# LlamaIndex

Jerry Liu

Apr 02, 2023

# GETTING STARTED

1 Overview 3

Python Module Index 227

Index 229

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

# OVERVIEW

## 1.1 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

## 1.2 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.

## 1.2.1 Installation and Setup

### Installation from Pip

You can simply do:

```
pip install llama-index
```

### 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).

### 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.

## 1.2.2 Starter Tutorial

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

### 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.

### Build and Query Index

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

```python
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

```python
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.`

### Viewing Queries and Events Using Logging

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

```python
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.

### Saving and Loading

To save to disk and load from disk, do

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

### 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*.

### 1.2.3 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

#### 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 to 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).

```python
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.

```python
from gpt_index.docstore import DocumentStore

docstore = DocumentStore()
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

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

```python
from llama_index import GPTSimpleVectorIndex

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

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

**NOTE**: An `insert_node` function is coming!

### Customizing LLM's

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

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

...

# define LLM
```

(continues on next page)

(continued from previous page)

```
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(
    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')
```

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

You can build indices on top of other indices!

```python
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:

```python
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.

```python
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"]
)
```

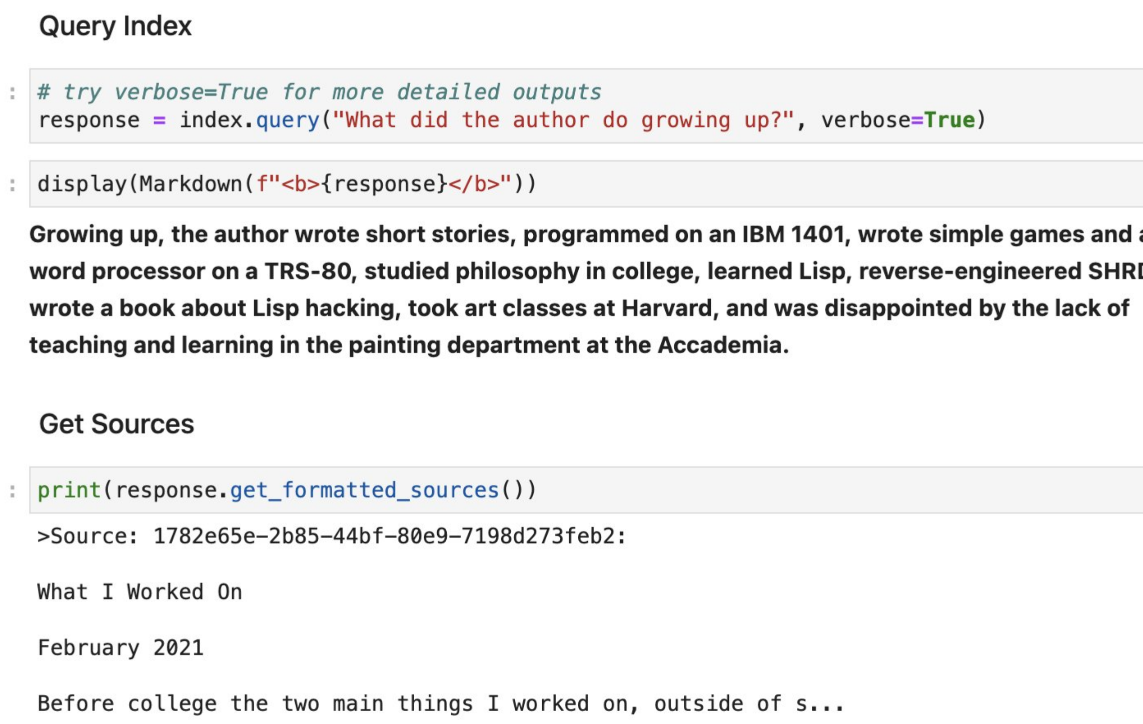
## 5. Parsing the response

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 SHRD 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.

### How Each Index Works

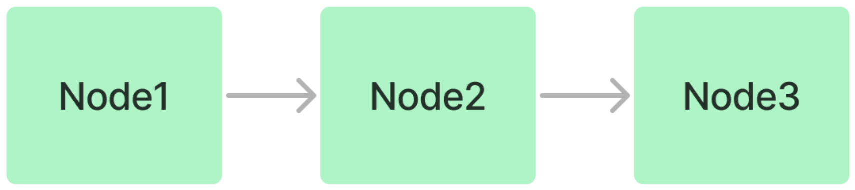
This guide describes how each index works with diagrams. We also visually highlight our "Response Synthesis" modes.

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.

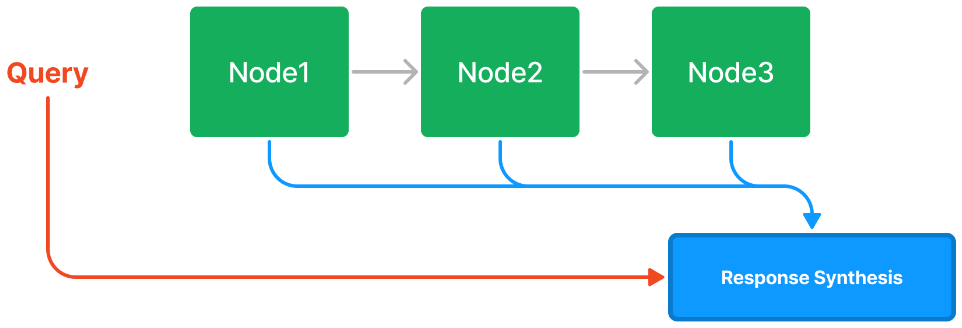
### List Index

The list index simply stores Nodes as a sequential chain.

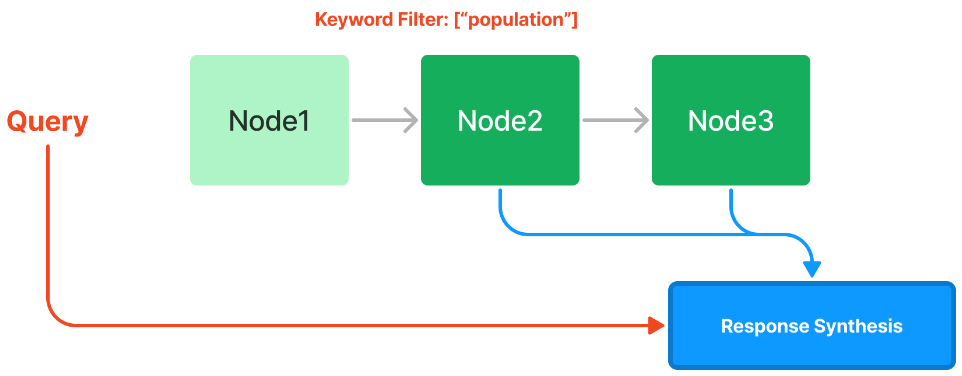


### Querying

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

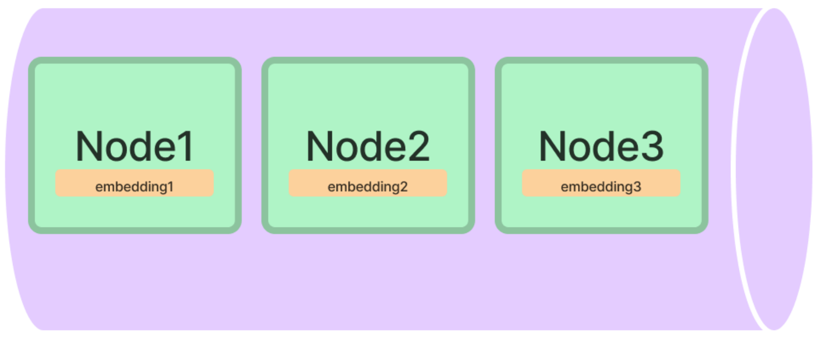


The list index does offer numerous ways of querying a list index, from an embedding-based query which will fetch the top-k neighbors, or with the addition of a keyword filter, as seen below:



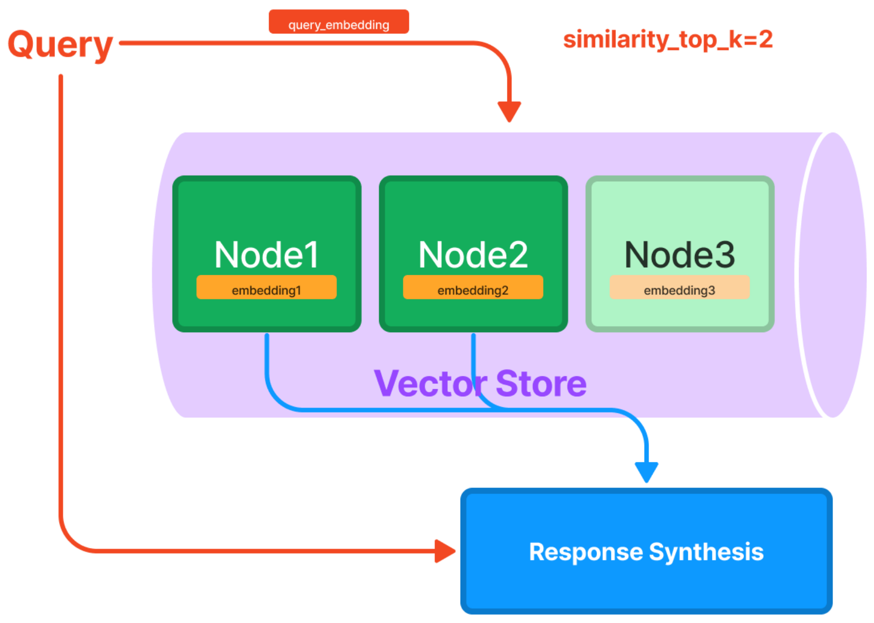
### Vector Store Index

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



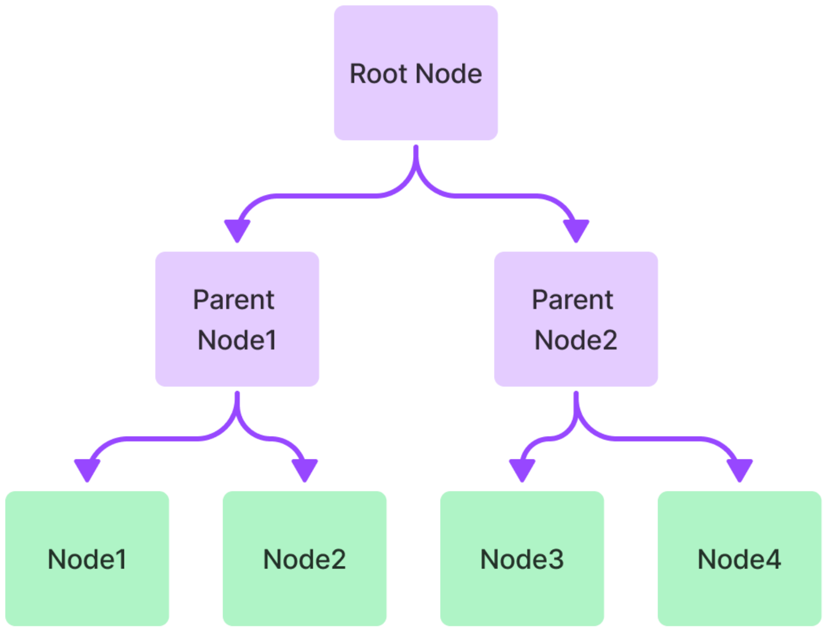
### Querying

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



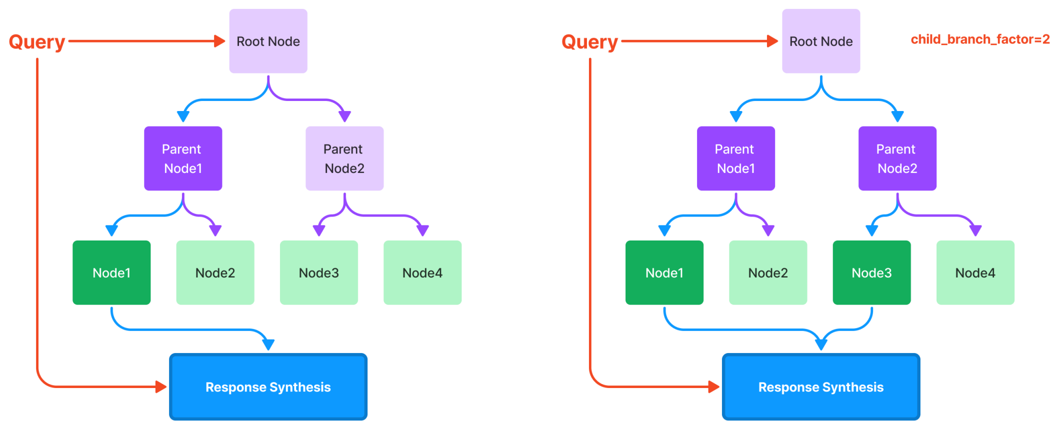
### Tree Index

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



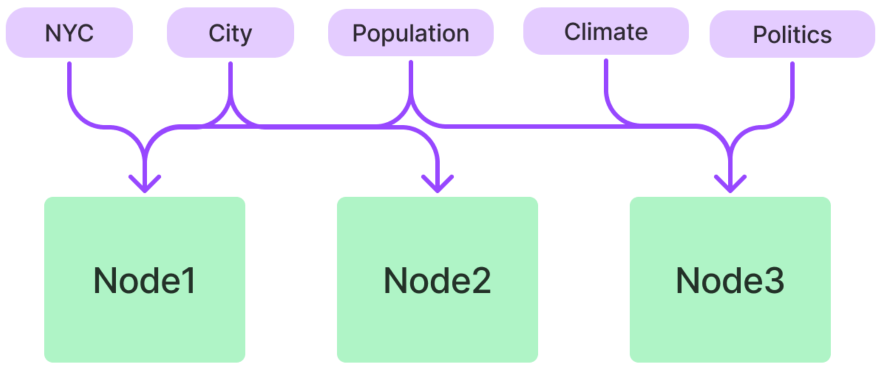
## Querying

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.



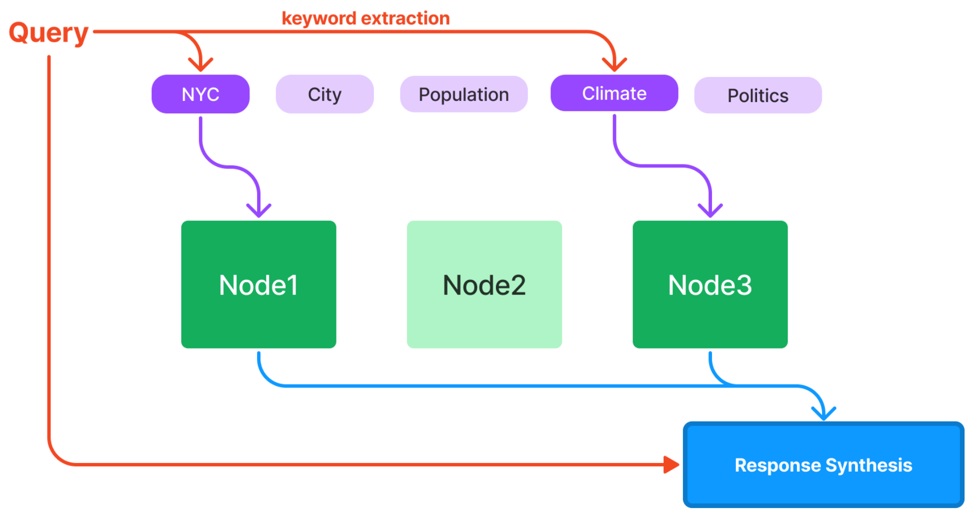
### Keyword Table Index

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.

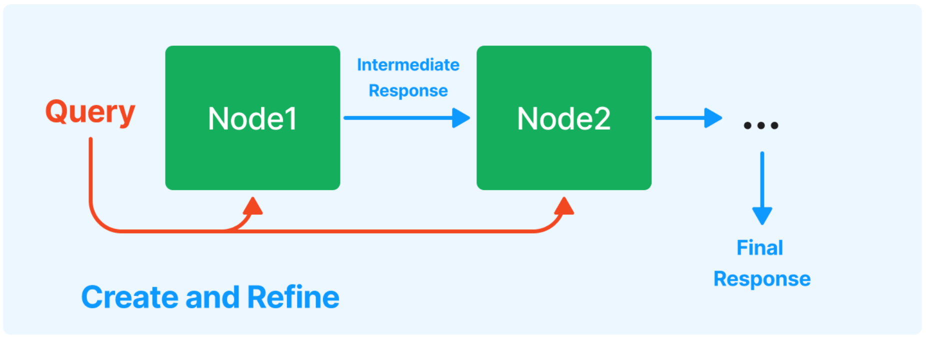


### Response Synthesis

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.

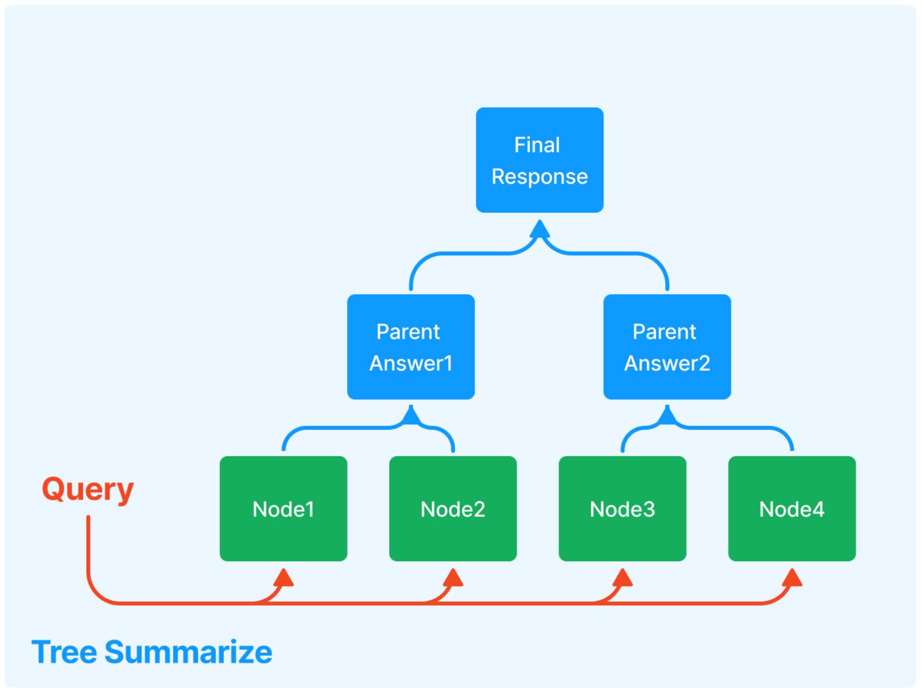
### Create and Refine

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.



### 1.2.4 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.

### 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 questions, 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,
)
```

(continues on next page)

(continued from previous page)

```
# [optional] save to disk
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": {
```

(continues on next page)

(continued from previous page)

```
            "response_mode": "tree_summarize",
            "verbose": True
        }
    },
]
# 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.

### 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:

```python
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, therequirements.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:

### 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": "United States"},
]
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,
```

(continues on next page)

(continued from previous page)

```
    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.

```python
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:

```python
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!

```python
# 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:

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

(continues on next page)

(continued from previous page)

```
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)
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
```

(continues on next page)

(continued from previous page)

```
↪"
)
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

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
)
```

(continues on next page)

(continued from previous page)

```
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.

### 1.2.5 Notebooks

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

Example notebooks are found here.

### 1.2.6 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.

#### 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

### 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")
```

### 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.

### Synthesis over Heterogenous Data

LlamaIndex supports synthesizing across heterogenous 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 heteregenous 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.

### Routing over Heterogenous Data

LlamaIndex also supports routing over heteregenous 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.

### 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

#### 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

### 1.2.7 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.

#### 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*

#### 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.

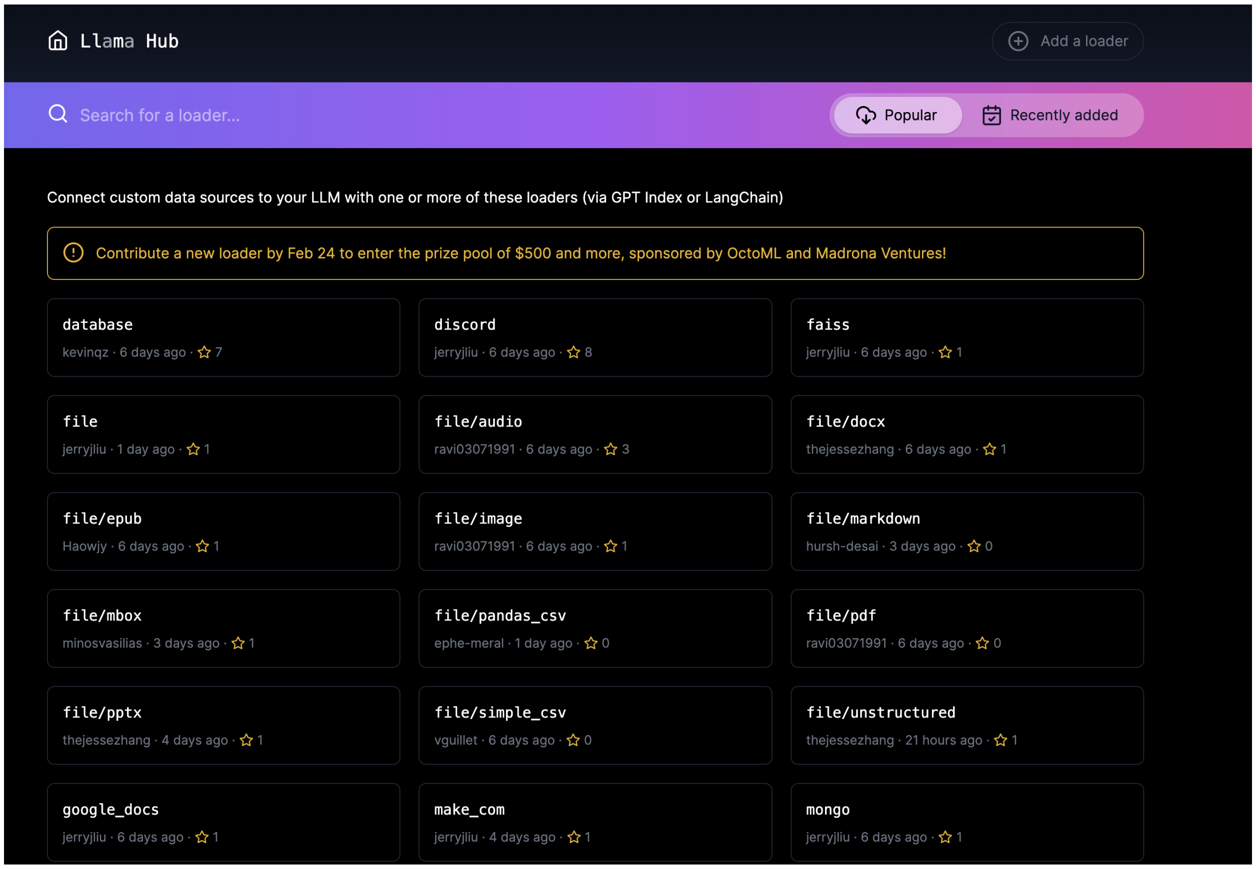
We provide tutorials and resources to help you get started in this area.

Relevant Resources:

- *Fullstack Application Guide*
- LlamaIndex Starter Pack

## 1.2.8 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.



Connect custom data sources to your LLM with one or more of these loaders (via GPT Index or LangChain)

Contribute a new loader by Feb 24 to enter the prize pool of $500 and more, sponsored by OctoML and Madrona Ventures!

| | | |
|---|---|---|
| database<br>kevinqz · 6 days ago · ☆ 7 | discord<br>jerryjliu · 6 days ago · ☆ 8 | faiss<br>jerryjliu · 6 days ago · ☆ 1 |
| file<br>jerryjliu · 1 day ago · ☆ 1 | file/audio<br>ravi03071991 · 6 days ago · ☆ 3 | file/docx<br>thejessezhang · 6 days ago · ☆ 1 |
| file/epub<br>Haowjy · 6 days ago · ☆ 1 | file/image<br>ravi03071991 · 6 days ago · ☆ 1 | file/markdown<br>hursh-desai · 3 days ago · ☆ 0 |
| file/mbox<br>minosvasilias · 3 days ago · ☆ 1 | file/pandas_csv<br>ephe-meral · 1 day ago · ☆ 0 | file/pdf<br>ravi03071991 · 6 days ago · ☆ 0 |
| file/pptx<br>thejessezhang · 4 days ago · ☆ 1 | file/simple_csv<br>vguillet · 6 days ago · ☆ 0 | file/unstructured<br>thejessezhang · 21 hours ago · ☆ 1 |
| google_docs<br>jerryjliu · 6 days ago · ☆ 1 | make_com<br>jerryjliu · 4 days ago · ☆ 1 | mongo<br>jerryjliu · 6 days ago · ☆ 1 |

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:

```python
from llama_index import GPTSimpleVectorIndex, download_loader

GoogleDocsReader = download_loader('GoogleDocsReader')

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

(continues on next page)

(continued from previous page)

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

## 1.2.9 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.

### 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])
```

### Composability

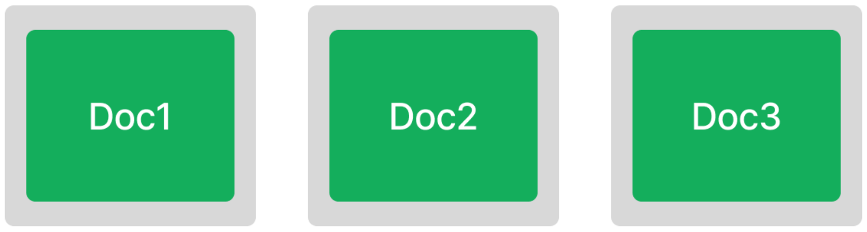
LlamaIndex offers **composability** of your indices, meaning that you can build indices on top of other indices. This allows you to more effectively index your entire document tree in order to feed custom knowledge to GPT.

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

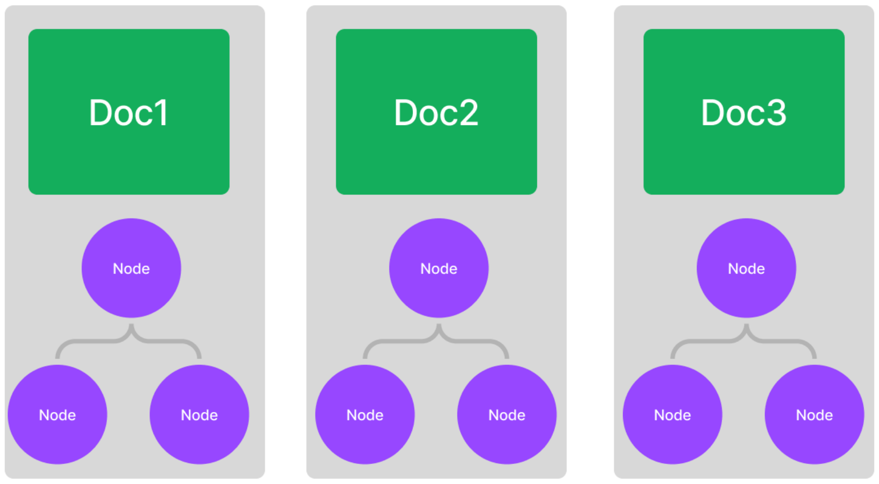
To see how this works, imagine you have 3 documents: `doc1`, `doc2`, and `doc3`.

```
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

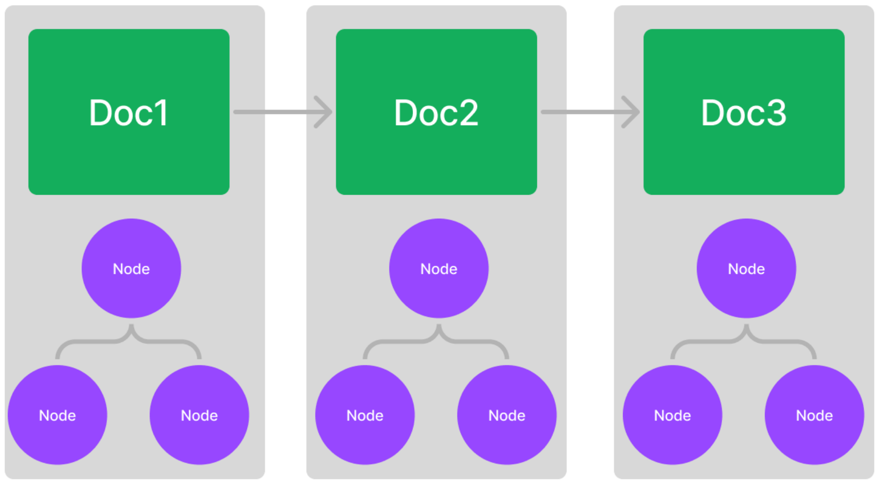
We can then create a graph with a list index on top of these 3 tree indices: We can query, save, and load the graph to/from disk as any other index.

```
from llama_index.indices.composability import ComposableGraph

graph = ComposableGraph.build_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")
```



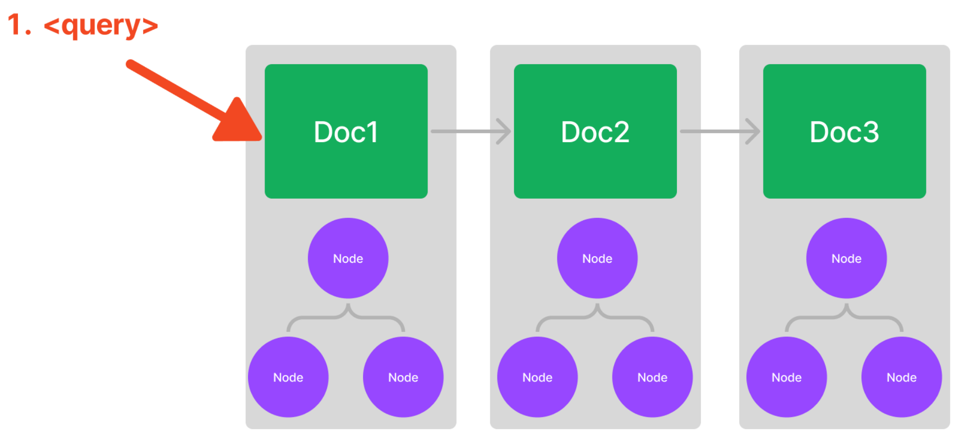
### Querying the Graph

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*.

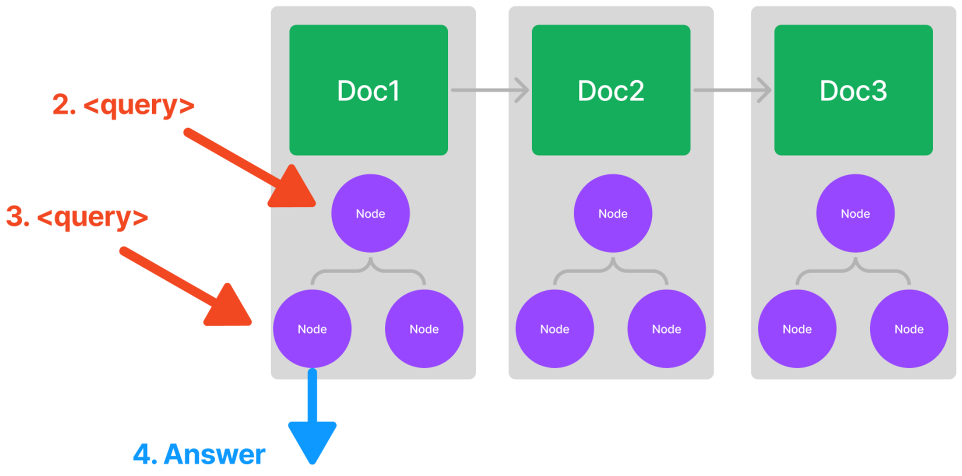
```
# set query config. An example is provided below
query_configs = [
    {
        # NOTE: index_struct_id is optional
        "index_struct_id": "<index_id_1>",
        "index_struct_type": "tree",
        "query_mode": "default",
        "query_kwargs": {
            "child_branch_factor": 2
   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

## 1.2.10 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.

### 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.

```python
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)
```

(continues on next page)

(continued from previous page)

```
# 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

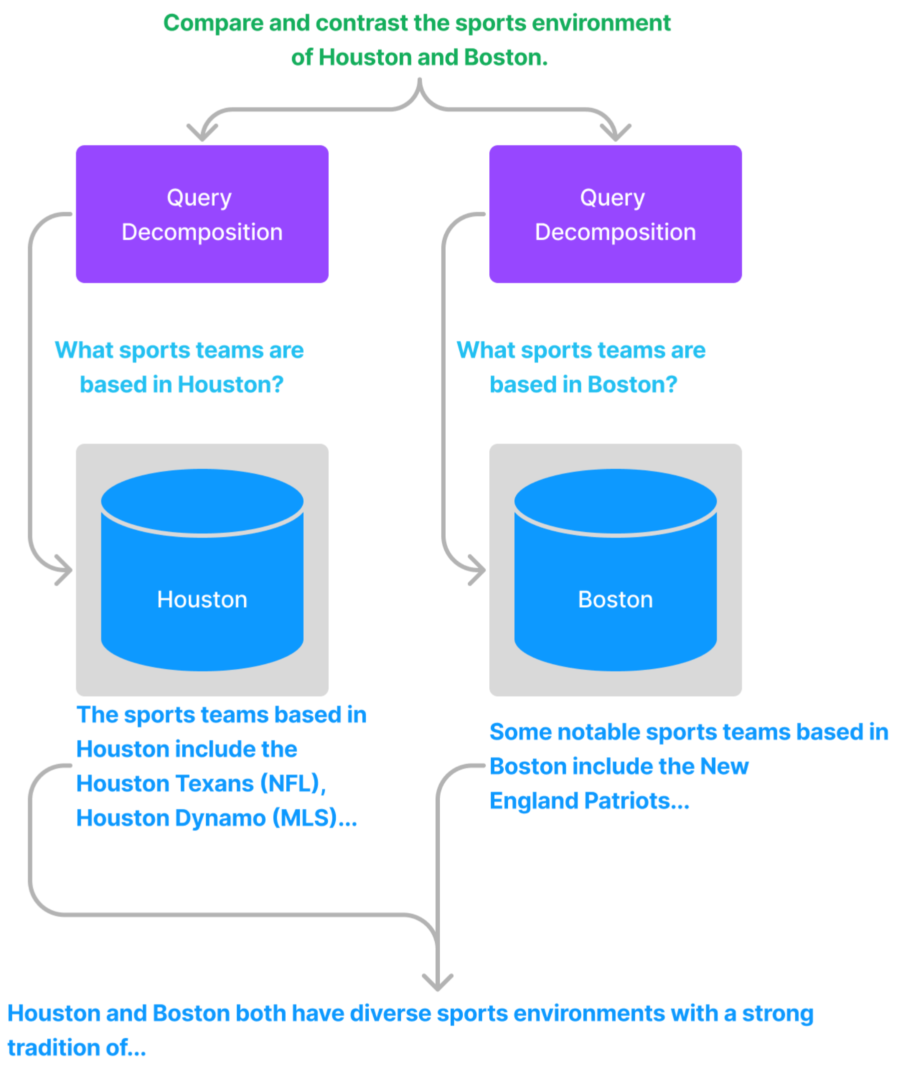
Some recent approaches (e.g. self-ask, ReAct) have suggested that LLM's perform better at answering complex questions when they break the question into smaller steps. We have found that this is true for queries that require knowledge augmentation as well.

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.


Compare and contrast the sports environment of Houston and Boston.
Query Decomposition
Query Decomposition
What sports teams are based in Houston?
What sports teams are based in Boston?
Houston
Boston
The sports teams based in Houston include the Houston Texans (NFL), Houston Dynamo (MLS)...
Some notable sports teams based in Boston include the New England Patriots...
Houston and Boston both have diverse sports environments with a strong tradition of...

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.

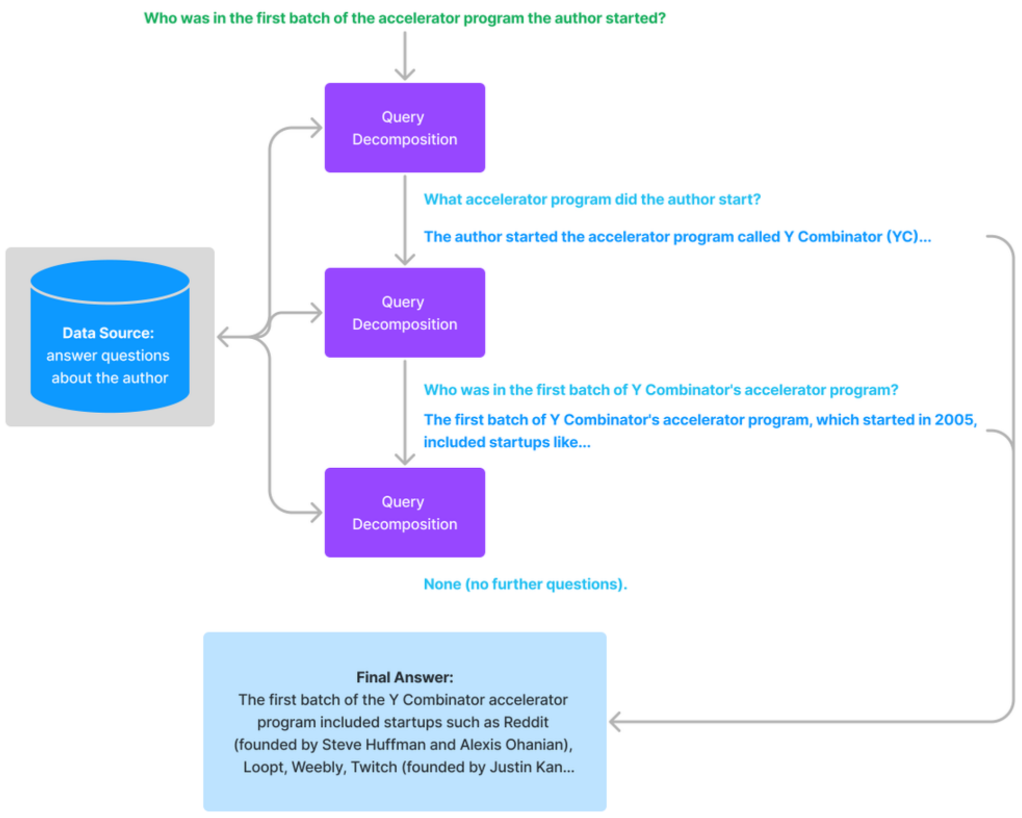
### Multi-Step Query Transformations

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.


Who was in the first batch of the accelerator program the author started?
Query Decomposition
What accelerator program did the author start?
The author started the accelerator program called Y Combinator (YC)...
Data Source: answer questions about the author
Query Decomposition
Who was in the first batch of Y Combinator's accelerator program?
The first batch of Y Combinator's accelerator program, which started in 2005, included startups like...
Query Decomposition
None (no further questions).
Final Answer: The first batch of the Y Combinator accelerator program included startups such as Reddit (founded by Steve Huffman and Alexis Ohanian), Loopt, Weebly, Twitch (founded by Justin Kan...

Here's a corresponding example code snippet.

```python
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.

### 1.2.11 Customization

LlamaIndex provides the ability to customize the following components:

- LLM
- Prompts
- Embedding model

These are described in their respective guides below.

#### 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,
    LLMPredictor,
    ServiceContext
)
from langchain import OpenAI

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

# define LLM
```

(continues on next page)

(continued from previous page)

```
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
```

(continues on next page)

(continued from previous page)

```
from transformers import pipeline
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 = new_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.

### 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)
```

(continues on next page)

(continued from previous page)

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

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

### 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

# 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",
```

(continues on next page)

(continued from previous page)

```
    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)
```

## 1.2.12 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.

### 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
)

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)
```

(continues on next page)

(continued from previous page)

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

Here is an example notebook.

### 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
playground.compare("What is the population of Berlin?")
```

### API Reference

*API Reference here*

### Example Notebook

Link to Example Notebook.

### 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.

```python
from gpt_index import GPTSimpleVectorIndex
from gpt_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.

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
```

(continues on next page)

(continued from previous page)

```
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*.

## 1.2.13 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`)

### 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()


# specify StructuredLLMPredictor
# this is a special LLMPredictor that allows for structured outputs

# define query / output spec
rail_spec = ("""
<rail version="0.1">
```

(continues on next page)

(continued from previous page)

```
<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'}]}
```

### 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␣
↪outside of school.'}
```

## 1.2.14 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.

### 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
- Qdrant (`QdrantReader`) Installation Python Client
- Weaviate (`WeaviateReader`). Installation. Python Client.
- Pinecone (`PineconeReader`). Installation/Quickstart.
- Faiss (`FaissReader`). Installation.

Chroma stores both documents and vectors. This is an example of how to use Chroma:

```python
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, ...]

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`

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?")
```

**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,
    metric="euclidean",
    pod_type="p1"
)
index = pinecone.Index("quickstart")
```

(continues on next page)

(continued from previous page)

```
# 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()
index = GPTChromaIndex.from_documents(documents, chroma_collection=chroma_collection)

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

Example notebooks can be found here.

### 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"
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)
```

(continues on next page)

(continued from previous page)

```
    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"
)
```

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()
```

(continues on next page)

(continued from previous page)

```
# 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.

### 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,
    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}
)
```

(continues on next page)

(continued from previous page)

```
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.

### 1.2.15 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.

### 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[*DocumentStore*] = 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*.

**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, *docstore: Optional[*DocumentStore*] = None*, *required_keywords: Optional[List[str]] = None*, *exclude_keywords: Optional[List[str]] = None*, *response_mode: ResponseMode = ResponseMode.DEFAULT*, *text_qa_template: Optional[*QuestionAnswerPrompt*] = None*, *refine_template: Optional[*RefinePrompt*] = None*, *include_summary: bool = False*, *response_kwargs: Optional[Dict] = None*, *similarity_cutoff: Optional[float] = None*, *use_async: bool = False*, *streaming: bool = False*, *doc_ids: Optional[List[str]] = None*, *optimizer: Optional[BaseTokenUsageOptimizer] = None*, *node_postprocessors: Optional[List[BaseNodePostprocessor]] = None*, *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.

## 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:** *DocumentStore*

Get the docstore corresponding to the index.

**classmethod from_documents**(*documents: Sequence[*Document*]*, *docstore: Optional[*DocumentStore*] = 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*.

**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:** *DocumentStore*

Get the docstore corresponding to the index.

**classmethod from_documents**(*documents: Sequence[Document]*, *docstore: Optional[DocumentStore] = 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*.

`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**: ***DocumentStore***

Get the docstore corresponding to the index.

classmethod **from_documents**(*documents: Sequence[Document], docstore: Optional[DocumentStore] = 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*.

**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

## 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:`** *`DocumentStore`*

Get the docstore corresponding to the index.

**`classmethod from_documents`**(*documents: Sequence[*Document*]*, *docstore: Optional[*DocumentStore*] = 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*.

**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*, *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*, *docstore: Optional[DocumentStore] = None*, *required_keywords: Optional[List[str]] = None*, *exclude_keywords: Optional[List[str]] = None*, *response_mode: ResponseMode = ResponseMode.DEFAULT*, *text_qa_template: Optional[QuestionAnswerPrompt] = None*, *refine_template: Optional[RefinePrompt] = None*, *include_summary: bool = False*, *response_kwargs: Optional[Dict] = None*, *similarity_cutoff: Optional[float] = None*, *use_async: bool = False*, *streaming: bool = False*, *doc_ids: Optional[List[str]] = None*, *optimizer: Optional[BaseTokenUsageOptimizer] = None*, *node_postprocessors: Optional[List[BaseNodePostprocessor]] = None*, *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.

## 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_embedding: List[float]*, *similarity_top_k: int*, *doc_ids: Optional[List[str]] = None*, *query_str: Optional[str] = None*) → 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_embedding: List[float]*, *similarity_top_k: int*, *doc_ids: Optional[List[str]] = None*, *query_str: Optional[str] = None*) → 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*)

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_embedding: List[float]*, *similarity_top_k: int*, *doc_ids: Optional[List[str]] = None*, *query_str: Optional[str] = None*) → 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.**OpensearchVectorClient**(*endpoint: str*, *index: str*, *dim: int*, *embedding_field: str = 'embedding'*, *text_field: str = 'content'*, *method: 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_embedding: List[float]*, *similarity_top_k: int*, *doc_ids: Optional[List[str]] = None*, *query_str: Optional[str] = None*) → 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, 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*)

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**(*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_embedding: List[float], similarity_top_k: int, doc_ids: Optional[List[str]] = None, query_str: Optional[str] = None*) → 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_embedding: List[float]*, *similarity_top_k: int*, *doc_ids: Optional[List[str]] = None*, *query_str: Optional[str] = None*) → 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.**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_embedding: List[float]*, *similarity_top_k: int*, *doc_ids: Optional[List[str]] = None*, *query_str: Optional[str] = None*) → 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_embedding: List[float]*, *similarity_top_k: int*, *doc_ids: Optional[List[str]] = None*, *query_str: Optional[str] = None*) → 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

## 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*, *text_qa_template: Optional[QuestionAnswerPrompt] = None*, *vector_store: Optional[VectorStore] = None*, *use_async: bool = False*, ***kwargs: Any*)

Base GPT Vector Store Index.

**Parameters**

- **text_qa_template** (*Optional[QuestionAnswerPrompt]*) – A Question-Answer Prompt (see *Prompt Templates*). NOTE: this is a deprecated field.
- **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[*DocumentStore*] = 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*.

**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[ChatGPTRetrievalPlug*
*= None*, *service_context: Optional[*ServiceContext*] = None*, *text_qa_template: Optional[*QuestionAnswerPrompt
*= None*, *endpoint_url: Optional[str] = None*, *bearer_token: Optional[str] = None*, *retries: Optional[Retry] = None*, *batch_size: int = 100*, ***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**

- **`text_qa_template`** (*`Optional[QuestionAnswerPrompt]`*) – A Question-Answer Prompt (see *Prompt Templates*). NOTE: this is a deprecated field.
- **`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[*DocumentStore*] = 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*.

`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*, *text_qa_template: Optional[*QuestionAnswerPrompt*] = 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**

- **text_qa_template** (*Optional[*QuestionAnswerPrompt*]*) – A Question-Answer Prompt (see *Prompt Templates*). NOTE: this is a deprecated field.
- **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[*DocumentStore*] = 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*.

**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*, *text_qa_template: Optional[QuestionAnswerPrompt] = 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**

- **text_qa_template** (*Optional[QuestionAnswerPrompt]*) – A Question-Answer Prompt (see *Prompt Templates*). NOTE: this is a deprecated field.
- **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[DocumentStore] = 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` – 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*.

**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.GPTOpensearchIndex`(*nodes: Optional[Sequence[Node]] = None*, *service_context: Optional[ServiceContext] = None*, *client: Optional[OpensearchVectorClient] = None*, *index_struct: Optional[IndexDict] = None*, *text_qa_template: Optional[QuestionAnswerPrompt] = 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**

- **text_qa_template** (*Optional[QuestionAnswerPrompt]*) – A Question-Answer Prompt (see *Prompt Templates*). NOTE: this is a deprecated field.
- **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[*DocumentStore*] = 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*.

**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*, *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*, *text_qa_template: Optional[*QuestionAnswerPrompt*] = None*, *service_context: Optional[*ServiceContext*] = 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**

- **`text_qa_template`** (*`Optional[QuestionAnswerPrompt]`*) – A Question-Answer Prompt (see *Prompt Templates*). NOTE: this is a deprecated field.
- **`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[*DocumentStore*] = 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*.

**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*, *text_qa_template: Optional[QuestionAnswerPrompt] = 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**

- **`text_qa_template`** (*`Optional[QuestionAnswerPrompt]`*) – A Question-Answer Prompt (see *Prompt Templates*). NOTE: this is a deprecated field.

- **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[*DocumentStore*] = 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*.

**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, text_qa_template: Optional[QuestionAnswerPrompt] = None, simple_vector_store_data_dict: Optional[dict] = 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**

- `text_qa_template` (*Optional[QuestionAnswerPrompt]*) – A Question-Answer Prompt (see *Prompt Templates*). NOTE: this is a deprecated field.
- `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[DocumentStore] = 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*.

`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*, *text_qa_template: Optional[QuestionAnswerPrompt] = 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**

- **text_qa_template** (*Optional[QuestionAnswerPrompt]*) – A Question-Answer Prompt (see *Prompt Templates*). NOTE: this is a deprecated field.

- **`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[*DocumentStore*] = 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*.

**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

## 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.

**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")
```

**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[*DocumentStore*] = 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*.

**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[*DocumentStore*] = 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*.

**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

## 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[*DocumentStore*] = 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*.

`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"

## 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[*DocumentStore*] = 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*.

**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.

**synthesize**(*query_bundle:* QueryBundle, *nodes: List[*NodeWithScore*]*, *additional_source_nodes: Optional[List[*NodeWithScore*]] = None*) → Union[*Response*, *StreamingResponse*]

Synthesize answer with relevant nodes.

## Base Index Class

Base index classes.

class gpt_index.indices.base.**BaseGPTIndex**(*nodes: Optional[Sequence[*Node*]] = None*, *index_struct: Optional[IS] = None*, *docstore: Optional[*DocumentStore*] = 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:** ***DocumentStore***

Get the docstore corresponding to the index.

**classmethod from_documents**(*documents: Sequence[*Document*], docstore: Optional[*DocumentStore*] = 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*.

**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

## 1.2.16 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.

### Querying a List Index

Default query for GPTListIndex.

**class** `gpt_index.indices.list.query.BaseGPTListIndexQuery`(*index_struct: IS*, *service_context:* ServiceContext, *docstore: Optional[*DocumentStore*] = None*, *required_keywords: Optional[List[str]] = None*, *exclude_keywords: Optional[List[str]] = None*, *response_mode: ResponseMode = ResponseMode.DEFAULT*, *text_qa_template: Optional[*QuestionAnswerPrompt*] = None*, *refine_template: Optional[*RefinePrompt*] = None*, *include_summary: bool = False*, *response_kwargs: Optional[Dict] = None*, *similarity_cutoff: Optional[float] = None*, *use_async: bool = False*, *streaming: bool = False*, *doc_ids: Optional[List[str]] = None*, *optimizer: Optional[BaseTokenUsageOptimizer] = None*, *node_postprocessors: Optional[List[BaseNodePostprocessor]] = None*, *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*, *docstore: Optional[DocumentStore] = None*, *required_keywords: Optional[List[str]] = None*, *exclude_keywords: Optional[List[str]] = None*, *response_mode: ResponseMode = ResponseMode.DEFAULT*, *text_qa_template: Optional[QuestionAnswerPrompt] = None*, *refine_template: Optional[RefinePrompt] = None*, *include_summary: bool = False*, *response_kwargs: Optional[Dict] = None*, *similarity_cutoff: Optional[float] = None*, *use_async: bool = False*, *streaming: bool = False*, *doc_ids: Optional[List[str]] = None*, *optimizer: Optional[BaseTokenUsageOptimizer] = None*, *node_postprocessors: Optional[List[BaseNodePostprocessor]] = None*, *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.

### 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.

## Querying a Tree Index

Leaf query mechanism.

**class** gpt_index.indices.tree.leaf_query.**GPTTreeIndexLeafQuery**(*index_struct: IndexGraph, query_template: Optional[TreeSelectPrompt] = 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.

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*, *docstore: Optional[DocumentStore] = None*, *required_keywords: Optional[List[str]] = None*, *exclude_keywords: Optional[List[str]] = None*, *response_mode: ResponseMode = ResponseMode.DEFAULT*, *text_qa_template: Optional[QuestionAnswerPrompt] = None*, *refine_template: Optional[RefinePrompt] = None*, *include_summary: bool = False*, *response_kwargs: Optional[Dict] = None*, *similarity_cutoff: Optional[float] = None*, *use_async: bool = False*, *streaming: bool = False*, *doc_ids: Optional[List[str]] = None*, *optimizer: Optional[BaseTokenUsageOptimizer] = None*, *node_postprocessors: Optional[List[BaseNodePostprocessor]] = None*, *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*, *num_children: int = 10*, ***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.

## 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, **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

Vector-store specific query classes.

`class gpt_index.indices.vector_store.queries.`**`ChatGPTRetrievalPluginQuery`**`(`*index_struct: IndexDict*, *endpoint_url: str*, *bearer_token: Optional[str] = None*, *retries: Optional[Retry] = None*, *batch_size: int = 100*, ***kwargs: Any*`)`

GPT retrieval plugin query.

**Parameters**

- **`text_qa_template`** (`Optional[QuestionAnswerPrompt]`) – A Question-Answer Prompt (see *Prompt Templates*).
- **`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.

`class gpt_index.indices.vector_store.queries.`**`GPTChromaIndexQuery`**`(`*index_struct: IndexDict*, *chroma_collection: Optional[Any] = None*, ***kwargs: Any*`)`

GPT Chroma vector index query.

**Parameters**

- **`text_qa_template`** (`Optional[QuestionAnswerPrompt]`) – A Question-Answer Prompt (see *Prompt Templates*).
- **`embed_model`** (`Optional[BaseEmbedding]`) – Embedding model to use for embedding similarity.
- **`chroma_collection`** (`Optional[Any]`) – Collection instance from *chromadb* package.

**`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.vector_store.queries.`**`GPTFaissIndexQuery`**`(`*index_struct: IndexDict*, *faiss_index: Optional[Any] = None*, ***kwargs: Any*`)`

GPT faiss vector index query.

**Parameters**

- **`embed_model`** (`Optional[BaseEmbedding]`) – embedding model

- **similarity_top_k** (*int*) – number of top k results to return
- **faiss_index** (*faiss.Index*) – A Faiss Index object (required). Note: the index will be reset during index construction.

**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.vector_store.queries.**GPTOpensearchIndexQuery**(*index_struct: IndexDict*, *client: Optional[OpensearchVectorClient] = None*, ***kwargs: Any*)

GPT Opensearch vector index query.

**Parameters**

- **text_qa_template** (*Optional[QuestionAnswerPrompt]*) – A Question-Answer Prompt (see *Prompt Templates*).
- **embed_model** (*Optional[BaseEmbedding]*) – Embedding model to use for embedding similarity.
- **client** (*Optional[OpensearchVectorClient]*) – Opensearch vector client.

**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.vector_store.queries.**GPTPineconeIndexQuery**(*index_struct: IndexDict*, *pinecone_index: Optional[Any] = 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*, ***kwargs: Any*)

GPT pinecone vector index query.

**Parameters**

- **embed_model** (*Optional[BaseEmbedding]*) – embedding model
- **similarity_top_k** (*int*) – number of top k results to return
- **pinecone_index** (*Optional[pinecone.Index]*) – Pinecone index instance
- **pinecone_kwargs** (*Optional[dict]*) – Pinecone index kwargs

**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.vector_store.queries.**GPTQdrantIndexQuery**(*index_struct: IndexDict*, *client: Optional[Any] = None*, *collection_name: Optional[str] = None*, ***kwargs: Any*)

GPT Qdrant vector index query.

**Parameters**

- **embed_model** (*Optional[BaseEmbedding]*) – embedding model
- **similarity_top_k** (*int*) – number of top k results to return
- **client** (*Optional[Any]*) – QdrantClient instance from *qdrant-client* package
- **collection_name** – (Optional[str]): name of the Qdrant collection

**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.vector_store.queries.**GPTSimpleVectorIndexQuery**(*index_struct: IndexDict*, *simple_vector_store_data_dict: Optional[Dict] = None*, ***kwargs: Any*)

GPT simple vector index query.

**Parameters**

- **embed_model** (*Optional[BaseEmbedding]*) – embedding model
- **similarity_top_k** (*int*) – number of top k results to return
- **simple_vector_store_data_dict** – (Optional[dict]): simple vector store data dict,

**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.vector_store.queries.**GPTWeaviateIndexQuery**(*index_struct: IndexDict*, *weaviate_client: Optional[Any] = None*, *class_prefix: Optional[str] = None*, ***kwargs: Any*)

GPT Weaviate vector index query.

**Parameters**

- **embed_model** (*Optional[BaseEmbedding]*) – embedding model
- **similarity_top_k** (*int*) – number of top k results to return
- **weaviate_client** (*Optional[Any]*) – Weaviate client instance

- **class_prefix** (*Optional[str]*) – Weaviate class prefix

**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.

## 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")
```

**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.

## 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"

### 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.

**synthesize**(*query_bundle:* QueryBundle, *nodes: List[*NodeWithScore*]*, *additional_source_nodes: Optional[List[*NodeWithScore*]] = None*) → Union[*Response*, *StreamingResponse*]

Synthesize answer with relevant nodes.

This section shows how to define a query config in order to recursively query multiple indices that are composed together.

### 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"

**QDRANT**

Qdrant Vector Store Index. See *Vector Store Index* for more information on the Qdrant vector store index.

**Type**
"qdrant"

**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"

`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"

## Base Query Class

Base query classes.

**class** `gpt_index.indices.query.base.`**`BaseGPTIndexQuery`**(*index_struct: IS*, *service_context:* ServiceContext, *docstore: Optional[*DocumentStore*] = None*, *required_keywords: Optional[List[str]] = None*, *exclude_keywords: Optional[List[str]] = None*, *response_mode: ResponseMode = ResponseMode.DEFAULT*, *text_qa_template: Optional[*QuestionAnswerPrompt*] = None*, *refine_template: Optional[*RefinePrompt*] = None*, *include_summary: bool = False*, *response_kwargs: Optional[Dict] = None*, *similarity_cutoff: Optional[float] = None*, *use_async: bool = False*, *streaming: bool = False*, *doc_ids: Optional[List[str]] = None*, *optimizer: Optional[BaseTokenUsageOptimizer] = None*, *node_postprocessors: Optional[List[BaseNodePostprocessor]] = None*, *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).

- **required_keywords** (*List[str]*) – Optional list of keywords that must be present in nodes. Can be used to query most indices (tree index is an exception).
- **exclude_keywords** (*List[str]*) – Optional list of keywords that must not be present in nodes. Can be used to query most indices (tree index is an exception).
- **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.
- **include_summary** (*bool*) – Optional bool. If True, will also use the summary text of the index when generating a response (the summary text can be set through *index.set_text("<text>")*).
- **similarity_cutoff** (*float*) – Optional float. If set, will filter out nodes with similarity below this cutoff threshold when computing the response
- **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.

## 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[StepDecomposeQueryTransforn* *= 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.

## 1.2.17 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.

`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, str] = <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, str]`*) – The relationships of the node.

**`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_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 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*)

### 1.2.18 Docstore

Document store.

**class** gpt_index.docstore.**DocumentStore**(*docs: ~typing.Dict[str, ~gpt_index.schema.BaseDocument] = <factory>, ref_doc_info: ~typing.Dict[str, ~typing.Dict[str, ~typing.Any]] = <factory>*)

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 = DocumentStore()
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*) → Optional[BaseDocument]

Delete a document from the store.

`document_exists`(*doc_id: str*) → bool

Check if document exists.

`classmethod from_documents`(*docs: Sequence[BaseDocument]*, *allow_update: bool = True*) → *DocumentStore*

Create from documents.

**Parameters**

- **docs** (*List[BaseDocument]*) – documents
- **allow_update** (*bool*) – allow update of docstore from document with same doc_id.

`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

**classmethod load_from_dict**(*docs_dict: Dict[str, Any]*) → *DocumentStore*

Load from dict.

**Parameters**
**docs_dict** (*Dict[str, Any]*) – dict of documents

**classmethod merge**(*docstores: Sequence[DocumentStore]*) → *DocumentStore*

Merge docstores.

**Parameters**
**docstores** (*List[DocumentStore]*) – docstores to merge

**serialize_to_dict**() → Dict[str, Any]

Serialize to dict.

**set_document_hash**(*doc_id: str*, *doc_hash: str*) → None

Set the hash for a given doc_id.

**update_docstore**(*other: DocumentStore*) → None

Update docstore.

**Parameters**
**other** (*DocumentStore*) – docstore to update from

## 1.2.19 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: DocumentStore*, *service_context: Optional[ServiceContext] = None*)

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.

NOTE: this is mostly syntactic sugar, roughly equivalent to directly calling *ComposableGraph.from_indices*.

**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[DocumentStore] = 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** (`DocumentStore`) – A DocumentStore 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.

### 1.2.20 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.**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.**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**(*host: str*, *port: 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*)

Qdrant reader.

Retrieve documents from existing Qdrant collections.

**Parameters**

- **host** – Host name of Qdrant service.
- **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

**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*, *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.
- **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.

## 1.2.21 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.

### 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.

### 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.

## 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.

## 1.2.22 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.

### 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_embeddding**(*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_embeddding**(*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.

## 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.

## 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: List[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.

## 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

### 1.2.23 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.

### 1.2.24 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.

## 1.2.25 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.

## 1.2.26 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.

## 1.2.27 Node Parser

Node parsers.

**class** gpt_index.node_parser.**NodeParser**(**args*, ***kwargs*)

Base interface for node parser.

**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*)

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

**get_nodes_from_documents**(*documents: Sequence[*Document*]*, *include_extra_info: bool = True*) → List[*Node*]

Parse document into nodes.

**Parameters**

- **documents** (*Sequence[Document]*) – documents to parse
- **include_extra_info** (*bool*) – whether to include extra info in nodes

### 1.2.28 Example Notebooks

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

Example notebooks are found here.

### 1.2.29 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*)

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*)

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[str] = 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.

## 1.2.30 App Showcase

Here is a sample of some of the incredible applications and tools built on top of LlamaIndex!

### 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]

### Algovera

Build AI workflows using building blocks. Many workflows built on top of LlamaIndex.

[Website].

### ChatGPT LlamaIndex

Interface that allows users to upload long docs and chat with the bot. [Tweet thread]

### AgentHQ

A web tool to build agents, interacting with LlamaIndex data structures.[Website]

### 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]

### 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].

### PaperBrain

A platform to access/understand research papers.

[Tweet thread].

### CACTUS

Contextual search on top of LinkedIn search results. [LinkedIn post here].

### Personal Note Chatbot

A chatbot that can answer questions over a directory of Obsidian notes. [Tweet thread].

### RHOBH AMA

Ask questions about the Real Housewives of Beverly Hills. [Tweet thread] [Website]

### Mynd

A journaling app that uses AI to uncover insights and patterns over time. [Website]

### AI-X by OpenExO

Your Digital Transformation Co-Pilot [Website]

### Blackmaria

Python package for webscraping in Natural language. [Tweet thread] [Github]

# PYTHON MODULE INDEX

g

gpt_index.composability, 175
gpt_index.data_structs.node_v2, 172
gpt_index.data_structs.struct_type, 165
gpt_index.docstore, 173
gpt_index.embeddings.langchain, 204
gpt_index.embeddings.openai, 202
gpt_index.indices.base, 146
gpt_index.indices.common.struct_store.base, 211
gpt_index.indices.empty, 144
gpt_index.indices.empty.query, 163
gpt_index.indices.keyword_table, 84
gpt_index.indices.keyword_table.query, 150
gpt_index.indices.knowledge_graph, 140
gpt_index.indices.knowledge_graph.query, 162
gpt_index.indices.list, 81
gpt_index.indices.list.query, 149
gpt_index.indices.prompt_helper, 206
gpt_index.indices.query.base, 167
gpt_index.indices.query.query_transform, 170
gpt_index.indices.query.schema, 168
gpt_index.indices.service_context, 208
gpt_index.indices.struct_store, 132
gpt_index.indices.struct_store.container_builder, 210
gpt_index.indices.struct_store.pandas_query, 161
gpt_index.indices.struct_store.sql_query, 160
gpt_index.indices.tree, 93
gpt_index.indices.tree.embedding_query, 153
gpt_index.indices.tree.leaf_query, 153
gpt_index.indices.tree.retrieve_query, 154
gpt_index.indices.tree.summarize_query, 155
gpt_index.indices.vector_store.base, 104
gpt_index.indices.vector_store.base_query, 156
gpt_index.indices.vector_store.queries, 157
gpt_index.indices.vector_store.vector_indices, 107
gpt_index.langchain_helpers.agents, 215
gpt_index.langchain_helpers.chain_wrapper, 205
gpt_index.langchain_helpers.memory_wrapper, 221
gpt_index.langchain_helpers.sql_wrapper, 209
gpt_index.logger, 207
gpt_index.node_parser, 214
gpt_index.optimization, 209
gpt_index.playground.base, 213
gpt_index.prompts, 201
gpt_index.prompts.prompts, 190
gpt_index.readers, 177
gpt_index.response.schema, 212
gpt_index.token_counter.mock_chain_wrapper, 205
gpt_index.vector_stores, 99

# INDEX

## A

add() (*gpt_index.vector_stores.ChatGPTRetrievalPluginClient method*), 99
add() (*gpt_index.vector_stores.ChromaVectorStore method*), 99
add() (*gpt_index.vector_stores.FaissVectorStore method*), 100
add() (*gpt_index.vector_stores.OpensearchVectorStore method*), 101
add() (*gpt_index.vector_stores.PineconeVectorStore method*), 102
add() (*gpt_index.vector_stores.QdrantVectorStore method*), 102
add() (*gpt_index.vector_stores.SimpleVectorStore method*), 103
add() (*gpt_index.vector_stores.WeaviateVectorStore method*), 104
add_documents() (*gpt_index.docstore.DocumentStore method*), 173
add_log() (*gpt_index.logger.LlamaLogger method*), 207
add_node() (*gpt_index.indices.knowledge_graph.GPTKnowledgeGraphIndex method*), 141
aget_embedding() (*in module gpt_index.embeddings.openai*), 204
aget_embeddings() (*in module gpt_index.embeddings.openai*), 204
aget_queued_text_embeddings() (*gpt_index.embeddings.langchain.LangchainEmbedding method*), 204
aget_queued_text_embeddings() (*gpt_index.embeddings.openai.OpenAIEmbedding method*), 203
apredict() (*gpt_index.token_counter.mock_chain_wrapper.MockLLMPredictor method*), 205
aquery() (*gpt_index.composability.ComposableGraph method*), 175
aquery() (*gpt_index.indices.base.BaseGPTIndex method*), 147
aquery() (*gpt_index.indices.empty.GPTEmptyIndex method*), 144
aquery() (*gpt_index.indices.keyword_table.GPTKeywordTableIndex method*), 85
aquery() (*gpt_index.indices.keyword_table.GPTRAKEKeywordTableIndex method*), 88
aquery() (*gpt_index.indices.keyword_table.GPTSimpleKeywordTableInde method*), 91
aquery() (*gpt_index.indices.knowledge_graph.GPTKnowledgeGraphIndex method*), 141
aquery() (*gpt_index.indices.list.GPTListIndex method*), 81
aquery() (*gpt_index.indices.struct_store.GPTPandasIndex method*), 133
aquery() (*gpt_index.indices.struct_store.GPTSQLStructStoreIndex method*), 136
aquery() (*gpt_index.indices.tree.GPTTreeIndex method*), 94
aquery() (*gpt_index.indices.vector_store.base.GPTVectorStoreIndex method*), 104
aquery() (*gpt_index.indices.vector_store.vector_indices.ChatGPTRetrieva method*), 108
aquery() (*gpt_index.indices.vector_store.vector_indices.GPTChromaInde method*), 111
aquery() (*gpt_index.indices.vector_store.vector_indices.GPTFaissIndex method*), 114
aquery() (*gpt_index.indices.vector_store.vector_indices.GPTOpensearchI method*), 117
aquery() (*gpt_index.indices.vector_store.vector_indices.GPTPineconeInd method*), 121
aquery() (*gpt_index.indices.vector_store.vector_indices.GPTQdrantIndex method*), 124
aquery() (*gpt_index.indices.vector_store.vector_indices.GPTSimpleVector method*), 127
aquery() (*gpt_index.indices.vector_store.vector_indices.GPTWeaviateInd method*), 130
arun() (*gpt_index.langchain_helpers.agents.LlamaGraphTool method*), 217
arun() (*gpt_index.langchain_helpers.agents.LlamaIndexTool method*), 218

## B

BaseGPTIndex (*class in gpt_index.indices.base*), 146
BaseGPTIndexQuery (*class in gpt_index.indices.query.base*), 167
BaseGPTKeywordTableQuery (*class in*

*gpt_index.indices.keyword_table.query*), 150
BaseGPTListIndexQuery (*class in gpt_index.indices.list.query*), 149
BaseStructDatapointExtractor (*class in gpt_index.indices.common.struct_store.base*), 211
BeautifulSoupWebReader (*class in gpt_index.readers*), 177
build_all_context_from_documents() (*gpt_index.indices.common.struct_store.base.SQLDocumentContextBuilder method*), 212
build_context_container() (*gpt_index.indices.struct_store.container_builder.SQLContextContainerBuilder method*), 211
build_context_container() (*gpt_index.indices.struct_store.SQLContextContainerBuilder method*), 139
build_graph_from_documents() (*gpt_index.composability.QASummaryGraphBuilder method*), 177
build_table_context_from_documents() (*gpt_index.indices.common.struct_store.base.SQLDocumentContextBuilder method*), 212

## C

CHATGPT_RETRIEVAL_PLUGIN (*gpt_index.data_structs.struct_type.IndexStructType attribute*), 167
ChatGPTRetrievalPluginClient (*class in gpt_index.vector_stores*), 99
ChatGPTRetrievalPluginIndex (*class in gpt_index.indices.vector_store.vector_indices*), 107
ChatGPTRetrievalPluginQuery (*class in gpt_index.indices.vector_store.queries*), 157
ChatGPTRetrievalPluginReader (*class in gpt_index.readers*), 177
CHROMA (*gpt_index.data_structs.struct_type.IndexStructType attribute*), 166
ChromaReader (*class in gpt_index.readers*), 178
ChromaVectorStore (*class in gpt_index.vector_stores*), 99
clear() (*gpt_index.langchain_helpers.memory_wrapper.GPTIndexChatMemory method*), 221
clear() (*gpt_index.langchain_helpers.memory_wrapper.GPTIndexMemory method*), 223
client (*gpt_index.vector_stores.ChatGPTRetrievalPluginClient property*), 99
client (*gpt_index.vector_stores.ChromaVectorStore property*), 99
client (*gpt_index.vector_stores.FaissVectorStore property*), 100
client (*gpt_index.vector_stores.OpensearchVectorStore property*), 101
client (*gpt_index.vector_stores.PineconeVectorStore property*), 102
client (*gpt_index.vector_stores.QdrantVectorStore property*), 103
client (*gpt_index.vector_stores.SimpleVectorStore property*), 103
client (*gpt_index.vector_stores.WeaviateVectorStore property*), 104
compact_text_chunks() (*gpt_index.indices.prompt_helper.PromptHelper method*), 207
compare() (*gpt_index.playground.base.Playground method*), 213
ComposableGraph (*class in gpt_index.composability*), 175
config_dict (*gpt_index.vector_stores.ChatGPTRetrievalPluginClient property*), 99
config_dict (*gpt_index.vector_stores.ChromaVectorStore property*), 99
config_dict (*gpt_index.vector_stores.FaissVectorStore property*), 100
config_dict (*gpt_index.vector_stores.OpensearchVectorStore property*), 101
config_dict (*gpt_index.vector_stores.PineconeVectorStore property*), 102
config_dict (*gpt_index.vector_stores.QdrantVectorStore property*), 103
config_dict (*gpt_index.vector_stores.SimpleVectorStore property*), 103
config_dict (*gpt_index.vector_stores.WeaviateVectorStore property*), 104
construct() (*gpt_index.langchain_helpers.agents.GraphToolConfig class method*), 215
construct() (*gpt_index.langchain_helpers.agents.IndexToolConfig class method*), 216
construct() (*gpt_index.langchain_helpers.agents.LlamaGraphTool class method*), 217
construct() (*gpt_index.langchain_helpers.agents.LlamaIndexTool class method*), 218
construct() (*gpt_index.langchain_helpers.agents.LlamaToolkit class method*), 219
construct() (*gpt_index.langchain_helpers.memory_wrapper.GPTIndexC class method*), 221
construct() (*gpt_index.langchain_helpers.memory_wrapper.GPTIndexM class method*), 223
copy() (*gpt_index.langchain_helpers.agents.GraphToolConfig method*), 215
copy() (*gpt_index.langchain_helpers.agents.IndexToolConfig method*), 216
copy() (*gpt_index.langchain_helpers.agents.LlamaGraphTool method*), 217
copy() (*gpt_index.langchain_helpers.agents.LlamaIndexTool*

*method*), 218
copy() (*gpt_index.langchain_helpers.agents.LlamaToolkit method*), 219
copy() (*gpt_index.langchain_helpers.memory_wrapper.GPTIndexChatMemory method*), 221
copy() (*gpt_index.langchain_helpers.memory_wrapper.GPTIndexMemory method*), 223
create_documents() (*gpt_index.readers.ChromaReader method*), 178
create_llama_agent() (*in module gpt_index.langchain_helpers.agents*), 220
create_llama_chat_agent() (*in module gpt_index.langchain_helpers.agents*), 220

## D

DecomposeQueryTransform (*class in gpt_index.indices.query.query_transform*), 170
DEFAULT (*gpt_index.indices.query.schema.QueryMode attribute*), 165, 170
default_output_processor() (*in module gpt_index.indices.struct_store.pandas_query*), 162
delete() (*gpt_index.indices.base.BaseGPTIndex method*), 147
delete() (*gpt_index.indices.keyword_table.GPTKeywordTableIndex method*), 85
delete() (*gpt_index.indices.keyword_table.GPTRAKEKeywordTableIndex method*), 88
delete() (*gpt_index.indices.keyword_table.GPTSimpleKeywordTableIndex method*), 91
delete() (*gpt_index.indices.tree.GPTTreeIndex method*), 94
delete_doc_id() (*gpt_index.vector_stores.OpensearchVectorClient method*), 101
delete_document() (*gpt_index.docstore.DocumentStore method*), 174
derive_index_from_context() (*gpt_index.indices.struct_store.container_builder.SQLContextContainerBuilder method*), 211
derive_index_from_context() (*gpt_index.indices.struct_store.SQLContextContainerBuilder method*), 139
dialect (*gpt_index.langchain_helpers.sql_wrapper.SQLDatabase property*), 209
DICT (*gpt_index.data_structs.struct_type.IndexStructType attribute*), 166
dict() (*gpt_index.langchain_helpers.agents.GraphToolConfig method*), 215
dict() (*gpt_index.langchain_helpers.agents.IndexToolConfig method*), 216
dict() (*gpt_index.langchain_helpers.agents.LlamaGraphTool method*), 217
dict() (*gpt_index.langchain_helpers.agents.LlamaIndexTool method*), 218
dict() (*gpt_index.langchain_helpers.agents.LlamaToolkit method*), 219
dict() (*gpt_index.langchain_helpers.memory_wrapper.GPTIndexChatMe method*), 222
dict() (*gpt_index.langchain_helpers.memory_wrapper.GPTIndexMemory method*), 223
DiscordReader (*class in gpt_index.readers*), 178
do_approx_knn() (*gpt_index.vector_stores.OpensearchVectorClient method*), 101
docstore (*gpt_index.indices.base.BaseGPTIndex property*), 147
docstore (*gpt_index.indices.keyword_table.GPTKeywordTableIndex property*), 85
docstore (*gpt_index.indices.keyword_table.GPTRAKEKeywordTableIndex property*), 89
docstore (*gpt_index.indices.keyword_table.GPTSimpleKeywordTableInde property*), 91
docstore (*gpt_index.indices.tree.GPTTreeIndex property*), 94
Document (*class in gpt_index.readers*), 179
document_exists() (*gpt_index.docstore.DocumentStore method*), 174
DocumentRelationship (*class in gpt_index.data_structs.node_v2*), 172
DocumentStore (*class in gpt_index.docstore*), 173

## E

ElasticsearchReader (*class in gpt_index.readers*), 179
EMBEDDING (*gpt_index.indices.knowledge_graph.KGQueryMode attribute*), 143
EMBEDDING (*gpt_index.indices.knowledge_graph.query.KGQueryMode attribute*), 163
EMBEDDING (*gpt_index.indices.query.schema.QueryMode attribute*), 165, 170
embedding_strs (*gpt_index.indices.query.schema.QueryBundle property*), 168
engine (*gpt_index.langchain_helpers.sql_wrapper.SQLDatabase property*), 209
extra_info_str (*gpt_index.data_structs.node_v2.Node property*), 172
extra_info_str (*gpt_index.readers.Document property*), 179

## F

FaissReader (*class in gpt_index.readers*), 180
FaissVectorStore (*class in gpt_index.vector_stores*), 100
format() (*gpt_index.prompts.Prompt method*), 202
format() (*gpt_index.prompts.prompts.KeywordExtractPrompt method*), 191

format() (*gpt_index.prompts.prompts.KnowledgeGraphPrompt method*), 191
format() (*gpt_index.prompts.prompts.PandasPrompt method*), 192
format() (*gpt_index.prompts.prompts.QueryKeywordExtractPrompt method*), 193
format() (*gpt_index.prompts.prompts.QuestionAnswerPrompt method*), 194
format() (*gpt_index.prompts.prompts.RefinePrompt method*), 194
format() (*gpt_index.prompts.prompts.RefineTableContextPrompt method*), 195
format() (*gpt_index.prompts.prompts.SchemaExtractPrompt method*), 196
format() (*gpt_index.prompts.prompts.SimpleInputPrompt method*), 196
format() (*gpt_index.prompts.prompts.SummaryPrompt method*), 197
format() (*gpt_index.prompts.prompts.TableContextPrompt method*), 198
format() (*gpt_index.prompts.prompts.TextToSQLPrompt method*), 199
format() (*gpt_index.prompts.prompts.TreeInsertPrompt method*), 199
format() (*gpt_index.prompts.prompts.TreeSelectMultiplePrompt method*), 200
format() (*gpt_index.prompts.prompts.TreeSelectPrompt method*), 201
from_defaults() (*gpt_index.indices.service_context.ServiceContext class method*), 208
from_docs() (*gpt_index.playground.base.Playground class method*), 213
from_documents() (*gpt_index.docstore.DocumentStore class method*), 174
from_documents() (*gpt_index.indices.base.BaseGPTIndex class method*), 147
from_documents() (*gpt_index.indices.empty.GPTEmptyIndex class method*), 144
from_documents() (*gpt_index.indices.keyword_table.GPTKeywordTableIndex class method*), 85
from_documents() (*gpt_index.indices.keyword_table.GPTRAKEKeywordTableIndex class method*), 89
from_documents() (*gpt_index.indices.keyword_table.GPTSimpleKeywordTableIndex class method*), 91
from_documents() (*gpt_index.indices.knowledge_graph.GPTKnowledgeGraphIndex class method*), 141
from_documents() (*gpt_index.indices.list.GPTListIndex class method*), 81
from_documents() (*gpt_index.indices.struct_store.container_builder.SQLContextContainerBuilder class method*), 211
from_documents() (*gpt_index.indices.struct_store.GPTPandasIndex class method*), 134
from_documents() (*gpt_index.indices.struct_store.GPTSQLStructStoreIndex class method*), 136
from_documents() (*gpt_index.indices.struct_store.SQLContextContainer class method*), 139
from_documents() (*gpt_index.indices.tree.GPTTreeIndex class method*), 94
from_documents() (*gpt_index.indices.vector_store.base.GPTVectorStore class method*), 105
from_documents() (*gpt_index.indices.vector_store.vector_indices.ChatG class method*), 109
from_documents() (*gpt_index.indices.vector_store.vector_indices.GPTCh class method*), 111
from_documents() (*gpt_index.indices.vector_store.vector_indices.GPTFa class method*), 114
from_documents() (*gpt_index.indices.vector_store.vector_indices.GPTOp class method*), 118
from_documents() (*gpt_index.indices.vector_store.vector_indices.GPTPi class method*), 121
from_documents() (*gpt_index.indices.vector_store.vector_indices.GPTQd class method*), 124
from_documents() (*gpt_index.indices.vector_store.vector_indices.GPTSi class method*), 127
from_documents() (*gpt_index.indices.vector_store.vector_indices.GPTWe class method*), 130
from_indices() (*gpt_index.composability.ComposableGraph class method*), 175
from_langchain_format() (*gpt_index.readers.Document class method*), 179
from_langchain_prompt() (*gpt_index.prompts.Prompt class method*), 202
from_langchain_prompt() (*gpt_index.prompts.prompts.KeywordExtractPrompt class method*), 191
from_langchain_prompt() (*gpt_index.prompts.prompts.KnowledgeGraphPrompt class method*), 191
from_langchain_prompt() (*gpt_index.prompts.prompts.PandasPrompt class method*), 192
from_langchain_prompt() (*gpt_index.prompts.prompts.QueryKeywordExtractPrompt class method*), 193
from_langchain_prompt() (*gpt_index.prompts.prompts.QuestionAnswerPrompt class method*), 194
from_langchain_prompt() (*gpt_index.prompts.prompts.RefinePrompt class method*), 194
from_langchain_prompt() (*gpt_index.prompts.prompts.RefineTableContextPrompt class method*), 195
from_langchain_prompt() (*gpt_index.prompts.prompts.SchemaExtractPrompt class method*), 196

from_langchain_prompt() (*gpt_index.prompts.prompts.SimpleInputPrompt class method*), 196
from_langchain_prompt() (*gpt_index.prompts.prompts.SummaryPrompt class method*), 197
from_langchain_prompt() (*gpt_index.prompts.prompts.TableContextPrompt class method*), 198
from_langchain_prompt() (*gpt_index.prompts.prompts.TextToSQLPrompt class method*), 199
from_langchain_prompt() (*gpt_index.prompts.prompts.TreeInsertPrompt class method*), 199
from_langchain_prompt() (*gpt_index.prompts.prompts.TreeSelectMultiplePrompt class method*), 200
from_langchain_prompt() (*gpt_index.prompts.prompts.TreeSelectPrompt class method*), 201
from_langchain_prompt_selector() (*gpt_index.prompts.Prompt class method*), 202
from_langchain_prompt_selector() (*gpt_index.prompts.prompts.KeywordExtractPrompt class method*), 191
from_langchain_prompt_selector() (*gpt_index.prompts.prompts.KnowledgeGraphPrompt class method*), 191
from_langchain_prompt_selector() (*gpt_index.prompts.prompts.PandasPrompt class method*), 192
from_langchain_prompt_selector() (*gpt_index.prompts.prompts.QueryKeywordExtractPrompt class method*), 193
from_langchain_prompt_selector() (*gpt_index.prompts.prompts.QuestionAnswerPrompt class method*), 194
from_langchain_prompt_selector() (*gpt_index.prompts.prompts.RefinePrompt class method*), 194
from_langchain_prompt_selector() (*gpt_index.prompts.prompts.RefineTableContextPrompt class method*), 195
from_langchain_prompt_selector() (*gpt_index.prompts.prompts.SchemaExtractPrompt class method*), 196
from_langchain_prompt_selector() (*gpt_index.prompts.prompts.SimpleInputPrompt class method*), 196
from_langchain_prompt_selector() (*gpt_index.prompts.prompts.SummaryPrompt class method*), 197
from_langchain_prompt_selector() (*gpt_index.prompts.prompts.TableContextPrompt class method*), 198
from_langchain_prompt_selector() (*gpt_index.prompts.prompts.TextToSQLPrompt class method*), 199
from_langchain_prompt_selector() (*gpt_index.prompts.prompts.TreeInsertPrompt class method*), 199
from_langchain_prompt_selector() (*gpt_index.prompts.prompts.TreeSelectMultiplePrompt class method*), 200
from_langchain_prompt_selector() (*gpt_index.prompts.prompts.TreeSelectPrompt class method*), 201
from_llm_predictor() (*gpt_index.indices.prompt_helper.PromptHelper class method*), 207
from_prompt() (*gpt_index.prompts.Prompt class method*), 202
from_prompt() (*gpt_index.prompts.prompts.KeywordExtractPrompt class method*), 191
from_prompt() (*gpt_index.prompts.prompts.KnowledgeGraphPrompt class method*), 191
from_prompt() (*gpt_index.prompts.prompts.PandasPrompt class method*), 192
from_prompt() (*gpt_index.prompts.prompts.QueryKeywordExtractPromp class method*), 193
from_prompt() (*gpt_index.prompts.prompts.QuestionAnswerPrompt class method*), 194
from_prompt() (*gpt_index.prompts.prompts.RefinePrompt class method*), 194
from_prompt() (*gpt_index.prompts.prompts.RefineTableContextPrompt class method*), 195
from_prompt() (*gpt_index.prompts.prompts.SchemaExtractPrompt class method*), 196
from_prompt() (*gpt_index.prompts.prompts.SimpleInputPrompt class method*), 197
from_prompt() (*gpt_index.prompts.prompts.SummaryPrompt class method*), 197
from_prompt() (*gpt_index.prompts.prompts.TableContextPrompt class method*), 198
from_prompt() (*gpt_index.prompts.prompts.TextToSQLPrompt class method*), 199
from_prompt() (*gpt_index.prompts.prompts.TreeInsertPrompt class method*), 199
from_prompt() (*gpt_index.prompts.prompts.TreeSelectMultiplePrompt class method*), 200
from_prompt() (*gpt_index.prompts.prompts.TreeSelectPrompt class method*), 201
from_tool_config() (*gpt_index.langchain_helpers.agents.LlamaGraphT class method*), 217
from_tool_config() (*gpt_index.langchain_helpers.agents.LlamaIndexTo class method*), 218

from_uri() (*gpt_index.langchain_helpers.sql_wrapper.SQLDatabase class method*), 209

G

get() (*gpt_index.vector_stores.SimpleVectorStore method*), 103
get_agg_embedding_from_queries() (*gpt_index.embeddings.langchain.LangchainEmbedding method*), 204
get_agg_embedding_from_queries() (*gpt_index.embeddings.openai.OpenAIEmbedding method*), 203
get_biggest_prompt() (*gpt_index.indices.prompt_helper.PromptHelper method*), 207
get_chunk_size_given_prompt() (*gpt_index.indices.prompt_helper.PromptHelper method*), 207
get_doc_hash() (*gpt_index.data_structs.node_v2.Node method*), 172
get_doc_hash() (*gpt_index.readers.Document method*), 179
get_doc_id() (*gpt_index.data_structs.node_v2.Node method*), 172
get_doc_id() (*gpt_index.readers.Document method*), 179
get_document() (*gpt_index.docstore.DocumentStore method*), 174
get_document_hash() (*gpt_index.docstore.DocumentStore method*), 174
get_embedding() (*gpt_index.data_structs.node_v2.Node method*), 172
get_embedding() (*gpt_index.readers.Document method*), 179
get_embedding() (*in module gpt_index.embeddings.openai*), 204
get_embeddings() (*in module gpt_index.embeddings.openai*), 204
get_formatted_sources() (*gpt_index.response.schema.Response method*), 213
get_formatted_sources() (*gpt_index.response.schema.StreamingResponse method*), 213
get_index() (*gpt_index.composability.ComposableGraph method*), 175
get_langchain_prompt() (*gpt_index.prompts.Prompt method*), 202
get_langchain_prompt() (*gpt_index.prompts.prompts.KeywordExtractPrompt method*), 191
get_langchain_prompt() (*gpt_index.prompts.prompts.KnowledgeGraphPrompt method*), 192
get_langchain_prompt() (*gpt_index.prompts.prompts.PandasPrompt method*), 192
get_langchain_prompt() (*gpt_index.prompts.prompts.QueryKeywordExtractPrompt method*), 193
get_langchain_prompt() (*gpt_index.prompts.prompts.QuestionAnswerPrompt method*), 194
get_langchain_prompt() (*gpt_index.prompts.prompts.RefinePrompt method*), 195
get_langchain_prompt() (*gpt_index.prompts.prompts.RefineTableContextPrompt method*), 195
get_langchain_prompt() (*gpt_index.prompts.prompts.SchemaExtractPrompt method*), 196
get_langchain_prompt() (*gpt_index.prompts.prompts.SimpleInputPrompt method*), 197
get_langchain_prompt() (*gpt_index.prompts.prompts.SummaryPrompt method*), 197
get_langchain_prompt() (*gpt_index.prompts.prompts.TableContextPrompt method*), 198
get_langchain_prompt() (*gpt_index.prompts.prompts.TextToSQLPrompt method*), 199
get_langchain_prompt() (*gpt_index.prompts.prompts.TreeInsertPrompt method*), 200
get_langchain_prompt() (*gpt_index.prompts.prompts.TreeSelectMultiplePrompt method*), 200
get_langchain_prompt() (*gpt_index.prompts.prompts.TreeSelectPrompt method*), 201
get_llm_metadata() (*gpt_index.token_counter.mock_chain_wrapper.Mo method*), 205
get_logs() (*gpt_index.logger.LlamaLogger method*), 207
get_metadata() (*gpt_index.logger.LlamaLogger method*), 208
get_networkx_graph() (*gpt_index.indices.knowledge_graph.GPTKnowledgeGraphIndex method*), 141
get_node() (*gpt_index.docstore.DocumentStore method*), 174
get_node_dict() (*gpt_index.docstore.DocumentStore method*), 174
get_nodes() (*gpt_index.docstore.DocumentStore

*method*), 174

get_nodes_from_documents() (*gpt_index.node_parser.NodeParser method*), 214

get_nodes_from_documents() (*gpt_index.node_parser.SimpleNodeParser method*), 214

get_numbered_text_from_nodes() (*gpt_index.indices.prompt_helper.PromptHelper method*), 207

get_prompt_input_key() (*in module gpt_index.langchain_helpers.memory_wrapper*), 224

get_query_embedding() (*gpt_index.embeddings.langchain.LangchainEmbedding method*), 205

get_query_embedding() (*gpt_index.embeddings.openai.OpenAIEmbedding method*), 203

get_query_map() (*gpt_index.indices.base.BaseGPTIndex class method*), 147

get_query_map() (*gpt_index.indices.empty.GPTEmptyIndex class method*), 144

get_query_map() (*gpt_index.indices.keyword_table.GPTKeywordTableIndex class method*), 85

get_query_map() (*gpt_index.indices.keyword_table.GPTRAKEKeywordTableIndex class method*), 89

get_query_map() (*gpt_index.indices.keyword_table.GPTSimpleKeywordTableIndex class method*), 91

get_query_map() (*gpt_index.indices.knowledge_graph.GPTKnowledgeGraphIndex class method*), 141

get_query_map() (*gpt_index.indices.list.GPTListIndex class method*), 81

get_query_map() (*gpt_index.indices.struct_store.GPTPandasIndex class method*), 134

get_query_map() (*gpt_index.indices.struct_store.GPTSQLStructStoreIndex class method*), 136

get_query_map() (*gpt_index.indices.tree.GPTTreeIndex class method*), 94

get_query_map() (*gpt_index.indices.vector_store.base.GPTVectorStoreIndex class method*), 105

get_query_map() (*gpt_index.indices.vector_store.vector_indices.GPTChatGPTRetrievalPluginIndex class method*), 109

get_query_map() (*gpt_index.indices.vector_store.vector_indices.GPTChromaIndex class method*), 112

get_query_map() (*gpt_index.indices.vector_store.vector_indices.GPTFaissIndex class method*), 115

get_query_map() (*gpt_index.indices.vector_store.vector_indices.GPTOpensearchIndex class method*), 118

get_query_map() (*gpt_index.indices.vector_store.vector_indices.GPTPineconeIndex class method*), 121

get_query_map() (*gpt_index.indices.vector_store.vector_indices.GPTQdrantIndex class method*), 124

get_query_map() (*gpt_index.indices.vector_store.vector_indices.GPTSimpleVectorIndex class method*), 127

get_query_map() (*gpt_index.indices.vector_store.vector_indices.GPTWea class method*), 130

get_queued_text_embeddings() (*gpt_index.embeddings.langchain.LangchainEmbedding method*), 205

get_queued_text_embeddings() (*gpt_index.embeddings.openai.OpenAIEmbedding method*), 203

get_response() (*gpt_index.response.schema.StreamingResponse method*), 213

get_single_table_info() (*gpt_index.langchain_helpers.sql_wrapper.SQLDatabase method*), 209

get_table_columns() (*gpt_index.langchain_helpers.sql_wrapper.SQLDatabase method*), 209

get_table_info() (*gpt_index.langchain_helpers.sql_wrapper.SQLDatab method*), 209

get_table_info_no_throw() (*gpt_index.langchain_helpers.sql_wrapper.SQLDatabase method*), 209

get_table_names() (*gpt_index.langchain_helpers.sql_wrapper.SQLData method*), 210

get_text() (*gpt_index.data_structs.node_v2.Node method*), 172

get_text() (*gpt_index.readers.Document method*), 179

get_text_embedding() (*gpt_index.embeddings.langchain.LangchainEmbedding method*), 205

get_text_embedding() (*gpt_index.embeddings.openai.OpenAIEmbedding method*), 203

get_text_from_nodes() (*gpt_index.indices.prompt_helper.PromptHelper method*), 207

get_text_splitter_given_prompt() (*gpt_index.indices.prompt_helper.PromptHelper method*), 207

get_tools() (*gpt_index.langchain_helpers.agents.LlamaToolkit method*), 219

get_type() (*gpt_index.data_structs.node_v2.Node class method*), 172

get_type() (*gpt_index.readers.Document class method*), 179

get_types() (*gpt_index.data_structs.node_v2.Node class method*), 173

get_types() (*gpt_index.readers.Document class method*), 179

get_usable_table_names() (*gpt_index.langchain_helpers.sql_wrapper.SQLDatabase method*), 210

GithubRepositoryReader (*class in gpt_index.readers*), 180

GoogleDocsReader (*class in gpt_index.readers*), 181
gpt_index.composability
    module, 175
gpt_index.data_structs.node_v2
    module, 172
gpt_index.data_structs.struct_type
    module, 165
gpt_index.docstore
    module, 173
gpt_index.embeddings.langchain
    module, 204
gpt_index.embeddings.openai
    module, 202
gpt_index.indices.base
    module, 146
gpt_index.indices.common.struct_store.base
    module, 211
gpt_index.indices.empty
    module, 144
gpt_index.indices.empty.query
    module, 163
gpt_index.indices.keyword_table
    module, 84
gpt_index.indices.keyword_table.query
    module, 150
gpt_index.indices.knowledge_graph
    module, 140
gpt_index.indices.knowledge_graph.query
    module, 162
gpt_index.indices.list
    module, 81
gpt_index.indices.list.query
    module, 149
gpt_index.indices.prompt_helper
    module, 206
gpt_index.indices.query.base
    module, 167
gpt_index.indices.query.query_transform
    module, 170
gpt_index.indices.query.schema
    module, 164, 168
gpt_index.indices.service_context
    module, 208
gpt_index.indices.struct_store
    module, 132
gpt_index.indices.struct_store.container_builder
    module, 210
gpt_index.indices.struct_store.pandas_query
    module, 161
gpt_index.indices.struct_store.sql_query
    module, 160
gpt_index.indices.tree
    module, 93
gpt_index.indices.tree.embedding_query
    module, 153
gpt_index.indices.tree.leaf_query
    module, 153
gpt_index.indices.tree.retrieve_query
    module, 154
gpt_index.indices.tree.summarize_query
    module, 155
gpt_index.indices.vector_store.base
    module, 104
gpt_index.indices.vector_store.base_query
    module, 156
gpt_index.indices.vector_store.queries
    module, 157
gpt_index.indices.vector_store.vector_indices
    module, 107
gpt_index.langchain_helpers.agents
    module, 215
gpt_index.langchain_helpers.chain_wrapper
    module, 205
gpt_index.langchain_helpers.memory_wrapper
    module, 221
gpt_index.langchain_helpers.sql_wrapper
    module, 209
gpt_index.logger
    module, 207
gpt_index.node_parser
    module, 214
gpt_index.optimization
    module, 209
gpt_index.playground.base
    module, 213
gpt_index.prompts
    module, 201
gpt_index.prompts.prompts
    module, 190
gpt_index.readers
    module, 177
gpt_index.response.schema
    module, 212
gpt_index.token_counter.mock_chain_wrapper
    module, 205
gpt_index.vector_stores
    module, 99
GPTChromaIndex (*class in gpt_index.indices.vector_store.vector_indices*), 111
GPTChromaIndexQuery (*class in gpt_index.indices.vector_store.queries*), 157
GPTEmptyIndex (*class in gpt_index.indices.empty*), 144
GPTEmptyIndexQuery (*class in gpt_index.indices.empty*), 146
GPTEmptyIndexQuery (*class in gpt_index.indices.empty.query*), 163
GPTFaissIndex (*class in*

*gpt_index.indices.vector_store.vector_indices*), 114
GPTFaissIndexQuery (*class in gpt_index.indices.vector_store.queries*), 157
GPTIndexChatMemory (*class in gpt_index.langchain_helpers.memory_wrapper*), 221
GPTIndexChatMemory.Config (*class in gpt_index.langchain_helpers.memory_wrapper*), 221
GPTIndexMemory (*class in gpt_index.langchain_helpers.memory_wrapper*), 222
GPTIndexMemory.Config (*class in gpt_index.langchain_helpers.memory_wrapper*), 223
GPTKeywordTableGPTQuery (*class in gpt_index.indices.keyword_table*), 84
GPTKeywordTableGPTQuery (*class in gpt_index.indices.keyword_table.query*), 151
GPTKeywordTableIndex (*class in gpt_index.indices.keyword_table*), 84
GPTKeywordTableRAKEQuery (*class in gpt_index.indices.keyword_table*), 87
GPTKeywordTableRAKEQuery (*class in gpt_index.indices.keyword_table.query*), 152
GPTKeywordTableSimpleQuery (*class in gpt_index.indices.keyword_table*), 87
GPTKeywordTableSimpleQuery (*class in gpt_index.indices.keyword_table.query*), 152
GPTKGTableQuery (*class in gpt_index.indices.knowledge_graph*), 140
GPTKGTableQuery (*class in gpt_index.indices.knowledge_graph.query*), 162
GPTKnowledgeGraphIndex (*class in gpt_index.indices.knowledge_graph*), 140
GPTListIndex (*class in gpt_index.indices.list*), 81
GPTListIndexEmbeddingQuery (*class in gpt_index.indices.list*), 83
GPTListIndexQuery (*class in gpt_index.indices.list*), 83
GPTListIndexQuery (*class in gpt_index.indices.list.query*), 150
GPTNLPandasIndexQuery (*class in gpt_index.indices.struct_store*), 132
GPTNLPandasIndexQuery (*class in gpt_index.indices.struct_store.pandas_query*), 161
GPTNLStructStoreIndexQuery (*class in gpt_index.indices.struct_store*), 133
GPTNLStructStoreIndexQuery (*class in gpt_index.indices.struct_store.sql_query*), 160
GPTOpensearchIndex (*class in gpt_index.indices.vector_store.vector_indices*), 117
GPTOpensearchIndexQuery (*class in gpt_index.indices.vector_store.queries*), 158
GPTPandasIndex (*class in gpt_index.indices.struct_store*), 133
GPTPineconeIndex (*class in gpt_index.indices.vector_store.vector_indices*), 120
GPTPineconeIndexQuery (*class in gpt_index.indices.vector_store.queries*), 158
GPTQdrantIndex (*class in gpt_index.indices.vector_store.vector_indices*), 123
GPTQdrantIndexQuery (*class in gpt_index.indices.vector_store.queries*), 159
GPTRAKEKeywordTableIndex (*class in gpt_index.indices.keyword_table*), 88
GPTSimpleKeywordTableIndex (*class in gpt_index.indices.keyword_table*), 91
GPTSimpleVectorIndex (*class in gpt_index.indices.vector_store.vector_indices*), 126
GPTSimpleVectorIndexQuery (*class in gpt_index.indices.vector_store.queries*), 159
GPTSQLStructStoreIndex (*class in gpt_index.indices.struct_store*), 135
GPTSQLStructStoreIndexQuery (*class in gpt_index.indices.struct_store*), 138
GPTSQLStructStoreIndexQuery (*class in gpt_index.indices.struct_store.sql_query*), 160
GPTTreeIndex (*class in gpt_index.indices.tree*), 93
GPTTreeIndexEmbeddingQuery (*class in gpt_index.indices.tree*), 96
GPTTreeIndexEmbeddingQuery (*class in gpt_index.indices.tree.embedding_query*), 153
GPTTreeIndexLeafQuery (*class in gpt_index.indices.tree*), 97
GPTTreeIndexLeafQuery (*class in gpt_index.indices.tree.leaf_query*), 153
GPTTreeIndexRetQuery (*class in gpt_index.indices.tree*), 97
GPTTreeIndexRetQuery (*class in gpt_index.indices.tree.retrieve_query*), 154
GPTTreeIndexSummarizeQuery (*class in gpt_index.indices.tree.summarize_query*), 155
GPTVectorStoreIndex (*class in gpt_index.indices.vector_store.base*), 104

GPTVectorStoreIndexQuery (*class in gpt_index.indices.vector_store.base_query*), 156
GPTWeaviateIndex (*class in gpt_index.indices.vector_store.vector_indices*), 129
GPTWeaviateIndexQuery (*class in gpt_index.indices.vector_store.queries*), 159
GraphToolConfig (*class in gpt_index.langchain_helpers.agents*), 215
GraphToolConfig.Config (*class in gpt_index.langchain_helpers.agents*), 215

## H

HYBRID (*gpt_index.indices.knowledge_graph.KGQueryMode attribute*), 143
HYBRID (*gpt_index.indices.knowledge_graph.query.KGQueryMode attribute*), 163
HyDEQueryTransform (*class in gpt_index.indices.query.query_transform*), 171

## I

index_results() (*gpt_index.vector_stores.OpensearchVectorClient method*), 101
index_struct (*gpt_index.indices.base.BaseGPTIndex property*), 147
index_struct (*gpt_index.indices.keyword_table.GPTKeywordTableGPTQuery property*), 84
index_struct (*gpt_index.indices.keyword_table.GPTKeywordTableIndex property*), 85
index_struct (*gpt_index.indices.keyword_table.GPTKeywordTableRAKEQuery property*), 87
index_struct (*gpt_index.indices.keyword_table.GPTKeywordTableSimpleQuery property*), 88
index_struct (*gpt_index.indices.keyword_table.GPTRAKEKeywordTableIndex property*), 89
index_struct (*gpt_index.indices.keyword_table.GPTSimpleKeywordTableIndex property*), 91
index_struct (*gpt_index.indices.query.base.BaseGPTIndexQuery property*), 168
index_struct (*gpt_index.indices.tree.GPTTreeIndex property*), 94
index_struct (*gpt_index.indices.tree.GPTTreeIndexEmbeddingQuery property*), 97
index_struct (*gpt_index.indices.tree.GPTTreeIndexLeafQuery property*), 97
index_struct (*gpt_index.indices.tree.GPTTreeIndexRetQuery property*), 98
index_struct_cls (*gpt_index.indices.keyword_table.GPTKeywordTableIndex attribute*), 85
index_struct_cls (*gpt_index.indices.keyword_table.GPTRAKEKeywordTableIndex attribute*), 89
index_struct_cls (*gpt_index.indices.keyword_table.GPTSimpleKeywor attribute*), 92
index_struct_cls (*gpt_index.indices.tree.GPTTreeIndex attribute*), 94
IndexStructType (*class in gpt_index.data_structs.struct_type*), 165
IndexToolConfig (*class in gpt_index.langchain_helpers.agents*), 216
IndexToolConfig.Config (*class in gpt_index.langchain_helpers.agents*), 216
indices (*gpt_index.playground.base.Playground property*), 214
insert() (*gpt_index.indices.base.BaseGPTIndex method*), 147
insert() (*gpt_index.indices.empty.GPTEmptyIndex method*), 144
insert() (*gpt_index.indices.keyword_table.GPTKeywordTableIndex method*), 85
insert() (*gpt_index.indices.keyword_table.GPTRAKEKeywordTableIndex method*), 89
insert() (*gpt_index.indices.keyword_table.GPTSimpleKeywordTableInde method*), 92
insert() (*gpt_index.indices.knowledge_graph.GPTKnowledgeGraphIndex method*), 141
insert() (*gpt_index.indices.list.GPTListIndex method*), 81
insert() (*gpt_index.indices.struct_store.GPTPandasIndex method*), 134
insert() (*gpt_index.indices.struct_store.GPTSQLStructStoreIndex method*), 137
insert() (*gpt_index.indices.tree.GPTTreeIndex method*), 94
insert() (*gpt_index.indices.vector_store.base.GPTVectorStoreIndex method*), 105
insert() (*gpt_index.indices.vector_store.vector_indices.ChatGPTRetrieva method*), 109
insert() (*gpt_index.indices.vector_store.vector_indices.GPTChromaIndex method*), 112
insert() (*gpt_index.indices.vector_store.vector_indices.GPTFaissIndex method*), 115
insert() (*gpt_index.indices.vector_store.vector_indices.GPTOpensearchI method*), 118
insert() (*gpt_index.indices.vector_store.vector_indices.GPTPineconeInd method*), 121
insert() (*gpt_index.indices.vector_store.vector_indices.GPTQdrantIndex method*), 124
insert() (*gpt_index.indices.vector_store.vector_indices.GPTSimpleVector method*), 127
insert() (*gpt_index.indices.vector_store.vector_indices.GPTWeaviateInd method*), 130
insert_datapoint_from_nodes() (*gpt_index.indices.common.struct_store.base.BaseStructDatapoi method*), 211
insert_into_table()

(*gpt_index.langchain_helpers.sql_wrapper.SQLDatabase method*), 210
is_doc_id_none (*gpt_index.data_structs.node_v2.Node property*), 173
is_doc_id_none (*gpt_index.readers.Document property*), 179
is_text_none (*gpt_index.data_structs.node_v2.Node property*), 173
is_text_none (*gpt_index.readers.Document property*), 179

J

json() (*gpt_index.langchain_helpers.agents.GraphToolConfig method*), 215
json() (*gpt_index.langchain_helpers.agents.IndexToolConfig method*), 216
json() (*gpt_index.langchain_helpers.agents.LlamaGraphTool method*), 217
json() (*gpt_index.langchain_helpers.agents.LlamaIndexTool method*), 218
json() (*gpt_index.langchain_helpers.agents.LlamaToolkit method*), 219
json() (*gpt_index.langchain_helpers.memory_wrapper.GPTIndexChatMemory method*), 222
json() (*gpt_index.langchain_helpers.memory_wrapper.GPTIndexMemory method*), 223
JSONReader (*class in gpt_index.readers*), 181

K

KEYWORD (*gpt_index.indices.knowledge_graph.KGQueryMode attribute*), 143
KEYWORD (*gpt_index.indices.knowledge_graph.query.KGQueryMode attribute*), 163
KEYWORD_TABLE (*gpt_index.data_structs.struct_type.IndexStructType attribute*), 166
KeywordExtractPrompt (*class in gpt_index.prompts.prompts*), 190
KG (*gpt_index.data_structs.struct_type.IndexStructType attribute*), 167
KGQueryMode (*class in gpt_index.indices.knowledge_graph*), 143
KGQueryMode (*class in gpt_index.indices.knowledge_graph.query*), 163
KnowledgeGraphPrompt (*class in gpt_index.prompts.prompts*), 191

L

LangchainEmbedding (*class in gpt_index.embeddings.langchain*), 204
last_token_usage (*gpt_index.embeddings.langchain.LangchainEmbedding property*), 205
last_token_usage (*gpt_index.embeddings.openai.OpenAIEmbedding property*), 203
last_token_usage (*gpt_index.token_counter.mock_chain_wrapper.MockL property*), 205
LIST (*gpt_index.data_structs.struct_type.IndexStructType attribute*), 166
LlamaGraphTool (*class in gpt_index.langchain_helpers.agents*), 216
LlamaGraphTool.Config (*class in gpt_index.langchain_helpers.agents*), 217
LlamaIndexTool (*class in gpt_index.langchain_helpers.agents*), 218
LlamaIndexTool.Config (*class in gpt_index.langchain_helpers.agents*), 218
LlamaLogger (*class in gpt_index.logger*), 207
LlamaToolkit (*class in gpt_index.langchain_helpers.agents*), 219
LlamaToolkit.Config (*class in gpt_index.langchain_helpers.agents*), 219
llm (*gpt_index.token_counter.mock_chain_wrapper.MockLLMPredictor property*), 206
load() (*gpt_index.vector_stores.FaissVectorStore class method*), 100
load_data() (*gpt_index.readers.BeautifulSoupWebReader method*), 177
load_data() (*gpt_index.readers.ChatGPTRetrievalPluginReader method*), 177
load_data() (*gpt_index.readers.ChromaReader method*), 178
load_data() (*gpt_index.readers.DiscordReader method*), 178
load_data() (*gpt_index.readers.ElasticsearchReader method*), 180
load_data() (*gpt_index.readers.FaissReader method*), 180
load_data() (*gpt_index.readers.GithubRepositoryReader method*), 181
load_data() (*gpt_index.readers.GoogleDocsReader method*), 181
load_data() (*gpt_index.readers.JSONReader method*), 182
load_data() (*gpt_index.readers.MakeWrapper method*), 182
load_data() (*gpt_index.readers.MboxReader method*), 182
load_data() (*gpt_index.readers.NotionPageReader method*), 183
load_data() (*gpt_index.readers.ObsidianReader method*), 183
load_data() (*gpt_index.readers.PineconeReader method*), 183
load_data() (*gpt_index.readers.QdrantReader method*), 184
load_data() (*gpt_index.readers.RssReader method*), 185
load_data() (*gpt_index.readers.SimpleDirectoryReader*

*method*), 185
load_data() (*gpt_index.readers.SimpleMongoReader method*), 186
load_data() (*gpt_index.readers.SimpleWebPageReader method*), 186
load_data() (*gpt_index.readers.SlackReader method*), 187
load_data() (*gpt_index.readers.SteamshipFileReader method*), 187
load_data() (*gpt_index.readers.StringIterableReader method*), 188
load_data() (*gpt_index.readers.TrafilaturaWebReader method*), 188
load_data() (*gpt_index.readers.TwitterTweetReader method*), 189
load_data() (*gpt_index.readers.WeaviateReader method*), 189
load_data() (*gpt_index.readers.WikipediaReader method*), 190
load_data() (*gpt_index.readers.YoutubeTranscriptReader method*), 190
load_from_dict() (*gpt_index.docstore.DocumentStore class method*), 174
load_from_dict() (*gpt_index.indices.base.BaseGPTIndex class method*), 147
load_from_dict() (*gpt_index.indices.empty.GPTEmptyIndex class method*), 144
load_from_dict() (*gpt_index.indices.keyword_table.GPTKeywordTableIndex class method*), 85
load_from_dict() (*gpt_index.indices.keyword_table.GPTRAKEKeywordTableIndex class method*), 89
load_from_dict() (*gpt_index.indices.keyword_table.GPTSimpleKeywordTableIndex class method*), 92
load_from_dict() (*gpt_index.indices.knowledge_graph.GPTKnowledgeGraphIndex class method*), 141
load_from_dict() (*gpt_index.indices.list.GPTListIndex class method*), 81
load_from_dict() (*gpt_index.indices.struct_store.GPTPandasIndex class method*), 134
load_from_dict() (*gpt_index.indices.struct_store.GPTSQLStructStoreIndex class method*), 137
load_from_dict() (*gpt_index.indices.tree.GPTTreeIndex class method*), 95
load_from_dict() (*gpt_index.indices.vector_store.base.GPTVectorStoreIndex class method*), 105
load_from_dict() (*gpt_index.indices.vector_store.vector_indices.ChatGPTRetrievalPluginIndex class method*), 109
load_from_dict() (*gpt_index.indices.vector_store.vector_indices.GPTChromaIndex class method*), 112
load_from_dict() (*gpt_index.indices.vector_store.vector_indices.GPTFaissIndex class method*), 115
load_from_dict() (*gpt_index.indices.vector_store.vector_indices.GPTOpensearchIndex class method*), 118
load_from_dict() (*gpt_index.indices.vector_store.vector_indices.GPTPineconeIndex*

*class method*), 121
load_from_dict() (*gpt_index.indices.vector_store.vector_indices.GPTQa*
*class method*), 124
load_from_dict() (*gpt_index.indices.vector_store.vector_indices.GPTSi*
*class method*), 127
load_from_dict() (*gpt_index.indices.vector_store.vector_indices.GPTWe*
*class method*), 130
load_from_disk() (*gpt_index.composability.ComposableGraph class method*), 175
load_from_disk() (*gpt_index.indices.base.BaseGPTIndex class method*), 147
load_from_disk() (*gpt_index.indices.empty.GPTEmptyIndex class method*), 144
load_from_disk() (*gpt_index.indices.keyword_table.GPTKeywordTableI*
*class method*), 85
load_from_disk() (*gpt_index.indices.keyword_table.GPTRAKEKeyword*
*class method*), 89
load_from_disk() (*gpt_index.indices.keyword_table.GPTSimpleKeyword*
*class method*), 92
load_from_disk() (*gpt_index.indices.knowledge_graph.GPTKnowledgeG*
*class method*), 141
load_from_disk() (*gpt_index.indices.list.GPTListIndex class method*), 81
load_from_disk() (*gpt_index.indices.struct_store.GPTPandasIndex class method*), 134
load_from_disk() (*gpt_index.indices.struct_store.GPTSQLStructStoreInd*
*class method*), 137
load_from_disk() (*gpt_index.indices.tree.GPTTreeIndex class method*), 95
load_from_disk() (*gpt_index.indices.vector_store.base.GPTVectorStoreI*
*class method*), 105
load_from_disk() (*gpt_index.indices.vector_store.vector_indices.ChatGI*
*class method*), 109
load_from_disk() (*gpt_index.indices.vector_store.vector_indices.GPTCh*
*class method*), 112
load_from_disk() (*gpt_index.indices.vector_store.vector_indices.GPTFa*
*class method*), 115
load_from_disk() (*gpt_index.indices.vector_store.vector_indices.GPTOp*
*class method*), 118
load_from_disk() (*gpt_index.indices.vector_store.vector_indices.GPTPi*
*class method*), 121
load_from_disk() (*gpt_index.indices.vector_store.vector_indices.GPTQa*
*class method*), 124
load_from_disk() (*gpt_index.indices.vector_store.vector_indices.GPTSi*
*class method*), 127
load_from_disk() (*gpt_index.indices.vector_store.vector_indices.GPTWe*
*class method*), 130
load_from_string() (*gpt_index.composability.ComposableGraph class method*), 176
load_from_string() (*gpt_index.indices.base.BaseGPTIndex class method*), 148
load_from_string() (*gpt_index.indices.empty.GPTEmptyIndex class method*), 145
load_from_string() (*gpt_index.indices.keyword_table.GPTKeywordTab*

*class method*), 86
`load_from_string()` (*gpt_index.indices.keyword_table.GPTRAKEKeywordTableIndex class method*), 89
`load_from_string()` (*gpt_index.indices.keyword_table.GPTSimpleKeywordTableIndex class method*), 92
`load_from_string()` (*gpt_index.indices.knowledge_graph.GPTKnowledgeGraphIndex class method*), 142
`load_from_string()` (*gpt_index.indices.list.GPTListIndex class method*), 82
`load_from_string()` (*gpt_index.indices.struct_store.GPTPandasIndex class method*), 134
`load_from_string()` (*gpt_index.indices.struct_store.GPTSQLStructStoreIndex class method*), 137
`load_from_string()` (*gpt_index.indices.tree.GPTTreeIndex class method*), 95
`load_from_string()` (*gpt_index.indices.vector_store.base.GPTVectorStoreIndex class method*), 106
`load_from_string()` (*gpt_index.indices.vector_store.vector_indices.ChatGPTRetrievalPluginIndex class method*), 109
`load_from_string()` (*gpt_index.indices.vector_store.vector_indices.GPTChromaIndex class method*), 112
`load_from_string()` (*gpt_index.indices.vector_store.vector_indices.GPTFaissIndex class method*), 115
`load_from_string()` (*gpt_index.indices.vector_store.vector_indices.GPTOpensearchIndex class method*), 119
`load_from_string()` (*gpt_index.indices.vector_store.vector_indices.GPTPineconeIndex class method*), 122
`load_from_string()` (*gpt_index.indices.vector_store.vector_indices.GPTQdrantIndex class method*), 125
`load_from_string()` (*gpt_index.indices.vector_store.vector_indices.GPTSimpleVectorIndex class method*), 128
`load_from_string()` (*gpt_index.indices.vector_store.vector_indices.GPTWeaviateIndex class method*), 131
`load_langchain_documents()` (*gpt_index.readers.BeautifulSoupWebReader method*), 177
`load_langchain_documents()` (*gpt_index.readers.ChatGPTRetrievalPluginReader method*), 178
`load_langchain_documents()` (*gpt_index.readers.ChromaReader method*), 178
`load_langchain_documents()` (*gpt_index.readers.DiscordReader method*), 179
`load_langchain_documents()` (*gpt_index.readers.ElasticsearchReader method*), 180
`load_langchain_documents()` (*gpt_index.readers.FaissReader method*), 180
`load_langchain_documents()` (*gpt_index.readers.GithubRepositoryReader method*), 181
`load_langchain_documents()` (*gpt_index.readers.GoogleDocsReader method*), 181
`load_langchain_documents()` (*gpt_index.readers.JSONReader method*), 182
`load_langchain_documents()` (*gpt_index.readers.MakeWrapper method*), 182
`load_langchain_documents()` (*gpt_index.readers.MboxReader method*), 182
`load_langchain_documents()` (*gpt_index.readers.NotionPageReader method*), 183
`load_langchain_documents()` (*gpt_index.readers.ObsidianReader method*), 183
`load_langchain_documents()` (*gpt_index.readers.PineconeReader method*), 184
`load_langchain_documents()` (*gpt_index.readers.QdrantReader method*), 184
`load_langchain_documents()` (*gpt_index.readers.RssReader method*), 185
`load_langchain_documents()` (*gpt_index.readers.SimpleDirectoryReader method*), 186
`load_langchain_documents()` (*gpt_index.readers.SimpleMongoReader method*), 186
`load_langchain_documents()` (*gpt_index.readers.SimpleWebPageReader method*), 187
`load_langchain_documents()` (*gpt_index.readers.SlackReader method*), 187
`load_langchain_documents()` (*gpt_index.readers.SteamshipFileReader method*), 188
`load_langchain_documents()` (*gpt_index.readers.StringIterableReader method*), 188
`load_langchain_documents()` (*gpt_index.readers.TrafilaturaWebReader method*), 188
`load_langchain_documents()` (*gpt_index.readers.TwitterTweetReader method*), 189
`load_langchain_documents()` (*gpt_index.readers.WeaviateReader method*), 189
`load_langchain_documents()`

(gpt_index.readers.WikipediaReader method), 190
load_langchain_documents() (gpt_index.readers.YoutubeTranscriptReader method), 190
load_memory_variables() (gpt_index.langchain_helpers.memory_wrapper.GPTIndexChatMemory method), 222
load_memory_variables() (gpt_index.langchain_helpers.memory_wrapper.GPTIndexMemory method), 223

## M

MakeWrapper (class in gpt_index.readers), 182
MboxReader (class in gpt_index.readers), 182
memory_variables (gpt_index.langchain_helpers.memory_wrapper.GPTIndexChatMemory property), 222
memory_variables (gpt_index.langchain_helpers.memory_wrapper.GPTIndexMemory property), 223
merge() (gpt_index.docstore.DocumentStore class method), 175
MockLLMPredictor (class in gpt_index.token_counter.mock_chain_wrapper), 205
modes (gpt_index.playground.base.Playground property), 214
module
- gpt_index.composability, 175
- gpt_index.data_structs.node_v2, 172
- gpt_index.data_structs.struct_type, 165
- gpt_index.docstore, 173
- gpt_index.embeddings.langchain, 204
- gpt_index.embeddings.openai, 202
- gpt_index.indices.base, 146
- gpt_index.indices.common.struct_store.base, 211
- gpt_index.indices.empty, 144
- gpt_index.indices.empty.query, 163
- gpt_index.indices.keyword_table, 84
- gpt_index.indices.keyword_table.query, 150
- gpt_index.indices.knowledge_graph, 140
- gpt_index.indices.knowledge_graph.query, 162
- gpt_index.indices.list, 81
- gpt_index.indices.list.query, 149
- gpt_index.indices.prompt_helper, 206
- gpt_index.indices.query.base, 167
- gpt_index.indices.query.query_transform, 170
- gpt_index.indices.query.schema, 164, 168
- gpt_index.indices.service_context, 208
- gpt_index.indices.struct_store, 132
- gpt_index.indices.struct_store.container_builder, 210
- gpt_index.indices.struct_store.pandas_query, 161
- gpt_index.indices.struct_store.sql_query, 160
- gpt_index.indices.tree, 93
- gpt_index.indices.tree.embedding_query, 153
- gpt_index.indices.tree.leaf_query, 153
- gpt_index.indices.tree.retrieve_query, 154
- gpt_index.indices.tree.summarize_query, 155
- gpt_index.indices.vector_store.base, 104
- gpt_index.indices.vector_store.base_query, 156
- gpt_index.indices.vector_store.queries, 157
- gpt_index.indices.vector_store.vector_indices, 107
- gpt_index.langchain_helpers.agents, 215
- gpt_index.langchain_helpers.chain_wrapper, 205
- gpt_index.langchain_helpers.memory_wrapper, 221
- gpt_index.langchain_helpers.sql_wrapper, 209
- gpt_index.logger, 207
- gpt_index.node_parser, 214
- gpt_index.optimization, 209
- gpt_index.playground.base, 213
- gpt_index.prompts, 201
- gpt_index.prompts.prompts, 190
- gpt_index.readers, 177
- gpt_index.response.schema, 212
- gpt_index.token_counter.mock_chain_wrapper, 205
- gpt_index.vector_stores, 99

## N

NEXT (gpt_index.data_structs.node_v2.DocumentRelationship attribute), 172
next_node_id (gpt_index.data_structs.node_v2.Node property), 173
Node (class in gpt_index.data_structs.node_v2), 172
NodeParser (class in gpt_index.node_parser), 214
NodeWithScore (class in gpt_index.data_structs.node_v2), 173
NotionPageReader (class in gpt_index.readers), 182

## O

OAEMM (in module gpt_index.embeddings.openai), 202
OAEMT (in module gpt_index.embeddings.openai), 202

ObsidianReader (*class in gpt_index.readers*), 183
OpenAIEmbedding (*class in gpt_index.embeddings.openai*), 202
OpenAIEmbeddingModelType (*class in gpt_index.embeddings.openai*), 204
OpenAIEmbeddingModeModel (*class in gpt_index.embeddings.openai*), 203
OPENSEARCH (*gpt_index.data_structs.struct_type.IndexStructType attribute*), 166
OpensearchVectorClient (*class in gpt_index.vector_stores*), 100
OpensearchVectorStore (*class in gpt_index.vector_stores*), 101
optimize() (*gpt_index.optimization.SentenceEmbeddingOptimizer method*), 209

## P

PandasPrompt (*class in gpt_index.prompts.prompts*), 192
partial_format() (*gpt_index.prompts.Prompt method*), 202
partial_format() (*gpt_index.prompts.prompts.KeywordExtractPrompt method*), 191
partial_format() (*gpt_index.prompts.prompts.KnowledgeGraphPrompt method*), 192
partial_format() (*gpt_index.prompts.prompts.PandasPrompt method*), 192
partial_format() (*gpt_index.prompts.prompts.QueryKeywordExtractPrompt method*), 193
partial_format() (*gpt_index.prompts.prompts.QuestionAnswerPrompt method*), 194
partial_format() (*gpt_index.prompts.prompts.RefinePrompt method*), 195
partial_format() (*gpt_index.prompts.prompts.RefineTableContextPrompt method*), 195
partial_format() (*gpt_index.prompts.prompts.SchemaExtractPrompt method*), 196
partial_format() (*gpt_index.prompts.prompts.SimpleInputPrompt method*), 197
partial_format() (*gpt_index.prompts.prompts.SummaryPrompt method*), 197
partial_format() (*gpt_index.prompts.prompts.TableContextPrompt method*), 198
partial_format() (*gpt_index.prompts.prompts.TextToSQLPrompt method*), 199
partial_format() (*gpt_index.prompts.prompts.TreeInsertPrompt method*), 200
partial_format() (*gpt_index.prompts.prompts.TreeSelectMultiplePrompt method*), 200
partial_format() (*gpt_index.prompts.prompts.TreeSelectPrompt method*), 201
pass_response_to_webhook() (*gpt_index.readers.MakeWrapper method*), 182
PINECONE (*gpt_index.data_structs.struct_type.IndexStructType attribute*), 166
PineconeReader (*class in gpt_index.readers*), 183
PineconeVectorStore (*class in gpt_index.vector_stores*), 101
Playground (*class in gpt_index.playground.base*), 213
predict() (*gpt_index.token_counter.mock_chain_wrapper.MockLLMPred method*), 206
prev_node_id (*gpt_index.data_structs.node_v2.Node property*), 173
PREVIOUS (*gpt_index.data_structs.node_v2.DocumentRelationship attribute*), 172
print_response_stream() (*gpt_index.response.schema.StreamingResponse method*), 213
Prompt (*class in gpt_index.prompts*), 201
PromptHelper (*class in gpt_index.indices.prompt_helper*), 206

## Q

QASummaryGraphBuilder (*class in gpt_index.composability*), 176
QDRANT (*gpt_index.data_structs.struct_type.IndexStructType attribute*), 166
QdrantReader (*class in gpt_index.readers*), 184
QdrantVectorStore (*class in gpt_index.vector_stores*), 102
query() (*gpt_index.composability.ComposableGraph method*), 176
query() (*gpt_index.indices.base.BaseGPTIndex method*), 148
query() (*gpt_index.indices.empty.GPTEmptyIndex method*), 145
query() (*gpt_index.indices.keyword_table.GPTKeywordTableIndex method*), 86
query() (*gpt_index.indices.keyword_table.GPTRAKEKeywordTableIndex method*), 90
query() (*gpt_index.indices.keyword_table.GPTSimpleKeywordTableIndex method*), 92
query() (*gpt_index.indices.knowledge_graph.GPTKnowledgeGraphIndex method*), 142
query() (*gpt_index.indices.list.GPTListIndex method*), 82
query() (*gpt_index.indices.struct_store.GPTPandasIndex method*), 134
query() (*gpt_index.indices.struct_store.GPTSQLStructStoreIndex method*), 137
query() (*gpt_index.indices.tree.GPTTreeIndex method*), 95
query() (*gpt_index.indices.vector_store.base.GPTVectorStoreIndex method*), 106
query() (*gpt_index.indices.vector_store.vector_indices.ChatGPTRetrieval method*), 110

query() (gpt_index.indices.vector_store.vector_indices.GPTChromaIndex method), 113
query() (gpt_index.indices.vector_store.vector_indices.GPTFaissIndex method), 116
query() (gpt_index.indices.vector_store.vector_indices.GPTOpensearchIndex method), 119
query() (gpt_index.indices.vector_store.vector_indices.GPTPineconeIndex method), 122
query() (gpt_index.indices.vector_store.vector_indices.GPTQdrantIndex method), 125
query() (gpt_index.indices.vector_store.vector_indices.GPTSimpleVectorIndex method), 128
query() (gpt_index.indices.vector_store.vector_indices.GPTWeaviateIndex method), 131
query() (gpt_index.vector_stores.ChatGPTRetrievalPluginClient method), 99
query() (gpt_index.vector_stores.ChromaVectorStore method), 99
query() (gpt_index.vector_stores.FaissVectorStore method), 100
query() (gpt_index.vector_stores.OpensearchVectorStore method), 101
query() (gpt_index.vector_stores.PineconeVectorStore method), 102
query() (gpt_index.vector_stores.QdrantVectorStore method), 103
query() (gpt_index.vector_stores.SimpleVectorStore method), 103
query() (gpt_index.vector_stores.WeaviateVectorStore method), 104
query_database() (gpt_index.readers.NotionPageReader method), 183
query_index_for_context() (gpt_index.indices.struct_store.container_builder.SQLContextContainerBuilder method), 211
query_index_for_context() (gpt_index.indices.struct_store.SQLContextContainerBuilder method), 139
QueryBundle (class in gpt_index.indices.query.schema), 168
QueryConfig (class in gpt_index.indices.query.schema), 164, 169
QueryKeywordExtractPrompt (class in gpt_index.prompts.prompts), 192
QueryMode (class in gpt_index.indices.query.schema), 165, 170
QuestionAnswerPrompt (class in gpt_index.prompts.prompts), 193
queue_text_for_embeddding() (gpt_index.embeddings.langchain.LangchainEmbedding method), 205
queue_text_for_embeddding() (gpt_index.embeddings.openai.OpenAIEmbedding method), 203

## R

RAKE (gpt_index.indices.query.schema.QueryMode attribute), 165, 170
read_page() (gpt_index.readers.NotionPageReader method), 183
RECURSIVE (gpt_index.indices.query.schema.QueryMode attribute), 165, 170
ref_doc_id (gpt_index.data_structs.node_v2.Node property), 173
RefinePrompt (class in gpt_index.prompts.prompts), 194
RefineTableContextPrompt (class in gpt_index.prompts.prompts), 195
refresh() (gpt_index.indices.base.BaseGPTIndex method), 148
refresh() (gpt_index.indices.empty.GPTEmptyIndex method), 145
refresh() (gpt_index.indices.keyword_table.GPTKeywordTableIndex method), 86
refresh() (gpt_index.indices.keyword_table.GPTRAKEKeywordTableInde method), 90
refresh() (gpt_index.indices.keyword_table.GPTSimpleKeywordTableInd method), 92
refresh() (gpt_index.indices.knowledge_graph.GPTKnowledgeGraphInde method), 142
refresh() (gpt_index.indices.list.GPTListIndex method), 82
refresh() (gpt_index.indices.struct_store.GPTPandasIndex method), 135
refresh() (gpt_index.indices.struct_store.GPTSQLStructStoreIndex method), 137
refresh() (gpt_index.indices.tree.GPTTreeIndex method), 95
refresh() (gpt_index.indices.vector_store.base.GPTVectorStoreIndex method), 106
refresh() (gpt_index.indices.vector_store.vector_indices.ChatGPTRetriev method), 110
refresh() (gpt_index.indices.vector_store.vector_indices.GPTChromaInd method), 113
refresh() (gpt_index.indices.vector_store.vector_indices.GPTFaissIndex method), 116
refresh() (gpt_index.indices.vector_store.vector_indices.GPTOpensearch method), 119
refresh() (gpt_index.indices.vector_store.vector_indices.GPTPineconeIn method), 122
refresh() (gpt_index.indices.vector_store.vector_indices.GPTQdrantInde method), 125
refresh() (gpt_index.indices.vector_store.vector_indices.GPTSimpleVecto method), 128
refresh() (gpt_index.indices.vector_store.vector_indices.GPTWeaviateIn method), 131
reset() (gpt_index.logger.LlamaLogger method), 208
Response (class in gpt_index.response.schema), 212
response (gpt_index.response.schema.Response at-

*tribute*), 212
response_gen (*gpt_index.response.schema.StreamingResponse attribute*), 213
RETRIEVE (*gpt_index.indices.query.schema.QueryMode attribute*), 165, 170
retrieve() (*gpt_index.indices.empty.GPTEmptyIndexQuery method*), 146
retrieve() (*gpt_index.indices.empty.query.GPTEmptyIndexQuery method*), 163
retrieve() (*gpt_index.indices.keyword_table.GPTKeywordTableGPTQuery method*), 84
retrieve() (*gpt_index.indices.keyword_table.GPTKeywordTableRAKEQuery method*), 87
retrieve() (*gpt_index.indices.keyword_table.GPTKeywordTableSimpleQuery method*), 88
retrieve() (*gpt_index.indices.keyword_table.query.BaseGPTKeywordTableQuery method*), 151
retrieve() (*gpt_index.indices.keyword_table.query.GPTKeywordTableGPTQuery method*), 152
retrieve() (*gpt_index.indices.keyword_table.query.GPTKeywordTableRAKEQuery method*), 152
retrieve() (*gpt_index.indices.keyword_table.query.GPTKeywordTableSimpleQuery method*), 153
retrieve() (*gpt_index.indices.knowledge_graph.GPTKGTableQuery method*), 140
retrieve() (*gpt_index.indices.knowledge_graph.query.GPTKGTableQuery method*), 162
retrieve() (*gpt_index.indices.list.GPTListIndexEmbeddingQuery method*), 83
retrieve() (*gpt_index.indices.list.GPTListIndexQuery method*), 84
retrieve() (*gpt_index.indices.list.query.BaseGPTListIndexQuery method*), 150
retrieve() (*gpt_index.indices.list.query.GPTListIndexQuery method*), 150
retrieve() (*gpt_index.indices.query.base.BaseGPTIndexQuery method*), 168
retrieve() (*gpt_index.indices.struct_store.GPTNLPandasIndexQuery method*), 133
retrieve() (*gpt_index.indices.struct_store.GPTNLStructStoreIndexQuery method*), 133
retrieve() (*gpt_index.indices.struct_store.GPTSQLStructStoreIndexQuery method*), 138
retrieve() (*gpt_index.indices.struct_store.pandas_query.GPTNLPandasIndexQuery method*), 162
retrieve() (*gpt_index.indices.struct_store.sql_query.GPTNLStructStoreIndexQuery method*), 160
retrieve() (*gpt_index.indices.struct_store.sql_query.GPTSQLStructStoreIndexQuery method*), 161
retrieve() (*gpt_index.indices.tree.embedding_query.GPTTreeIndexEmbeddingQuery method*), 154
retrieve() (*gpt_index.indices.tree.GPTTreeIndexEmbeddingQuery method*), 97
retrieve() (*gpt_index.indices.tree.GPTTreeIndexLeafQuery method*), 97
retrieve() (*gpt_index.indices.tree.GPTTreeIndexRetQuery method*), 98
retrieve() (*gpt_index.indices.tree.leaf_query.GPTTreeIndexLeafQuery method*), 153
retrieve() (*gpt_index.indices.tree.retrieve_query.GPTTreeIndexRetQuer method*), 155
retrieve() (*gpt_index.indices.tree.summarize_query.GPTTreeIndexSumm method*), 156
retrieve() (*gpt_index.indices.vector_store.base_query.GPTVectorStoreIn method*), 156
retrieve() (*gpt_index.indices.vector_store.queries.ChatGPTRetrievalPlu method*), 157
retrieve() (*gpt_index.indices.vector_store.queries.GPTChromaIndexQue method*), 157
retrieve() (*gpt_index.indices.vector_store.queries.GPTFaissIndexQuery method*), 158
retrieve() (*gpt_index.indices.vector_store.queries.GPTOpensearchIndex method*), 158
retrieve() (*gpt_index.indices.vector_store.queries.GPTPineconeIndexQu method*), 158
retrieve() (*gpt_index.indices.vector_store.queries.GPTQdrantIndexQuer method*), 159
retrieve() (*gpt_index.indices.vector_store.queries.GPTSimpleVectorInde method*), 159
retrieve() (*gpt_index.indices.vector_store.queries.GPTWeaviateIndexQu method*), 160
RssReader (*class in gpt_index.readers*), 184
run() (*gpt_index.indices.query.query_transform.DecomposeQueryTransfor method*), 171
run() (*gpt_index.indices.query.query_transform.HyDEQueryTransform method*), 171
run() (*gpt_index.indices.query.query_transform.StepDecomposeQueryTra method*), 171
run() (*gpt_index.langchain_helpers.agents.LlamaGraphTool method*), 217
run() (*gpt_index.langchain_helpers.agents.LlamaIndexTool method*), 219
run() (*gpt_index.langchain_helpers.sql_wrapper.SQLDatabase method*), 210
run_no_throw() (*gpt_index.langchain_helpers.sql_wrapper.SQLDatabase method*), 210
run_sql() (*gpt_index.langchain_helpers.sql_wrapper.SQLDatabase method*), 210

S

save() (*gpt_index.vector_stores.FaissVectorStore method*), 100
save_context() (*gpt_index.langchain_helpers.memory_wrapper.GPTInd method*), 222
save_context() (*gpt_index.langchain_helpers.memory_wrapper.GPTInd method*), 224
save_to_dict() (*gpt_index.indices.base.BaseGPTIndex method*), 148

save_to_dict() (gpt_index.indices.empty.GPTEmptyIndex method), 145
save_to_dict() (gpt_index.indices.keyword_table.GPTKeywordTableIndex method), 86
save_to_dict() (gpt_index.indices.keyword_table.GPTRAKEKeywordTableIndex method), 90
save_to_dict() (gpt_index.indices.keyword_table.GPTSimpleKeywordTableIndex method), 93
save_to_dict() (gpt_index.indices.knowledge_graph.GPTKnowledgeGraphIndex method), 142
save_to_dict() (gpt_index.indices.list.GPTListIndex method), 82
save_to_dict() (gpt_index.indices.struct_store.GPTPandasIndex method), 135
save_to_dict() (gpt_index.indices.struct_store.GPTSQLStructStoreIndex method), 138
save_to_dict() (gpt_index.indices.tree.GPTTreeIndex method), 95
save_to_dict() (gpt_index.indices.vector_store.base.GPTVectorStoreIndex method), 106
save_to_dict() (gpt_index.indices.vector_store.vector_indices.ChatGPTRetrievalPluginIndex method), 110
save_to_dict() (gpt_index.indices.vector_store.vector_indices.GPTChromaIndex method), 113
save_to_dict() (gpt_index.indices.vector_store.vector_indices.GPTFaissIndex method), 116
save_to_dict() (gpt_index.indices.vector_store.vector_indices.GPTOpensearchIndex method), 119
save_to_dict() (gpt_index.indices.vector_store.vector_indices.GPTPineconeIndex method), 122
save_to_dict() (gpt_index.indices.vector_store.vector_indices.GPTQdrantIndex method), 125
save_to_dict() (gpt_index.indices.vector_store.vector_indices.GPTSimpleVectorIndex method), 128
save_to_dict() (gpt_index.indices.vector_store.vector_indices.GPTWeaviateIndex method), 131
save_to_disk() (gpt_index.composability.ComposableGraph method), 176
save_to_disk() (gpt_index.indices.base.BaseGPTIndex method), 148
save_to_disk() (gpt_index.indices.empty.GPTEmptyIndex method), 145
save_to_disk() (gpt_index.indices.keyword_table.GPTKeywordTableIndex method), 86
save_to_disk() (gpt_index.indices.keyword_table.GPTRAKEKeywordTableIndex method), 90
save_to_disk() (gpt_index.indices.keyword_table.GPTSimpleKeywordTableIndex method), 93
save_to_disk() (gpt_index.indices.knowledge_graph.GPTKnowledgeGraphIndex method), 142
save_to_disk() (gpt_index.indices.list.GPTListIndex method), 82
save_to_disk() (gpt_index.indices.struct_store.GPTPandasIndex method), 135
save_to_disk() (gpt_index.indices.struct_store.GPTSQLStructStoreIndex method), 138
save_to_disk() (gpt_index.indices.tree.GPTTreeIndex method), 96
save_to_disk() (gpt_index.indices.vector_store.base.GPTVectorStoreInd
method), 106
save_to_disk() (gpt_index.indices.vector_store.vector_indices.ChatGPT
method), 110
save_to_disk() (gpt_index.indices.vector_store.vector_indices.GPTChro
method), 113
save_to_disk() (gpt_index.indices.vector_store.vector_indices.GPTFaiss
method), 116
save_to_disk() (gpt_index.indices.vector_store.vector_indices.GPTOpen
method), 119
save_to_disk() (gpt_index.indices.vector_store.vector_indices.GPTPine
method), 122
save_to_disk() (gpt_index.indices.vector_store.vector_indices.GPTQdra
method), 125
save_to_disk() (gpt_index.indices.vector_store.vector_indices.GPTSimp
method), 128
save_to_disk() (gpt_index.indices.vector_store.vector_indices.GPTWeav
method), 131
save_to_string() (gpt_index.composability.ComposableGraph method), 176
save_to_string() (gpt_index.indices.base.BaseGPTIndex method), 148
save_to_string() (gpt_index.indices.empty.GPTEmptyIndex method), 145
save_to_string() (gpt_index.indices.keyword_table.GPTKeywordTableI
method), 86
save_to_string() (gpt_index.indices.keyword_table.GPTRAKEKeyword
method), 90
save_to_string() (gpt_index.indices.keyword_table.GPTSimpleKeyword
method), 93
save_to_string() (gpt_index.indices.knowledge_graph.GPTKnowledgeC
method), 142
save_to_string() (gpt_index.indices.list.GPTListIndex method), 82
save_to_string() (gpt_index.indices.struct_store.GPTPandasIndex method), 135
save_to_string() (gpt_index.indices.struct_store.GPTSQLStructStoreInd
method), 138
save_to_string() (gpt_index.indices.tree.GPTTreeIndex method), 96
save_to_string() (gpt_index.indices.vector_store.base.GPTVectorStoreI
method), 107
save_to_string() (gpt_index.indices.vector_store.vector_indices.ChatG
method), 110
save_to_string() (gpt_index.indices.vector_store.vector_indices.GPTCh
method), 113
save_to_string() (gpt_index.indices.vector_store.vector_indices.GPTFa
method), 116
save_to_string() (gpt_index.indices.vector_store.vector_indices.GPTOp
method), 120

save_to_string() (*gpt_index.indices.vector_store.vector_*
*method*), 123
save_to_string() (*gpt_index.indices.vector_store.vector_*
*method*), 126
save_to_string() (*gpt_index.indices.vector_store.vector_indices.GPTSimpleVectorIndex*
*method*), 129
save_to_string() (*gpt_index.indices.vector_store.vector_indices.GPTWeaviateIndex*
*method*), 132
SchemaExtractPrompt (*class in gpt_index.prompts.prompts*), 195
search() (*gpt_index.readers.NotionPageReader method*), 183
SentenceEmbeddingOptimizer (*class in gpt_index.optimization*), 209
serialize_to_dict() (*gpt_index.docstore.DocumentStore method*), 175
ServiceContext (*class in gpt_index.indices.service_context*), 208
set_callback_manager() (*gpt_index.langchain_helpers.agents.LlamaGraphTool*
*class method*), 217
set_callback_manager() (*gpt_index.langchain_helpers.agents.LlamaIndexTool*
*class method*), 219
set_document_hash() (*gpt_index.docstore.DocumentStore method*), 175
set_metadata() (*gpt_index.logger.LlamaLogger method*), 208
similarity() (*gpt_index.embeddings.langchain.LangchainEmbedding*
*method*), 205
similarity() (*gpt_index.embeddings.openai.OpenAIEmbedding*
*method*), 203
SIMPLE (*gpt_index.indices.query.schema.QueryMode attribute*), 165, 170
SIMPLE_DICT (*gpt_index.data_structs.struct_type.IndexStructType*
*attribute*), 166
SimpleDirectoryReader (*class in gpt_index.readers*), 185
SimpleInputPrompt (*class in gpt_index.prompts.prompts*), 196
SimpleMongoReader (*class in gpt_index.readers*), 186
SimpleNodeParser (*class in gpt_index.node_parser*), 214
SimpleVectorStore (*class in gpt_index.vector_stores*), 103
SimpleWebPageReader (*class in gpt_index.readers*), 186
SlackReader (*class in gpt_index.readers*), 187
SOURCE (*gpt_index.data_structs.node_v2.DocumentRelationship*
*attribute*), 172
SQL (*gpt_index.data_structs.struct_type.IndexStructType attribute*), 167
SQLContextContainerBuilder (*class in gpt_index.indices.struct_store*), 139
SQLContextContainerBuilder (*class in gpt_index.indices.struct_store.container_builder*), 210
SQLDatabase (*class in gpt_index.langchain_helpers.sql_wrapper*), 209
SQLDocumentContextBuilder (*class in gpt_index.indices.common.struct_store.base*), 211
SteamshipFileReader (*class in gpt_index.readers*), 187
StepDecomposeQueryTransform (*class in gpt_index.indices.query.query_transform*), 171
stream() (*gpt_index.token_counter.mock_chain_wrapper.MockLLMPredic*
*method*), 206
StreamingResponse (*class in gpt_index.response.schema*), 213
StringIterableReader (*class in gpt_index.readers*), 188
SUMMARIZE (*gpt_index.indices.query.schema.QueryMode attribute*), 165, 170
SummaryPrompt (*class in gpt_index.prompts.prompts*), 197
synthesize() (*gpt_index.indices.empty.GPTEmptyIndexQuery*
*method*), 146
synthesize() (*gpt_index.indices.empty.query.GPTEmptyIndexQuery*
*method*), 163

T

table_info (*gpt_index.langchain_helpers.sql_wrapper.SQLDatabase*
*property*), 210
TableContextPrompt (*class in gpt_index.prompts.prompts*), 197
TextToSQLPrompt (*class in gpt_index.prompts.prompts*), 198
to_langchain_format() (*gpt_index.readers.Document method*), 179
total_tokens_used (*gpt_index.embeddings.langchain.LangchainEmbedd*
*property*), 205
total_tokens_used (*gpt_index.embeddings.openai.OpenAIEmbedding*
*property*), 203
total_tokens_used (*gpt_index.token_counter.mock_chain_wrapper.Mock*
*property*), 206
TrafilaturaWebReader (*class in gpt_index.readers*), 188
TREE (*gpt_index.data_structs.struct_type.IndexStructType*
*attribute*), 166
TreeInsertPrompt (*class in gpt_index.prompts.prompts*), 199
TreeSelectMultiplePrompt (*class in gpt_index.prompts.prompts*), 200

TreeSelectPrompt (*class in gpt_index.prompts.prompts*), 200
TwitterTweetReader (*class in gpt_index.readers*), 189

U

unset_metadata() (*gpt_index.logger.LlamaLogger method*), 208
update() (*gpt_index.indices.base.BaseGPTIndex method*), 149
update() (*gpt_index.indices.empty.GPTEmptyIndex method*), 146
update() (*gpt_index.indices.keyword_table.GPTKeywordTableIndex method*), 87
update() (*gpt_index.indices.keyword_table.GPTRAKEKeywordTableIndex method*), 90
update() (*gpt_index.indices.keyword_table.GPTSimpleKeywordTableIndex method*), 93
update() (*gpt_index.indices.knowledge_graph.GPTKnowledgeGraphIndex method*), 143
update() (*gpt_index.indices.list.GPTListIndex method*), 83
update() (*gpt_index.indices.struct_store.GPTPandasIndex method*), 135
update() (*gpt_index.indices.struct_store.GPTSQLStructStoreIndex method*), 138
update() (*gpt_index.indices.tree.GPTTreeIndex method*), 96
update() (*gpt_index.indices.vector_store.base.GPTVectorStoreIndex method*), 107
update() (*gpt_index.indices.vector_store.vector_indices.ChatGPTRetrievalPluginIndex method*), 111
update() (*gpt_index.indices.vector_store.vector_indices.GPTChromaIndex method*), 114
update() (*gpt_index.indices.vector_store.vector_indices.GPTFaissIndex method*), 117
update() (*gpt_index.indices.vector_store.vector_indices.GPTOpensearchIndex method*), 120
update() (*gpt_index.indices.vector_store.vector_indices.GPTPineconeIndex method*), 123
update() (*gpt_index.indices.vector_store.vector_indices.GPTQdrantIndex method*), 126
update() (*gpt_index.indices.vector_store.vector_indices.GPTSimpleVectorIndex method*), 129
update() (*gpt_index.indices.vector_store.vector_indices.GPTWeaviateIndex method*), 132
update_docstore() (*gpt_index.docstore.DocumentStore method*), 175
update_forward_refs() (*gpt_index.langchain_helpers.agents.GraphToolConfig class method*), 215
update_forward_refs() (*gpt_index.langchain_helpers.agents.IndexToolConfig class method*), 216
update_forward_refs() (*gpt_index.langchain_helpers.agents.LlamaGraphTool class method*), 218
update_forward_refs() (*gpt_index.langchain_helpers.agents.LlamaIndexTool class method*), 219
update_forward_refs() (*gpt_index.langchain_helpers.agents.LlamaToolkit class method*), 220
update_forward_refs() (*gpt_index.langchain_helpers.memory_wrapper.GPTIndexChatM class method*), 222
update_forward_refs() (*gpt_index.langchain_helpers.memory_wrapper.GPTIndexMemor class method*), 224
upsert_triplet() (*gpt_index.indices.knowledge_graph.GPTKnowledgeG method*), 143
upsert_triplet_and_node() (*gpt_index.indices.knowledge_graph.GPTKnowledgeGraphIndex method*), 143

W

WEAVIATE (*gpt_index.data_structs.struct_type.IndexStructType attribute*), 166
WeaviateReader (*class in gpt_index.readers*), 189
WeaviateVectorStore (*class in gpt_index.vector_stores*), 103
WikipediaReader (*class in gpt_index.readers*), 190

Y

YoutubeTranscriptReader (*class in gpt_index.readers*), 190