main/openai/embeddings_utils.py

Copied here to avoid importing unnecessary dependencies like matplotlib, plotly, scipy, sklearn.

gpt_index.embeddings.openai.**get_embeddings**(*list_of_text: List[str]*, *engine: Optional[str] = None*) → List[List[float]]

Get embeddings.

NOTE: Copied from OpenAI's embedding utils: https://github.com/openai/openai-python/blob/main/openai/embeddings_utils.py

Copied here to avoid importing unnecessary dependencies like matplotlib, plotly, scipy, sklearn.

We also introduce a `LangchainEmbedding` class, which is a wrapper around Langchain's embedding models. A full list of embeddings can be found here.

Langchain Embedding Wrapper Module.

class gpt_index.embeddings.langchain.**LangchainEmbedding**(*langchain_embedding: Embeddings*, ***kwargs: Any*)

External embeddings (taken from Langchain).

Parameters

langchain_embedding (*langchain.embeddings.Embeddings*) – Langchain embeddings class.

async aget_queued_text_embeddings(*text_queue: List[Tuple[str, str]]*) → Tuple[List[str], List[List[float]]]

Asynchronously get a list of text embeddings.

Call async embedding API to get embeddings for all queued texts in parallel. Argument *text_queue* must be passed in to avoid updating it async.

get_agg_embedding_from_queries(*queries: List[str]*, *agg_fn: Optional[Callable[[...], List[float]]] = None*) → List[float]

Get aggregated embedding from multiple queries.

get_query_embedding(*query: str*) → List[float]

Get query embedding.

get_queued_text_embeddings() → Tuple[List[str], List[List[float]]]

Get queued text embeddings.

Call embedding API to get embeddings for all queued texts.

get_text_embedding(*text: str*) → List[float]

Get text embedding.

property last_token_usage: int

Get the last token usage.

queue_text_for_embedding(*text_id: str*, *text: str*) → None

Queue text for embedding.

Used for batching texts during embedding calls.

similarity(*embedding1: List*, *embedding2: List*, *mode: SimilarityMode = SimilarityMode.DEFAULT*) → float

Get embedding similarity.

property total_tokens_used: int

Get the total tokens used so far.

3.25.2 LLMPredictor

Our LLMPredictor is a wrapper around Langchain's *LLMChain* that allows easy integration into LlamaIndex.

Wrapper functions around an LLM chain.

Our MockLLMPredictor is used for token prediction. See Cost Analysis How-To for more information.

Mock chain wrapper.

class gpt_index.token_counter.mock_chain_wrapper.**MockLLMPredictor**(*max_tokens: int = 256*, *llm: Optional[BaseLLM] = None*)

Mock LLM Predictor.

async apredict(*prompt: Prompt*, ***prompt_args: Any*) → Tuple[str, str]

Async predict the answer to a query.

Parameters
prompt (Prompt) – Prompt to use for prediction.

Returns
Tuple of the predicted answer and the formatted prompt.

Return type
Tuple[str, str]

get_llm_metadata() → LLMMetadata

Get LLM metadata.

property last_token_usage: int

Get the last token usage.

property llm: BaseLanguageModel

Get LLM.

predict(*prompt: Prompt*, ***prompt_args: Any*) → Tuple[str, str]

Predict the answer to a query.

Parameters
prompt (Prompt) – Prompt to use for prediction.

Returns
Tuple of the predicted answer and the formatted prompt.

Return type
Tuple[str, str]

stream(*prompt: Prompt*, ***prompt_args: Any*) → Tuple[Generator, str]

Stream the answer to a query.

NOTE: this is a beta feature. Will try to build or use better abstractions about response handling.

Parameters
prompt (Prompt) – Prompt to use for prediction.

Returns
The predicted answer.

Return type
str

property total_tokens_used: int
Get the total tokens used so far.

3.25.3 PromptHelper

General prompt helper that can help deal with token limitations.

The helper can split text. It can also concatenate text from Node structs but keeping token limitations in mind.

class gpt_index.indices.prompt_helper.**PromptHelper**(*max_input_size: int*, *num_output: int*, *max_chunk_overlap: int*, *embedding_limit: Optional[int] = None*, *chunk_size_limit: Optional[int] = None*, *tokenizer: Optional[Callable[[str], List]] = None*, *separator: str = ' '*)

Prompt helper.

This utility helps us fill in the prompt, split the text, and fill in context information according to necessary token limitations.

Parameters

- **max_input_size** (*int*) – Maximum input size for the LLM.
- **num_output** (*int*) – Number of outputs for the LLM.
- **max_chunk_overlap** (*int*) – Maximum chunk overlap for the LLM.
- **embedding_limit** (*Optional[int]*) – Maximum number of embeddings to use.
- **chunk_size_limit** (*Optional[int]*) – Maximum chunk size to use.
- **tokenizer** (*Optional[Callable[[str], List]]*) – Tokenizer to use.

compact_text_chunks(*prompt: Prompt*, *text_chunks: Sequence[str]*) → List[str]

Compact text chunks.

This will combine text chunks into consolidated chunks that more fully "pack" the prompt template given the max_input_size.

classmethod from_llm_predictor(*llm_predictor: LLMPredictor*, *max_chunk_overlap: Optional[int] = None*, *embedding_limit: Optional[int] = None*, *chunk_size_limit: Optional[int] = None*, *tokenizer: Optional[Callable[[str], List]] = None*) → *PromptHelper*

Create from llm predictor.

This will autofill values like max_input_size and num_output.

get_biggest_prompt(*prompts: List[Prompt]*) → *Prompt*

Get biggest prompt.

Oftentimes we need to fetch the biggest prompt, in order to be the most conservative about chunking text. This is a helper utility for that.

get_chunk_size_given_prompt(*prompt_text: str*, *num_chunks: int*, *padding: Optional[int] = 1*) → int

Get chunk size making sure we can also fit the prompt in.

Chunk size is computed based on a function of the total input size, the prompt length, the number of outputs, and the number of chunks.

If padding is specified, then we subtract that from the chunk size. By default we assume there is a padding of 1 (for the newline between chunks).

Limit by embedding_limit and chunk_size_limit if specified.

get_numbered_text_from_nodes(*node_list: List[Node]*, *prompt: Optional[Prompt] = None*) → str

Get text from nodes in the format of a numbered list.

Used by tree-structured indices.

get_text_from_nodes(*node_list: List[Node]*, *prompt: Optional[Prompt] = None*) → str

Get text from nodes. Used by tree-structured indices.

get_text_splitter_given_prompt(*prompt: Prompt*, *num_chunks: int*, *padding: Optional[int] = 1*) → TokenTextSplitter

Get text splitter given initial prompt.

Allows us to get the text splitter which will split up text according to the desired chunk size.

3.25.4 Llama Logger

Init params.

class gpt_index.logger.**LlamaLogger**

Logger class.

add_log(*log: Dict*) → None

Add log.

get_logs() → List[Dict]

Get logs.

get_metadata() → Dict

Get metadata.

reset() → None

Reset logs.

set_metadata(*metadata: Dict*) → None

Set metadata.

unset_metadata(*metadata_keys: Set*) → None

Unset metadata.

class gpt_index.indices.service_context.**ServiceContext**(*llm_predictor: LLMPredictor*, *prompt_helper:* PromptHelper, *embed_model: BaseEmbedding*, *node_parser:* NodeParser, *llama_logger:* LlamaLogger, *chunk_size_limit: Optional[int] = None*)

Service Context container.

The service context container is a utility container for LlamaIndex index and query classes. It contains the following: - llm_predictor: LLMPredictor - prompt_helper: PromptHelper - embed_model: BaseEmbedding - node_parser: NodeParser - llama_logger: LlamaLogger - chunk_size_limit: chunk size limit

classmethod from_defaults(*llm_predictor: Optional[LLMPredictor] = None*, *prompt_helper: Optional[*PromptHelper*] = None*, *embed_model: Optional[BaseEmbedding] = None*, *node_parser: Optional[*NodeParser*] = None*, *llama_logger: Optional[*LlamaLogger*] = None*, *chunk_size_limit: Optional[int] = None*) → *ServiceContext*

Create a ServiceContext from defaults. If an argument is specified, then use the argument value provided for that parameter. If an argument is not specified, then use the default value.

Parameters

- **llm_predictor** (*Optional[LLMPredictor]*) – LLMPredictor
- **prompt_helper** (*Optional[*PromptHelper*]*) – PromptHelper
- **embed_model** (*Optional[BaseEmbedding]*) – BaseEmbedding
- **node_parser** (*Optional[*NodeParser*]*) – NodeParser
- **llama_logger** (*Optional[*LlamaLogger*]*) – LlamaLogger
- **chunk_size_limit** (*Optional[int]*) – chunk_size_limit

3.26 Optimizers

Optimization.

class gpt_index.optimization.**SentenceEmbeddingOptimizer**(*embed_model: Optional[BaseEmbedding] = None*, *percentile_cutoff: Optional[float] = None*, *threshold_cutoff: Optional[float] = None*, *tokenizer_fn: Optional[Callable[[str], List[str]]] = None*)

Optimization of a text chunk given the query by shortening the input text.

optimize(*query_bundle:* QueryBundle, *text: str*) → str

Optimize a text chunk given the query by shortening the input text.

3.27 Structured Index Configuration

Our structured indices are documented in *Structured Store Index*. Below, we provide a reference of the classes that are used to configure our structured indices.

SQL wrapper around SQLDatabase in langchain.

class gpt_index.langchain_helpers.sql_wrapper.**SQLDatabase**(**args: Any*, ***kwargs: Any*)

SQL Database.

Wrapper around SQLDatabase object from langchain. Offers some helper utilities for insertion and querying. See langchain documentation for more details:

Parameters

- ***args** – Arguments to pass to langchain SQLDatabase.
- ****kwargs** – Keyword arguments to pass to langchain SQLDatabase.

property dialect: str

Return string representation of dialect to use.

property engine: Engine

Return SQL Alchemy engine.

classmethod from_uri(*database_uri: str*, *engine_args: Optional[dict] = None*, ***kwargs: Any*) → *SQLDatabase*

Construct a SQLAlchemy engine from URI.

get_single_table_info(*table_name: str*) → str

Get table info for a single table.

get_table_columns(*table_name: str*) → List[dict]

Get table columns.

get_table_info(*table_names: Optional[List[str]] = None*) → str

Get information about specified tables.

Follows best practices as specified in: Rajkumar et al, 2022 (https://arxiv.org/abs/2204.00498)

If *sample_rows_in_table_info*, the specified number of sample rows will be appended to each table description. This can increase performance as demonstrated in the paper.

get_table_info_no_throw(*table_names: Optional[List[str]] = None*) → str

Get information about specified tables.

Follows best practices as specified in: Rajkumar et al, 2022 (https://arxiv.org/abs/2204.00498)

If *sample_rows_in_table_info*, the specified number of sample rows will be appended to each table description. This can increase performance as demonstrated in the paper.

get_table_names() → Iterable[str]

Get names of tables available.

get_usable_table_names() → Iterable[str]

Get names of tables available.

insert_into_table(*table_name: str*, *data: dict*) → None

Insert data into a table.

run(*command: str*, *fetch: str = 'all'*) → str

Execute a SQL command and return a string representing the results.

If the statement returns rows, a string of the results is returned. If the statement returns no rows, an empty string is returned.

run_no_throw(*command: str*, *fetch: str = 'all'*) → str

Execute a SQL command and return a string representing the results.

If the statement returns rows, a string of the results is returned. If the statement returns no rows, an empty string is returned.

If the statement throws an error, the error message is returned.

run_sql(*command: str*) → Tuple[str, Dict]

Execute a SQL statement and return a string representing the results.

If the statement returns rows, a string of the results is returned. If the statement returns no rows, an empty string is returned.

property table_info: str

Information about all tables in the database.

SQL Container builder.

class gpt_index.indices.struct_store.container_builder.**SQLContextContainerBuilder**(*sql_database: SQLDatabase*, *context_dict: Optional[Dict[str, str]] = None*, *context_str: Optional[str] = None*)

SQLContextContainerBuilder.

Build a SQLContextContainer that can be passed to the SQL index during index construction or during querytime.

NOTE: if context_str is specified, that will be used as context instead of context_dict

Parameters

- **sql_database** (SQLDatabase) – SQL database
- **context_dict** (*Optional[Dict[str, str]]*) – context dict

build_context_container(*ignore_db_schema: bool = False*) → SQLContextContainer

Build index structure.

derive_index_from_context(*index_cls: Type[BaseGPTIndex]*, *ignore_db_schema: bool = False*, ***index_kwargs: Any*) → *BaseGPTIndex*

Derive index from context.

classmethod from_documents(*documents_dict: Dict[str, List[BaseDocument]]*, *sql_database:* SQLDatabase, ***context_builder_kwargs: Any*) → *SQLContextContainerBuilder*

Build context from documents.

query_index_for_context(*index:* BaseGPTIndex, *query_str: Union[str,* QueryBundle*]*, *query_tmpl: Optional[str] = 'Please return the relevant tables (including the full schema) for the following query: {orig_query_str}'*, *store_context_str: bool = True*, ***index_kwargs: Any*) → str

Query index for context.

A simple wrapper around the index.query call which injects a query template to specifically fetch table information, and can store a context_str.

Parameters

- **index** (BaseGPTIndex) – index data structure
- **query_str** (*Union[str,* QueryBundle*]*) – query string
- **query_tmpl** (*Optional[str]*) – query template
- **store_context_str** (*bool*) – store context_str

Common classes for structured operations.

class gpt_index.indices.common.struct_store.base.**BaseStructDatapointExtractor**(*llm_predictor: LLMPredictor*, *schema_extract_prompt:* SchemaExtractPrompt, *output_parser: Callable[[str], Optional[Dict[str, Any]]]*)

Extracts datapoints from a structured document.

insert_datapoint_from_nodes(*nodes: Sequence[*Node*]*) → None

Extract datapoint from a document and insert it.

class gpt_index.indices.common.struct_store.base.**SQLDocumentContextBuilder**(*sql_database: SQLDatabase, service_context: Optional[ServiceContext] = None, text_splitter: Optional[TextSplitter] = None, table_context_prompt: Optional[TableContextPrompt] = None, refine_table_context_prompt: Optional[RefineTableContextPrompt] = None, table_context_task: Optional[str] = None*)

Builder that builds context for a given set of SQL tables.

Parameters

- **sql_database** (*Optional[SQLDatabase]*) – SQL database to use,
- **llm_predictor** (*Optional[LLMPredictor]*) – LLM Predictor to use.
- **prompt_helper** (*Optional[PromptHelper]*) – Prompt Helper to use.
- **text_splitter** (*Optional[TextSplitter]*) – Text Splitter to use.
- **table_context_prompt** (*Optional[TableContextPrompt]*) – A Table Context Prompt (see *Prompt Templates*).
- **refine_table_context_prompt** (*Optional[RefineTableContextPrompt]*) – A Refine Table Context Prompt (see *Prompt Templates*).
- **table_context_task** (*Optional[str]*) – The query to perform on the table context. A default query string is used if none is provided by the user.

build_all_context_from_documents(*documents_dict: Dict[str, List[BaseDocument]]*) → Dict[str, str]

Build context for all tables in the database.

build_table_context_from_documents(*documents: Sequence[BaseDocument]*, *table_name: str*) → str

Build context from documents for a single table.

3.28 Response

Response schema.

class gpt_index.response.schema.**Response**(*response: ~typing.Optional[str], source_nodes: ~typing.List[~gpt_index.data_structs.node_v2.NodeWithScore] = <factory>, extra_info: ~typing.Optional[~typing.Dict[str, ~typing.Any]] = None*)

Response object.

Returned if streaming=False during the *index.query()* call.

response

The response text.

Type
Optional[str]

get_formatted_sources(*length: int = 100*) → str

Get formatted sources text.

class gpt_index.response.schema.**StreamingResponse**(*response_gen: ~typing.Optional[~typing.Generator], source_nodes: ~typing.List[~gpt_index.data_structs.node_v2.NodeWithScore] = <factory>, extra_info: ~typing.Optional[~typing.Dict[str, ~typing.Any]] = None, response_txt: ~typing.Optional[str] = None*)

StreamingResponse object.

Returned if streaming=True during the *index.query()* call.

response_gen

The response generator.

Type
Optional[Generator]

get_formatted_sources(*length: int = 100*) → str

Get formatted sources text.

get_response() → *Response*

Get a standard response object.

print_response_stream() → None

Print the response stream.

3.29 Playground

Experiment with different indices, models, and more.

class gpt_index.playground.base.**Playground**(*indices: List[BaseGPTIndex], modes: List[str] = ['default', 'summarize', 'embedding', 'retrieve', 'recursive']*)

Experiment with indices, models, embeddings, modes, and more.

compare(*query_text: str, to_pandas: Optional[bool] = True*) → Union[DataFrame, List[Dict[str, Any]]]

Compare index outputs on an input query.

Parameters

- **query_text** (*str*) – Query to run all indices on.
- **to_pandas** (*Optional[bool]*) – Return results in a pandas dataframe. True by default.

Returns
The output of each index along with other data, such as the time it took to compute. Results are stored in a Pandas Dataframe or a list of Dicts.

classmethod from_docs(*documents: ~typing.List[~gpt_index.readers.schema.base.Document], index_classes: ~typing.List[~typing.Type[~gpt_index.indices.base.BaseGPTIndex]] = [<class 'gpt_index.indices.vector_store.vector_indices.GPTSimpleVectorIndex'>, <class 'gpt_index.indices.tree.base.GPTTreeIndex'>, <class 'gpt_index.indices.list.base.GPTListIndex'>], **kwargs: ~typing.Any*) → *Playground*

Initialize with Documents using the default list of indices.

Parameters
documents – A List of Documents to experiment with.

property indices: List[*BaseGPTIndex*]

Get Playground's indices.

property modes: List[str]

Get Playground's indices.

3.30 Node Parser

Node parsers.

class gpt_index.node_parser.**NodeParser**

Base interface for node parser.

abstract get_nodes_from_documents(*documents: Sequence[*Document*]*) → List[*Node*]

Parse documents into nodes.

Parameters
documents (*Sequence[Document]*) – documents to parse

class gpt_index.node_parser.**SimpleNodeParser**(*text_splitter: Optional[TextSplitter] = None, include_extra_info: bool = True, include_prev_next_rel: bool = True*)

Simple node parser.

Splits a document into Nodes using a TextSplitter.

Parameters

- **text_splitter** (*Optional[TextSplitter]*) – text splitter
- **include_extra_info** (*bool*) – whether to include extra info in nodes
- **include_prev_next_rel** (*bool*) – whether to include prev/next relationships

get_nodes_from_documents(*documents: Sequence[*Document*]*) → List[*Node*]

Parse document into nodes.

Parameters

- **documents** (*Sequence[Document]*) – documents to parse
- **include_extra_info** (*bool*) – whether to include extra info in nodes

3.31 Example Notebooks

We offer a wide variety of example notebooks. They are referenced throughout the documentation.

Example notebooks are found here.

3.32 Langchain Integrations

Agent Tools + Functions

Llama integration with Langchain agents.

class gpt_index.langchain_helpers.agents.**GraphToolConfig**(*, *graph:* ComposableGraph, *name: str*, *description: str*, *query_configs: List[Dict] = None*, *tool_kwargs: Dict = None*)

Configuration for LlamaIndex graph tool.

class Config

Configuration for this pydantic object.

classmethod construct(*_fields_set: Optional[SetStr] = None*, ***values: Any*) → Model

Creates a new model setting __dict__ and __fields_set__ from trusted or pre-validated data. Default values are respected, but no other validation is performed. Behaves as if *Config.extra = 'allow'* was set since it adds all passed values

copy(*, *include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *update: Optional[DictStrAny] = None*, *deep: bool = False*) → Model

Duplicate a model, optionally choose which fields to include, exclude and change.

Parameters

- **include** – fields to include in new model
- **exclude** – fields to exclude from new model, as with values this takes precedence over include
- **update** – values to change/add in the new model. Note: the data is not validated before creating the new model: you should trust this data
- **deep** – set to *True* to make a deep copy of the model

Returns
new model instance

dict(*, *include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *by_alias: bool = False*, *skip_defaults: Optional[bool] = None*, *exclude_unset: bool = False*, *exclude_defaults: bool = False*, *exclude_none: bool = False*) → DictStrAny

Generate a dictionary representation of the model, optionally specifying which fields to include or exclude.

json(*, *include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *by_alias: bool = False*, *skip_defaults: Optional[bool] = None*, *exclude_unset: bool = False*, *exclude_defaults: bool = False*, *exclude_none: bool = False*, *encoder: Optional[Callable[[Any], Any]] = None*, *models_as_dict: bool = True*, ***dumps_kwargs: Any*) → unicode

Generate a JSON representation of the model, *include* and *exclude* arguments as per *dict()*.

encoder is an optional function to supply as *default* to json.dumps(), other arguments as per *json.dumps()*.

classmethod update_forward_refs(***localns: Any*) → None

Try to update ForwardRefs on fields based on this Model, globalns and localns.

class gpt_index.langchain_helpers.agents.**IndexToolConfig**(*, *index:* BaseGPTIndex, *name: str*, *description: str*, *index_query_kwargs: Dict = None*, *tool_kwargs: Dict = None*)

Configuration for LlamaIndex index tool.

class Config

Configuration for this pydantic object.

classmethod construct(*_fields_set: Optional[SetStr] = None*, ***values: Any*) → Model

Creates a new model setting __dict__ and __fields_set__ from trusted or pre-validated data. Default values are respected, but no other validation is performed. Behaves as if *Config.extra = 'allow'* was set since it adds all passed values

copy(*, *include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *update: Optional[DictStrAny] = None*, *deep: bool = False*) → Model

Duplicate a model, optionally choose which fields to include, exclude and change.

Parameters

- **include** – fields to include in new model
- **exclude** – fields to exclude from new model, as with values this takes precedence over include
- **update** – values to change/add in the new model. Note: the data is not validated before creating the new model: you should trust this data
- **deep** – set to *True* to make a deep copy of the model

Returns

new model instance

dict(*, *include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *by_alias: bool = False*, *skip_defaults: Optional[bool] = None*, *exclude_unset: bool = False*, *exclude_defaults: bool = False*, *exclude_none: bool = False*) → DictStrAny

Generate a dictionary representation of the model, optionally specifying which fields to include or exclude.

json(*, *include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *by_alias: bool = False*, *skip_defaults: Optional[bool] = None*, *exclude_unset: bool = False*, *exclude_defaults: bool = False*, *exclude_none: bool = False*, *encoder: Optional[Callable[[Any], Any]] = None*, *models_as_dict: bool = True*, ***dumps_kwargs: Any*) → unicode

Generate a JSON representation of the model, *include* and *exclude* arguments as per *dict()*.

encoder is an optional function to supply as *default* to json.dumps(), other arguments as per *json.dumps()*.

classmethod update_forward_refs(***localns: Any*) → None

Try to update ForwardRefs on fields based on this Model, globalns and localns.

class gpt_index.langchain_helpers.agents.**LlamaGraphTool**(*, *name: str*, *description: str*, *return_direct: bool = False*, *verbose: bool = False*, *callback_manager: BaseCallbackManager = None*, *graph:* ComposableGraph, *query_configs: List[Dict] = None*, *return_sources: bool = False*)

Tool for querying a ComposableGraph.

class **Config**

Configuration for this pydantic object.

async **arun**(*tool_input: str*, *verbose: Optional[bool] = None*, *start_color: Optional[str] = 'green'*, *color: Optional[str] = 'green'*, ***kwargs: Any*) → str

Run the tool asynchronously.

classmethod **construct**(*_fields_set: Optional[SetStr] = None*, ***values: Any*) → Model

Creates a new model setting __dict__ and __fields_set__ from trusted or pre-validated data. Default values are respected, but no other validation is performed. Behaves as if *Config.extra = 'allow'* was set since it adds all passed values

copy(*, *include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *update: Optional[DictStrAny] = None*, *deep: bool = False*) → Model

Duplicate a model, optionally choose which fields to include, exclude and change.

Parameters

- **include** – fields to include in new model
- **exclude** – fields to exclude from new model, as with values this takes precedence over include
- **update** – values to change/add in the new model. Note: the data is not validated before creating the new model: you should trust this data
- **deep** – set to *True* to make a deep copy of the model

Returns

new model instance

dict(*, *include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *by_alias: bool = False*, *skip_defaults: Optional[bool] = None*, *exclude_unset: bool = False*, *exclude_defaults: bool = False*, *exclude_none: bool = False*) → DictStrAny

Generate a dictionary representation of the model, optionally specifying which fields to include or exclude.

classmethod **from_tool_config**(*tool_config:* GraphToolConfig) → *LlamaGraphTool*

Create a tool from a tool config.

json(*, *include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *by_alias: bool = False*, *skip_defaults: Optional[bool] = None*, *exclude_unset: bool = False*, *exclude_defaults: bool = False*, *exclude_none: bool = False*, *encoder: Optional[Callable[[Any], Any]] = None*, *models_as_dict: bool = True*, ***dumps_kwargs: Any*) → unicode

Generate a JSON representation of the model, *include* and *exclude* arguments as per *dict()*.

encoder is an optional function to supply as *default* to json.dumps(), other arguments as per *json.dumps()*.

run(*tool_input: str*, *verbose: Optional[bool] = None*, *start_color: Optional[str] = 'green'*, *color: Optional[str] = 'green'*, ***kwargs: Any*) → str

Run the tool.

classmethod set_callback_manager(*callback_manager: Optional[BaseCallbackManager]*) → BaseCallbackManager

If callback manager is None, set it.

This allows users to pass in None as callback manager, which is a nice UX.

classmethod update_forward_refs(***localns: Any*) → None

Try to update ForwardRefs on fields based on this Model, globalns and localns.

class gpt_index.langchain_helpers.agents.**LlamaIndexTool**(***, *name: str*, *description: str*, *return_direct: bool = False*, *verbose: bool = False*, *callback_manager: BaseCallbackManager = None*, *index: BaseGPTIndex*, *query_kwargs: Dict = None*, *return_sources: bool = False*)

Tool for querying a LlamaIndex.

class Config

Configuration for this pydantic object.

async arun(*tool_input: str*, *verbose: Optional[bool] = None*, *start_color: Optional[str] = 'green'*, *color: Optional[str] = 'green'*, ***kwargs: Any*) → str

Run the tool asynchronously.

classmethod construct(*_fields_set: Optional[SetStr] = None*, ***values: Any*) → Model

Creates a new model setting __dict__ and __fields_set__ from trusted or pre-validated data. Default values are respected, but no other validation is performed. Behaves as if *Config.extra = 'allow'* was set since it adds all passed values

copy(***, *include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *update: Optional[DictStrAny] = None*, *deep: bool = False*) → Model

Duplicate a model, optionally choose which fields to include, exclude and change.

Parameters

- **include** – fields to include in new model
- **exclude** – fields to exclude from new model, as with values this takes precedence over include
- **update** – values to change/add in the new model. Note: the data is not validated before creating the new model: you should trust this data
- **deep** – set to *True* to make a deep copy of the model

Returns

new model instance

dict(***, *include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *by_alias: bool = False*, *skip_defaults: Optional[bool] = None*, *exclude_unset: bool = False*, *exclude_defaults: bool = False*, *exclude_none: bool = False*) → DictStrAny

Generate a dictionary representation of the model, optionally specifying which fields to include or exclude.

classmethod from_tool_config(*tool_config:* IndexToolConfig) → *LlamaIndexTool*

Create a tool from a tool config.

json(*, *include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *by_alias: bool = False*, *skip_defaults: Optional[bool] = None*, *exclude_unset: bool = False*, *exclude_defaults: bool = False*, *exclude_none: bool = False*, *encoder: Optional[Callable[[Any], Any]] = None*, *models_as_dict: bool = True*, ***dumps_kwargs: Any*) → unicode

Generate a JSON representation of the model, *include* and *exclude* arguments as per *dict()*.

encoder is an optional function to supply as *default* to json.dumps(), other arguments as per *json.dumps()*.

run(*tool_input: str*, *verbose: Optional[bool] = None*, *start_color: Optional[str] = 'green'*, *color: Optional[str] = 'green'*, ***kwargs: Any*) → str

Run the tool.

classmethod set_callback_manager(*callback_manager: Optional[BaseCallbackManager]*) → BaseCallbackManager

If callback manager is None, set it.

This allows users to pass in None as callback manager, which is a nice UX.

classmethod update_forward_refs(***localns: Any*) → None

Try to update ForwardRefs on fields based on this Model, globalns and localns.

class gpt_index.langchain_helpers.agents.**LlamaToolkit**(*, *index_configs: List[*IndexToolConfig*] = None*, *graph_configs: List[*GraphToolConfig*] = None*)

Toolkit for interacting with Llama indices.

class Config

Configuration for this pydantic object.

classmethod construct(*_fields_set: Optional[SetStr] = None*, ***values: Any*) → Model

Creates a new model setting __dict__ and __fields_set__ from trusted or pre-validated data. Default values are respected, but no other validation is performed. Behaves as if *Config.extra = 'allow'* was set since it adds all passed values

copy(*, *include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *update: Optional[DictStrAny] = None*, *deep: bool = False*) → Model

Duplicate a model, optionally choose which fields to include, exclude and change.

Parameters

- **include** – fields to include in new model
- **exclude** – fields to exclude from new model, as with values this takes precedence over include
- **update** – values to change/add in the new model. Note: the data is not validated before creating the new model: you should trust this data
- **deep** – set to *True* to make a deep copy of the model

Returns

new model instance

dict(*, *include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *by_alias: bool = False*, *skip_defaults: Optional[bool] = None*, *exclude_unset: bool = False*, *exclude_defaults: bool = False*, *exclude_none: bool = False*) → DictStrAny

Generate a dictionary representation of the model, optionally specifying which fields to include or exclude.

get_tools() → List[BaseTool]

Get the tools in the toolkit.

json(*, *include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *by_alias: bool = False*, *skip_defaults: Optional[bool] = None*, *exclude_unset: bool = False*, *exclude_defaults: bool = False*, *exclude_none: bool = False*, *encoder: Optional[Callable[[Any], Any]] = None*, *models_as_dict: bool = True*, ***dumps_kwargs: Any*) → unicode

Generate a JSON representation of the model, *include* and *exclude* arguments as per *dict()*.

encoder is an optional function to supply as *default* to json.dumps(), other arguments as per *json.dumps()*.

classmethod update_forward_refs(***localns: Any*) → None

Try to update ForwardRefs on fields based on this Model, globalns and localns.

gpt_index.langchain_helpers.agents.**create_llama_agent**(*toolkit:* LlamaToolkit, *llm: BaseLLM*, *agent: Optional[AgentType] = None*, *callback_manager: Optional[BaseCallbackManager] = None*, *agent_path: Optional[str] = None*, *agent_kwargs: Optional[dict] = None*, ***kwargs: Any*) → AgentExecutor

Load an agent executor given a Llama Toolkit and LLM.

NOTE: this is a light wrapper around initialize_agent in langchain.

Parameters

- **toolkit** – LlamaToolkit to use.
- **llm** – Language model to use as the agent.
- **agent** –

  A string that specified the agent type to use. Valid options are:
  zero-shot-react-description react-docstore self-ask-with-search conversational-react-description chat-zero-shot-react-description, *chat-conversational-react-description*,

  If None and agent_path is also None, will default to
  zero-shot-react-description.
- **callback_manager** – CallbackManager to use. Global callback manager is used if not provided. Defaults to None.
- **agent_path** – Path to serialized agent to use.
- **agent_kwargs** – Additional key word arguments to pass to the underlying agent
- ****kwargs** – Additional key word arguments passed to the agent executor

Returns
An agent executor

gpt_index.langchain_helpers.agents.**create_llama_chat_agent**(*toolkit: LlamaToolkit*, *llm: BaseLLM*, *callback_manager: Optional[BaseCallbackManager] = None*, *agent_kwargs: Optional[dict] = None*, ***kwargs: Any*) → AgentExecutor

Load a chat llama agent given a Llama Toolkit and LLM.

Parameters

- **toolkit** – LlamaToolkit to use.
- **llm** – Language model to use as the agent.
- **callback_manager** – CallbackManager to use. Global callback manager is used if not provided. Defaults to None.
- **agent_kwargs** – Additional key word arguments to pass to the underlying agent
- ****kwargs** – Additional key word arguments passed to the agent executor

Returns

An agent executor

Memory Module

Langchain memory wrapper (for LlamaIndex).

class gpt_index.langchain_helpers.memory_wrapper.**GPTIndexChatMemory**(***, *chat_memory: BaseChatMessageHistory = None*, *output_key: Optional[str] = None*, *input_key: Optional[str] = None*, *return_messages: bool = False*, *human_prefix: str = 'Human'*, *ai_prefix: str = 'AI'*, *memory_key: str = 'history'*, *index: BaseGPTIndex*, *query_kwargs: Dict = None*, *return_source: bool = False*, *id_to_message: Dict[str, BaseMessage] = None*)

Langchain chat memory wrapper (for LlamaIndex).

Parameters

- **human_prefix** (*str*) – Prefix for human input. Defaults to "Human".
- **ai_prefix** (*str*) – Prefix for AI output. Defaults to "AI".
- **memory_key** (*str*) – Key for memory. Defaults to "history".
- **index** (BaseGPTIndex) – LlamaIndex instance.
- **query_kwargs** (*Dict[str, Any]*) – Keyword arguments for LlamaIndex query.
- **input_key** (*Optional[str]*) – Input key. Defaults to None.
- **output_key** (*Optional[str]*) – Output key. Defaults to None.

class Config

Configuration for this pydantic object.

clear() → None

Clear memory contents.

classmethod construct(*_fields_set: Optional[SetStr] = None*, ***values: Any*) → Model

Creates a new model setting __dict__ and __fields_set__ from trusted or pre-validated data. Default values are respected, but no other validation is performed. Behaves as if *Config.extra = 'allow'* was set since it adds all passed values

copy(***, *include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *update: Optional[DictStrAny] = None*, *deep: bool = False*) → Model

Duplicate a model, optionally choose which fields to include, exclude and change.

Parameters

- **include** – fields to include in new model
- **exclude** – fields to exclude from new model, as with values this takes precedence over include
- **update** – values to change/add in the new model. Note: the data is not validated before creating the new model: you should trust this data
- **deep** – set to *True* to make a deep copy of the model

Returns

new model instance

dict(***, *include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *by_alias: bool = False*, *skip_defaults: Optional[bool] = None*, *exclude_unset: bool = False*, *exclude_defaults: bool = False*, *exclude_none: bool = False*) → DictStrAny

Generate a dictionary representation of the model, optionally specifying which fields to include or exclude.

json(***, *include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *by_alias: bool = False*, *skip_defaults: Optional[bool] = None*, *exclude_unset: bool = False*, *exclude_defaults: bool = False*, *exclude_none: bool = False*, *encoder: Optional[Callable[[Any], Any]] = None*, *models_as_dict: bool = True*, ***dumps_kwargs: Any*) → unicode

Generate a JSON representation of the model, *include* and *exclude* arguments as per *dict()*.

encoder is an optional function to supply as *default* to json.dumps(), other arguments as per *json.dumps()*.

load_memory_variables(*inputs: Dict[str, Any]*) → Dict[str, str]

Return key-value pairs given the text input to the chain.

property memory_variables: List[str]

Return memory variables.

save_context(*inputs: Dict[str, Any]*, *outputs: Dict[str, str]*) → None

Save the context of this model run to memory.

classmethod update_forward_refs(***localns: Any*) → None

Try to update ForwardRefs on fields based on this Model, globalns and localns.

class gpt_index.langchain_helpers.memory_wrapper.**GPTIndexMemory**(*, *human_prefix: str = 'Human'*, *ai_prefix: str = 'AI'*, *memory_key: str = 'history'*, *index:* BaseGPTIndex, *query_kwargs: Dict = None*, *output_key: Optional[str] = None*, *input_key: Optional[str] = None*)

Langchain memory wrapper (for LlamaIndex).

Parameters

- **human_prefix** (*str*) – Prefix for human input. Defaults to "Human".
- **ai_prefix** (*str*) – Prefix for AI output. Defaults to "AI".
- **memory_key** (*str*) – Key for memory. Defaults to "history".
- **index** (BaseGPTIndex) – LlamaIndex instance.
- **query_kwargs** (*Dict[str, Any]*) – Keyword arguments for LlamaIndex query.
- **input_key** (*Optional[str]*) – Input key. Defaults to None.
- **output_key** (*Optional[str]*) – Output key. Defaults to None.

class Config

Configuration for this pydantic object.

clear() → None

Clear memory contents.

classmethod construct(*_fields_set: Optional[SetStr] = None*, ***values: Any*) → Model

Creates a new model setting __dict__ and __fields_set__ from trusted or pre-validated data. Default values are respected, but no other validation is performed. Behaves as if *Config.extra = 'allow'* was set since it adds all passed values

copy(*, *include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *update: Optional[DictStrAny] = None*, *deep: bool = False*) → Model

Duplicate a model, optionally choose which fields to include, exclude and change.

Parameters

- **include** – fields to include in new model
- **exclude** – fields to exclude from new model, as with values this takes precedence over include
- **update** – values to change/add in the new model. Note: the data is not validated before creating the new model: you should trust this data
- **deep** – set to *True* to make a deep copy of the model

Returns

new model instance

dict(*, *include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *by_alias: bool = False*, *skip_defaults: Optional[bool] = None*, *exclude_unset: bool = False*, *exclude_defaults: bool = False*, *exclude_none: bool = False*) → DictStrAny

Generate a dictionary representation of the model, optionally specifying which fields to include or exclude.

json(*, *include: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *exclude: Optional[Union[AbstractSetIntStr, MappingIntStrAny]] = None*, *by_alias: bool = False*, *skip_defaults: Optional[bool] = None*, *exclude_unset: bool = False*, *exclude_defaults: bool = False*, *exclude_none: bool = False*, *encoder: Optional[Callable[[Any], Any]] = None*, *models_as_dict: bool = True*, ***dumps_kwargs: Any*) → unicode

Generate a JSON representation of the model, *include* and *exclude* arguments as per *dict()*.

encoder is an optional function to supply as *default* to json.dumps(), other arguments as per *json.dumps()*.

load_memory_variables(*inputs: Dict[str, Any]*) → Dict[str, str]

Return key-value pairs given the text input to the chain.

property memory_variables: List[str]

Return memory variables.

save_context(*inputs: Dict[str, Any]*, *outputs: Dict[str, str]*) → None

Save the context of this model run to memory.

classmethod update_forward_refs(***localns: Any*) → None

Try to update ForwardRefs on fields based on this Model, globalns and localns.

gpt_index.langchain_helpers.memory_wrapper.**get_prompt_input_key**(*inputs: Dict[str, Any]*, *memory_variables: List[str]*) → str

Get prompt input key.

Copied over from langchain.

3.33 App Showcase

Here is a sample of some of the incredible applications and tools built on top of LlamaIndex!

3.33.1 Meru - Dense Data Retrieval API

Hosted API service. Includes a "Dense Data Retrieval" API built on top of LlamaIndex where users can upload their documents and query them. [Website]

3.33.2 Algovera

Build AI workflows using building blocks. Many workflows built on top of LlamaIndex.

[Website].

3.33.3 ChatGPT LlamaIndex

Interface that allows users to upload long docs and chat with the bot. [Tweet thread]

3.33.4 AgentHQ

A web tool to build agents, interacting with LlamaIndex data structures.[Website]

3.33.5 PapersGPT

Feed any of the following content into GPT to give it deep customized knowledge:

- Scientific Papers
- Substack Articles
- Podcasts
- Github Repos and more.

[Tweet thread] [Website]

3.33.6 VideoQues + DocsQues

VideoQues: A tool that answers your queries on YouTube videos. [LinkedIn post here].

DocsQues: A tool that answers your questions on longer documents (including .pdfs!) [LinkedIn post here].

3.33.7 PaperBrain

A platform to access/understand research papers.

[Tweet thread].

3.33.8 CACTUS

Contextual search on top of LinkedIn search results. [LinkedIn post here].

3.33.9 Personal Note Chatbot

A chatbot that can answer questions over a directory of Obsidian notes. [Tweet thread].

3.33.10 RHOBH AMA

Ask questions about the Real Housewives of Beverly Hills. [Tweet thread] [Website]

3.33.11 Mynd

A journaling app that uses AI to uncover insights and patterns over time. [Website]

3.33.12 AI-X by OpenExO

Your Digital Transformation Co-Pilot [Website]

3.33.13 AnySummary

Summarize any document, audio or video with AI [Website]

3.33.14 Blackmaria

Python package for webscraping in Natural language. [Tweet thread] [Github]

PYTHON MODULE INDEX

g

INDEX

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

W

Y