redis/redis-vl-python

Redis Vector Library (RedisVL) interfaces with Redis' vector database for realtime semantic search, RAG, and recommendation systems.

ref: 376742a71ee0de5cab2f9dcde8836a72048fc234

License: MIT License

Stars: 215

Forks: 34

Top Contributors

tylerhutcherson (88)

justin-cechmanek (17)

bsbodden (6)

chayim (5)

ajac-zero (1)

dwdougherty (1)

eltociear (1)

MSFTeegarden (1)

mankerious (1)

rbs333 (1)

joshrotenberg (1)

nabsabraham (1)

Chapter 0.0.0

root

README.md

```
<div align="center" style="margin-bottom: 20px;">
    <div><img src="docs/_static/Redis_Logo_Red_RGB.svg" style="width: 200px; margin-
bottom: 20px;"></div>
    <h1>🔥 Vector Library</h1>
</div>

<div align="center" style="margin-top: 20px;">
    <span style="display: block; margin-bottom: 10px;">the *AI-native* Redis
Python client</span>
    <br />


[![Codecov](https://img.shields.io/codecov/c/github/redis/redis-vl-python/dev?
label=Codecov&logo=codecov&token=E30WxqBeJJ)](https://codecov.io/gh/redis/redis-
vl-python)
[![License: MIT](https://img.shields.io/badge/License-MIT-yellow.svg)]
(https://opensource.org/licenses/MIT)
![Language](https://img.shields.io/github/languages/top/redis/redis-vl-python)
[![Code style: black](https://img.shields.io/badge/code%20style-black-000000.svg)]
(https://github.com/psf/black)
![GitHub last commit](https://img.shields.io/github/last-commit/redis/redis-vl-python)
![GitHub deployments](https://img.shields.io/github/deployments/redis/redis-vl-
python/github-pages?label=doc%20build)
[![pypi](https://badge.fury.io/py/redisvl.svg)](https://pypi.org/project/redisvl/)

</div>

<div align="center">
<div display="inline-block">
    <a href="https://github.com/redis/redis-vl-python">
<b>Home</b></a>   
    <a href="https://www.redisvl.com"><b>Documentation</b></a>   
    <a href="https://github.com/redis-developer/redis-ai-resources">
<b>Recipes</b></a>   
  </div>
    <br />
</div>


# Introduction

Welcome to the Redis Vector Library – the ultimate Python client designed for AI
applications harnessing the power of [Redis](https://redis.io).

[redisvl](https://pypi.org/project/redisvl/) is your go-to tool for:

- Lightning-fast information retrieval & vector similarity search
- Real-time RAG pipelines
- Agentic memory structures
- Smart recommendation engines


# 💪 Getting Started

## Installation
```

Install `redisvl` into your Python (>=3.8) environment using `pip`:

```bash
pip install redisvl
```

> For more detailed instructions, visit the [installation guide](https://www.redisvl.com/overview/installation.html).

Setting up Redis

Choose from multiple Redis deployment options:

1. [Redis Cloud](https://redis.io/try-free): Managed cloud database (free tier available)
2. [Redis Stack](https://redis.io/docs/getting-started/install-stack/docker/): Docker image for development
    ```bash
    docker run -d --name redis-stack -p 6379:6379 -p 8001:8001 redis/redis-stack:latest
    ```
3. [Redis Enterprise](https://redis.io/enterprise/): Commercial, self-hosted database
4. [Azure Cache for Redis Enterprise](https://learn.microsoft.com/azure/azure-cache-for-redis/quickstart-create-redis-enterprise): Fully managed Redis Enterprise on Azure

> Enhance your experience and observability with the free [Redis Insight GUI](https://redis.com/redis-enterprise/redis-insight/).

Overview

🗃️ Redis Index Management
1. [Design a schema for your use case](https://www.redisvl.com/user_guide/getting_started_01.html#define-an-indexschema) that models your dataset with built-in Redis and indexable fields (*e.g. text, tags, numerics, geo, and vectors*). [Load a schema](https://www.redisvl.com/user_guide/getting_started_01.html#example-schema-creation) from a YAML file:
    ```yaml
    index:
      name: user-idx
      prefix: user
      storage_type: json

    fields:
      - name: user
        type: tag
      - name: credit_score
        type: tag
      - name: embedding
        type: vector
        attrs:
          algorithm: flat
          dims: 4
          distance_metric: cosine
          datatype: float32
    ```
    ```python
    from redisvl.schema import IndexSchema

    schema = IndexSchema.from_yaml("schemas/schema.yaml")
    ```
    Or load directly from a Python dictionary:

    ```python
    schema = IndexSchema.from_dict({
        "index": {
            "name": "user-idx",
            "prefix": "user",
            "storage_type": "json"
        },
        "fields": [
            {"name": "user", "type": "tag"},
            {"name": "credit_score", "type": "tag"},
            {
                "name": "embedding",
                "type": "vector",
                "attrs": {
                    "algorithm": "flat",
                    "datatype": "float32",
                    "dims": 4,
                    "distance_metric": "cosine"
                }
            }
        ]
    })
    ```

2. [Create a SearchIndex](https://www.redisvl.com/user_guide/getting_started_01.html#create-a-searchindex) class with an input schema and client connection in order to perform admin and search operations on your index in Redis:
    ```python
    from redis import Redis
    from redisvl.index import SearchIndex

    # Establish Redis connection and define index
    client = Redis.from_url("redis://localhost:6379")
    index = SearchIndex(schema, client)

    # Create the index in Redis
    index.create()
    ```
    > Async compliant search index class also available: [AsyncSearchIndex](https://www.redisvl.com/api/searchindex.html#redisvl.index.AsyncSearchIndex).

3. [Load](https://www.redisvl.com/user_guide/getting_started_01.html#load-data-to-searchindex) and [fetch](https://www.redisvl.com/user_guide/getting_started_01.html#fetch-an-object-from-redis) data to/from your Redis instance:
    ```python
    data = {"user": "john", "credit_score": "high", "embedding": [0.23, 0.49, -0.18, 0.95]}

    # load list of dictionaries, specify the "id" field
    index.load([data], id_field="user")

    # fetch by "id"
    john = index.fetch("john")
    ```

🔍 Retrieval

Define queries and perform advanced searches over your indices, including the combination of vectors, metadata filters, and more.

- [VectorQuery](https://www.redisvl.com/api/query.html#vectorquery) - Flexible vector queries with customizable filters enabling semantic search:

    ```python
    from redisvl.query import VectorQuery

    query = VectorQuery(
      vector=[0.16, -0.34, 0.98, 0.23],
      vector_field_name="embedding",
      num_results=3
    )
    # run the vector search query against the embedding field
    results = index.query(query)
    ```

    Incorporate complex metadata filters on your queries:
    ```python
    from redisvl.query.filter import Tag

    # define a tag match filter
    tag_filter = Tag("user") == "john"

    # update query definition
    query.set_filter(tag_filter)

    # execute query
    results = index.query(query)
    ```

- [RangeQuery](https://www.redisvl.com/api/query.html#rangequery) - Vector search within a defined range paired with customizable filters
- [FilterQuery](https://www.redisvl.com/api/query.html#filterquery) - Standard search using filters and the full-text search
- [CountQuery](https://www.redisvl.com/api/query.html#countquery) - Count the number of indexed records given attributes

> Read more about building [advanced Redis queries](https://www.redisvl.com/user_guide/hybrid_queries_02.html).

🔧 Utilities

Vectorizers
Integrate with popular embedding providers to greatly simplify the process of vectorizing unstructured data for your index and queries:
- [AzureOpenAI](https://www.redisvl.com/api/vectorizer.html#azureopenaitextvectorizer)
- [Cohere](https://www.redisvl.com/api/vectorizer.html#coheretextvectorizer)
- [Custom](https://www.redisvl.com/api/vectorizer.html#customtextvectorizer)
- [GCP VertexAI](https://www.redisvl.com/api/vectorizer.html#vertexaitextvectorizer)
- [HuggingFace](https://www.redisvl.com/api/vectorizer.html#hftextvectorizer)
- [Mistral](https://www.redisvl.com/api/vectorizer/html#mistralaitextvectorizer)
- [OpenAI](https://www.redisvl.com/api/vectorizer.html#openaitextvectorizer)

```python
from redisvl.utils.vectorize import CohereTextVectorizer

# set COHERE_API_KEY in your environment
co = CohereTextVectorizer()

embedding = co.embed(
    text="What is the capital city of France?",
    input_type="search_query"
)

embeddings = co.embed_many(
    texts=["my document chunk content", "my other document chunk content"],
```

```
    input_type="search_document"
)
```

> Learn more about using [vectorizers]
((https://www.redisvl.com/user_guide/vectorizers_04.html)) in your embedding workflows.

Rerankers
[Integrate with popular reranking providers]
(https://www.redisvl.com/user_guide/rerankers_06.html) to improve the relevancy of the
initial search results from Redis

💫 Extensions
We're excited to announce the support for **RedisVL Extensions**. These modules
implement interfaces exposing best practices and design patterns for working with LLM
memory and agents. We've taken the best from what we've learned from our users (that's
you) as well as bleeding-edge customers, and packaged it up.

*Have an idea for another extension? Open a PR or reach out to us at
applied.ai@redis.com. We're always open to feedback.*

LLM Semantic Caching
Increase application throughput and reduce the cost of using LLM models in production
by leveraging previously generated knowledge with the [`SemanticCache`]
(https://www.redisvl.com/api/cache.html#semanticcache).

```python
from redisvl.extensions.llmcache import SemanticCache

# init cache with TTL and semantic distance threshold
llmcache = SemanticCache(
    name="llmcache",
    ttl=360,
    redis_url="redis://localhost:6379",
    distance_threshold=0.1
)

# store user queries and LLM responses in the semantic cache
llmcache.store(
    prompt="What is the capital city of France?",
    response="Paris"
)

# quickly check the cache with a slightly different prompt (before invoking an LLM)
response = llmcache.check(prompt="What is France's capital city?")
print(response[0]["response"])
```
```stdout
>>> Paris
```

> Learn more about [semantic caching]
((https://www.redisvl.com/user_guide/llmcache_03.html)) for LLMs.

LLM Session Management

Improve personalization and accuracy of LLM responses by providing user chat history as
context. Manage access to the session data using recency or relevancy, *powered by
vector search* with the [`SemanticSessionManager`]
(https://www.redisvl.com/api/session_manager.html).

```python
from redisvl.extensions.session_manager import SemanticSessionManager

session = SemanticSessionManager(
    name="my-session",
    redis_url="redis://localhost:6379",
    distance_threshold=0.7
)

session.add_messages([
    {"role": "user", "content": "hello, how are you?"},
    {"role": "assistant", "content": "I'm doing fine, thanks."},
    {"role": "user", "content": "what is the weather going to be today?"},
    {"role": "assistant", "content": "I don't know"}
])
```
Get recent chat history:
```python
session.get_recent(top_k=1)
```
```stdout
>>> [{"role": "assistant", "content": "I don't know"}]
```
Get relevant chat history (powered by vector search):
```python
session.get_relevant("weather", top_k=1)
```
```stdout
>>> [{"role": "user", "content": "what is the weather going to be today?"}]
```
> Learn more about [LLM session management]
((https://www.redisvl.com/user_guide/session_manager_07.html)).

LLM Semantic Routing
Build fast decision models that run directly in Redis and route user queries to the
nearest "route" or "topic".

```python
from redisvl.extensions.router import Route, SemanticRouter

routes = [
    Route(
        name="greeting",
        references=["hello", "hi"],
        metadata={"type": "greeting"},
        distance_threshold=0.3,
    ),
    Route(
        name="farewell",
        references=["bye", "goodbye"],
        metadata={"type": "farewell"},
        distance_threshold=0.3,
    ),
]

# build semantic router from routes
router = SemanticRouter(
    name="topic-router",
    routes=routes,
    redis_url="redis://localhost:6379",
)
```

````
router("Hi, good morning")
```

```stdout
>>> RouteMatch(name='greeting', distance=0.273891836405)
```
> Learn more about [semantic routing]
(https://www.redisvl.com/user_guide/semantic_router_08.html).

## 🖥️ Command Line Interface
Create, destroy, and manage Redis index configurations from a purpose-built CLI
interface: `rvl`.

```bash
$ rvl -h

usage: rvl <command> [<args>]

Commands:
        index       Index manipulation (create, delete, etc.)
        version     Obtain the version of RedisVL
        stats       Obtain statistics about an index
```

> Read more about [using the CLI](https://www.redisvl.com/user_guide/cli.html).

## 🚀 Why RedisVL?

In the age of GenAI, **vector databases** and **LLMs** are transforming information
retrieval systems. With emerging and popular frameworks like [LangChain]
(https://github.com/langchain-ai/langchain) and [LlamaIndex]
(https://www.llamaindex.ai/), innovation is rapid. Yet, many organizations face the
challenge of delivering AI solutions **quickly** and at **scale**.

Enter [Redis](https://redis.io) – a cornerstone of the NoSQL world, renowned for its
versatile [data structures](https://redis.io/docs/data-types/) and [processing engines]
(https://redis.io/docs/interact/). Redis excels in real-time workloads like caching,
session management, and search. It's also a powerhouse as a vector database for RAG, an
LLM cache, and a chat session memory store for conversational AI.

The Redis Vector Library bridges the gap between the AI-native developer ecosystem and
Redis's robust capabilities. With a lightweight, elegant, and intuitive interface,
RedisVL makes it easy to leverage Redis's power. Built on the [Redis Python]
(https://github.com/redis/redis-py/tree/master) client, `redisvl` transforms Redis's
features into a grammar perfectly aligned with the needs of today's AI/ML Engineers and
Data Scientists.


## 😁 Helpful Links

For additional help, check out the following resources:
 - [Getting Started Guide](https://www.redisvl.com/user_guide/getting_started_01.html)
 - [API Reference](https://www.redisvl.com/api/index.html)
 - [Example Gallery](https://www.redisvl.com/examples/index.html)
 - [Redis AI Recipes](https://github.com/redis-developer/redis-ai-resources)
 - [Official Redis Vector API Docs](https://redis.io/docs/interact/search-and-
query/advanced-concepts/vectors/)


## 🤝 Contributing

Please help us by contributing PRs, opening GitHub issues for bugs or new feature
ideas, improving documentation, or increasing test coverage. [Read more about how to
contribute!](CONTRIBUTING.md)
````
````

```
## 🚧 Maintenance
This project is supported by [Redis, Inc](https://redis.com) on a good faith effort
basis. To report bugs, request features, or receive assistance, please [file an issue]
(https://github.com/redis/redis-vl-python/issues).
}
```

CONTRIBUTING.md

```
# Contributing

## Introduction

First off, thank you for considering contributions. We value community contributions!

## Contributions We Need

You may already know what you want to contribute \-- a fix for a bug you
encountered, or a new feature your team wants to use.

If you don't know what to contribute, keep an open mind! Improving
documentation, bug triaging, and writing tutorials are all examples of
helpful contributions that mean less work for you.

## Your First Contribution

Unsure where to begin contributing? You can start by looking through some of our issues
[listed here](https://github.com/redis/redis-vl-python/issues).

## Getting Started

Here's how to get started with your code contribution:

1.  Create your own fork of this repo
2.  Set up your developer environment
2.  Apply the changes in your forked codebase / environment
4.  If you like the change and think the project could use it, send us a
    pull request.

### Dev Environment
RedisVL uses [Poetry](https://python-poetry.org/) for dependency management.

Follow the instructions to [install Poetry](https://python-
poetry.org/docs/#installation).

Then install the required libraries:

```bash
poetry install --all-extras
```

Linting and Tests

Check formatting, linting, and typing:
```bash
poetry run format
poetry run sort-imports
poetry run mypy
```

TestContainers
```

RedisVL uses Testcontainers Python for integration tests. Testcontainers is an open-source framework for provisioning throwaway, on-demand containers for development and testing use cases.

To run Testcontainers-based tests you need a local Docker installation such as:
- [Docker Desktop](https://www.docker.com/products/docker-desktop/)
- [Docker Engine on Linux](https://docs.docker.com/engine/install/)

Running the Tests

Tests w/ vectorizers:
```bash
poetry run test-cov
```

Tests w/out vectorizers:
```bash
SKIP_VECTORIZERS=true poetry run test-cov
```

Tests w/out rerankers:
```bash
SKIP_RERANKERS=true poetry run test-cov
```

Documentation
Docs are served from the `docs/` directory.

Build the docs. Generates the `_build/html` contents:
```bash
poetry run build-docs
```

Serve the documentation with a local webserver:
```bash
poetry run serve-docs
```

Getting Redis

In order for your applications to use RedisVL, you must have [Redis](https://redis.io) accessible with Search & Query features enabled on [Redis Cloud](https://redis.com/try-free) or locally in docker with [Redis Stack](https://redis.io/docs/getting-started/install-stack/docker/):

```bash
docker run -d --name redis-stack -p 6379:6379 -p 8001:8001 redis/redis-stack:latest
```

This will also spin up the [FREE RedisInsight GUI](https://redis.com/redis-enterprise/redis-insight/) at `http://localhost:8001`.

How to Report a Bug

Security Vulnerabilities

NOTE: If you find a security vulnerability, do NOT open an issue.
Email [Redis OSS (<oss@redis.com>)](mailto:oss@redis.com) instead.

In order to determine whether you are dealing with a security issue, ask yourself these two questions:

-   Can I access something that's not mine, or something I shouldn't

```
    have access to?
-   Can I disable something for other people?

If the answer to either of those two questions are *yes*, then you're
probably dealing with a security issue. Note that even if you answer
*no*  to both questions, you may still be dealing with a security
issue, so if you're unsure, just email us.

### Everything Else

When filing an issue, make sure to answer these five questions:

1.  What version of python are you using?
2.  What version of `redis` and `redisvl` are you using?
3.  What did you do?
4.  What did you expect to see?
5.  What did you see instead?

## How to Suggest a Feature or Enhancement

If you'd like to contribute a new feature, make sure you check our
issue list to see if someone has already proposed it. Work may already
be under way on the feature you want -- or we may have rejected a
feature like it already.

If you don't see anything, open a new issue that describes the feature
you would like and how it should work.

## Code Review Process

The core team looks at Pull Requests on a regular basis. We will give
feedback as as soon as possible. After feedback, we expect a response
within two weeks. After that time, we may close your PR if it isn't
showing any activity.
}
```

LICENSE

```
SOFTWARE.
}
```

conftest.py

```
import os
import pytest
import asyncio

from redisvl.redis.connection import RedisConnectionFactory
from testcontainers.compose import DockerCompose



@pytest.fixture(scope="session", autouse=True)
def redis_container():
    # Set the default Redis version if not already set
    os.environ.setdefault("REDIS_VERSION", "edge")

    compose = DockerCompose("tests", compose_file_name="docker-compose.yml", pull=True)
    compose.start()

    redis_host, redis_port = compose.get_service_host_and_port("redis", 6379)
    redis_url = f"redis://{redis_host}:{redis_port}"
    os.environ["REDIS_URL"] = redis_url

    yield compose

    compose.stop()

@pytest.fixture(scope="session")
def redis_url():
    return os.getenv("REDIS_URL", "redis://localhost:6379")

@pytest.fixture
async def async_client(redis_url):
    client = await RedisConnectionFactory.get_async_redis_connection(redis_url)
    yield client
    try:
        await client.aclose()
    except RuntimeError as e:
        if "Event loop is closed" not in str(e):
            raise

@pytest.fixture
def client():
    conn = RedisConnectionFactory.get_redis_connection(os.environ["REDIS_URL"])
    yield conn
    conn.close()

@pytest.fixture
def openai_key():
    return os.getenv("OPENAI_API_KEY")

@pytest.fixture
def openai_version():
    return os.getenv("OPENAI_API_VERSION")

@pytest.fixture
def azure_endpoint():
```

```
    return os.getenv("AZURE_OPENAI_ENDPOINT")

@pytest.fixture
def cohere_key():
    return os.getenv("COHERE_API_KEY")

@pytest.fixture
def mistral_key():
    return os.getenv("MISTRAL_API_KEY")

@pytest.fixture
def gcp_location():
    return os.getenv("GCP_LOCATION")

@pytest.fixture
def gcp_project_id():
    return os.getenv("GCP_PROJECT_ID")


@pytest.fixture
def sample_data():
    return [
    {
        "user": "john",
        "age": 18,
        "job": "engineer",
        "credit_score": "high",
        "location": "-122.4194,37.7749",
        "user_embedding": [0.1, 0.1, 0.5]
    },
    {
        "user": "mary",
        "age": 14,
        "job": "doctor",
        "credit_score": "low",
        "location": "-122.4194,37.7749",
        "user_embedding": [0.1, 0.1, 0.5]
    },
    {
        "user": "nancy",
        "age": 94,
        "job": "doctor",
        "credit_score": "high",
        "location": "-122.4194,37.7749",
        "user_embedding": [0.7, 0.1, 0.5]
    },
    {
        "user": "tyler",
        "age": 100,
        "job": "engineer",
        "credit_score": "high",
        "location": "-110.0839,37.3861",
        "user_embedding": [0.1, 0.4, 0.5]
    },
    {
        "user": "tim",
        "age": 12,
        "job": "dermatologist",
        "credit_score": "high",
        "location": "-110.0839,37.3861",
        "user_embedding": [0.4, 0.4, 0.5]
    },
    {
        "user": "taimur",
```

```
        "age": 15,
        "job": "CEO",
        "credit_score": "low",
        "location": "-110.0839,37.3861",
        "user_embedding": [0.6, 0.1, 0.5]
    },
    {
        "user": "joe",
        "age": 35,
        "job": "dentist",
        "credit_score": "medium",
        "location": "-110.0839,37.3861",
        "user_embedding": [0.9, 0.9, 0.1]
    },
]

@pytest.fixture
def clear_db(redis):
    redis.flushall()
    yield
    redis.flushall()

@pytest.fixture
def app_name():
    return "test_app"

}
```

Chapter 1.0.0

doctests

doctests/query_vector.py

```
# EXAMPLE: query_vector
# HIDE_START
import json
import warnings
import redis
import numpy as np
from redisvl.index import SearchIndex
from redisvl.query import RangeQuery, VectorQuery
from redisvl.schema import IndexSchema
from sentence_transformers import SentenceTransformer


def embed_text(model, text):
    return np.array(model.encode(text)).astype(np.float32).tobytes()

r = redis.Redis(decode_responses=True)

warnings.filterwarnings("ignore", category=FutureWarning,
message=r".*clean_up_tokenization_spaces.*")
model = SentenceTransformer('sentence-transformers/all-MiniLM-L6-v2')
```

```
# create index
schema = IndexSchema.from_yaml('data/query_vector_idx.yaml')
index = SearchIndex(schema, r)
index.create(overwrite=True, drop=True)

# load data
with open("data/query_vector.json") as f:
    bicycles = json.load(f)
index.load(bicycles)
# HIDE_END

# STEP_START vector1
query = "Bike for small kids"
query_vector = embed_text(model, query)
print(query_vector[:10]) # >>> b'\x02=c=\x93\x0e\xe0=aC'

vquery = VectorQuery(
    vector=query_vector,
    vector_field_name="description_embeddings",
    num_results=3,
    return_score=True,
    return_fields=["description"]
)
res = index.query(vquery)
print(res) # >>> [{'id': 'bicycle:6b702e8b...', 'vector_distance': '0.399111807346',
'description': 'Kids want...
# REMOVE_START
assert len(res) == 3
# REMOVE_END
# STEP_END

# STEP_START vector2
vquery = RangeQuery(
    vector=query_vector,
    vector_field_name="description_embeddings",
    distance_threshold=0.5,
    return_score=True
).return_fields("description").dialect(2)
res = index.query(vquery)
print(res) # >>> [{'id': 'bicycle:6bcb1bb4...', 'vector_distance': '0.399111807346',
'description': 'Kids want...
# REMOVE_START
assert len(res) == 2
# REMOVE_END
# STEP_END

# REMOVE_START
# destroy index and data
index.delete(drop=True)
# REMOVE_END
}
```

pyproject.toml

```
[tool.poetry]
name = "redisvl"
version = "0.3.5"
description = "Python client library and CLI for using Redis as a vector database"
authors = ["Redis Inc. <applied.ai@redis.com>"]
```

```
license = "MIT"
readme = "README.md"
homepage = "https://github.com/redis/redis-vl-python"
repository = "https://github.com/redis/redis-vl-python"
documentation = "https://www.redisvl.com"
keywords = ["ai", "redis", "redis-client", "vector-database", "vector-search"]
classifiers = [
  "Programming Language :: Python :: 3.9",
  "Programming Language :: Python :: 3.10",
  "Programming Language :: Python :: 3.11",
  "License :: OSI Approved :: MIT License",
]
packages = [{ include = "redisvl", from = "." }]

[tool.poetry.dependencies]
python = ">=3.9,<4.0"
numpy = "*"
pyyaml = "*"
coloredlogs = "*"
redis = ">=5.0.0"
pydantic = { version = ">=2,<3" }
tenacity = ">=8.2.2"
tabulate = { version = ">=0.9.0,<1" }
ml-dtypes = "^0.4.0"
openai = { version = ">=1.13.0", optional = true }
sentence-transformers = { version = ">=2.2.2", optional = true }
google-cloud-aiplatform = { version = ">=1.26", optional = true }
cohere = { version = ">=4.44", optional = true }
mistralai = { version = ">=0.2.0", optional = true }

[tool.poetry.extras]
openai = ["openai"]
sentence-transformers = ["sentence-transformers"]
google_cloud_aiplatform = ["google_cloud_aiplatform"]
cohere = ["cohere"]
mistralai = ["mistralai"]

[tool.poetry.group.dev.dependencies]
black = ">=20.8b1"
isort = ">=5.6.4"
pylint = "3.1.0"
pytest = "8.1.1"
pytest-cov = "5.0.0"
pytest-asyncio = "0.23.6"
mypy = "1.9.0"
types-redis = "*"
types-pyyaml = "*"
types-tabulate = "*"
treon = "*"

[tool.poetry.group.test.dependencies]
testcontainers = "^4.3.1"

[tool.poetry.group.docs.dependencies]
sphinx = ">=4.4.0"
pydata-sphinx-theme = "^0.15.2"
nbsphinx = "^0.9.3"
jinja2 = "^3.1.3"
sphinx-copybutton = "^0.5.2"
sphinx-favicon = "^1.0.1"
sphinx-design = "^0.5.0"
myst-nb = "^1.1.0"

[tool.poetry.scripts]
```

```
rvl = "redisvl.cli.runner:main"
format = "scripts:format"
check-format = "scripts:check_format"
sort-imports = "scripts:sort_imports"
check-sort-imports = "scripts:check_sort_imports"
check-lint = "scripts:check_lint"
mypy = "scripts:mypy"
test = "scripts:test"
test-verbose = "scripts:test_verbose"
test-cov = "scripts:test_cov"
cov = "scripts:cov"
test-notebooks = "scripts:test_notebooks"
build-docs = "scripts:build_docs"
serve-docs = "scripts:serve_docs"

[build-system]
requires = ["poetry-core>=1.0.0"]
build-backend = "poetry.core.masonry.api"

[tool.black]
target-version = ['py38', 'py39', 'py310', 'py311']
exclude = '''
(
  | \.egg
  | \.git
  | \.hg
  | \.mypy_cache
  | \.nox
  | \.tox
  | \.venv
  | _build
  | build
  | dist
  | setup.py
)
'''

[tool.pytest.ini_options]
log_cli = true
asyncio_mode = "auto"

[tool.coverage.run]
source = ["redisvl"]

[tool.coverage.report]
ignore_errors = true

[tool.coverage.html]
directory = "htmlcov"

[tool.mypy]
warn_unused_configs = true
ignore_missing_imports = true
}
```

Chapter 2.0.0

redisvl

redisvl/__init__.py

```
from redisvl.version import __version__

all = ["__version__"]
}
```

Chapter 2.1.0

redisvl/cli

redisvl/cli/index.py

```
import argparse
import sys
from argparse import Namespace

from tabulate import tabulate

from redisvl.cli.utils import add_index_parsing_options, create_redis_url
from redisvl.index import SearchIndex
from redisvl.redis.connection import RedisConnectionFactory
from redisvl.redis.utils import convert_bytes, make_dict
from redisvl.schema.schema import IndexSchema
from redisvl.utils.log import get_logger

logger = get_logger("[RedisVL]")


class Index:
    usage = "\n".join(
        [
            "rvl index <command> [<args>]\n",
            "Commands:",
            "\tinfo        Obtain information about an index",
            "\tcreate      Create a new index",
            "\tdelete      Delete an existing index",
            "\tdestroy     Delete an existing index and all of its data",
            "\tlistall     List all indexes",
            "\n",
        ]
    )

    def __init__(self):
        parser = argparse.ArgumentParser(usage=self.usage)

        parser.add_argument("command", help="Subcommand to run")
```

```
        parser.add_argument(
            "-f",
            "--format",
            help="Output format for info command",
            type=str,
            default="rounded_outline",
        )
        parser = add_index_parsing_options(parser)

        args = parser.parse_args(sys.argv[2:])
        if not hasattr(self, args.command):
            parser.print_help()
            exit(0)
        try:
            getattr(self, args.command)(args)
        except Exception as e:
            logger.error(e)
            exit(0)

    def create(self, args: Namespace):
        """Create an index.

        Usage:
            rvl index create -i <index_name> | -s <schema_path>
        """
        if not args.schema:
            logger.error("Schema must be provided to create an index")
        index = SearchIndex.from_yaml(args.schema)
        redis_url = create_redis_url(args)
        index.connect(redis_url)
        index.create()
        logger.info("Index created successfully")

    def info(self, args: Namespace):
        """Obtain information about an index.

        Usage:
            rvl index info -i <index_name> | -s <schema_path>
        """
        index = self._connect_to_index(args)
        _display_in_table(index.info(), output_format=args.format)

    def listall(self, args: Namespace):
        """List all indices.

        Usage:
            rvl index listall
        """
        redis_url = create_redis_url(args)
        conn = RedisConnectionFactory.get_redis_connection(redis_url)
        indices = convert_bytes(conn.execute_command("FT._LIST"))
        logger.info("Indices:")
        for i, index in enumerate(indices):
            logger.info(str(i + 1) + ". " + index)

    def delete(self, args: Namespace, drop=False):
        """Delete an index.

        Usage:
            rvl index delete -i <index_name> | -s <schema_path>
        """
        index = self._connect_to_index(args)
        index.delete(drop=drop)
        logger.info("Index deleted successfully")
```

```
    def destroy(self, args: Namespace):
        """Delete an index and the documents within it.

        Usage:
            rvl index destroy -i <index_name> | -s <schema_path>
        """
        self.delete(args, drop=True)

    def _connect_to_index(self, args: Namespace) -> SearchIndex:
        # connect to redis
        try:
            redis_url = create_redis_url(args)
            conn = RedisConnectionFactory.get_redis_connection(url=redis_url)
        except ValueError:
            logger.error(
                "Must set REDIS_URL environment variable or provide host and port"
            )
            exit(0)

        if args.index:
            schema = IndexSchema.from_dict({"index": {"name": args.index}})
            index = SearchIndex(schema=schema, redis_url=redis_url)
        elif args.schema:
            index = SearchIndex.from_yaml(args.schema)
            index.set_client(conn)
        else:
            logger.error("Index name or schema must be provided")
            exit(0)

        return index


def _display_in_table(index_info, output_format="rounded_outline"):
    print("\n")
    attributes = index_info.get("attributes", [])
    definition = make_dict(index_info.get("index_definition"))
    index_info = [
        index_info.get("index_name"),
        definition.get("key_type"),
        definition.get("prefixes"),
        index_info.get("index_options"),
        index_info.get("indexing"),
    ]

    # Display the index information in tabular format
    print("Index Information:")
    print(
        tabulate(
            [index_info],
            headers=[
                "Index Name",
                "Storage Type",
                "Prefixes",
                "Index Options",
                "Indexing",
            ],
            tablefmt=output_format,
        )
    )

    attr_values = []
    headers = [
        "Name",
```

```
        "Attribute",
        "Type",
    ]

    for attrs in attributes:
        attr = make_dict(attrs)

        values = [attr.get("identifier"), attr.get("attribute"), attr.get("type")]
        if len(attrs) > 5:
            options = make_dict(attrs)
            for k, v in options.items():
                if k not in ["identifier", "attribute", "type"]:
                    headers.append("Field Option")
                    headers.append("Option Value")
                    values.append(k)
                    values.append(v)
        attr_values.append(values)

    # Display the attributes in tabular format
    print("Index Fields:")
    print(
        tabulate(
            attr_values,
            headers=headers,
            tablefmt=output_format,
        )
    )
}
```

redisvl/cli/main.py

```
import argparse
import sys

from redisvl.cli.index import Index
from redisvl.cli.stats import Stats
from redisvl.cli.version import Version
from redisvl.utils.log import get_logger

logger = get_logger(__name__)


def _usage():
    usage = [
        "rvl <command> [<args>]\n",
        "Commands:",
        "\tindex       Index manipulation (create, delete, etc.)",
        "\tversion     Obtain the version of RedisVL",
        "\tstats       Obtain statistics about an index",
    ]
    return "\n".join(usage) + "\n"


class RedisVlCLI:
    def __init__(self):
        parser = argparse.ArgumentParser(
            description="Redis Vector Library CLI", usage=_usage()
        )

        parser.add_argument("command", help="Subcommand to run")
```

```
        if len(sys.argv) < 2:
            parser.print_help()
            exit(0)

        args = parser.parse_args(sys.argv[1:2])
        if not hasattr(self, args.command):
            parser.print_help()
            exit(0)
        getattr(self, args.command)()

    def index(self):
        Index()
        exit(0)

    def version(self):
        Version()
        exit(0)

    def stats(self):
        Stats()
        exit(0)
}
```

redisvl/cli/runner.py

```
from redisvl.cli.main import RedisVlCLI


def main():
    """Main call to init the RedisVL CLI tool."""
    RedisVlCLI()


if __name__ == "__main__":
    main()
}
```

redisvl/cli/stats.py

```
import argparse
import sys
from argparse import Namespace

from tabulate import tabulate

from redisvl.cli.utils import add_index_parsing_options, create_redis_url
from redisvl.index import SearchIndex
from redisvl.schema.schema import IndexSchema
from redisvl.utils.log import get_logger

logger = get_logger("[RedisVL]")

STATS_KEYS = [
    "num_docs",
```

```
    "num_terms",
    "max_doc_id",
    "num_records",
    "percent_indexed",
    "hash_indexing_failures",
    "number_of_uses",
    "bytes_per_record_avg",
    "doc_table_size_mb",
    "inverted_sz_mb",
    "key_table_size_mb",
    "offset_bits_per_record_avg",
    "offset_vectors_sz_mb",
    "offsets_per_term_avg",
    "records_per_doc_avg",
    "sortable_values_size_mb",
    "total_indexing_time",
    "total_inverted_index_blocks",
    "vector_index_sz_mb",
]


class Stats:
    usage = "\n".join(
        [
            "rvl stats [<args>]\n",
        ]
    )

    def __init__(self):
        parser = argparse.ArgumentParser(usage=self.usage)

        parser.add_argument(
            "-f", "--format", help="Output format", type=str, default="rounded_outline"
        )
        parser = add_index_parsing_options(parser)
        args = parser.parse_args(sys.argv[2:])
        try:
            self.stats(args)
        except Exception as e:
            logger.error(e)
            exit(0)

    def stats(self, args: Namespace):
        """Obtain stats about an index.

        Usage:
            rvl stats -i <index_name> | -s <schema_path>
        """
        index = self._connect_to_index(args)
        _display_stats(index.info(), output_format=args.format)

    def _connect_to_index(self, args: Namespace) -> SearchIndex:
        # connect to redis
        try:
            redis_url = create_redis_url(args)
        except ValueError:
            logger.error(
                "Must set REDIS_ADDRESS environment variable or provide host and port"
            )
            exit(0)

        if args.index:
            schema = IndexSchema.from_dict({"index": {"name": args.index}})
            index = SearchIndex(schema=schema, redis_url=redis_url)
```

```
        elif args.schema:
            index = SearchIndex.from_yaml(args.schema, redis_url=redis_url)
        else:
            logger.error("Index name or schema must be provided")
            exit(0)

        return index


def _display_stats(index_info, output_format="rounded_outline"):
    # Extracting the statistics
    stats_data = [(key, str(index_info.get(key))) for key in STATS_KEYS]

    # Display the statistics in tabular format
    print("\nStatistics:")
    print(
        tabulate(
            stats_data,
            headers=["Stat Key", "Value"],
            tablefmt=output_format,
            colalign=("left", "left"),
        )
    )
}
```

redisvl/cli/utils.py

```
import os
from argparse import ArgumentParser, Namespace

from redisvl.utils.log import get_logger

logger = get_logger("[RedisVL]")


def create_redis_url(args: Namespace) -> str:
    env_address = os.getenv("REDIS_URL")
    if env_address:
        logger.info(f"Using Redis address from environment variable, REDIS_URL")
        return env_address
    elif args.url:
        return args.url
    else:
        url = "redis://"
        if args.ssl:
            url += "rediss://"
        if args.user:
            url += args.user
            if args.password:
                url += ":" + args.password
            url += "@"
        url += args.host + ":" + str(args.port)
        return url


def add_index_parsing_options(parser: ArgumentParser) -> ArgumentParser:
    parser.add_argument("-i", "--index", help="Index name", type=str, required=False)
    parser.add_argument(
        "-s", "--schema", help="Path to schema file", type=str, required=False
    )
```

```
    parser.add_argument("-u", "--url", help="Redis URL", type=str, required=False)
    parser.add_argument("--host", help="Redis host", type=str, default="localhost")
    parser.add_argument("-p", "--port", help="Redis port", type=int, default=6379)
    parser.add_argument("--user", help="Redis username", type=str, default="default")
    parser.add_argument("--ssl", help="Use SSL", action="store_true")
    parser.add_argument("-a", "--password", help="Redis password",
type=str, default="")
    return parser
}
```

redisvl/cli/version.py

```
import argparse
import sys
from argparse import Namespace

from redisvl import __version__
from redisvl.utils.log import get_logger

logger = get_logger("[RedisVL]")


class Version:
    usage = "\n".join(
        [
            "rvl version [<args>]\n",
            "\n",
        ]
    )

    def __init__(self):
        parser = argparse.ArgumentParser(usage=self.usage)
        parser.add_argument(
            "-s", "--short", help="Print only the version number", action="store_true"
        )

        args = parser.parse_args(sys.argv[2:])
        self.version(args)

    def version(self, args: Namespace):
        if args.short:
            print(__version__)
        else:
            logger.info(f"RedisVL version {__version__}")
}
```

redisvl/exceptions.py

```
class RedisVLException(Exception):
    """Base RedisVL exception"""


class RedisModuleVersionError(RedisVLException):
    """Invalid module versions installed"""
```

```
class RedisSearchError(RedisVLException):
    """Error while performing a search or aggregate request"""
}
```

Chapter 2.2.0

redisvl/extensions

redisvl/extensions/constants.py

```
"""
Constants used within the extension classes SemanticCache, BaseSessionManager,
StandardSessionManager,SemanticSessionManager and SemanticRouter.
These constants are also used within theses classes corresponding schema.
"""

# BaseSessionManager
ID_FIELD_NAME: str = "entry_id"
ROLE_FIELD_NAME: str = "role"
CONTENT_FIELD_NAME: str = "content"
TOOL_FIELD_NAME: str = "tool_call_id"
TIMESTAMP_FIELD_NAME: str = "timestamp"
SESSION_FIELD_NAME: str = "session_tag"

# SemanticSessionManager
SESSION_VECTOR_FIELD_NAME: str = "vector_field"

# SemanticCache
REDIS_KEY_FIELD_NAME: str = "key"
ENTRY_ID_FIELD_NAME: str = "entry_id"
PROMPT_FIELD_NAME: str = "prompt"
RESPONSE_FIELD_NAME: str = "response"
CACHE_VECTOR_FIELD_NAME: str = "prompt_vector"
INSERTED_AT_FIELD_NAME: str = "inserted_at"
UPDATED_AT_FIELD_NAME: str = "updated_at"
METADATA_FIELD_NAME: str = "metadata"

# SemanticRouter
ROUTE_VECTOR_FIELD_NAME: str = "vector"
}
```

Chapter 2.2.1

redisvl/extensions/llmcache

redisvl/extensions/llmcache/__init__.py

```
from redisvl.extensions.llmcache.semantic import SemanticCache

__all__ = ["SemanticCache"]
}
```

redisvl/extensions/llmcache/base.py

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


class BaseLLMCache:
    def __init__(self, ttl: Optional[int] = None):
        self._ttl: Optional[int] = None
        self.set_ttl(ttl)

    @property
    def ttl(self) -> Optional[int]:
        """The default TTL, in seconds, for entries in the cache."""
        return self._ttl

    def set_ttl(self, ttl: Optional[int] = None):
        """Set the default TTL, in seconds, for entries in the cache.

        Args:
            ttl (Optional[int], optional): The optional time-to-live expiration
                for the cache, in seconds.

        Raises:
            ValueError: If the time-to-live value is not an integer.
        """
        if ttl:
            if not isinstance(ttl, int):
                raise ValueError(f"TTL must be an integer value, got {ttl}")
            self._ttl = int(ttl)
        else:
            self._ttl = None

    def clear(self) -> None:
        """Clear the cache of all keys in the index."""
        raise NotImplementedError

    async def aclear(self) -> None:
        """Async clear the cache of all keys in the index."""
        raise NotImplementedError

    def check(
        self,
        prompt: Optional[str] = None,
        vector: Optional[List[float]] = None,
        num_results: int = 1,
        return_fields: Optional[List[str]] = None,
    ) -> List[dict]:
        """Check the cache based on a prompt or vector."""
        raise NotImplementedError

    async def acheck(
```

```
        self,
        prompt: Optional[str] = None,
        vector: Optional[List[float]] = None,
        num_results: int = 1,
        return_fields: Optional[List[str]] = None,
    ) -> List[dict]:
        """Async check the cache based on a prompt or vector."""
        raise NotImplementedError

    def store(
        self,
        prompt: str,
        response: str,
        vector: Optional[List[float]] = None,
        metadata: Optional[dict] = {},
    ) -> str:
        """Store the specified key-value pair in the cache along with
        metadata."""
        raise NotImplementedError

    async def astore(
        self,
        prompt: str,
        response: str,
        vector: Optional[List[float]] = None,
        metadata: Optional[dict] = {},
    ) -> str:
        """Async store the specified key-value pair in the cache along with
        metadata."""
        raise NotImplementedError
}
```

redisvl/extensions/llmcache/schema.py

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

from pydantic.v1 import BaseModel, Field, root_validator, validator

from redisvl.extensions.constants import (
    CACHE_VECTOR_FIELD_NAME,
    INSERTED_AT_FIELD_NAME,
    PROMPT_FIELD_NAME,
    RESPONSE_FIELD_NAME,
    UPDATED_AT_FIELD_NAME,
)
from redisvl.redis.utils import array_to_buffer, hashify
from redisvl.schema import IndexSchema
from redisvl.utils.utils import current_timestamp, deserialize, serialize


class CacheEntry(BaseModel):
    """A single cache entry in Redis"""

    entry_id: Optional[str] = Field(default=None)
    """Cache entry identifier"""
    prompt: str
    """Input prompt or question cached in Redis"""
    response: str
    """Response or answer to the question, cached in Redis"""
    prompt_vector: List[float]
```

```
    """Text embedding representation of the prompt"""
    inserted_at: float = Field(default_factory=current_timestamp)
    """Timestamp of when the entry was added to the cache"""
    updated_at: float = Field(default_factory=current_timestamp)
    """Timestamp of when the entry was updated in the cache"""
    metadata: Optional[Dict[str, Any]] = Field(default=None)
    """Optional metadata stored on the cache entry"""
    filters: Optional[Dict[str, Any]] = Field(default=None)
    """Optional filter data stored on the cache entry for customizing retrieval"""

    @root_validator(pre=True)
    @classmethod
    def generate_id(cls, values):
        # Ensure entry_id is set
        if not values.get("entry_id"):
            values["entry_id"] = hashify(values["prompt"], values.get("filters"))
        return values

    @validator("metadata")
    def non_empty_metadata(cls, v):
        if v is not None and not isinstance(v, dict):
            raise TypeError("Metadata must be a dictionary.")
        return v

    def to_dict(self, dtype: str) -> Dict:
        data = self.dict(exclude_none=True)
        data["prompt_vector"] = array_to_buffer(self.prompt_vector, dtype)
        if self.metadata is not None:
            data["metadata"] = serialize(self.metadata)
        if self.filters is not None:
            data.update(self.filters)
            del data["filters"]
        return data


class CacheHit(BaseModel):
    """A cache hit based on some input query"""

    entry_id: str
    """Cache entry identifier"""
    prompt: str
    """Input prompt or question cached in Redis"""
    response: str
    """Response or answer to the question, cached in Redis"""
    vector_distance: float
    """The semantic distance between the query vector and the stored prompt vector"""
    inserted_at: float
    """Timestamp of when the entry was added to the cache"""
    updated_at: float
    """Timestamp of when the entry was updated in the cache"""
    metadata: Optional[Dict[str, Any]] = Field(default=None)
    """Optional metadata stored on the cache entry"""
    filters: Optional[Dict[str, Any]] = Field(default=None)
    """Optional filter data stored on the cache entry for customizing retrieval"""

    @root_validator(pre=True)
    @classmethod
    def validate_cache_hit(cls, values):
        # Deserialize metadata if necessary
        if "metadata" in values and isinstance(values["metadata"], str):
            values["metadata"] = deserialize(values["metadata"])

        # Separate filters from other fields
        known_fields = set(cls.__fields__.keys())
```

```
        filters = {k: v for k, v in values.items() if k not in known_fields}

        # Add filters to values
        if filters:
            values["filters"] = filters

        # Remove filter fields from the main values
        for k in filters:
            values.pop(k)

        return values

    def to_dict(self) -> Dict:
        data = self.dict(exclude_none=True)
        if self.filters:
            data.update(self.filters)
            del data["filters"]

        return data


class SemanticCacheIndexSchema(IndexSchema):

    @classmethod
    def from_params(cls, name: str, prefix: str, vector_dims: int, dtype: str):

        return cls(
            index={"name": name, "prefix": prefix},  # type: ignore
            fields=[  # type: ignore
                {"name": PROMPT_FIELD_NAME, "type": "text"},
                {"name": RESPONSE_FIELD_NAME, "type": "text"},
                {"name": INSERTED_AT_FIELD_NAME, "type": "numeric"},
                {"name": UPDATED_AT_FIELD_NAME, "type": "numeric"},
                {
                    "name": CACHE_VECTOR_FIELD_NAME,
                    "type": "vector",
                    "attrs": {
                        "dims": vector_dims,
                        "datatype": dtype,
                        "distance_metric": "cosine",
                        "algorithm": "flat",
                    },
                },
            ],
        )
}
```

redisvl/extensions/llmcache/semantic.py

```
import asyncio
from typing import Any, Dict, List, Optional

from redis import Redis

from redisvl.extensions.constants import (
    CACHE_VECTOR_FIELD_NAME,
    ENTRY_ID_FIELD_NAME,
    INSERTED_AT_FIELD_NAME,
    METADATA_FIELD_NAME,
    PROMPT_FIELD_NAME,
```

```
    REDIS_KEY_FIELD_NAME,
    RESPONSE_FIELD_NAME,
    UPDATED_AT_FIELD_NAME,
)
from redisvl.extensions.llmcache.base import BaseLLMCache
from redisvl.extensions.llmcache.schema import (
    CacheEntry,
    CacheHit,
    SemanticCacheIndexSchema,
)
from redisvl.index import AsyncSearchIndex, SearchIndex
from redisvl.query import RangeQuery
from redisvl.query.filter import FilterExpression
from redisvl.utils.utils import current_timestamp, serialize, validate_vector_dims
from redisvl.utils.vectorize import BaseVectorizer, HFTextVectorizer


class SemanticCache(BaseLLMCache):
    """Semantic Cache for Large Language Models."""

    _index: SearchIndex
    _aindex: Optional[AsyncSearchIndex] = None

    def __init__(
        self,
        name: str = "llmcache",
        distance_threshold: float = 0.1,
        ttl: Optional[int] = None,
        vectorizer: Optional[BaseVectorizer] = None,
        filterable_fields: Optional[List[Dict[str, Any]]] = None,
        redis_client: Optional[Redis] = None,
        redis_url: str = "redis://localhost:6379",
        connection_kwargs: Dict[str, Any] = {},
        overwrite: bool = False,
        **kwargs,
    ):
        """Semantic Cache for Large Language Models.

        Args:
            name (str, optional): The name of the semantic cache search index.
                Defaults to "llmcache".
            distance_threshold (float, optional): Semantic threshold for the
                cache. Defaults to 0.1.
            ttl (Optional[int], optional): The time-to-live for records cached
                in Redis. Defaults to None.
            vectorizer (Optional[BaseVectorizer], optional): The vectorizer for
the cache.
                Defaults to HFTextVectorizer.
            filterable_fields (Optional[List[Dict[str, Any]]]): An optional list of
RedisVL fields
                that can be used to customize cache retrieval with filters.
            redis_client(Optional[Redis], optional): A redis client
connection instance.
                Defaults to None.
            redis_url (str, optional): The redis url. Defaults
to redis://localhost:6379.
            connection_kwargs (Dict[str, Any]): The connection arguments
                for the redis client. Defaults to empty {}.
            overwrite (bool): Whether or not to force overwrite the schema for
                the semantic cache index. Defaults to false.

        Raises:
            TypeError: If an invalid vectorizer is provided.
            TypeError: If the TTL value is not an int.
```

```
            ValueError: If the threshold is not between 0 and 1.
            ValueError: If existing schema does not match new schema and overwrite
is False.
        """
        super().__init__(ttl)

        self.redis_kwargs = {
            "redis_client": redis_client,
            "redis_url": redis_url,
            "connection_kwargs": connection_kwargs,
        }

        # Use the index name as the key prefix by default
        if "prefix" in kwargs:
            prefix = kwargs["prefix"]
        else:
            prefix = name

        # Set vectorizer default
        if vectorizer is None:
            vectorizer = HFTextVectorizer(
                model="sentence-transformers/all-mpnet-base-v2"
            )

        # Process fields and other settings
        self.set_threshold(distance_threshold)
        self.return_fields = [
            ENTRY_ID_FIELD_NAME,
            PROMPT_FIELD_NAME,
            RESPONSE_FIELD_NAME,
            INSERTED_AT_FIELD_NAME,
            UPDATED_AT_FIELD_NAME,
            METADATA_FIELD_NAME,
        ]

        # Create semantic cache schema and index
        dtype = kwargs.get("dtype", "float32")
        schema = SemanticCacheIndexSchema.from_params(
            name, prefix, vectorizer.dims, dtype
        )
        schema = self._modify_schema(schema, filterable_fields)
        self._index = SearchIndex(schema=schema)

        # Handle redis connection
        if redis_client:
            self._index.set_client(redis_client)
        elif redis_url:
            self._index.connect(redis_url=redis_url, **connection_kwargs)

        # Check for existing cache index
        if not overwrite and self._index.exists():
            existing_index = SearchIndex.from_existing(
                name, redis_client=self._index.client
            )
            if existing_index.schema != self._index.schema:
                raise ValueError(
                    f"Existing index {name} schema does not match the user provided
schema for the semantic cache. "
                    "If you wish to overwrite the index schema, set overwrite=True
during initialization."
                )

        # Create the search index
        self._index.create(overwrite=overwrite, drop=False)
```

```
        # Initialize and validate vectorizer
        if not isinstance(vectorizer, BaseVectorizer):
            raise TypeError("Must provide a valid redisvl.vectorizer class.")

        validate_vector_dims(
            vectorizer.dims,
            self._index.schema.fields[CACHE_VECTOR_FIELD_NAME].attrs.dims,  #
type: ignore
        )
        self._vectorizer = vectorizer
        self._dtype = self.index.schema.fields[CACHE_VECTOR_FIELD_NAME].attrs.datatype
# type: ignore[union-attr]

    def _modify_schema(
        self,
        schema: SemanticCacheIndexSchema,
        filterable_fields: Optional[List[Dict[str, Any]]] = None,
    ) -> SemanticCacheIndexSchema:
        """Modify the base cache schema using the provided filterable fields"""

        if filterable_fields is not None:
            protected_field_names = set(self.return_fields + [REDIS_KEY_FIELD_NAME])
            for filter_field in filterable_fields:
                field_name = filter_field["name"]
                if field_name in protected_field_names:
                    raise ValueError(
                        f"{field_name} is a reserved field name for the semantic
cache schema"
                    )
                # Add to schema
                schema.add_field(filter_field)
                # Add to return fields too
                self.return_fields.append(field_name)

        return schema

    async def _get_async_index(self) -> AsyncSearchIndex:
        """Lazily construct the async search index class."""
        if not self._aindex:
            # Construct async index if necessary
            self._aindex = AsyncSearchIndex(schema=self._index.schema)
            # Connect Redis async client
            redis_client = self.redis_kwargs["redis_client"]
            redis_url = self.redis_kwargs["redis_url"]
            connection_kwargs = self.redis_kwargs["connection_kwargs"]
            if redis_client is not None:
                await self._aindex.set_client(redis_client)
            elif redis_url:
                await self._aindex.connect(redis_url, **connection_kwargs)  #
type: ignore
        return self._aindex

    @property
    def index(self) -> SearchIndex:
        """The underlying SearchIndex for the cache.

        Returns:
            SearchIndex: The search index.
        """
        return self._index

    @property
    def aindex(self) -> Optional[AsyncSearchIndex]:
```

```
        """The underlying AsyncSearchIndex for the cache.

        Returns:
            AsyncSearchIndex: The async search index.
        """
        return self._aindex

    @property
    def distance_threshold(self) -> float:
        """The semantic distance threshold for the cache.

        Returns:
            float: The semantic distance threshold.
        """
        return self._distance_threshold

    def set_threshold(self, distance_threshold: float) -> None:
        """Sets the semantic distance threshold for the cache.

        Args:
            distance_threshold (float): The semantic distance threshold for
                the cache.

        Raises:
            ValueError: If the threshold is not between 0 and 1.
        """
        if not 0 <= float(distance_threshold) <= 1:
            raise ValueError(
                f"Distance must be between 0 and 1, got {distance_threshold}"
            )
        self._distance_threshold = float(distance_threshold)

    def clear(self) -> None:
        """Clear the cache of all keys while preserving the index."""
        self._index.clear()

    async def aclear(self) -> None:
        """Clear the cache of all keys while preserving the index.
        """
        aindex = await self._get_async_index()
        await aindex.clear()

    def delete(self) -> None:
        """Clear the semantic cache of all keys and remove the underlying search
        index."""
        self._index.delete(drop=True)

    async def adelete(self) -> None:
        """Clear the semantic cache of all keys and remove the underlying search
        index.
        """
        aindex = await self._get_async_index()
        await aindex.delete(drop=True)

    def drop(
        self, ids: Optional[List[str]] = None, keys: Optional[List[str]] = None
    ) -> None:
        """Manually expire specific entries from the cache by id or specific
        Redis key.

        Args:
            ids (Optional[str]): The document ID or IDs to remove from the cache.
            keys (Optional[str]): The Redis keys to remove from the cache.
        """
        if ids is not None:
            self._index.drop_keys([self._index.key(id) for id in ids])
        if keys is not None:
```

```
            self._index.drop_keys(keys)

    async def adrop(
        self, ids: Optional[List[str]] = None, keys: Optional[List[str]] = None
    ) -> None:
        """Async expire specific entries from the cache by id or specific
        Redis key.

        Args:
            ids (Optional[str]): The document ID or IDs to remove from the cache.
            keys (Optional[str]): The Redis keys to remove from the cache.
        """
        aindex = await self._get_async_index()

        if ids is not None:
            await aindex.drop_keys([self._index.key(id) for id in ids])
        if keys is not None:
            await aindex.drop_keys(keys)

    def _refresh_ttl(self, key: str) -> None:
        """Refresh the time-to-live for the specified key."""
        if self._ttl:
            self._index.client.expire(key, self._ttl)  # type: ignore

    async def _async_refresh_ttl(self, key: str) -> None:
        """Async refresh the time-to-live for the specified key."""
        aindex = await self._get_async_index()
        if self._ttl:
            await aindex.client.expire(key, self._ttl)  # type: ignore

    def _vectorize_prompt(self, prompt: Optional[str]) -> List[float]:
        """Converts a text prompt to its vector representation using the
        configured vectorizer."""
        if not isinstance(prompt, str):
            raise TypeError("Prompt must be a string.")

        return self._vectorizer.embed(prompt, dtype=self._dtype)

    async def _avectorize_prompt(self, prompt: Optional[str]) -> List[float]:
        """Converts a text prompt to its vector representation using the
        configured vectorizer."""
        if not isinstance(prompt, str):
            raise TypeError("Prompt must be a string.")

        return await self._vectorizer.aembed(prompt)

    def _check_vector_dims(self, vector: List[float]):
        """Checks the size of the provided vector and raises an error if it
        doesn't match the search index vector dimensions."""
        schema_vector_dims =
self._index.schema.fields[CACHE_VECTOR_FIELD_NAME].attrs.dims  # type: ignore
        validate_vector_dims(len(vector), schema_vector_dims)

    def check(
        self,
        prompt: Optional[str] = None,
        vector: Optional[List[float]] = None,
        num_results: int = 1,
        return_fields: Optional[List[str]] = None,
        filter_expression: Optional[FilterExpression] = None,
        distance_threshold: Optional[float] = None,
    ) -> List[Dict[str, Any]]:
        """Checks the semantic cache for results similar to the specified prompt
        or vector.
```

```
        This method searches the cache using vector similarity with
        either a raw text prompt (converted to a vector) or a provided vector as
        input. It checks for semantically similar prompts and fetches the cached
        LLM responses.

        Args:
            prompt (Optional[str], optional): The text prompt to search for in
                the cache.
            vector (Optional[List[float]], optional): The vector representation
                of the prompt to search for in the cache.
            num_results (int, optional): The number of cached results to return.
                Defaults to 1.
            return_fields (Optional[List[str]], optional): The fields to include
                in each returned result. If None, defaults to all available
                fields in the cached entry.
            filter_expression (Optional[FilterExpression]) : Optional filter expression
                that can be used to filter cache results. Defaults to None and
                the full cache will be searched.
            distance_threshold (Optional[float]): The threshold for semantic
                vector distance.

        Returns:
            List[Dict[str, Any]]: A list of dicts containing the requested
                return fields for each similar cached response.

        Raises:
            ValueError: If neither a `prompt` nor a `vector` is specified.
            ValueError: if 'vector' has incorrect dimensions.
            TypeError: If `return_fields` is not a list when provided.

        .. code-block:: python

            response = cache.check(
                prompt="What is the captial city of France?"
            )
        """
        if not any([prompt, vector]):
            raise ValueError("Either prompt or vector must be specified.")
        if return_fields and not isinstance(return_fields, list):
            raise TypeError("Return fields must be a list of values.")

        # overrides
        distance_threshold = distance_threshold or self._distance_threshold
        vector = vector or self._vectorize_prompt(prompt)
        self._check_vector_dims(vector)

        query = RangeQuery(
            vector=vector,
            vector_field_name=CACHE_VECTOR_FIELD_NAME,
            return_fields=self.return_fields,
            distance_threshold=distance_threshold,
            num_results=num_results,
            return_score=True,
            filter_expression=filter_expression,
            dtype=self._dtype,
        )

        # Search the cache!
        cache_search_results = self._index.query(query)
        redis_keys, cache_hits = self._process_cache_results(
            cache_search_results, return_fields  # type: ignore
        )
        # Extend TTL on keys
```

```
        for key in redis_keys:
            self._refresh_ttl(key)

        return cache_hits

    async def acheck(
        self,
        prompt: Optional[str] = None,
        vector: Optional[List[float]] = None,
        num_results: int = 1,
        return_fields: Optional[List[str]] = None,
        filter_expression: Optional[FilterExpression] = None,
        distance_threshold: Optional[float] = None,
    ) -> List[Dict[str, Any]]:
        """Async check the semantic cache for results similar to the specified prompt
        or vector.

        This method searches the cache using vector similarity with
        either a raw text prompt (converted to a vector) or a provided vector as
        input. It checks for semantically similar prompts and fetches the cached
        LLM responses.

        Args:
            prompt (Optional[str], optional): The text prompt to search for in
                the cache.
            vector (Optional[List[float]], optional): The vector representation
                of the prompt to search for in the cache.
            num_results (int, optional): The number of cached results to return.
                Defaults to 1.
            return_fields (Optional[List[str]], optional): The fields to include
                in each returned result. If None, defaults to all available
                fields in the cached entry.
            filter_expression (Optional[FilterExpression]) : Optional filter expression
                that can be used to filter cache results. Defaults to None and
                the full cache will be searched.
            distance_threshold (Optional[float]): The threshold for semantic
                vector distance.

        Returns:
            List[Dict[str, Any]]: A list of dicts containing the requested
                return fields for each similar cached response.

        Raises:
            ValueError: If neither a `prompt` nor a `vector` is specified.
            ValueError: if 'vector' has incorrect dimensions.
            TypeError: If `return_fields` is not a list when provided.

        .. code-block:: python

            response = await cache.acheck(
                prompt="What is the captial city of France?"
            )
        """
        aindex = await self._get_async_index()

        if not any([prompt, vector]):
            raise ValueError("Either prompt or vector must be specified.")
        if return_fields and not isinstance(return_fields, list):
            raise TypeError("Return fields must be a list of values.")

        # overrides
        distance_threshold = distance_threshold or self._distance_threshold
        vector = vector or await self._avectorize_prompt(prompt)
        self._check_vector_dims(vector)
```

```
        query = RangeQuery(
            vector=vector,
            vector_field_name=CACHE_VECTOR_FIELD_NAME,
            return_fields=self.return_fields,
            distance_threshold=distance_threshold,
            num_results=num_results,
            return_score=True,
            filter_expression=filter_expression,
        )

        # Search the cache!
        cache_search_results = await aindex.query(query)
        redis_keys, cache_hits = self._process_cache_results(
            cache_search_results, return_fields  # type: ignore
        )
        # Extend TTL on keys
        asyncio.gather(*[self._async_refresh_ttl(key) for key in redis_keys])

        return cache_hits

    def _process_cache_results(
        self, cache_search_results: List[Dict[str, Any]], return_fields: List[str]
    ):
        redis_keys: List[str] = []
        cache_hits: List[Dict[Any, str]] = []
        for cache_search_result in cache_search_results:
            # Pop the redis key from the result
            redis_key = cache_search_result.pop("id")
            redis_keys.append(redis_key)
            # Create and process cache hit
            cache_hit = CacheHit(**cache_search_result)
            cache_hit_dict = cache_hit.to_dict()
            # Filter down to only selected return fields if needed
            if isinstance(return_fields, list) and len(return_fields) > 0:
                cache_hit_dict = {
                    k: v for k, v in cache_hit_dict.items() if k in return_fields
                }
            cache_hit_dict[REDIS_KEY_FIELD_NAME] = redis_key
            cache_hits.append(cache_hit_dict)
        return redis_keys, cache_hits

    def store(
        self,
        prompt: str,
        response: str,
        vector: Optional[List[float]] = None,
        metadata: Optional[Dict[str, Any]] = None,
        filters: Optional[Dict[str, Any]] = None,
        ttl: Optional[int] = None,
    ) -> str:
        """Stores the specified key-value pair in the cache along with metadata.

        Args:
            prompt (str): The user prompt to cache.
            response (str): The LLM response to cache.
            vector (Optional[List[float]], optional): The prompt vector to
                cache. Defaults to None, and the prompt vector is generated on
                demand.
            metadata (Optional[Dict[str, Any]], optional): The optional metadata
to cache
                alongside the prompt and response. Defaults to None.
            filters (Optional[Dict[str, Any]]): The optional tag to assign to the
cache entry.
```

```
                Defaults to None.
            ttl (Optional[int]): The optional TTL override to use on this
individual cache
                entry. Defaults to the global TTL setting.

        Returns:
            str: The Redis key for the entries added to the semantic cache.

        Raises:
            ValueError: If neither prompt nor vector is specified.
            ValueError: if vector has incorrect dimensions.
            TypeError: If provided metadata is not a dictionary.

        .. code-block:: python

            key = cache.store(
                prompt="What is the captial city of France?",
                response="Paris",
                metadata={"city": "Paris", "country": "France"}
            )
        """
        # Vectorize prompt if necessary and create cache payload
        vector = vector or self._vectorize_prompt(prompt)
        self._check_vector_dims(vector)

        # Build cache entry for the cache
        cache_entry = CacheEntry(
            prompt=prompt,
            response=response,
            prompt_vector=vector,
            metadata=metadata,
            filters=filters,
        )

        # Load cache entry with TTL
        ttl = ttl or self._ttl
        keys = self._index.load(
            data=[cache_entry.to_dict(self._dtype)],
            ttl=ttl,
            id_field=ENTRY_ID_FIELD_NAME,
        )
        return keys[0]

    async def astore(
        self,
        prompt: str,
        response: str,
        vector: Optional[List[float]] = None,
        metadata: Optional[Dict[str, Any]] = None,
        filters: Optional[Dict[str, Any]] = None,
        ttl: Optional[int] = None,
    ) -> str:
        """Async stores the specified key-value pair in the cache along with metadata.

        Args:
            prompt (str): The user prompt to cache.
            response (str): The LLM response to cache.
            vector (Optional[List[float]], optional): The prompt vector to
                cache. Defaults to None, and the prompt vector is generated on
                demand.
            metadata (Optional[Dict[str, Any]], optional): The optional metadata
to cache
                alongside the prompt and response. Defaults to None.
            filters (Optional[Dict[str, Any]]): The optional tag to assign to the
```

```
cache entry.
                Defaults to None.
            ttl (Optional[int]): The optional TTL override to use on this
individual cache
                entry. Defaults to the global TTL setting.

        Returns:
            str: The Redis key for the entries added to the semantic cache.

        Raises:
            ValueError: If neither prompt nor vector is specified.
            ValueError: if vector has incorrect dimensions.
            TypeError: If provided metadata is not a dictionary.

        .. code-block:: python

            key = await cache.astore(
                prompt="What is the captial city of France?",
                response="Paris",
                metadata={"city": "Paris", "country": "France"}
            )
        """
        aindex = await self._get_async_index()

        # Vectorize prompt if necessary and create cache payload
        vector = vector or self._vectorize_prompt(prompt)
        self._check_vector_dims(vector)

        # Build cache entry for the cache
        cache_entry = CacheEntry(
            prompt=prompt,
            response=response,
            prompt_vector=vector,
            metadata=metadata,
            filters=filters,
        )

        # Load cache entry with TTL
        ttl = ttl or self._ttl
        keys = await aindex.load(
            data=[cache_entry.to_dict(self._dtype)],
            ttl=ttl,
            id_field=ENTRY_ID_FIELD_NAME,
        )
        return keys[0]

    def update(self, key: str, **kwargs) -> None:
        """Update specific fields within an existing cache entry. If no fields
        are passed, then only the document TTL is refreshed.

        Args:
            key (str): the key of the document to update using kwargs.

        Raises:
            ValueError if an incorrect mapping is provided as a kwarg.
            TypeError if metadata is provided and not of type dict.

        .. code-block:: python

            key = cache.store('this is a prompt', 'this is a response')
            cache.update(key, metadata={"hit_count": 1, "model_name": "Llama-2-7b"})
            )
        """
        if kwargs:
```

```
            for k, v in kwargs.items():

                # Make sure the item is in the index schema
                if k not in set(self._index.schema.field_names
+ [METADATA_FIELD_NAME]):
                    raise ValueError(f"{k} is not a valid field within the
cache entry")

                # Check for metadata and deserialize
                if k == METADATA_FIELD_NAME:
                    if isinstance(v, dict):
                        kwargs[k] = serialize(v)
                    else:
                        raise TypeError(
                            "If specified, cached metadata must be a dictionary."
                        )

            kwargs.update({UPDATED_AT_FIELD_NAME: current_timestamp()})

            self._index.client.hset(key, mapping=kwargs)  # type: ignore

        self._refresh_ttl(key)

    async def aupdate(self, key: str, **kwargs) -> None:
        """Async update specific fields within an existing cache entry. If no fields
        are passed, then only the document TTL is refreshed.

        Args:
            key (str): the key of the document to update using kwargs.

        Raises:
            ValueError if an incorrect mapping is provided as a kwarg.
            TypeError if metadata is provided and not of type dict.

        .. code-block:: python

            key = await cache.astore('this is a prompt', 'this is a response')
            await cache.aupdate(
                key,
                metadata={"hit_count": 1, "model_name": "Llama-2-7b"}
            )
        """
        aindex = await self._get_async_index()

        if kwargs:
            for k, v in kwargs.items():

                # Make sure the item is in the index schema
                if k not in set(self._index.schema.field_names
+ [METADATA_FIELD_NAME]):
                    raise ValueError(f"{k} is not a valid field within the
cache entry")

                # Check for metadata and deserialize
                if k == METADATA_FIELD_NAME:
                    if isinstance(v, dict):
                        kwargs[k] = serialize(v)
                    else:
                        raise TypeError(
                            "If specified, cached metadata must be a dictionary."
                        )

            kwargs.update({UPDATED_AT_FIELD_NAME: current_timestamp()})
```

```
            await aindex.load(data=[kwargs], keys=[key])

        await self._async_refresh_ttl(key)
}
```

Chapter 2.2.2

redisvl/extensions/router

redisvl/extensions/router/__init__.py

```
from redisvl.extensions.router.schema import Route, RoutingConfig
from redisvl.extensions.router.semantic import SemanticRouter

__all__ = ["SemanticRouter", "Route", "RoutingConfig"]
}
```

redisvl/extensions/router/schema.py

```
from enum import Enum
from typing import Dict, List, Optional

from pydantic.v1 import BaseModel, Field, validator

from redisvl.extensions.constants import ROUTE_VECTOR_FIELD_NAME
from redisvl.schema import IndexSchema


class Route(BaseModel):
    """Model representing a routing path with associated metadata and thresholds."""

    name: str
    """The name of the route."""
    references: List[str]
    """List of reference phrases for the route."""
    metadata: Dict[str, str] = Field(default={})
    """Metadata associated with the route."""
    distance_threshold: Optional[float] = Field(default=None)
    """Distance threshold for matching the route."""

    @validator("name")
    def name_must_not_be_empty(cls, v):
        if not v or not v.strip():
            raise ValueError("Route name must not be empty")
        return v

    @validator("references")
    def references_must_not_be_empty(cls, v):
        if not v:
```

```
            raise ValueError("References must not be empty")
        if any(not ref.strip() for ref in v):
            raise ValueError("All references must be non-empty strings")
        return v

    @validator("distance_threshold")
    def distance_threshold_must_be_positive(cls, v):
        if v is not None and v <= 0:
            raise ValueError("Route distance threshold must be greater than zero")
        return v


class RouteMatch(BaseModel):
    """Model representing a matched route with distance information."""

    name: Optional[str] = None
    """The matched route name."""
    distance: Optional[float] = Field(default=None)
    """The vector distance between the statement and the matched route."""


class DistanceAggregationMethod(Enum):
    """Enumeration for distance aggregation methods."""

    avg = "avg"
    """Compute the average of the vector distances."""
    min = "min"
    """Compute the minimum of the vector distances."""
    sum = "sum"
    """Compute the sum of the vector distances."""


class RoutingConfig(BaseModel):
    """Configuration for routing behavior."""

    distance_threshold: float = Field(default=0.5)
    """The threshold for semantic distance."""
    max_k: int = Field(default=1)
    """The maximum number of top matches to return."""
    aggregation_method: DistanceAggregationMethod = Field(
        default=DistanceAggregationMethod.avg
    )
    """Aggregation method to use to classify queries."""

    @validator("max_k")
    def max_k_must_be_positive(cls, v):
        if v <= 0:
            raise ValueError("max_k must be a positive integer")
        return v

    @validator("distance_threshold")
    def distance_threshold_must_be_valid(cls, v):
        if v <= 0 or v > 1:
            raise ValueError("distance_threshold must be between 0 and 1")
        return v


class SemanticRouterIndexSchema(IndexSchema):
    """Customized index schema for SemanticRouter."""

    @classmethod
    def from_params(cls, name: str, vector_dims: int, dtype: str):
        """Create an index schema based on router name and vector dimensions.
```

```
        Args:
            name (str): The name of the index.
            vector_dims (int): The dimensions of the vectors.

        Returns:
            SemanticRouterIndexSchema: The constructed index schema.
        """
        return cls(
            index={"name": name, "prefix": name},  # type: ignore
            fields=[  # type: ignore
                {"name": "route_name", "type": "tag"},
                {"name": "reference", "type": "text"},
                {
                    "name": ROUTE_VECTOR_FIELD_NAME,
                    "type": "vector",
                    "attrs": {
                        "algorithm": "flat",
                        "dims": vector_dims,
                        "distance_metric": "cosine",
                        "datatype": dtype,
                    },
                },
            ],
        )
}
```

redisvl/extensions/router/semantic.py

```
from pathlib import Path
from typing import Any, Dict, List, Optional, Type

import redis.commands.search.reducers as reducers
import yaml
from pydantic.v1 import BaseModel, Field, PrivateAttr
from redis import Redis
from redis.commands.search.aggregation import AggregateRequest,
AggregateResult, Reducer
from redis.exceptions import ResponseError

from redisvl.extensions.constants import ROUTE_VECTOR_FIELD_NAME
from redisvl.extensions.router.schema import (
    DistanceAggregationMethod,
    Route,
    RouteMatch,
    RoutingConfig,
    SemanticRouterIndexSchema,
)
from redisvl.index import SearchIndex
from redisvl.query import RangeQuery
from redisvl.redis.utils import convert_bytes, hashify, make_dict
from redisvl.utils.log import get_logger
from redisvl.utils.utils import model_to_dict
from redisvl.utils.vectorize import (
    BaseVectorizer,
    HFTextVectorizer,
    vectorizer_from_dict,
)

logger = get_logger(__name__)
```

```
class SemanticRouter(BaseModel):
    """Semantic Router for managing and querying route vectors."""

    name: str
    """The name of the semantic router."""
    routes: List[Route]
    """List of Route objects."""
    vectorizer: BaseVectorizer = Field(default_factory=HFTextVectorizer)
    """The vectorizer used to embed route references."""
    routing_config: RoutingConfig = Field(default_factory=RoutingConfig)
    """Configuration for routing behavior."""

    _index: SearchIndex = PrivateAttr()

    class Config:
        arbitrary_types_allowed = True

    def __init__(
        self,
        name: str,
        routes: List[Route],
        vectorizer: Optional[BaseVectorizer] = None,
        routing_config: Optional[RoutingConfig] = None,
        redis_client: Optional[Redis] = None,
        redis_url: str = "redis://localhost:6379",
        overwrite: bool = False,
        connection_kwargs: Dict[str, Any] = {},
        **kwargs,
    ):
        """Initialize the SemanticRouter.

        Args:
            name (str): The name of the semantic router.
            routes (List[Route]): List of Route objects.
            vectorizer (BaseVectorizer, optional): The vectorizer used to embed route
references. Defaults to default HFTextVectorizer.
            routing_config (RoutingConfig, optional): Configuration for routing
behavior. Defaults to the default RoutingConfig.
            redis_client (Optional[Redis], optional): Redis client for connection.
Defaults to None.
            redis_url (str, optional): The redis url. Defaults
to redis://localhost:6379.
            overwrite (bool, optional): Whether to overwrite existing index. Defaults
to False.
            connection_kwargs (Dict[str, Any]): The connection arguments
                for the redis client. Defaults to empty {}.
        """
        # Set vectorizer default
        if vectorizer is None:
            vectorizer = HFTextVectorizer()

        if routing_config is None:
            routing_config = RoutingConfig()

        super().__init__(
            name=name,
            routes=routes,
            vectorizer=vectorizer,
            routing_config=routing_config,
        )
        dtype = kwargs.get("dtype", "float32")
        self._initialize_index(
            redis_client, redis_url, overwrite, dtype, **connection_kwargs
```

```
        )

    def _initialize_index(
        self,
        redis_client: Optional[Redis] = None,
        redis_url: str = "redis://localhost:6379",
        overwrite: bool = False,
        dtype: str = "float32",
        **connection_kwargs,
    ):
        """Initialize the search index and handle Redis connection."""
        schema = SemanticRouterIndexSchema.from_params(
            self.name, self.vectorizer.dims, dtype
        )
        self._index = SearchIndex(schema=schema)

        if redis_client:
            self._index.set_client(redis_client)
        elif redis_url:
            self._index.connect(redis_url=redis_url, **connection_kwargs)

        # Check for existing router index
        existed = self._index.exists()
        if not overwrite and existed:
            existing_index = SearchIndex.from_existing(
                self.name, redis_client=self._index.client
            )
            if existing_index.schema != self._index.schema:
                raise ValueError(
                    f"Existing index {self.name} schema does not match the user
provided schema for the semantic router. "
                    "If you wish to overwrite the index schema, set overwrite=True
during initialization."
                )
        self._index.create(overwrite=overwrite, drop=False)

        if not existed or overwrite:
            # write the routes to Redis
            self._add_routes(self.routes)

    @property
    def route_names(self) -> List[str]:
        """Get the list of route names.

        Returns:
            List[str]: List of route names.
        """
        return [route.name for route in self.routes]

    @property
    def route_thresholds(self) -> Dict[str, Optional[float]]:
        """Get the distance thresholds for each route.

        Returns:
            Dict[str, float]: Dictionary of route names and their distance thresholds.
        """
        return {route.name: route.distance_threshold for route in self.routes}

    def update_routing_config(self, routing_config: RoutingConfig):
        """Update the routing configuration.

        Args:
            routing_config (RoutingConfig): The new routing configuration.
        """
```

```
        self.routing_config = routing_config

    def _route_ref_key(self, route_name: str, reference: str) -> str:
        """Generate the route reference key."""
        reference_hash = hashify(reference)
        return f"{self._index.prefix}:{route_name}:{reference_hash}"

    def _add_routes(self, routes: List[Route]):
        """Add routes to the router and index.

        Args:
            routes (List[Route]): List of routes to be added.
        """
        route_references: List[Dict[str, Any]] = []
        keys: List[str] = []

        for route in routes:
            # embed route references as a single batch
            reference_vectors = self.vectorizer.embed_many(
                [reference for reference in route.references],
                as_buffer=True,

dtype=self._index.schema.fields[ROUTE_VECTOR_FIELD_NAME].attrs.datatype,  # type:
ignore[union-attr]
            )
            # set route references
            for i, reference in enumerate(route.references):
                route_references.append(
                    {
                        "route_name": route.name,
                        "reference": reference,
                        "vector": reference_vectors[i],
                    }
                )
                keys.append(self._route_ref_key(route.name, reference))

            # set route if does not yet exist client side
            if not self.get(route.name):
                self.routes.append(route)

        self._index.load(route_references, keys=keys)

    def get(self, route_name: str) -> Optional[Route]:
        """Get a route by its name.

        Args:
            route_name (str): Name of the route.

        Returns:
            Optional[Route]: The selected Route object or None if not found.
        """
        return next((route for route in self.routes if route.name == route_name), None)

    def _process_route(self, result: Dict[str, Any]) -> RouteMatch:
        """Process resulting route objects and metadata."""
        route_dict = make_dict(convert_bytes(result))
        return RouteMatch(
            name=route_dict["route_name"], distance=float(route_dict["distance"])
        )

    def _build_aggregate_request(
        self,
        vector_range_query: RangeQuery,
        aggregation_method: DistanceAggregationMethod,
```

```
        max_k: int,
    ) -> AggregateRequest:
        """Build the Redis aggregation request."""
        aggregation_func: Type[Reducer]

        if aggregation_method == DistanceAggregationMethod.min:
            aggregation_func = reducers.min
        elif aggregation_method == DistanceAggregationMethod.sum:
            aggregation_func = reducers.sum
        else:
            aggregation_func = reducers.avg

        aggregate_query = str(vector_range_query).split(" RETURN")[0]
        aggregate_request = (
            AggregateRequest(aggregate_query)
            .group_by(
                "@route_name", aggregation_func("vector_distance").alias("distance")
            )
            .sort_by("@distance", max=max_k)
            .dialect(2)
        )

        return aggregate_request

    def _classify_route(
        self,
        vector: List[float],
        distance_threshold: float,
        aggregation_method: DistanceAggregationMethod,
    ) -> RouteMatch:
        """Classify to a single route using a vector."""
        vector_range_query = RangeQuery(
            vector=vector,
            vector_field_name=ROUTE_VECTOR_FIELD_NAME,
            distance_threshold=distance_threshold,
            return_fields=["route_name"],
            dtype=self._index.schema.fields[ROUTE_VECTOR_FIELD_NAME].attrs.datatype,  #
type: ignore[union-attr]
        )

        aggregate_request = self._build_aggregate_request(
            vector_range_query, aggregation_method, max_k=1
        )

        try:
            aggregation_result: AggregateResult = self._index.aggregate(
                aggregate_request, vector_range_query.params
            )
        except ResponseError as e:
            if "VSS is not yet supported on FT.AGGREGATE" in str(e):
                raise RuntimeError(
                    "Semantic routing is only available on Redis version 7.x.x
or greater"
                )
            raise e

        # process aggregation results into route matches
        route_matches = [
            self._process_route(route_match) for route_match in aggregation_result.rows
        ]

        # process route matches
        if route_matches:
            top_route_match = route_matches[0]
```

```
            if top_route_match.name is not None:
                if route := self.get(top_route_match.name):
                    # use the matched route's distance threshold
                    _distance_threshold = route.distance_threshold
or distance_threshold
                    if self._pass_threshold(top_route_match, _distance_threshold):
                        return top_route_match
                else:
                    raise ValueError(
                        f"{top_route_match.name} not a supported route for the
{self.name} semantic router."
                    )

        # fallback to empty route match if no hits
        return RouteMatch()

    def _classify_multi_route(
        self,
        vector: List[float],
        max_k: int,
        distance_threshold: float,
        aggregation_method: DistanceAggregationMethod,
    ) -> List[RouteMatch]:
        """Classify to multiple routes, up to max_k (int), using a vector."""
        vector_range_query = RangeQuery(
            vector=vector,
            vector_field_name=ROUTE_VECTOR_FIELD_NAME,
            distance_threshold=distance_threshold,
            return_fields=["route_name"],
            dtype=self._index.schema.fields[ROUTE_VECTOR_FIELD_NAME].attrs.datatype,  #
type: ignore[union-attr]
        )
        aggregate_request = self._build_aggregate_request(
            vector_range_query, aggregation_method, max_k
        )

        try:
            aggregation_result: AggregateResult = self._index.aggregate(
                aggregate_request, vector_range_query.params
            )
        except ResponseError as e:
            if "VSS is not yet supported on FT.AGGREGATE" in str(e):
                raise RuntimeError(
                    "Semantic routing is only available on Redis version 7.x.x
or greater"
                )
            raise e

        # process aggregation results into route matches
        route_matches = [
            self._process_route(route_match) for route_match in aggregation_result.rows
        ]

        # process route matches
        top_route_matches: List[RouteMatch] = []
        if route_matches:
            for route_match in route_matches:
                if route_match.name is not None:
                    if route := self.get(route_match.name):
                        # use the matched route's distance threshold
                        _distance_threshold = (
                            route.distance_threshold or distance_threshold
                        )
                        if self._pass_threshold(route_match, _distance_threshold):
```

```
                            top_route_matches.append(route_match)
                    else:
                        raise ValueError(
                            f"{route_match.name} not a supported route for the
{self.name} semantic router."
                        )

        return top_route_matches

    def _pass_threshold(
        self, route_match: Optional[RouteMatch], distance_threshold: float
    ) -> bool:
        """Check if a route match passes the distance threshold.

        Args:
            route_match (Optional[RouteMatch]): The route match to check.
            distance_threshold (float): The fallback distance threshold to use if not
assigned to a route.

        Returns:
            bool: True if the route match passes the threshold, False otherwise.
        """
        if route_match and distance_threshold:
            if route_match.distance is not None:
                return route_match.distance <= distance_threshold
        return False

    def __call__(
        self,
        statement: Optional[str] = None,
        vector: Optional[List[float]] = None,
        distance_threshold: Optional[float] = None,
        aggregation_method: Optional[DistanceAggregationMethod] = None,
    ) -> RouteMatch:
        """Query the semantic router with a given statement or vector.

        Args:
            statement (Optional[str]): The input statement to be queried.
            vector (Optional[List[float]]): The input vector to be queried.
            distance_threshold (Optional[float]): The threshold for semantic distance.
            aggregation_method (Optional[DistanceAggregationMethod]): The aggregation
method used for vector distances.

        Returns:
            RouteMatch: The matching route.
        """
        if not vector:
            if not statement:
                raise ValueError("Must provide a vector or statement to the router")
            vector = self.vectorizer.embed(statement)

        # override routing config
        distance_threshold = (
            distance_threshold or self.routing_config.distance_threshold
        )
        aggregation_method = (
            aggregation_method or self.routing_config.aggregation_method
        )

        # perform route classification
        top_route_match = self._classify_route(
            vector, distance_threshold, aggregation_method
        )
        return top_route_match
```

```
    def route_many(
        self,
        statement: Optional[str] = None,
        vector: Optional[List[float]] = None,
        max_k: Optional[int] = None,
        distance_threshold: Optional[float] = None,
        aggregation_method: Optional[DistanceAggregationMethod] = None,
    ) -> List[RouteMatch]:
        """Query the semantic router with a given statement or vector for
multiple matches.

        Args:
            statement (Optional[str]): The input statement to be queried.
            vector (Optional[List[float]]): The input vector to be queried.
            max_k (Optional[int]): The maximum number of top matches to return.
            distance_threshold (Optional[float]): The threshold for semantic distance.
            aggregation_method (Optional[DistanceAggregationMethod]): The aggregation
method used for vector distances.

        Returns:
            List[RouteMatch]: The matching routes and their details.
        """
        if not vector:
            if not statement:
                raise ValueError("Must provide a vector or statement to the router")
            vector = self.vectorizer.embed(statement)

        # override routing config defaults
        distance_threshold = (
            distance_threshold or self.routing_config.distance_threshold
        )
        max_k = max_k or self.routing_config.max_k
        aggregation_method = (
            aggregation_method or self.routing_config.aggregation_method
        )

        # classify routes
        top_route_matches = self._classify_multi_route(
            vector, max_k, distance_threshold, aggregation_method
        )
        return top_route_matches

    def remove_route(self, route_name: str) -> None:
        """Remove a route and all references from the semantic router.

        Args:
            route_name (str): Name of the route to remove.
        """
        route = self.get(route_name)
        if route is None:
            logger.warning(f"Route {route_name} is not found in the SemanticRouter")
        else:
            self._index.drop_keys(
                [
                    self._route_ref_key(route.name, reference)
                    for reference in route.references
                ]
            )
            self.routes = [route for route in self.routes if route.name != route_name]

    def delete(self) -> None:
        """Delete the semantic router index."""
        self._index.delete(drop=True)
```

```
    def clear(self) -> None:
        """Flush all routes from the semantic router index."""
        self._index.clear()
        self.routes = []

    @classmethod
    def from_dict(
        cls,
        data: Dict[str, Any],
        **kwargs,
    ) -> "SemanticRouter":
        """Create a SemanticRouter from a dictionary.

        Args:
            data (Dict[str, Any]): The dictionary containing the semantic router data.

        Returns:
            SemanticRouter: The semantic router instance.

        Raises:
            ValueError: If required data is missing or invalid.

        .. code-block:: python

            from redisvl.extensions.router import SemanticRouter
            router_data = {
                "name": "example_router",
                "routes": [{"name": "route1", "references": ["ref1"],
"distance_threshold": 0.5}],
                "vectorizer": {"type": "openai", "model": "text-embedding-ada-002"},
            }
            router = SemanticRouter.from_dict(router_data)
        """
        try:
            name = data["name"]
            routes_data = data["routes"]
            vectorizer_data = data["vectorizer"]
            routing_config_data = data["routing_config"]
        except KeyError as e:
            raise ValueError(f"Unable to load semantic router from dict: {str(e)}")

        try:
            vectorizer = vectorizer_from_dict(vectorizer_data)
        except Exception as e:
            raise ValueError(f"Unable to load vectorizer: {str(e)}")

        if not vectorizer:
            raise ValueError(f"Unable to load vectorizer: {vectorizer_data}")

        routes = [Route(**route) for route in routes_data]
        routing_config = RoutingConfig(**routing_config_data)

        return cls(
            name=name,
            routes=routes,
            vectorizer=vectorizer,
            routing_config=routing_config,
            **kwargs,
        )

    def to_dict(self) -> Dict[str, Any]:
        """Convert the SemanticRouter instance to a dictionary.
```

```
        Returns:
            Dict[str, Any]: The dictionary representation of the SemanticRouter.

        .. code-block:: python

            from redisvl.extensions.router import SemanticRouter
            router = SemanticRouter(name="example_router", routes=[],
redis_url="redis://localhost:6379")
            router_dict = router.to_dict()
        """
        return {
            "name": self.name,
            "routes": [model_to_dict(route) for route in self.routes],
            "vectorizer": {
                "type": self.vectorizer.type,
                "model": self.vectorizer.model,
            },
            "routing_config": model_to_dict(self.routing_config),
        }

    @classmethod
    def from_yaml(
        cls,
        file_path: str,
        **kwargs,
    ) -> "SemanticRouter":
        """Create a SemanticRouter from a YAML file.

        Args:
            file_path (str): The path to the YAML file.

        Returns:
            SemanticRouter: The semantic router instance.

        Raises:
            ValueError: If the file path is invalid.
            FileNotFoundError: If the file does not exist.

        .. code-block:: python

            from redisvl.extensions.router import SemanticRouter
            router = SemanticRouter.from_yaml("router.yaml",
redis_url="redis://localhost:6379")
        """
        try:
            fp = Path(file_path).resolve()
        except OSError as e:
            raise ValueError(f"Invalid file path: {file_path}") from e

        if not fp.exists():
            raise FileNotFoundError(f"File {file_path} does not exist")

        with open(fp, "r") as f:
            yaml_data = yaml.safe_load(f)
            return cls.from_dict(
                yaml_data,
                **kwargs,
            )

    def to_yaml(self, file_path: str, overwrite: bool = True) -> None:
        """Write the semantic router to a YAML file.

        Args:
            file_path (str): The path to the YAML file.
```

```
            overwrite (bool): Whether to overwrite the file if it already exists.

        Raises:
            FileExistsError: If the file already exists and overwrite is False.

        .. code-block:: python

            from redisvl.extensions.router import SemanticRouter
            router = SemanticRouter(
                name="example_router",
                routes=[],
                redis_url="redis://localhost:6379"
            )
            router.to_yaml("router.yaml")
        """
        fp = Path(file_path).resolve()
        if fp.exists() and not overwrite:
            raise FileExistsError(f"Schema file {file_path} already exists.")

        with open(fp, "w") as f:
            yaml_data = self.to_dict()
            yaml.dump(yaml_data, f, sort_keys=False)
}
```

Chapter 2.2.3

redisvl/extensions/session_manager

redisvl/extensions/session_manager/__init__.py

```
from redisvl.extensions.session_manager.base_session import BaseSessionManager
from redisvl.extensions.session_manager.semantic_session import SemanticSessionManager
from redisvl.extensions.session_manager.standard_session import StandardSessionManager

__all__ = ["BaseSessionManager", "StandardSessionManager", "SemanticSessionManager"]
}
```

redisvl/extensions/session_manager/base_session.py

```
from typing import Any, Dict, List, Optional, Union

from redisvl.extensions.constants import (
    CONTENT_FIELD_NAME,
    ROLE_FIELD_NAME,
    TOOL_FIELD_NAME,
)
from redisvl.extensions.session_manager.schema import ChatMessage
from redisvl.utils.utils import create_uuid
```

```
class BaseSessionManager:

    def __init__(
        self,
        name: str,
        session_tag: Optional[str] = None,
    ):
        """Initialize session memory with index

        Session Manager stores the current and previous user text prompts and
        LLM responses to allow for enriching future prompts with session
        context. Session history is stored in individual user or LLM prompts and
        responses.

        Args:
            name (str): The name of the session manager index.
            session_tag (str): Tag to be added to entries to link to a specific
                session. Defaults to instance uuid.
        """
        self._name = name
        self._session_tag = session_tag or create_uuid()

    def clear(self) -> None:
        """Clears the chat session history."""
        raise NotImplementedError

    def delete(self) -> None:
        """Clear all conversation history and remove any search indices."""
        raise NotImplementedError

    def drop(self, id_field: Optional[str] = None) -> None:
        """Remove a specific exchange from the conversation history.

        Args:
            id_field (Optional[str]): The id_field of the entry to delete.
                If None then the last entry is deleted.
        """
        raise NotImplementedError

    @property
    def messages(self) -> Union[List[str], List[Dict[str, str]]]:
        """Returns the full chat history."""
        raise NotImplementedError

    def get_recent(
        self,
        top_k: int = 5,
        as_text: bool = False,
        raw: bool = False,
        session_tag: Optional[str] = None,
    ) -> Union[List[str], List[Dict[str, str]]]:
        """Retreive the recent conversation history in sequential order.

        Args:
            top_k (int): The number of previous exchanges to return. Default is 5.
                Note that one exchange contains both a prompt and response.
            as_text (bool): Whether to return the conversation as a single string,
                or list of alternating prompts and responses.
            raw (bool): Whether to return the full Redis hash entry or just the
                prompt and response
            session_tag (str): Tag to be added to entries to link to a specific
                session. Defaults to instance uuid.
```

```
        Returns:
            Union[str, List[str]]: A single string transcription of the session
                                   or list of strings if as_text is false.

        Raises:
            ValueError: If top_k is not an integer greater than or equal to 0.
        """
        raise NotImplementedError

    def _format_context(
        self, messages: List[Dict[str, Any]], as_text: bool
    ) -> Union[List[str], List[Dict[str, str]]]:
        """Extracts the prompt and response fields from the Redis hashes and
           formats them as either flat dictionaries or strings.

        Args:
            messages (List[Dict[str, Any]]): The messages from the session index.
            as_text (bool): Whether to return the conversation as a single string,
                or list of alternating prompts and responses.

        Returns:
            Union[str, List[str]]: A single string transcription of the session
                or list of strings if as_text is false.
        """
        context = []

        for message in messages:

            chat_message = ChatMessage(**message)

            if as_text:
                context.append(chat_message.content)
            else:
                chat_message_dict = {
                    ROLE_FIELD_NAME: chat_message.role,
                    CONTENT_FIELD_NAME: chat_message.content,
                }
                if chat_message.tool_call_id is not None:
                    chat_message_dict[TOOL_FIELD_NAME] = chat_message.tool_call_id

                context.append(chat_message_dict)  # type: ignore

        return context

    def store(
        self, prompt: str, response: str, session_tag: Optional[str] = None
    ) -> None:
        """Insert a prompt:response pair into the session memory. A timestamp
        is associated with each exchange so that they can be later sorted
        in sequential ordering after retrieval.

        Args:
            prompt (str): The user prompt to the LLM.
            response (str): The corresponding LLM response.
            session_tag (Optional[str]): The tag to mark the message with. Defaults
to None.
        """
        raise NotImplementedError

    def add_messages(
        self, messages: List[Dict[str, str]], session_tag: Optional[str] = None
    ) -> None:
        """Insert a list of prompts and responses into the session memory.
        A timestamp is associated with each so that they can be later sorted
```

```
        in sequential ordering after retrieval.

        Args:
            messages (List[Dict[str, str]]): The list of user prompts and
LLM responses.
            session_tag (Optional[str]): The tag to mark the messages with. Defaults
to None.
        """
        raise NotImplementedError

    def add_message(
        self, message: Dict[str, str], session_tag: Optional[str] = None
    ) -> None:
        """Insert a single prompt or response into the session memory.
        A timestamp is associated with it so that it can be later sorted
        in sequential ordering after retrieval.

        Args:
            message (Dict[str,str]): The user prompt or LLM response.
            session_tag (Optional[str]): The tag to mark the message with. Defaults
to None.
        """
        raise NotImplementedError
}
```

redisvl/extensions/session_manager/schema.py

```
from typing import Dict, List, Optional

from pydantic.v1 import BaseModel, Field, root_validator

from redisvl.extensions.constants import (
    CONTENT_FIELD_NAME,
    ID_FIELD_NAME,
    ROLE_FIELD_NAME,
    SESSION_FIELD_NAME,
    SESSION_VECTOR_FIELD_NAME,
    TIMESTAMP_FIELD_NAME,
    TOOL_FIELD_NAME,
)
from redisvl.redis.utils import array_to_buffer
from redisvl.schema import IndexSchema
from redisvl.utils.utils import current_timestamp


class ChatMessage(BaseModel):
    """A single chat message exchanged between a user and an LLM."""

    entry_id: Optional[str] = Field(default=None)
    """A unique identifier for the message."""
    role: str  # TODO -- do we enumify this?
    """The role of the message sender (e.g., 'user' or 'llm')."""
    content: str
    """The content of the message."""
    session_tag: str
    """Tag associated with the current session."""
    timestamp: Optional[float] = Field(default=None)
    """The time the message was sent, in UTC, rounded to milliseconds."""
    tool_call_id: Optional[str] = Field(default=None)
    """An optional identifier for a tool call associated with the message."""
```

```
    vector_field: Optional[List[float]] = Field(default=None)
    """The vector representation of the message content."""

    class Config:
        arbitrary_types_allowed = True

    @root_validator(pre=True)
    @classmethod
    def generate_id(cls, values):
        if TIMESTAMP_FIELD_NAME not in values:
            values[TIMESTAMP_FIELD_NAME] = current_timestamp()
        if ID_FIELD_NAME not in values:
            values[ID_FIELD_NAME] = (
                f"{values[SESSION_FIELD_NAME]}:{values[TIMESTAMP_FIELD_NAME]}"
            )
        return values

    def to_dict(self, dtype: Optional[str] = None) -> Dict:
        data = self.dict(exclude_none=True)

        # handle optional fields
        if SESSION_VECTOR_FIELD_NAME in data:
            data[SESSION_VECTOR_FIELD_NAME] = array_to_buffer(
                data[SESSION_VECTOR_FIELD_NAME], dtype  # type: ignore[arg-type]
            )
        return data


class StandardSessionIndexSchema(IndexSchema):

    @classmethod
    def from_params(cls, name: str, prefix: str):

        return cls(
            index={"name": name, "prefix": prefix},  # type: ignore
            fields=[  # type: ignore
                {"name": ROLE_FIELD_NAME, "type": "tag"},
                {"name": CONTENT_FIELD_NAME, "type": "text"},
                {"name": TOOL_FIELD_NAME, "type": "tag"},
                {"name": TIMESTAMP_FIELD_NAME, "type": "numeric"},
                {"name": SESSION_FIELD_NAME, "type": "tag"},
            ],
        )


class SemanticSessionIndexSchema(IndexSchema):

    @classmethod
    def from_params(cls, name: str, prefix: str, vectorizer_dims: int, dtype: str):

        return cls(
            index={"name": name, "prefix": prefix},  # type: ignore
            fields=[  # type: ignore
                {"name": ROLE_FIELD_NAME, "type": "tag"},
                {"name": CONTENT_FIELD_NAME, "type": "text"},
                {"name": TOOL_FIELD_NAME, "type": "tag"},
                {"name": TIMESTAMP_FIELD_NAME, "type": "numeric"},
                {"name": SESSION_FIELD_NAME, "type": "tag"},
                {
                    "name": SESSION_VECTOR_FIELD_NAME,
                    "type": "vector",
                    "attrs": {
                        "dims": vectorizer_dims,
                        "datatype": dtype,
```

```
                        "distance_metric": "cosine",
                        "algorithm": "flat",
                    },
                },
            ],
        )
}
```

redisvl/extensions/session_manager/semantic_session.py

```
from typing import Any, Dict, List, Optional, Union

from redis import Redis

from redisvl.extensions.constants import (
    CONTENT_FIELD_NAME,
    ID_FIELD_NAME,
    ROLE_FIELD_NAME,
    SESSION_FIELD_NAME,
    SESSION_VECTOR_FIELD_NAME,
    TIMESTAMP_FIELD_NAME,
    TOOL_FIELD_NAME,
)
from redisvl.extensions.session_manager import BaseSessionManager
from redisvl.extensions.session_manager.schema import (
    ChatMessage,
    SemanticSessionIndexSchema,
)
from redisvl.index import SearchIndex
from redisvl.query import FilterQuery, RangeQuery
from redisvl.query.filter import Tag
from redisvl.utils.utils import validate_vector_dims
from redisvl.utils.vectorize import BaseVectorizer, HFTextVectorizer


class SemanticSessionManager(BaseSessionManager):

    def __init__(
        self,
        name: str,
        session_tag: Optional[str] = None,
        prefix: Optional[str] = None,
        vectorizer: Optional[BaseVectorizer] = None,
        distance_threshold: float = 0.3,
        redis_client: Optional[Redis] = None,
        redis_url: str = "redis://localhost:6379",
        connection_kwargs: Dict[str, Any] = {},
        overwrite: bool = False,
        **kwargs,
    ):
        """Initialize session memory with index

        Session Manager stores the current and previous user text prompts and
        LLM responses to allow for enriching future prompts with session
        context. Session history is stored in individual user or LLM prompts and
        responses.


        Args:
            name (str): The name of the session manager index.
```

```
            session_tag (Optional[str]): Tag to be added to entries to link to
a specific
                session. Defaults to instance uuid.
            prefix (Optional[str]): Prefix for the keys for this session data.
                Defaults to None and will be replaced with the index name.
            vectorizer (Optional[BaseVectorizer]): The vectorizer used to
create embeddings.
            distance_threshold (float): The maximum semantic distance to be
                included in the context. Defaults to 0.3.
            redis_client (Optional[Redis]): A Redis client instance. Defaults to
                None.
            redis_url (str, optional): The redis url. Defaults
to redis://localhost:6379.
            connection_kwargs (Dict[str, Any]): The connection arguments
                for the redis client. Defaults to empty {}.
            overwrite (bool): Whether or not to force overwrite the schema for
                the semantic session index. Defaults to false.

        The proposed schema will support a single vector embedding constructed
        from either the prompt or response in a single string.

        """
        super().__init__(name, session_tag)

        prefix = prefix or name

        self._vectorizer = vectorizer or HFTextVectorizer(
            model="sentence-transformers/msmarco-distilbert-cos-v5"
        )

        self.set_distance_threshold(distance_threshold)

        dtype = kwargs.get("dtype", "float32")
        schema = SemanticSessionIndexSchema.from_params(
            name, prefix, self._vectorizer.dims, dtype
        )

        self._index = SearchIndex(schema=schema)

        # handle redis connection
        if redis_client:
            self._index.set_client(redis_client)
        elif redis_url:
            self._index.connect(redis_url=redis_url, **connection_kwargs)

        # Check for existing session index
        if not overwrite and self._index.exists():
            existing_index = SearchIndex.from_existing(
                name, redis_client=self._index.client
            )
            if existing_index.schema != self._index.schema:
                raise ValueError(
                    f"Existing index {name} schema does not match the user provided
schema for the semantic session. "
                    "If you wish to overwrite the index schema, set overwrite=True
during initialization."
                )
        self._index.create(overwrite=overwrite, drop=False)

        self._default_session_filter = Tag(SESSION_FIELD_NAME) == self._session_tag

    def clear(self) -> None:
        """Clears the chat session history."""
        self._index.clear()
```

```
    def delete(self) -> None:
        """Clear all conversation keys and remove the search index."""
        self._index.delete(drop=True)

    def drop(self, id: Optional[str] = None) -> None:
        """Remove a specific exchange from the conversation history.

        Args:
            id (Optional[str]): The id of the session entry to delete.
                If None then the last entry is deleted.
        """
        if id is None:
            id = self.get_recent(top_k=1, raw=True)[0][ID_FIELD_NAME]  # type: ignore

        self._index.client.delete(self._index.key(id))  # type: ignore

    @property
    def messages(self) -> Union[List[str], List[Dict[str, str]]]:
        """Returns the full chat history."""
        # TODO raw or as_text?
        # TODO refactor method to use get_recent and support other session tags
        return_fields = [
            ID_FIELD_NAME,
            SESSION_FIELD_NAME,
            ROLE_FIELD_NAME,
            CONTENT_FIELD_NAME,
            TOOL_FIELD_NAME,
            TIMESTAMP_FIELD_NAME,
        ]

        query = FilterQuery(
            filter_expression=self._default_session_filter,
            return_fields=return_fields,
        )
        query.sort_by(TIMESTAMP_FIELD_NAME, asc=True)
        messages = self._index.query(query)

        return self._format_context(messages, as_text=False)

    def get_relevant(
        self,
        prompt: str,
        as_text: bool = False,
        top_k: int = 5,
        fall_back: bool = False,
        session_tag: Optional[str] = None,
        raw: bool = False,
        distance_threshold: Optional[float] = None,
    ) -> Union[List[str], List[Dict[str, str]]]:
        """Searches the chat history for information semantically related to
        the specified prompt.

        This method uses vector similarity search with a text prompt as input.
        It checks for semantically similar prompts and responses and gets
        the top k most relevant previous prompts or responses to include as
        context to the next LLM call.

        Args:
            prompt (str): The message text to search for in session memory
            as_text (bool): Whether to return the prompts and responses as text
            or as JSON
            top_k (int): The number of previous messages to return. Default is 5.
            session_tag (Optional[str]): Tag to be added to entries to link to
```

```
a specific
                session. Defaults to instance uuid.
            distance_threshold (Optional[float]): The threshold for semantic
                vector distance.
            fall_back (bool): Whether to drop back to recent conversation history
                if no relevant context is found.
            raw (bool): Whether to return the full Redis hash entry or just the
                message.

        Returns:
            Union[List[str], List[Dict[str,str]]: Either a list of strings, or a
            list of prompts and responses in JSON containing the most relevant.

        Raises ValueError: if top_k is not an integer greater or equal to 0.
        """
        if type(top_k) != int or top_k < 0:
            raise ValueError("top_k must be an integer greater than or equal to -1")
        if top_k == 0:
            return []

        # override distance threshold
        distance_threshold = distance_threshold or self._distance_threshold

        return_fields = [
            SESSION_FIELD_NAME,
            ROLE_FIELD_NAME,
            CONTENT_FIELD_NAME,
            TIMESTAMP_FIELD_NAME,
            TOOL_FIELD_NAME,
        ]

        session_filter = (
            Tag(SESSION_FIELD_NAME) == session_tag
            if session_tag
            else self._default_session_filter
        )

        query = RangeQuery(
            vector=self._vectorizer.embed(prompt),
            vector_field_name=SESSION_VECTOR_FIELD_NAME,
            return_fields=return_fields,
            distance_threshold=distance_threshold,
            num_results=top_k,
            return_score=True,
            filter_expression=session_filter,
            dtype=self._index.schema.fields[SESSION_VECTOR_FIELD_NAME].attrs.datatype,
# type: ignore[union-attr]
        )
        messages = self._index.query(query)

        # if we don't find semantic matches fallback to returning recent context
        if not messages and fall_back:
            return self.get_recent(as_text=as_text, top_k=top_k, raw=raw)
        if raw:
            return messages
        return self._format_context(messages, as_text)

    def get_recent(
        self,
        top_k: int = 5,
        as_text: bool = False,
        raw: bool = False,
        session_tag: Optional[str] = None,
    ) -> Union[List[str], List[Dict[str, str]]]:
```

```
        """Retreive the recent conversation history in sequential order.

        Args:
            top_k (int): The number of previous exchanges to return. Default is 5.
            as_text (bool): Whether to return the conversation as a single string,
                or list of alternating prompts and responses.
            raw (bool): Whether to return the full Redis hash entry or just the
                prompt and response
            session_tag (Optional[str]): Tag to be added to entries to link to
a specific
                session. Defaults to instance uuid.

        Returns:
            Union[str, List[str]]: A single string transcription of the session
                or list of strings if as_text is false.

        Raises:
            ValueError: if top_k is not an integer greater than or equal to 0.
        """
        if type(top_k) != int or top_k < 0:
            raise ValueError("top_k must be an integer greater than or equal to 0")

        return_fields = [
            ID_FIELD_NAME,
            SESSION_FIELD_NAME,
            ROLE_FIELD_NAME,
            CONTENT_FIELD_NAME,
            TOOL_FIELD_NAME,
            TIMESTAMP_FIELD_NAME,
        ]

        session_filter = (
            Tag(SESSION_FIELD_NAME) == session_tag
            if session_tag
            else self._default_session_filter
        )

        query = FilterQuery(
            filter_expression=session_filter,
            return_fields=return_fields,
            num_results=top_k,
        )
        query.sort_by(TIMESTAMP_FIELD_NAME, asc=False)
        messages = self._index.query(query)

        if raw:
            return messages[::-1]
        return self._format_context(messages[::-1], as_text)

    @property
    def distance_threshold(self):
        return self._distance_threshold

    def set_distance_threshold(self, threshold):
        self._distance_threshold = threshold

    def store(
        self, prompt: str, response: str, session_tag: Optional[str] = None
    ) -> None:
        """Insert a prompt:response pair into the session memory. A timestamp
        is associated with each message so that they can be later sorted
        in sequential ordering after retrieval.

        Args:
```

```
            prompt (str): The user prompt to the LLM.
            response (str): The corresponding LLM response.
            session_tag (Optional[str]): Tag to be added to entries to link to
a specific
                session. Defaults to instance uuid.
        """
        self.add_messages(
            [
                {ROLE_FIELD_NAME: "user", CONTENT_FIELD_NAME: prompt},
                {ROLE_FIELD_NAME: "llm", CONTENT_FIELD_NAME: response},
            ],
            session_tag,
        )

    def add_messages(
        self, messages: List[Dict[str, str]], session_tag: Optional[str] = None
    ) -> None:
        """Insert a list of prompts and responses into the session memory.
        A timestamp is associated with each so that they can be later sorted
        in sequential ordering after retrieval.

        Args:
            messages (List[Dict[str, str]]): The list of user prompts and
LLM responses.
            session_tag (Optional[str]): Tag to be added to entries to link to
a specific
                session. Defaults to instance uuid.
        """
        session_tag = session_tag or self._session_tag
        chat_messages: List[Dict[str, Any]] = []

        for message in messages:
            content_vector = self._vectorizer.embed(message[CONTENT_FIELD_NAME])
            validate_vector_dims(
                len(content_vector),
                self._index.schema.fields[SESSION_VECTOR_FIELD_NAME].attrs.dims,  #
type: ignore
            )

            chat_message = ChatMessage(
                role=message[ROLE_FIELD_NAME],
                content=message[CONTENT_FIELD_NAME],
                session_tag=session_tag,
                vector_field=content_vector,
            )

            if TOOL_FIELD_NAME in message:
                chat_message.tool_call_id = message[TOOL_FIELD_NAME]


chat_messages.append(chat_message.to_dict(dtype=self._index.schema.fields[SESSION_VECT…
# type: ignore[union-attr]

        self._index.load(data=chat_messages, id_field=ID_FIELD_NAME)

    def add_message(
        self, message: Dict[str, str], session_tag: Optional[str] = None
    ) -> None:
        """Insert a single prompt or response into the session memory.
        A timestamp is associated with it so that it can be later sorted
        in sequential ordering after retrieval.

        Args:
            message (Dict[str,str]): The user prompt or LLM response.
```

```
            session_tag (Optional[str]): Tag to be added to entries to link to
a specific
                session. Defaults to instance uuid.
        """
        self.add_messages([message], session_tag)
}
```

redisvl/extensions/session_manager/standard_session.py

```
from typing import Any, Dict, List, Optional, Union

from redis import Redis

from redisvl.extensions.constants import (
    CONTENT_FIELD_NAME,
    ID_FIELD_NAME,
    ROLE_FIELD_NAME,
    SESSION_FIELD_NAME,
    TIMESTAMP_FIELD_NAME,
    TOOL_FIELD_NAME,
)
from redisvl.extensions.session_manager import BaseSessionManager
from redisvl.extensions.session_manager.schema import (
    ChatMessage,
    StandardSessionIndexSchema,
)
from redisvl.index import SearchIndex
from redisvl.query import FilterQuery
from redisvl.query.filter import Tag


class StandardSessionManager(BaseSessionManager):

    def __init__(
        self,
        name: str,
        session_tag: Optional[str] = None,
        prefix: Optional[str] = None,
        redis_client: Optional[Redis] = None,
        redis_url: str = "redis://localhost:6379",
        connection_kwargs: Dict[str, Any] = {},
        **kwargs,
    ):
        """Initialize session memory

        Session Manager stores the current and previous user text prompts and
        LLM responses to allow for enriching future prompts with session
        context.Session history is stored in individual user or LLM prompts and
        responses.

        Args:
            name (str): The name of the session manager index.
            session_tag (Optional[str]): Tag to be added to entries to link to
a specific
                session. Defaults to instance uuid.
            prefix (Optional[str]): Prefix for the keys for this session data.
                Defaults to None and will be replaced with the index name.
            redis_client (Optional[Redis]): A Redis client instance. Defaults to
                None.
            redis_url (str, optional): The redis url. Defaults
```

```
to redis://localhost:6379.
            connection_kwargs (Dict[str, Any]): The connection arguments
                for the redis client. Defaults to empty {}.

        The proposed schema will support a single combined vector embedding
        constructed from the prompt & response in a single string.

        """
        super().__init__(name, session_tag)

        prefix = prefix or name

        schema = StandardSessionIndexSchema.from_params(name, prefix)

        self._index = SearchIndex(schema=schema)

        if redis_client:
            self._index.set_client(redis_client)
        else:
            self._index.connect(redis_url=redis_url, **connection_kwargs)

        self._index.create(overwrite=False)

        self._default_session_filter = Tag(SESSION_FIELD_NAME) == self._session_tag

    def clear(self) -> None:
        """Clears the chat session history."""
        self._index.clear()

    def delete(self) -> None:
        """Clear all conversation keys and remove the search index."""
        self._index.delete(drop=True)

    def drop(self, id: Optional[str] = None) -> None:
        """Remove a specific exchange from the conversation history.

        Args:
            id (Optional[str]): The id of the session entry to delete.
                If None then the last entry is deleted.
        """
        if id is None:
            id = self.get_recent(top_k=1, raw=True)[0][ID_FIELD_NAME]  # type: ignore

        self._index.client.delete(self._index.key(id))  # type: ignore

    @property
    def messages(self) -> Union[List[str], List[Dict[str, str]]]:
        """Returns the full chat history."""
        # TODO raw or as_text?
        # TODO refactor this method to use get_recent and support other session tags?
        return_fields = [
            ID_FIELD_NAME,
            SESSION_FIELD_NAME,
            ROLE_FIELD_NAME,
            CONTENT_FIELD_NAME,
            TOOL_FIELD_NAME,
            TIMESTAMP_FIELD_NAME,
        ]

        query = FilterQuery(
            filter_expression=self._default_session_filter,
            return_fields=return_fields,
        )
        query.sort_by(TIMESTAMP_FIELD_NAME, asc=True)
```

```
        messages = self._index.query(query)

        return self._format_context(messages, as_text=False)

    def get_recent(
        self,
        top_k: int = 5,
        as_text: bool = False,
        raw: bool = False,
        session_tag: Optional[str] = None,
    ) -> Union[List[str], List[Dict[str, str]]]:
        """Retrieve the recent conversation history in sequential order.

        Args:
            top_k (int): The number of previous messages to return. Default is 5.
            as_text (bool): Whether to return the conversation as a single string,
                or list of alternating prompts and responses.
            raw (bool): Whether to return the full Redis hash entry or just the
                prompt and response
            session_tag (Optional[str]): Tag to be added to entries to link to
a specific
                session. Defaults to instance uuid.

        Returns:
            Union[str, List[str]]: A single string transcription of the session
                or list of strings if as_text is false.

        Raises:
            ValueError: if top_k is not an integer greater than or equal to 0.
        """
        if type(top_k) != int or top_k < 0:
            raise ValueError("top_k must be an integer greater than or equal to 0")

        return_fields = [
            ID_FIELD_NAME,
            SESSION_FIELD_NAME,
            ROLE_FIELD_NAME,
            CONTENT_FIELD_NAME,
            TOOL_FIELD_NAME,
            TIMESTAMP_FIELD_NAME,
        ]

        session_filter = (
            Tag(SESSION_FIELD_NAME) == session_tag
            if session_tag
            else self._default_session_filter
        )

        query = FilterQuery(
            filter_expression=session_filter,
            return_fields=return_fields,
            num_results=top_k,
        )
        query.sort_by(TIMESTAMP_FIELD_NAME, asc=False)
        messages = self._index.query(query)

        if raw:
            return messages[::-1]
        return self._format_context(messages[::-1], as_text)

    def store(
        self, prompt: str, response: str, session_tag: Optional[str] = None
    ) -> None:
        """Insert a prompt:response pair into the session memory. A timestamp
```

```
        is associated with each exchange so that they can be later sorted
        in sequential ordering after retrieval.

        Args:
            prompt (str): The user prompt to the LLM.
            response (str): The corresponding LLM response.
            session_tag (Optional[str]): Tag to be added to entries to link to
a specific
                session. Defaults to instance uuid.
        """
        self.add_messages(
            [
                {ROLE_FIELD_NAME: "user", CONTENT_FIELD_NAME: prompt},
                {ROLE_FIELD_NAME: "llm", CONTENT_FIELD_NAME: response},
            ],
            session_tag,
        )

    def add_messages(
        self, messages: List[Dict[str, str]], session_tag: Optional[str] = None
    ) -> None:
        """Insert a list of prompts and responses into the session memory.
        A timestamp is associated with each so that they can be later sorted
        in sequential ordering after retrieval.

        Args:
            messages (List[Dict[str, str]]): The list of user prompts and
LLM responses.
            session_tag (Optional[str]): Tag to be added to entries to link to
a specific
                session. Defaults to instance uuid.
        """
        session_tag = session_tag or self._session_tag
        chat_messages: List[Dict[str, Any]] = []

        for message in messages:

            chat_message = ChatMessage(
                role=message[ROLE_FIELD_NAME],
                content=message[CONTENT_FIELD_NAME],
                session_tag=session_tag,
            )

            if TOOL_FIELD_NAME in message:
                chat_message.tool_call_id = message[TOOL_FIELD_NAME]

            chat_messages.append(chat_message.to_dict())

        self._index.load(data=chat_messages, id_field=ID_FIELD_NAME)

    def add_message(
        self, message: Dict[str, str], session_tag: Optional[str] = None
    ) -> None:
        """Insert a single prompt or response into the session memory.
        A timestamp is associated with it so that it can be later sorted
        in sequential ordering after retrieval.

        Args:
            message (Dict[str,str]): The user prompt or LLM response.
            session_tag (Optional[str]): Tag to be added to entries to link to
a specific
                session. Defaults to instance uuid.
        """
```

```
        self.add_messages([message], session_tag)
}
```

Chapter 2.3.0

redisvl/index

redisvl/index/__init__.py

```
from redisvl.index.index import AsyncSearchIndex, SearchIndex

__all__ = ["SearchIndex", "AsyncSearchIndex"]
}
```

redisvl/index/index.py

```
import asyncio
import atexit
import json
import threading
from functools import wraps
from typing import (
    TYPE_CHECKING,
    Any,
    AsyncGenerator,
    Callable,
    Dict,
    Generator,
    Iterable,
    List,
    Optional,
    Union,
)

if TYPE_CHECKING:
    from redis.commands.search.aggregation import AggregateResult
    from redis.commands.search.document import Document
    from redis.commands.search.result import Result
    from redisvl.query.query import BaseQuery

import redis
import redis.asyncio as aredis
from redis.commands.search.indexDefinition import IndexDefinition

from redisvl.exceptions import RedisModuleVersionError, RedisSearchError
from redisvl.index.storage import BaseStorage, HashStorage, JsonStorage
from redisvl.query import BaseQuery, CountQuery, FilterQuery
from redisvl.query.filter import FilterExpression
from redisvl.redis.connection import (
```

```
    RedisConnectionFactory,
    convert_index_info_to_schema,
    validate_modules,
)
from redisvl.redis.utils import convert_bytes
from redisvl.schema import IndexSchema, StorageType
from redisvl.utils.log import get_logger

logger = get_logger(__name__)


def process_results(
    results: "Result", query: BaseQuery, storage_type: StorageType
) -> List[Dict[str, Any]]:
    """Convert a list of search Result objects into a list of document
    dictionaries.

    This function processes results from Redis, handling different storage
    types and query types. For JSON storage with empty return fields, it
    unpacks the JSON object while retaining the document ID. The 'payload'
    field is also removed from all resulting documents for consistency.

    Args:
        results (Result): The search results from Redis.
        query (BaseQuery): The query object used for the search.
        storage_type (StorageType): The storage type of the search
            index (json or hash).

    Returns:
        List[Dict[str, Any]]: A list of processed document dictionaries.
    """
    # Handle count queries
    if isinstance(query, CountQuery):
        return results.total

    # Determine if unpacking JSON is needed
    unpack_json = (
        (storage_type == StorageType.JSON)
        and isinstance(query, FilterQuery)
        and not query._return_fields  # type: ignore
    )

    # Process records
    def _process(doc: "Document") -> Dict[str, Any]:
        doc_dict = doc.__dict__

        # Unpack and Project JSON fields properly
        if unpack_json and "json" in doc_dict:
            json_data = doc_dict.get("json", {})
            if isinstance(json_data, str):
                json_data = json.loads(json_data)
            if isinstance(json_data, dict):
                return {"id": doc_dict.get("id"), **json_data}
            raise ValueError(f"Unable to parse json data from Redis {json_data}")

        # Remove 'payload' if present
        doc_dict.pop("payload", None)

        return doc_dict

    return [_process(doc) for doc in results.docs]


def setup_redis():
```

```
    def decorator(func):
        @wraps(func)
        def wrapper(self, *args, **kwargs):
            result = func(self, *args, **kwargs)
            RedisConnectionFactory.validate_sync_redis(
                self._redis_client, self._lib_name
            )
            return result

        return wrapper

    return decorator


def setup_async_redis():
    def decorator(func):
        @wraps(func)
        async def wrapper(self, *args, **kwargs):
            result = await func(self, *args, **kwargs)
            await RedisConnectionFactory.validate_async_redis(
                self._redis_client, self._lib_name
            )
            return result

        return wrapper

    return decorator


class BaseSearchIndex:
    """Base search engine class"""

    _STORAGE_MAP = {
        StorageType.HASH: HashStorage,
        StorageType.JSON: JsonStorage,
    }

    schema: IndexSchema

    def __init__(*args, **kwargs):
        pass

    @property
    def _storage(self) -> BaseStorage:
        """The storage type for the index schema."""
        return self._STORAGE_MAP[self.schema.index.storage_type](
            prefix=self.schema.index.prefix,
            key_separator=self.schema.index.key_separator,
        )

    @property
    def name(self) -> str:
        """The name of the Redis search index."""
        return self.schema.index.name

    @property
    def prefix(self) -> str:
        """The optional key prefix that comes before a unique key value in
        forming a Redis key."""
        return self.schema.index.prefix

    @property
    def key_separator(self) -> str:
        """The optional separator between a defined prefix and key value in
```

```
        forming a Redis key."""
        return self.schema.index.key_separator

    @property
    def storage_type(self) -> StorageType:
        """The underlying storage type for the search index; either
        hash or json."""
        return self.schema.index.storage_type

    @classmethod
    def from_yaml(cls, schema_path: str, **kwargs):
        """Create a SearchIndex from a YAML schema file.

        Args:
            schema_path (str): Path to the YAML schema file.

        Returns:
            SearchIndex: A RedisVL SearchIndex object.

        .. code-block:: python

            from redisvl.index import SearchIndex

            index = SearchIndex.from_yaml("schemas/schema.yaml")
        """
        schema = IndexSchema.from_yaml(schema_path)
        return cls(schema=schema, **kwargs)

    @classmethod
    def from_dict(cls, schema_dict: Dict[str, Any], **kwargs):
        """Create a SearchIndex from a dictionary.

        Args:
            schema_dict (Dict[str, Any]): A dictionary containing the schema.

        Returns:
            SearchIndex: A RedisVL SearchIndex object.

        .. code-block:: python

            from redisvl.index import SearchIndex

            index = SearchIndex.from_dict({
                "index": {
                    "name": "my-index",
                    "prefix": "rvl",
                    "storage_type": "hash",
                },
                "fields": [
                    {"name": "doc-id", "type": "tag"}
                ]
            })

        """
        schema = IndexSchema.from_dict(schema_dict)
        return cls(schema=schema, **kwargs)

    def disconnect(self):
        """Disconnect from the Redis database."""
        self._redis_client = None
        return self

    def key(self, id: str) -> str:
        """Construct a redis key as a combination of an index key prefix (optional)
```

```
        and specified id.

        The id is typically either a unique identifier, or
        derived from some domain-specific metadata combination (like a document
        id or chunk id).

        Args:
            id (str): The specified unique identifier for a particular
                document indexed in Redis.

        Returns:
            str: The full Redis key including key prefix and value as a string.
        """
        return self._storage._key(
            id=id,
            prefix=self.schema.index.prefix,
            key_separator=self.schema.index.key_separator,
        )


class SearchIndex(BaseSearchIndex):
    """A search index class for interacting with Redis as a vector database.

    The SearchIndex is instantiated with a reference to a Redis database and an
    IndexSchema (YAML path or dictionary object) that describes the various
    settings and field configurations.

    .. code-block:: python

        from redisvl.index import SearchIndex

        # initialize the index object with schema from file
        index = SearchIndex.from_yaml("schemas/schema.yaml")
        index.connect(redis_url="redis://localhost:6379")

        # create the index
        index.create(overwrite=True)

        # data is an iterable of dictionaries
        index.load(data)

        # delete index and data
        index.delete(drop=True)

    """

    def __init__(
        self,
        schema: IndexSchema,
        redis_client: Optional[redis.Redis] = None,
        redis_url: Optional[str] = None,
        connection_args: Dict[str, Any] = {},
        **kwargs,
    ):
        """Initialize the RedisVL search index with a schema, Redis client
        (or URL string with other connection args), connection_args, and other
        kwargs.

        Args:
            schema (IndexSchema): Index schema object.
            redis_client(Optional[redis.Redis]): An
                instantiated redis client.
            redis_url (Optional[str]): The URL of the Redis server to
                connect to.
```

```
            connection_args (Dict[str, Any], optional): Redis client connection
                args.
        """
        # final validation on schema object
        if not isinstance(schema, IndexSchema):
            raise ValueError("Must provide a valid IndexSchema object")

        self.schema = schema

        self._lib_name: Optional[str] = kwargs.pop("lib_name", None)

        # set up redis connection
        self._redis_client: Optional[redis.Redis] = None

        if redis_client is not None:
            self.set_client(redis_client)
        elif redis_url is not None:
            self.connect(redis_url, **connection_args)

    @classmethod
    def from_existing(
        cls,
        name: str,
        redis_client: Optional[redis.Redis] = None,
        redis_url: Optional[str] = None,
        **kwargs,
    ):
        # Handle redis instance
        if redis_url:
            redis_client = RedisConnectionFactory.connect(
                redis_url=redis_url, use_async=False, **kwargs
            )
        if not redis_client:
            raise ValueError(
                "Must provide either a redis_url or redis_client to fetch Redis
index info."
            )

        # Validate modules
        installed_modules = RedisConnectionFactory.get_modules(redis_client)

        try:
            required_modules = [
                {"name": "search", "ver": 20810},
                {"name": "searchlight", "ver": 20810},
            ]
            validate_modules(installed_modules, required_modules)
        except RedisModuleVersionError as e:
            raise RedisModuleVersionError(
                f"Loading from existing index failed. {str(e)}"
            )

        # Fetch index info and convert to schema
        index_info = cls._info(name, redis_client)
        schema_dict = convert_index_info_to_schema(index_info)
        schema = IndexSchema.from_dict(schema_dict)
        return cls(schema, redis_client, **kwargs)

    @property
    def client(self) -> Optional[redis.Redis]:
        """The underlying redis-py client object."""
        return self._redis_client

    def connect(self, redis_url: Optional[str] = None, **kwargs):
```

```
        """Connect to a Redis instance using the provided `redis_url`, falling
        back to the `REDIS_URL` environment variable (if available).

        Note: Additional keyword arguments (`**kwargs`) can be used to provide
        extra options specific to the Redis connection.

        Args:
            redis_url (Optional[str], optional): The URL of the Redis server to
                connect to. If not provided, the method defaults to using the
                `REDIS_URL` environment variable.

        Raises:
            redis.exceptions.ConnectionError: If the connection to the Redis
                server fails.
            ValueError: If the Redis URL is not provided nor accessible
                through the `REDIS_URL` environment variable.

        .. code-block:: python

            index.connect(redis_url="redis://localhost:6379")

        """
        client = RedisConnectionFactory.connect(
            redis_url=redis_url, use_async=False, **kwargs
        )
        return self.set_client(client)

    @setup_redis()
    def set_client(self, redis_client: redis.Redis, **kwargs):
        """Manually set the Redis client to use with the search index.

        This method configures the search index to use a specific Redis or
        Async Redis client. It is useful for cases where an external,
        custom-configured client is preferred instead of creating a new one.

        Args:
            redis_client (redis.Redis): A Redis or Async Redis
                client instance to be used for the connection.

        Raises:
            TypeError: If the provided client is not valid.

        .. code-block:: python

            import redis
            from redisvl.index import SearchIndex

            client = redis.Redis.from_url("redis://localhost:6379")
            index = SearchIndex.from_yaml("schemas/schema.yaml")
            index.set_client(client)

        """
        if not isinstance(redis_client, redis.Redis):
            raise TypeError("Invalid Redis client instance")

        self._redis_client = redis_client
        return self

    def create(self, overwrite: bool = False, drop: bool = False) -> None:
        """Create an index in Redis with the current schema and properties.

        Args:
            overwrite (bool, optional): Whether to overwrite the index if it
                already exists. Defaults to False.
```

```
            drop (bool, optional): Whether to drop all keys associated with the
                index in the case of overwriting. Defaults to False.

        Raises:
            RuntimeError: If the index already exists and 'overwrite' is False.
            ValueError: If no fields are defined for the index.

        .. code-block:: python

            # create an index in Redis; only if one does not exist with given name
            index.create()

            # overwrite an index in Redis without dropping associated data
            index.create(overwrite=True)

            # overwrite an index in Redis; drop associated data (clean slate)
            index.create(overwrite=True, drop=True)
        """
        # Check that fields are defined.
        redis_fields = self.schema.redis_fields
        if not redis_fields:
            raise ValueError("No fields defined for index")
        if not isinstance(overwrite, bool):
            raise TypeError("overwrite must be of type bool")

        if self.exists():
            if not overwrite:
                logger.info("Index already exists, not overwriting.")
                return None
            logger.info("Index already exists, overwriting.")
            self.delete(drop=drop)

        try:
            self._redis_client.ft(self.name).create_index(  # type: ignore
                fields=redis_fields,
                definition=IndexDefinition(
                    prefix=[self.schema.index.prefix], index_type=self._storage.type
                ),
            )
        except:
            logger.exception("Error while trying to create the index")
            raise

    def delete(self, drop: bool = True):
        """Delete the search index while optionally dropping all keys associated
        with the index.

        Args:
            drop (bool, optional): Delete the key / documents pairs in the
                index. Defaults to True.

        raises:
            redis.exceptions.ResponseError: If the index does not exist.
        """
        try:
            self._redis_client.ft(self.schema.index.name).dropindex(  # type: ignore
                delete_documents=drop
            )
        except Exception as e:
            raise RedisSearchError(f"Error while deleting index: {str(e)}") from e

    def clear(self) -> int:
        """Clear all keys in Redis associated with the index, leaving the index
        available and in-place for future insertions or updates.
```

```
        Returns:
            int: Count of records deleted from Redis.
        """
        # Track deleted records
        total_records_deleted: int = 0

        # Paginate using queries and delete in batches
        for batch in self.paginate(
            FilterQuery(FilterExpression("*"), return_fields=["id"]), page_size=500
        ):
            batch_keys = [record["id"] for record in batch]
            record_deleted = self._redis_client.delete(*batch_keys)  # type: ignore
            total_records_deleted += record_deleted  # type: ignore

        return total_records_deleted

    def drop_keys(self, keys: Union[str, List[str]]) -> int:
        """Remove a specific entry or entries from the index by it's key ID.

        Args:
            keys (Union[str, List[str]]): The document ID or IDs to remove from
the index.

        Returns:
            int: Count of records deleted from Redis.
        """
        if isinstance(keys, List):
            return self._redis_client.delete(*keys)  # type: ignore
        else:
            return self._redis_client.delete(keys)  # type: ignore

    def load(
        self,
        data: Iterable[Any],
        id_field: Optional[str] = None,
        keys: Optional[Iterable[str]] = None,
        ttl: Optional[int] = None,
        preprocess: Optional[Callable] = None,
        batch_size: Optional[int] = None,
    ) -> List[str]:
        """Load objects to the Redis database. Returns the list of keys loaded
        to Redis.

        RedisVL automatically handles constructing the object keys, batching,
        optional preprocessing steps, and setting optional expiration
        (TTL policies) on keys.

        Args:
            data (Iterable[Any]): An iterable of objects to store.
            id_field (Optional[str], optional): Specified field used as the id
                portion of the redis key (after the prefix) for each
                object. Defaults to None.
            keys (Optional[Iterable[str]], optional): Optional iterable of keys.
                Must match the length of objects if provided. Defaults to None.
            ttl (Optional[int], optional): Time-to-live in seconds for each key.
                Defaults to None.
            preprocess (Optional[Callable], optional): A function to preprocess
                objects before storage. Defaults to None.
            batch_size (Optional[int], optional): Number of objects to write in
                a single Redis pipeline execution. Defaults to class's
                default batch size.

        Returns:
```

```
            List[str]: List of keys loaded to Redis.

        Raises:
            ValueError: If the length of provided keys does not match the length
                of objects.

        .. code-block:: python

            data = [{"test": "foo"}, {"test": "bar"}]

            # simple case
            keys = index.load(data)

            # set 360 second ttl policy on data
            keys = index.load(data, ttl=360)

            # load data with predefined keys
            keys = index.load(data, keys=["rvl:foo", "rvl:bar"])

            # load data with preprocessing step
            def add_field(d):
                d["new_field"] = 123
                return d
            keys = index.load(data, preprocess=add_field)
        """
        try:
            return self._storage.write(
                self._redis_client,  # type: ignore
                objects=data,
                id_field=id_field,
                keys=keys,
                ttl=ttl,
                preprocess=preprocess,
                batch_size=batch_size,
            )
        except:
            logger.exception("Error while loading data to Redis")
            raise

    def fetch(self, id: str) -> Optional[Dict[str, Any]]:
        """Fetch an object from Redis by id.

        The id is typically either a unique identifier,
        or derived from some domain-specific metadata combination
        (like a document id or chunk id).

        Args:
            id (str): The specified unique identifier for a particular
                document indexed in Redis.

        Returns:
            Dict[str, Any]: The fetched object.
        """
        obj = self._storage.get(self._redis_client, [self.key(id)])  # type: ignore
        if obj:
            return convert_bytes(obj[0])
        return None

    def aggregate(self, *args, **kwargs) -> "AggregateResult":
        """Perform an aggregation operation against the index.

        Wrapper around the aggregation API that adds the index name
        to the query and passes along the rest of the arguments
        to the redis-py ft().aggregate() method.
```

```
        Returns:
            Result: Raw Redis aggregation results.
        """
        try:
            return self._redis_client.ft(self.schema.index.name).aggregate(  #
type: ignore
                *args, **kwargs
            )
        except Exception as e:
            raise RedisSearchError(f"Error while aggregating: {str(e)}") from e

    def search(self, *args, **kwargs) -> "Result":
        """Perform a search against the index.

        Wrapper around the search API that adds the index name
        to the query and passes along the rest of the arguments
        to the redis-py ft().search() method.

        Returns:
            Result: Raw Redis search results.
        """
        try:
            return self._redis_client.ft(self.schema.index.name).search(  #
type: ignore
                *args, **kwargs
            )
        except Exception as e:
            raise RedisSearchError(f"Error while searching: {str(e)}") from e

    def _query(self, query: BaseQuery) -> List[Dict[str, Any]]:
        """Execute a query and process results."""
        results = self.search(query.query, query_params=query.params)
        return process_results(
            results, query=query, storage_type=self.schema.index.storage_type
        )

    def query(self, query: BaseQuery) -> List[Dict[str, Any]]:
        """Execute a query on the index.

        This method takes a BaseQuery object directly, runs the search, and
        handles post-processing of the search.

        Args:
            query (BaseQuery): The query to run.

        Returns:
            List[Result]: A list of search results.

        .. code-block:: python

            from redisvl.query import VectorQuery

            query = VectorQuery(
                vector=[0.16, -0.34, 0.98, 0.23],
                vector_field_name="embedding",
                num_results=3
            )

            results = index.query(query)

        """
        return self._query(query)
```

```
    def paginate(self, query: BaseQuery, page_size: int = 30) -> Generator:
        """Execute a given query against the index and return results in
        paginated batches.

        This method accepts a RedisVL query instance, enabling pagination of
        results which allows for subsequent processing over each batch with a
        generator.

        Args:
            query (BaseQuery): The search query to be executed.
            page_size (int, optional): The number of results to return in each
                batch. Defaults to 30.

        Yields:
            A generator yielding batches of search results.

        Raises:
            TypeError: If the page_size argument is not of type int.
            ValueError: If the page_size argument is less than or equal to zero.

        .. code-block:: python

            # Iterate over paginated search results in batches of 10
            for result_batch in index.paginate(query, page_size=10):
                # Process each batch of results
                pass

        Note:
            The page_size parameter controls the number of items each result
            batch contains. Adjust this value based on performance
            considerations and the expected volume of search results.

        """
        if not isinstance(page_size, int):
            raise TypeError("page_size must be an integer")

        if page_size <= 0:
            raise ValueError("page_size must be greater than 0")

        offset = 0
        while True:
            query.paging(offset, page_size)
            results = self._query(query)
            if not results:
                break
            yield results
            # Increment the offset for the next batch of pagination
            offset += page_size

    def listall(self) -> List[str]:
        """List all search indices in Redis database.

        Returns:
            List[str]: The list of indices in the database.
        """
        return convert_bytes(self._redis_client.execute_command("FT._LIST"))  #
type: ignore

    def exists(self) -> bool:
        """Check if the index exists in Redis.

        Returns:
            bool: True if the index exists, False otherwise.
        """
```

```
        return self.schema.index.name in self.listall()

    @staticmethod
    def _info(name: str, redis_client: redis.Redis) -> Dict[str, Any]:
        """Run FT.INFO to fetch information about the index."""
        try:
            return convert_bytes(redis_client.ft(name).info())  # type: ignore
        except Exception as e:
            raise RedisSearchError(
                f"Error while fetching {name} index info: {str(e)}"
            ) from e

    def info(self, name: Optional[str] = None) -> Dict[str, Any]:
        """Get information about the index.

        Args:
            name (str, optional): Index name to fetch info about.
                Defaults to None.

        Returns:
            dict: A dictionary containing the information about the index.
        """
        index_name = name or self.schema.index.name
        return self._info(index_name, self._redis_client)  # type: ignore


class AsyncSearchIndex(BaseSearchIndex):
    """A search index class for interacting with Redis as a vector database in
    async-mode.

    The AsyncSearchIndex is instantiated with a reference to a Redis database
    and an IndexSchema (YAML path or dictionary object) that describes the
    various settings and field configurations.

    .. code-block:: python

        from redisvl.index import AsyncSearchIndex

        # initialize the index object with schema from file
        index = AsyncSearchIndex.from_yaml("schemas/schema.yaml")
        await index.connect(redis_url="redis://localhost:6379")

        # create the index
        await index.create(overwrite=True)

        # data is an iterable of dictionaries
        await index.load(data)

        # delete index and data
        await index.delete(drop=True)

    """

    def __init__(
        self,
        schema: IndexSchema,
        **kwargs,
    ):
        """Initialize the RedisVL async search index with a schema.

        Args:
            schema (IndexSchema): Index schema object.
            connection_args (Dict[str, Any], optional): Redis client connection
                args.
```

```
        """
        # final validation on schema object
        if not isinstance(schema, IndexSchema):
            raise ValueError("Must provide a valid IndexSchema object")

        self.schema = schema

        self._lib_name: Optional[str] = kwargs.pop("lib_name", None)

        # set up empty redis connection
        self._redis_client: Optional[aredis.Redis] = None

        if "redis_client" in kwargs or "redis_url" in kwargs:
            logger.warning(
                "Must use set_client() or connect() methods to provide a Redis
connection to AsyncSearchIndex"
            )

        atexit.register(self._cleanup_connection)

    def _cleanup_connection(self):
        if self._redis_client:

            def run_in_thread():
                try:
                    loop = asyncio.new_event_loop()
                    asyncio.set_event_loop(loop)
                    loop.run_until_complete(self._redis_client.aclose())
                    loop.close()
                except RuntimeError:
                    pass

            # Run cleanup in a background thread to avoid event loop issues
            thread = threading.Thread(target=run_in_thread)
            thread.start()
            thread.join()

        self._redis_client = None

    def disconnect(self):
        """Disconnect and cleanup the underlying async redis connection."""
        self._cleanup_connection()

    @classmethod
    async def from_existing(
        cls,
        name: str,
        redis_client: Optional[aredis.Redis] = None,
        redis_url: Optional[str] = None,
        **kwargs,
    ):
        if redis_url:
            redis_client = RedisConnectionFactory.connect(
                redis_url=redis_url, use_async=True, **kwargs
            )

        if not redis_client:
            raise ValueError(
                "Must provide either a redis_url or redis_client to fetch Redis
index info."
            )

        # Validate modules
        installed_modules = await
```

```
RedisConnectionFactory.get_modules_async(redis_client)

        try:
            required_modules = [
                {"name": "search", "ver": 20810},
                {"name": "searchlight", "ver": 20810},
            ]
            validate_modules(installed_modules, required_modules)
        except RedisModuleVersionError as e:
            raise RedisModuleVersionError(
                f"Loading from existing index failed. {str(e)}"
            ) from e

        # Fetch index info and convert to schema
        index_info = await cls._info(name, redis_client)
        schema_dict = convert_index_info_to_schema(index_info)
        schema = IndexSchema.from_dict(schema_dict)
        index = cls(schema, **kwargs)
        await index.set_client(redis_client)
        return index

    @property
    def client(self) -> Optional[aredis.Redis]:
        """The underlying redis-py client object."""
        return self._redis_client

    async def connect(self, redis_url: Optional[str] = None, **kwargs):
        """Connect to a Redis instance using the provided `redis_url`, falling
        back to the `REDIS_URL` environment variable (if available).

        Note: Additional keyword arguments (`**kwargs`) can be used to provide
        extra options specific to the Redis connection.

        Args:
            redis_url (Optional[str], optional): The URL of the Redis server to
                connect to. If not provided, the method defaults to using the
                `REDIS_URL` environment variable.

        Raises:
            redis.exceptions.ConnectionError: If the connection to the Redis
                server fails.
            ValueError: If the Redis URL is not provided nor accessible
                through the `REDIS_URL` environment variable.

        .. code-block:: python

            index.connect(redis_url="redis://localhost:6379")

        """
        client = RedisConnectionFactory.connect(
            redis_url=redis_url, use_async=True, **kwargs
        )
        return await self.set_client(client)

    @setup_async_redis()
    async def set_client(self, redis_client: aredis.Redis):
        """Manually set the Redis client to use with the search index.

        This method configures the search index to use a specific
        Async Redis client. It is useful for cases where an external,
        custom-configured client is preferred instead of creating a new one.

        Args:
            redis_client (aredis.Redis): An Async Redis
```

```
            client instance to be used for the connection.

        Raises:
            TypeError: If the provided client is not valid.

        .. code-block:: python

            import redis.asyncio as aredis
            from redisvl.index import AsyncSearchIndex

            # async Redis client and index
            client = aredis.Redis.from_url("redis://localhost:6379")
            index = AsyncSearchIndex.from_yaml("schemas/schema.yaml")
            await index.set_client(client)

        """
        if isinstance(redis_client, redis.Redis):
            print("Setting client and converting from async", flush=True)
            self._redis_client = RedisConnectionFactory.sync_to_async_redis(
                redis_client
            )
        else:
            self._redis_client = redis_client

        return self

    async def create(self, overwrite: bool = False, drop: bool = False) -> None:
        """Asynchronously create an index in Redis with the current schema
            and properties.

        Args:
            overwrite (bool, optional): Whether to overwrite the index if it
                already exists. Defaults to False.
            drop (bool, optional): Whether to drop all keys associated with the
                index in the case of overwriting. Defaults to False.

        Raises:
            RuntimeError: If the index already exists and 'overwrite' is False.
            ValueError: If no fields are defined for the index.

        .. code-block:: python

            # create an index in Redis; only if one does not exist with given name
            await index.create()

            # overwrite an index in Redis without dropping associated data
            await index.create(overwrite=True)

            # overwrite an index in Redis; drop associated data (clean slate)
            await index.create(overwrite=True, drop=True)
        """
        redis_fields = self.schema.redis_fields

        if not redis_fields:
            raise ValueError("No fields defined for index")
        if not isinstance(overwrite, bool):
            raise TypeError("overwrite must be of type bool")

        if await self.exists():
            if not overwrite:
                logger.info("Index already exists, not overwriting.")
                return None
            logger.info("Index already exists, overwriting.")
            await self.delete(drop)
```

```
        try:
            await self._redis_client.ft(self.schema.index.name).create_index(  #
type: ignore
                fields=redis_fields,
                definition=IndexDefinition(
                    prefix=[self.schema.index.prefix], index_type=self._storage.type
                ),
            )
        except:
            logger.exception("Error while trying to create the index")
            raise

    async def delete(self, drop: bool = True):
        """Delete the search index.

        Args:
            drop (bool, optional): Delete the documents in the index.
                Defaults to True.

        Raises:
            redis.exceptions.ResponseError: If the index does not exist.
        """
        try:
            await self._redis_client.ft(self.schema.index.name).dropindex(  #
type: ignore
                delete_documents=drop
            )
        except Exception as e:
            raise RedisSearchError(f"Error while deleting index: {str(e)}") from e

    async def clear(self) -> int:
        """Clear all keys in Redis associated with the index, leaving the index
        available and in-place for future insertions or updates.

        Returns:
            int: Count of records deleted from Redis.
        """
        # Track deleted records
        total_records_deleted: int = 0

        # Paginate using queries and delete in batches
        async for batch in self.paginate(
            FilterQuery(FilterExpression("*"), return_fields=["id"]), page_size=500
        ):
            batch_keys = [record["id"] for record in batch]
            records_deleted = await self._redis_client.delete(*batch_keys)  #
type: ignore
            total_records_deleted += records_deleted  # type: ignore

        return total_records_deleted

    async def drop_keys(self, keys: Union[str, List[str]]) -> int:
        """Remove a specific entry or entries from the index by it's key ID.

        Args:
            keys (Union[str, List[str]]): The document ID or IDs to remove from
the index.

        Returns:
            int: Count of records deleted from Redis.
        """
        if isinstance(keys, List):
            return await self._redis_client.delete(*keys)  # type: ignore
```

```
        else:
            return await self._redis_client.delete(keys)  # type: ignore

    async def load(
        self,
        data: Iterable[Any],
        id_field: Optional[str] = None,
        keys: Optional[Iterable[str]] = None,
        ttl: Optional[int] = None,
        preprocess: Optional[Callable] = None,
        concurrency: Optional[int] = None,
    ) -> List[str]:
        """Asynchronously load objects to Redis with concurrency control.
        Returns the list of keys loaded to Redis.

        RedisVL automatically handles constructing the object keys, batching,
        optional preprocessing steps, and setting optional expiration
        (TTL policies) on keys.

        Args:
            data (Iterable[Any]): An iterable of objects to store.
            id_field (Optional[str], optional): Specified field used as the id
                portion of the redis key (after the prefix) for each
                object. Defaults to None.
            keys (Optional[Iterable[str]], optional): Optional iterable of keys.
                Must match the length of objects if provided. Defaults to None.
            ttl (Optional[int], optional): Time-to-live in seconds for each key.
                Defaults to None.
            preprocess (Optional[Callable], optional): An async function to
                preprocess objects before storage. Defaults to None.
            concurrency (Optional[int], optional): The maximum number of
                concurrent write operations. Defaults to class's default
                concurrency level.

        Returns:
            List[str]: List of keys loaded to Redis.

        Raises:
            ValueError: If the length of provided keys does not match the
                length of objects.

        .. code-block:: python

            data = [{"test": "foo"}, {"test": "bar"}]

            # simple case
            keys = await index.load(data)

            # set 360 second ttl policy on data
            keys = await index.load(data, ttl=360)

            # load data with predefined keys
            keys = await index.load(data, keys=["rvl:foo", "rvl:bar"])

            # load data with preprocessing step
            async def add_field(d):
                d["new_field"] = 123
                return d
            keys = await index.load(data, preprocess=add_field)

        """
        try:
            return await self._storage.awrite(
                self._redis_client,  # type: ignore
```

```
                objects=data,
                id_field=id_field,
                keys=keys,
                ttl=ttl,
                preprocess=preprocess,
                concurrency=concurrency,
            )
        except:
            logger.exception("Error while loading data to Redis")
            raise

    async def fetch(self, id: str) -> Optional[Dict[str, Any]]:
        """Asynchronously etch an object from Redis by id. The id is typically
        either a unique identifier, or derived from some domain-specific
        metadata combination (like a document id or chunk id).

        Args:
            id (str): The specified unique identifier for a particular
                document indexed in Redis.

        Returns:
            Dict[str, Any]: The fetched object.
        """
        obj = await self._storage.aget(self._redis_client, [self.key(id)])  #
type: ignore
        if obj:
            return convert_bytes(obj[0])
        return None

    async def aggregate(self, *args, **kwargs) -> "AggregateResult":
        """Perform an aggregation operation against the index.

        Wrapper around the aggregation API that adds the index name
        to the query and passes along the rest of the arguments
        to the redis-py ft().aggregate() method.

        Returns:
            Result: Raw Redis aggregation results.
        """
        try:
            return await self._redis_client.ft(self.schema.index.name).aggregate(  #
type: ignore
                *args, **kwargs
            )
        except Exception as e:
            raise RedisSearchError(f"Error while aggregating: {str(e)}") from e

    async def search(self, *args, **kwargs) -> "Result":
        """Perform a search on this index.

        Wrapper around redis.search.Search that adds the index name
        to the search query and passes along the rest of the arguments
        to the redis-py ft.search() method.

        Returns:
            Result: Raw Redis search results.
        """
        try:
            return await self._redis_client.ft(self.schema.index.name).search(  #
type: ignore
                *args, **kwargs
            )
        except Exception as e:
            raise RedisSearchError(f"Error while searching: {str(e)}") from e
```

```
async def _query(self, query: BaseQuery) -> List[Dict[str, Any]]:
    """Asynchronously execute a query and process results."""
    results = await self.search(query.query, query_params=query.params)
    return process_results(
        results, query=query, storage_type=self.schema.index.storage_type
    )

async def query(self, query: BaseQuery) -> List[Dict[str, Any]]:
    """Asynchronously execute a query on the index.

    This method takes a BaseQuery object directly, runs the search, and
    handles post-processing of the search.

    Args:
        query (BaseQuery): The query to run.

    Returns:
        List[Result]: A list of search results.

    .. code-block:: python

        from redisvl.query import VectorQuery

        query = VectorQuery(
            vector=[0.16, -0.34, 0.98, 0.23],
            vector_field_name="embedding",
            num_results=3
        )

        results = await index.query(query)
    """
    return await self._query(query)

async def paginate(self, query: BaseQuery, page_size: int = 30) -> AsyncGenerator:
    """Execute a given query against the index and return results in
    paginated batches.

    This method accepts a RedisVL query instance, enabling async pagination
    of results which allows for subsequent processing over each batch with a
    generator.

    Args:
        query (BaseQuery): The search query to be executed.
        page_size (int, optional): The number of results to return in each
            batch. Defaults to 30.

    Yields:
        An async generator yielding batches of search results.

    Raises:
        TypeError: If the page_size argument is not of type int.
        ValueError: If the page_size argument is less than or equal to zero.

    .. code-block:: python

        # Iterate over paginated search results in batches of 10
        async for result_batch in index.paginate(query, page_size=10):
            # Process each batch of results
            pass

    Note:
        The page_size parameter controls the number of items each result
        batch contains. Adjust this value based on performance
```

```
            considerations and the expected volume of search results.

        """
        if not isinstance(page_size, int):
            raise TypeError("page_size must be an integer")

        if page_size <= 0:
            raise ValueError("page_size must be greater than 0")

        first = 0
        while True:
            query.paging(first, page_size)
            results = await self._query(query)
            if not results:
                break
            yield results
            # increment the pagination tracker
            first += page_size

    async def listall(self) -> List[str]:
        """List all search indices in Redis database.

        Returns:
            List[str]: The list of indices in the database.
        """
        return convert_bytes(
            await self._redis_client.execute_command("FT._LIST")  # type: ignore
        )

    async def exists(self) -> bool:
        """Check if the index exists in Redis.

        Returns:
            bool: True if the index exists, False otherwise.
        """
        return self.schema.index.name in await self.listall()

    @staticmethod
    async def _info(name: str, redis_client: aredis.Redis) -> Dict[str, Any]:
        try:
            return convert_bytes(await redis_client.ft(name).info())  # type: ignore
        except Exception as e:
            raise RedisSearchError(
                f"Error while fetching {name} index info: {str(e)}"
            ) from e

    async def info(self, name: Optional[str] = None) -> Dict[str, Any]:
        """Get information about the index.

        Args:
            name (str, optional): Index name to fetch info about.
                Defaults to None.

        Returns:
            dict: A dictionary containing the information about the index.
        """
        index_name = name or self.schema.index.name
        return await self._info(index_name, self._redis_client)  # type: ignore
}
```

redisvl/index/storage.py

```
import asyncio
import uuid
from typing import Any, Callable, Dict, Iterable, List, Optional

from pydantic.v1 import BaseModel
from redis import Redis
from redis.asyncio import Redis as AsyncRedis
from redis.commands.search.indexDefinition import IndexType

from redisvl.redis.utils import convert_bytes


class BaseStorage(BaseModel):
    """
    Base class for internal storage handling in Redis.

    Provides foundational methods for key management, data preprocessing,
    validation, and basic read/write operations (both sync and async).
    """

    type: IndexType
    """Type of index used in storage"""
    prefix: str
    """Prefix for Redis keys"""
    key_separator: str
    """Separator between prefix and key value"""
    default_batch_size: int = 200
    """Default size for batch operations"""
    default_write_concurrency: int = 20
    """Default concurrency for async ops"""

    @staticmethod
    def _key(id: str, prefix: str, key_separator: str) -> str:
        """Create a Redis key using a combination of a prefix, separator, and
        the identifider.

        Args:
            id (str): The unique identifier for the Redis entry.
            prefix (str): A prefix to append before the key value.
            key_separator (str): A separator to insert between prefix
                and key value.

        Returns:
            str: The fully formed Redis key.
        """
        if not prefix:
            return id
        else:
            return f"{prefix}{key_separator}{id}"

    def _create_key(self, obj: Dict[str, Any], id_field: Optional[str] = None) -> str:
        """Construct a Redis key for a given object, optionally using a
        specified field from the object as the key.

        Args:
            obj (Dict[str, Any]): The object from which to construct the key.
            id_field (Optional[str], optional): The field to use as the
                key, if provided.

        Returns:
            str: The constructed Redis key for the object.
```

```
        Raises:
            ValueError: If the id_field is not found in the object.
        """
        if id_field is None:
            key_value = uuid.uuid4().hex
        else:
            try:
                key_value = obj[id_field]  # type: ignore
            except KeyError:
                raise ValueError(f"Key field {id_field} not found in record {obj}")

        return self._key(
            key_value, prefix=self.prefix, key_separator=self.key_separator
        )

    @staticmethod
    def _preprocess(obj: Any, preprocess: Optional[Callable] = None) -> Dict[str, Any]:
        """Apply a preprocessing function to the object if provided.

        Args:
            preprocess (Optional[Callable], optional): Function to
                process the object.
            obj (Any): Object to preprocess.

        Returns:
            Dict[str, Any]: Processed object as a dictionary.
        """
        # optionally preprocess object
        if preprocess:
            obj = preprocess(obj)
        return obj

    @staticmethod
    async def _apreprocess(
        obj: Any, preprocess: Optional[Callable] = None
    ) -> Dict[str, Any]:
        """Asynchronously apply a preprocessing function to the object if
        provided.

        Args:
            preprocess (Optional[Callable], optional): Async function to
                process the object.
            obj (Any): Object to preprocess.

        Returns:
            Dict[str, Any]: Processed object as a dictionary.
        """
        # optionally async preprocess object
        if preprocess:
            obj = await preprocess(obj)
        return obj

    def _validate(self, obj: Dict[str, Any]):
        """Validate the object before writing to Redis. This method should be
        implemented by subclasses.

        Args:
            obj (Dict[str, Any]): The object to validate.
        """
        raise NotImplementedError

    @staticmethod
    def _set(client: Redis, key: str, obj: Dict[str, Any]):
        """Synchronously set the value in Redis for the given key.
```

```
        Args:
            client (Redis): The Redis client instance.
            key (str): The key under which to store the object.
            obj (Dict[str, Any]): The object to store in Redis.
        """
        raise NotImplementedError

    @staticmethod
    async def _aset(client: AsyncRedis, key: str, obj: Dict[str, Any]):
        """Asynchronously set the value in Redis for the given key.

        Args:
            client (AsyncRedis): The Redis client instance.
            key (str): The key under which to store the object.
            obj (Dict[str, Any]): The object to store in Redis.
        """
        raise NotImplementedError

    @staticmethod
    def _get(client: Redis, key: str) -> Dict[str, Any]:
        """Synchronously get the value from Redis for the given key.

        Args:
            client (Redis): The Redis client instance.
            key (str): The key for which to retrieve the object.

        Returns:
            Dict[str, Any]: The retrieved object from Redis.
        """
        raise NotImplementedError

    @staticmethod
    async def _aget(client: AsyncRedis, key: str) -> Dict[str, Any]:
        """Asynchronously get the value from Redis for the given key.

        Args:
            client (AsyncRedis): The Redis client instance.
            key (str): The key for which to retrieve the object.

        Returns:
            Dict[str, Any]: The retrieved object from Redis.
        """
        raise NotImplementedError

    def write(
        self,
        redis_client: Redis,
        objects: Iterable[Any],
        id_field: Optional[str] = None,
        keys: Optional[Iterable[str]] = None,
        ttl: Optional[int] = None,
        preprocess: Optional[Callable] = None,
        batch_size: Optional[int] = None,
    ) -> List[str]:
        """Write a batch of objects to Redis as hash entries. This method
        returns a list of Redis keys written to the database.

        Args:
            redis_client (Redis): A Redis client used for writing data.
            objects (Iterable[Any]): An iterable of objects to store.
            id_field (Optional[str], optional): Field used as the key for
                each object. Defaults to None.
            keys (Optional[Iterable[str]], optional): Optional iterable of
```

```
            keys, must match the length of objects if provided.
        ttl (Optional[int], optional): Time-to-live in seconds for each
            key. Defaults to None.
        preprocess (Optional[Callable], optional): A function to preprocess
            objects before storage. Defaults to None.
        batch_size (Optional[int], optional): Number of objects to write
            in a single Redis pipeline execution.

    Raises:
        ValueError: If the length of provided keys does not match the
            length of objects.
    """
    if keys and len(keys) != len(objects):  # type: ignore
        raise ValueError("Length of keys does not match the length of objects")

    if batch_size is None:
        # Use default or calculate based on the input data
        batch_size = self.default_batch_size

    keys_iterator = iter(keys) if keys else None
    added_keys: List[str] = []

    if objects:
        with redis_client.pipeline(transaction=False) as pipe:
            for i, obj in enumerate(objects, start=1):
                # Construct key, validate, and write
                key = (
                    next(keys_iterator)
                    if keys_iterator
                    else self._create_key(obj, id_field)
                )
                obj = self._preprocess(obj, preprocess)
                self._validate(obj)
                self._set(pipe, key, obj)
                # Set TTL if provided
                if ttl:
                    pipe.expire(key, ttl)
                # Execute mini batch
                if i % batch_size == 0:
                    pipe.execute()
                added_keys.append(key)
            # Clean up batches if needed
            if i % batch_size != 0:
                pipe.execute()

    return added_keys

async def awrite(
    self,
    redis_client: AsyncRedis,
    objects: Iterable[Any],
    id_field: Optional[str] = None,
    keys: Optional[Iterable[str]] = None,
    ttl: Optional[int] = None,
    preprocess: Optional[Callable] = None,
    concurrency: Optional[int] = None,
) -> List[str]:
    """Asynchronously write objects to Redis as hash entries with
    concurrency control. The method returns a list of keys written to the
    database.

    Args:
        redis_client (AsyncRedis): An asynchronous Redis client used
            for writing data.
```

```
            objects (Iterable[Any]): An iterable of objects to store.
            id_field (Optional[str], optional): Field used as the key for each
                object. Defaults to None.
            keys (Optional[Iterable[str]], optional): Optional iterable of keys.
                Must match the length of objects if provided.
            ttl (Optional[int], optional): Time-to-live in seconds for each key.
                Defaults to None.
            preprocess (Optional[Callable], optional): An async function to
                preprocess objects before storage. Defaults to None.
            concurrency (Optional[int], optional): The maximum number of
                concurrent write operations. Defaults to class's default
                concurrency level.

        Returns:
            List[str]: List of Redis keys loaded to the databases.

        Raises:
            ValueError: If the length of provided keys does not match the
                length of objects.
        """
        if keys and len(keys) != len(objects):  # type: ignore
            raise ValueError("Length of keys does not match the length of objects")

        if not concurrency:
            concurrency = self.default_write_concurrency

        semaphore = asyncio.Semaphore(concurrency)
        keys_iterator = iter(keys) if keys else None

        async def _load(obj: Dict[str, Any], key: Optional[str] = None) -> str:
            async with semaphore:
                if key is None:
                    key = self._create_key(obj, id_field)
                obj = await self._apreprocess(obj, preprocess)
                self._validate(obj)
                await self._aset(redis_client, key, obj)
                if ttl:
                    await redis_client.expire(key, ttl)
                return key

        if keys_iterator:
            tasks = [
                asyncio.create_task(_load(obj, next(keys_iterator))) for obj in objects
            ]
        else:
            tasks = [asyncio.create_task(_load(obj)) for obj in objects]

        return await asyncio.gather(*tasks)

    def get(
        self, redis_client: Redis, keys: Iterable[str], batch_size: Optional[int]
= None
    ) -> List[Dict[str, Any]]:
        """Retrieve objects from Redis by keys.

        Args:
            redis_client (Redis): Synchronous Redis client.
            keys (Iterable[str]): Keys to retrieve from Redis.
            batch_size (Optional[int], optional): Number of objects to write
                in a single Redis pipeline execution. Defaults to class's
                default batch size.

        Returns:
            List[Dict[str, Any]]: List of objects pulled from redis.
```

```
        """
        results: List = []

        if not isinstance(keys, Iterable):  # type: ignore
            raise TypeError("Keys must be an iterable of strings")

        if len(keys) == 0:  # type: ignore
            return []

        if batch_size is None:
            batch_size = (
                self.default_batch_size
            )  # Use default or calculate based on the input data

        # Use a pipeline to batch the retrieval
        with redis_client.pipeline(transaction=False) as pipe:
            for i, key in enumerate(keys, start=1):
                self._get(pipe, key)
                if i % batch_size == 0:
                    results.extend(pipe.execute())
            if i % batch_size != 0:
                results.extend(pipe.execute())

        # Process results
        return convert_bytes(results)

    async def aget(
        self,
        redis_client: AsyncRedis,
        keys: Iterable[str],
        concurrency: Optional[int] = None,
    ) -> List[Dict[str, Any]]:
        """Asynchronously retrieve objects from Redis by keys, with concurrency
        control.

        Args:
            redis_client (AsyncRedis): Asynchronous Redis client.
            keys (Iterable[str]): Keys to retrieve from Redis.
            concurrency (Optional[int], optional): The number of concurrent
                requests to make.

        Returns:
            Dict[str, Any]: Dictionary with keys and their corresponding
                objects.
        """
        if not isinstance(keys, Iterable):  # type: ignore
            raise TypeError("Keys must be an iterable of strings")

        if len(keys) == 0:  # type: ignore
            return []

        if not concurrency:
            concurrency = self.default_write_concurrency

        semaphore = asyncio.Semaphore(concurrency)

        async def _get(key: str) -> Dict[str, Any]:
            async with semaphore:
                result = await self._aget(redis_client, key)
                return result

        tasks = [asyncio.create_task(_get(key)) for key in keys]
        results = await asyncio.gather(*tasks)
        return convert_bytes(results)
```

```
class HashStorage(BaseStorage):
    """
    Internal subclass of BaseStorage for the Redis hash data type.

    Implements hash-specific logic for validation and read/write operations
    (both sync and async) in Redis.
    """

    type: IndexType = IndexType.HASH
    """Hash data type for the index"""

    def _validate(self, obj: Dict[str, Any]):
        """Validate that the given object is a dictionary, suitable for storage
        as a Redis hash.

        Args:
            obj (Dict[str, Any]): The object to validate.

        Raises:
            TypeError: If the object is not a dictionary.
        """
        if not isinstance(obj, dict):
            raise TypeError("Object must be a dictionary.")

    @staticmethod
    def _set(client: Redis, key: str, obj: Dict[str, Any]):
        """Synchronously set a hash value in Redis for the given key.

        Args:
            client (Redis): The Redis client instance.
            key (str): The key under which to store the hash.
            obj (Dict[str, Any]): The hash to store in Redis.
        """
        client.hset(name=key, mapping=obj)  # type: ignore

    @staticmethod
    async def _aset(client: AsyncRedis, key: str, obj: Dict[str, Any]):
        """Asynchronously set a hash value in Redis for the given key.

        Args:
            client (AsyncRedis): The Redis client instance.
            key (str): The key under which to store the hash.
            obj (Dict[str, Any]): The hash to store in Redis.
        """
        await client.hset(name=key, mapping=obj)  # type: ignore

    @staticmethod
    def _get(client: Redis, key: str) -> Dict[str, Any]:
        """Synchronously retrieve a hash value from Redis for the given key.

        Args:
            client (Redis): The Redis client instance.
            key (str): The key for which to retrieve the hash.

        Returns:
            Dict[str, Any]: The retrieved hash from Redis.
        """
        return client.hgetall(key)

    @staticmethod
    async def _aget(client: AsyncRedis, key: str) -> Dict[str, Any]:
        """Asynchronously retrieve a hash value from Redis for the given key.
```

```
        Args:
            client (AsyncRedis): The Redis client instance.
            key (str): The key for which to retrieve the hash.

        Returns:
            Dict[str, Any]: The retrieved hash from Redis.
        """
        return await client.hgetall(key)


class JsonStorage(BaseStorage):
    """
    Internal subclass of BaseStorage for the Redis JSON data type.

    Implements json-specific logic for validation and read/write operations
    (both sync and async) in Redis.
    """

    type: IndexType = IndexType.JSON
    """JSON data type for the index"""

    def _validate(self, obj: Dict[str, Any]):
        """Validate that the given object is a dictionary, suitable for JSON
        serialization.

        Args:
            obj (Dict[str, Any]): The object to validate.

        Raises:
            TypeError: If the object is not a dictionary.
        """
        if not isinstance(obj, dict):
            raise TypeError("Object must be a dictionary.")

    @staticmethod
    def _set(client: Redis, key: str, obj: Dict[str, Any]):
        """Synchronously set a JSON obj in Redis for the given key.

        Args:
            client (AsyncRedis): The Redis client instance.
            key (str): The key under which to store the JSON obj.
            obj (Dict[str, Any]): The JSON obj to store in Redis.
        """
        client.json().set(key, "$", obj)

    @staticmethod
    async def _aset(client: AsyncRedis, key: str, obj: Dict[str, Any]):
        """Asynchronously set a JSON obj in Redis for the given key.

        Args:
            client (AsyncRedis): The Redis client instance.
            key (str): The key under which to store the JSON obj.
            obj (Dict[str, Any]): The JSON obj to store in Redis.
        """
        await client.json().set(key, "$", obj)

    @staticmethod
    def _get(client: Redis, key: str) -> Dict[str, Any]:
        """Synchronously retrieve a JSON obj from Redis for the given key.

        Args:
            client (AsyncRedis): The Redis client instance.
            key (str): The key for which to retrieve the JSON obj.
```

```
        Returns:
            Dict[str, Any]: The retrieved JSON obj from Redis.
        """
        return client.json().get(key)

    @staticmethod
    async def _aget(client: AsyncRedis, key: str) -> Dict[str, Any]:
        """Asynchronously retrieve a JSON obj from Redis for the given key.

        Args:
            client (AsyncRedis): The Redis client instance.
            key (str): The key for which to retrieve the JSON obj.

        Returns:
            Dict[str, Any]: The retrieved JSON obj from Redis.
        """
        return await client.json().get(key)
}
```

Chapter 2.4.0

redisvl/query

redisvl/query/__init__.py

```
from redisvl.query.query import (
    BaseQuery,
    CountQuery,
    FilterQuery,
    RangeQuery,
    VectorQuery,
    VectorRangeQuery,
)

__all__ = [
    "BaseQuery",
    "VectorQuery",
    "FilterQuery",
    "RangeQuery",
    "VectorRangeQuery",
    "CountQuery",
]
}
```

redisvl/query/filter.py

```
from enum import Enum
from functools import wraps
```

```
from typing import Any, Callable, Dict, List, Optional, Set, Tuple, Union

from redisvl.utils.token_escaper import TokenEscaper

# disable mypy error for dunder method overrides
# mypy: disable-error-code="override"


class FilterOperator(Enum):
    EQ = 1
    NE = 2
    LT = 3
    GT = 4
    LE = 5
    GE = 6
    OR = 7
    AND = 8
    LIKE = 9
    IN = 10


class FilterField:
    escaper: TokenEscaper = TokenEscaper()
    OPERATORS: Dict[FilterOperator, str] = {}

    def __init__(self, field: str):
        self._field = field
        self._value: Any = None
        self._operator: FilterOperator = FilterOperator.EQ

    def equals(self, other: "FilterField") -> bool:
        if not isinstance(other, type(self)):
            return False
        return (self._field == other._field) and (self._value == other._value)

    def _set_value(
        self,
        val: Any,
        val_type: Union[type, Tuple[type, ...]],
        operator: FilterOperator,
    ):
        # check that the operator is supported by this class
        if operator not in self.OPERATORS:
            raise ValueError(
                f"Operator {operator} not supported by {self.__class__.__name__}. "
                + f"Supported operators are {self.OPERATORS.values()}"
            )
        # check that the value is of the proper type
        if not isinstance(val, val_type):
            raise TypeError(
                f"Right side argument passed to operator {self.OPERATORS[operator]} "
                f"with left side "
                f"argument {self.__class__.__name__} must be of type {val_type}"
            )
        self._value = val
        self._operator = operator


def check_operator_misuse(func: Callable) -> Callable:
    @wraps(func)
    def wrapper(instance: Any, *args: List[Any], **kwargs: Dict[str, Any]) -> Any:
        # Extracting 'other' from positional arguments or keyword arguments
        other = kwargs.get("other") if "other" in kwargs else None
        if not other:
```

```
            for arg in args:
                if isinstance(arg, type(instance)):
                    other = arg
                    break

        if isinstance(other, type(instance)):
            raise ValueError(
                "Equality operators are overridden for FilterExpression creation. Use "
                ".equals() for equality checks"
            )
        return func(instance, *args, **kwargs)

    return wrapper


class Tag(FilterField):
    """A Tag filter can be applied to Tag fields"""

    OPERATORS: Dict[FilterOperator, str] = {
        FilterOperator.EQ: "==",
        FilterOperator.NE: "!=",
        FilterOperator.IN: "==",
    }
    OPERATOR_MAP: Dict[FilterOperator, str] = {
        FilterOperator.EQ: "@%s:{%s}",
        FilterOperator.NE: "(-@%s:{%s})",
        FilterOperator.IN: "@%s:{%s}",
    }
    SUPPORTED_VAL_TYPES = (list, set, tuple, str, type(None))

    def _set_tag_value(
        self, other: Union[List[str], Set[str], str], operator: FilterOperator
    ):
        if isinstance(other, (list, set, tuple)):
            try:
                # "if val" clause removes non-truthy values from list
                other = [str(val) for val in other if val]
            except ValueError:
                raise ValueError("All tags within collection must be strings")
        # above to catch the "" case
        elif not other:
            other = []
        elif isinstance(other, str):
            other = [other]

        self._set_value(other, self.SUPPORTED_VAL_TYPES, operator)

    @check_operator_misuse
    def __eq__(self, other: Union[List[str], str]) -> "FilterExpression":
        """Create a Tag equality filter expression.

        Args:
            other (Union[List[str], str]): The tag(s) to filter on.

        .. code-block:: python

            from redisvl.query.filter import Tag

            f = Tag("brand") == "nike"
        """
        self._set_tag_value(other, FilterOperator.EQ)
        return FilterExpression(str(self))

    @check_operator_misuse
```

```
    def __ne__(self, other) -> "FilterExpression":
        """Create a Tag inequality filter expression.

        Args:
            other (Union[List[str], str]): The tag(s) to filter on.

        .. code-block:: python

            from redisvl.query.filter import Tag
            f = Tag("brand") != "nike"

        """
        self._set_tag_value(other, FilterOperator.NE)
        return FilterExpression(str(self))

    @property
    def _formatted_tag_value(self) -> str:
        return "|".join([self.escaper.escape(tag) for tag in self._value])

    def __str__(self) -> str:
        """Return the Redis Query string for the Tag filter"""
        if not self._value:
            return "*"

        return self.OPERATOR_MAP[self._operator] % (
            self._field,
            self._formatted_tag_value,
        )


class GeoSpec:
    GEO_UNITS = ["m", "km", "mi", "ft"]

    # class for the operand for FilterExpressions with Geo
    def __init__(self, longitude: float, latitude: float, unit: str = "km"):
        if unit.lower() not in self.GEO_UNITS:
            raise ValueError(f"Unit must be one of {self.GEO_UNITS}")
        self._longitude = longitude
        self._latitude = latitude
        self._unit = unit.lower()


class GeoRadius(GeoSpec):
    """A GeoRadius is a GeoSpec representing a geographic radius."""

    def __init__(
        self,
        longitude: float,
        latitude: float,
        radius: int = 1,
        unit: str = "km",
    ):
        """Create a GeoRadius specification (GeoSpec)

        Args:
            longitude (float): The longitude of the center of the radius.
            latitude (float): The latitude of the center of the radius.
            radius (int, optional): The radius of the circle. Defaults to 1.
            unit (str, optional): The unit of the radius. Defaults to "km".

        Raises:
            ValueError: If the unit is not one of "m", "km", "mi", or "ft".
        """
        super().__init__(longitude, latitude, unit)
```

```
        self._radius = radius

    def get_args(self) -> List[Union[float, int, str]]:
        return [self._longitude, self._latitude, self._radius, self._unit]


class Geo(FilterField):
    """A Geo is a FilterField representing a geographic (lat/lon) field in a
    Redis index."""

    OPERATORS: Dict[FilterOperator, str] = {
        FilterOperator.EQ: "==",
        FilterOperator.NE: "!=",
    }
    OPERATOR_MAP: Dict[FilterOperator, str] = {
        FilterOperator.EQ: "@%s:[%s %s %i %s]",
        FilterOperator.NE: "(-@%s:[%s %s %i %s])",
    }
    SUPPORTED_VAL_TYPES = (GeoSpec, type(None))

    @check_operator_misuse
    def __eq__(self, other) -> "FilterExpression":
        """Create a geographic filter within a specified GeoRadius.

        Args:
            other (GeoRadius): The geographic spec to filter on.

        .. code-block:: python

            from redisvl.query.filter import Geo, GeoRadius

            f = Geo("location") == GeoRadius(-122.4194, 37.7749, 1, unit="m")

        """
        self._set_value(other, self.SUPPORTED_VAL_TYPES, FilterOperator.EQ)  #
type: ignore
        return FilterExpression(str(self))

    @check_operator_misuse
    def __ne__(self, other) -> "FilterExpression":
        """Create a geographic filter outside of a specified GeoRadius.

        Args:
            other (GeoRadius): The geographic spec to filter on.

        .. code-block:: python

            from redisvl.query.filter import Geo, GeoRadius

            f = Geo("location") != GeoRadius(-122.4194, 37.7749, 1, unit="m")

        """
        self._set_value(other, self.SUPPORTED_VAL_TYPES, FilterOperator.NE)  #
type: ignore
        return FilterExpression(str(self))

    def __str__(self) -> str:
        """Return the Redis Query string for the Geo filter"""
        if not self._value:
            return "*"

        return self.OPERATOR_MAP[self._operator] % (
            self._field,
            *self._value.get_args(),
```

```
        )


class Num(FilterField):
    """A Num is a FilterField representing a numeric field in a Redis index."""

    OPERATORS: Dict[FilterOperator, str] = {
        FilterOperator.EQ: "==",
        FilterOperator.NE: "!=",
        FilterOperator.LT: "<",
        FilterOperator.GT: ">",
        FilterOperator.LE: "<=",
        FilterOperator.GE: ">=",
    }
    OPERATOR_MAP: Dict[FilterOperator, str] = {
        FilterOperator.EQ: "@%s:[%s %s]",
        FilterOperator.NE: "(-@%s:[%s %s])",
        FilterOperator.GT: "@%s:[(%s +inf]",
        FilterOperator.LT: "@%s:[-inf (%s]",
        FilterOperator.GE: "@%s:[%s +inf]",
        FilterOperator.LE: "@%s:[-inf %s]",
    }
    SUPPORTED_VAL_TYPES = (int, float, type(None))

    def __eq__(self, other: int) -> "FilterExpression":
        """Create a Numeric equality filter expression.

        Args:
            other (int): The value to filter on.

        .. code-block:: python

            from redisvl.query.filter import Num
            f = Num("zipcode") == 90210

        """
        self._set_value(other, self.SUPPORTED_VAL_TYPES, FilterOperator.EQ)
        return FilterExpression(str(self))

    def __ne__(self, other: int) -> "FilterExpression":
        """Create a Numeric inequality filter expression.

        Args:
            other (int): The value to filter on.

        .. code-block:: python

            from redisvl.query.filter import Num

            f = Num("zipcode") != 90210

        """
        self._set_value(other, self.SUPPORTED_VAL_TYPES, FilterOperator.NE)
        return FilterExpression(str(self))

    def __gt__(self, other: int) -> "FilterExpression":
        """Create a Numeric greater than filter expression.

        Args:
            other (int): The value to filter on.

        .. code-block:: python

            from redisvl.query.filter import Num
```

```
        f = Num("age") > 18

    """
    self._set_value(other, self.SUPPORTED_VAL_TYPES, FilterOperator.GT)
    return FilterExpression(str(self))

def __lt__(self, other: int) -> "FilterExpression":
    """Create a Numeric less than filter expression.

    Args:
        other (int): The value to filter on.

    .. code-block:: python

        from redisvl.query.filter import Num

        f = Num("age") < 18

    """
    self._set_value(other, self.SUPPORTED_VAL_TYPES, FilterOperator.LT)
    return FilterExpression(str(self))

def __ge__(self, other: int) -> "FilterExpression":
    """Create a Numeric greater than or equal to filter expression.

    Args:
        other (int): The value to filter on.

    .. code-block:: python

        from redisvl.query.filter import Num

        f = Num("age") >= 18

    """
    self._set_value(other, self.SUPPORTED_VAL_TYPES, FilterOperator.GE)
    return FilterExpression(str(self))

def __le__(self, other: int) -> "FilterExpression":
    """Create a Numeric less than or equal to filter expression.

    Args:
        other (int): The value to filter on.

    .. code-block:: python

        from redisvl.query.filter import Num

        f = Num("age") <= 18

    """
    self._set_value(other, self.SUPPORTED_VAL_TYPES, FilterOperator.LE)
    return FilterExpression(str(self))

def __str__(self) -> str:
    """Return the Redis Query string for the Numeric filter"""
    if self._value is None:
        return "*"
    if self._operator == FilterOperator.EQ or self._operator == FilterOperator.NE:
        return self.OPERATOR_MAP[self._operator] % (
            self._field,
            self._value,
            self._value,
```

```
            )
        else:
            return self.OPERATOR_MAP[self._operator] % (self._field, self._value)


class Text(FilterField):
    """A Text is a FilterField representing a text field in a Redis index."""

    OPERATORS: Dict[FilterOperator, str] = {
        FilterOperator.EQ: "==",
        FilterOperator.NE: "!=",
        FilterOperator.LIKE: "%",
    }
    OPERATOR_MAP: Dict[FilterOperator, str] = {
        FilterOperator.EQ: '@%s:("%s")',
        FilterOperator.NE: '(-@%s:"%s")',
        FilterOperator.LIKE: "@%s:(%s)",
    }
    SUPPORTED_VAL_TYPES = (str, type(None))

    @check_operator_misuse
    def __eq__(self, other: str) -> "FilterExpression":
        """Create a Text equality filter expression. These expressions yield
        filters that enforce an exact match on the supplied term(s).

        Args:
            other (str): The text value to filter on.

        .. code-block:: python

            from redisvl.query.filter import Text

            f = Text("job") == "engineer"

        """
        self._set_value(other, self.SUPPORTED_VAL_TYPES, FilterOperator.EQ)
        return FilterExpression(str(self))

    @check_operator_misuse
    def __ne__(self, other: str) -> "FilterExpression":
        """Create a Text inequality filter expression. These expressions yield
        negated filters on exact matches on the supplied term(s). Opposite of an
        equality filter expression.

        Args:
            other (str): The text value to filter on.

        .. code-block:: python

            from redisvl.query.filter import Text

            f = Text("job") != "engineer"

        """
        self._set_value(other, self.SUPPORTED_VAL_TYPES, FilterOperator.NE)
        return FilterExpression(str(self))

    def __mod__(self, other: str) -> "FilterExpression":
        """Create a Text "LIKE" filter expression. A flexible expression that
        yields filters that can use a variety of additional operators like
        wildcards (*), fuzzy matches (%%), or combinatorics (|) of the supplied
        term(s).

        Args:
```

```
            other (str): The text value to filter on.

        .. code-block:: python

            from redisvl.query.filter import Text

            f = Text("job") % "engine*"         # suffix wild card match
            f = Text("job") % "%%engine%%"      # fuzzy match w/ Levenshtein Distance
            f = Text("job") % "engineer|doctor" # contains either term in field
            f = Text("job") % "engineer doctor" # contains both terms in field

        """
        self._set_value(other, self.SUPPORTED_VAL_TYPES, FilterOperator.LIKE)
        return FilterExpression(str(self))

    def __str__(self) -> str:
        """Return the Redis Query string for the Text filter"""
        if not self._value:
            return "*"

        return self.OPERATOR_MAP[self._operator] % (
            self._field,
            self._value,
        )


class FilterExpression:
    """A FilterExpression is a logical combination of filters in RedisVL.

    FilterExpressions can be combined using the & and | operators to create
    complex expressions that evaluate to the Redis Query language.

    This presents an interface by which users can create complex queries
    without having to know the Redis Query language.

    .. code-block:: python

        from redisvl.query.filter import Tag, Num

        brand_is_nike = Tag("brand") == "nike"
        price_is_over_100 = Num("price") < 100
        f = brand_is_nike & price_is_over_100

        print(str(f))

        >>> (@brand:{nike} @price:[-inf (100)])

    This can be combined with the VectorQuery class to create a query:

    .. code-block:: python

        from redisvl.query import VectorQuery

        v = VectorQuery(
            vector=[0.1, 0.1, 0.5, ...],
            vector_field_name="product_embedding",
            return_fields=["product_id", "brand", "price"],
            filter_expression=f,
        )

    Note:
        Filter expressions are typically not called directly. Instead they are
        built by combining filter statements using the & and | operators.
```

```
    """

    def __init__(
        self,
        _filter: Optional[str] = None,
        operator: Optional[FilterOperator] = None,
        left: Optional["FilterExpression"] = None,
        right: Optional["FilterExpression"] = None,
    ):
        self._filter = _filter
        self._operator = operator
        self._left = left
        self._right = right

    def __and__(self, other) -> "FilterExpression":
        return FilterExpression(operator=FilterOperator.AND, left=self, right=other)

    def __or__(self, other) -> "FilterExpression":
        return FilterExpression(operator=FilterOperator.OR, left=self, right=other)

    @staticmethod
    def format_expression(left, right, operator_str) -> str:
        _left, _right = str(left), str(right)
        if _left == _right == "*":
            return _left
        if _left == "*" != _right:
            return _right
        if _right == "*" != _left:
            return _left
        return f"({_left}{operator_str}{_right})"

    def __str__(self) -> str:
        # top level check that allows recursive calls to __str__
        if not self._filter and not self._operator:
            raise ValueError("Improperly initialized FilterExpression")

        # if theres an operator, combine expressions accordingly
        if self._operator:
            if not isinstance(self._left, FilterExpression) or not isinstance(
                self._right, FilterExpression
            ):
                raise TypeError(
                    "Improper combination of filters. Both left and right should be
type FilterExpression"
                )

            operator_str = " | " if self._operator == FilterOperator.OR else " "
            return self.format_expression(self._left, self._right, operator_str)

        # check that base case, the filter is set
        if not self._filter:
            raise ValueError("Improperly initialized FilterExpression")
        return self._filter
}
```

redisvl/query/query.py

```
from typing import Any, Dict, List, Optional, Union

from redis.commands.search.query import Query as RedisQuery
```

```
from redisvl.query.filter import FilterExpression
from redisvl.redis.utils import array_to_buffer


class BaseQuery(RedisQuery):
    """Base query class used to subclass many query types."""

    _params: Dict[str, Any] = {}
    _filter_expression: Union[str, FilterExpression] = FilterExpression("*")

    def __init__(self, query_string: str = "*"):
        """
        Initialize the BaseQuery class.

        Args:
            query_string (str, optional): The query string to use. Defaults to '*'.
        """
        super().__init__(query_string)

    def __str__(self) -> str:
        """Return the string representation of the query."""
        return " ".join([str(x) for x in self.get_args()])

    def _build_query_string(self) -> str:
        """Build the full Redis query string."""
        raise NotImplementedError("Must be implemented by subclasses")

    def set_filter(
        self, filter_expression: Optional[Union[str, FilterExpression]] = None
    ):
        """Set the filter expression for the query.

        Args:
            filter_expression (Optional[Union[str, FilterExpression]], optional):
The filter
                expression or query string to use on the query.

        Raises:
            TypeError: If filter_expression is not a valid FilterExpression or string.
        """
        if filter_expression is None:
            # Default filter to match everything
            self._filter_expression = FilterExpression("*")
        elif isinstance(filter_expression, (FilterExpression, str)):
            self._filter_expression = filter_expression
        else:
            raise TypeError(
                "filter_expression must be of type FilterExpression or string or None"
            )

        # Reset the query string
        self._query_string = self._build_query_string()

    @property
    def filter(self) -> Union[str, FilterExpression]:
        """The filter expression for the query."""
        return self._filter_expression

    @property
    def query(self) -> "BaseQuery":
        """Return self as the query object."""
        return self
```

```
    @property
    def params(self) -> Dict[str, Any]:
        """Return the query parameters."""
        return self._params


class FilterQuery(BaseQuery):
    def __init__(
        self,
        filter_expression: Optional[Union[str, FilterExpression]] = None,
        return_fields: Optional[List[str]] = None,
        num_results: int = 10,
        dialect: int = 2,
        sort_by: Optional[str] = None,
        in_order: bool = False,
        params: Optional[Dict[str, Any]] = None,
    ):
        """A query for running a filtered search with a filter expression.

        Args:
            filter_expression (Optional[Union[str, FilterExpression]]): The
optional filter
                expression to query with. Defaults to '*'.
            return_fields (Optional[List[str]], optional): The fields to return.
            num_results (Optional[int], optional): The number of results to return.
Defaults to 10.
            dialect (int, optional): The query dialect. Defaults to 2.
            sort_by (Optional[str], optional): The field to order the results by.
Defaults to None.
            in_order (bool, optional): Requires the terms in the field to have the same
order as the terms in the query filter. Defaults to False.
            params (Optional[Dict[str, Any]], optional): The parameters for the query.
Defaults to None.

        Raises:
            TypeError: If filter_expression is not of type
redisvl.query.FilterExpression
        """
        self.set_filter(filter_expression)

        if params:
            self._params = params

        self._num_results = num_results

        # Initialize the base query with the full query string constructed from the
filter expression
        query_string = self._build_query_string()
        super().__init__(query_string)

        # Handle query settings
        if return_fields:
            self.return_fields(*return_fields)
        self.paging(0, self._num_results).dialect(dialect)

        if sort_by:
            self.sort_by(sort_by)

        if in_order:
            self.in_order()

    def _build_query_string(self) -> str:
        """Build the full query string based on the filter and other components."""
        if isinstance(self._filter_expression, FilterExpression):
```

```
            return str(self._filter_expression)
        return self._filter_expression


class CountQuery(BaseQuery):
    def __init__(
        self,
        filter_expression: Optional[Union[str, FilterExpression]] = None,
        dialect: int = 2,
        params: Optional[Dict[str, Any]] = None,
    ):
        """A query for a simple count operation provided some filter expression.

        Args:
            filter_expression (Optional[Union[str, FilterExpression]]): The filter
expression to query with. Defaults to None.
            params (Optional[Dict[str, Any]], optional): The parameters for the query.
Defaults to None.

        Raises:
            TypeError: If filter_expression is not of type
redisvl.query.FilterExpression

        .. code-block:: python

            from redisvl.query import CountQuery
            from redisvl.query.filter import Tag

            t = Tag("brand") == "Nike"
            query = CountQuery(filter_expression=t)

            count = index.query(query)
        """
        self.set_filter(filter_expression)

        if params:
            self._params = params

        # Initialize the base query with the full query string constructed from the
filter expression
        query_string = self._build_query_string()
        super().__init__(query_string)

        # Query specific modifications
        self.no_content().paging(0, 0).dialect(dialect)

    def _build_query_string(self) -> str:
        """Build the full query string based on the filter and other components."""
        if isinstance(self._filter_expression, FilterExpression):
            return str(self._filter_expression)
        return self._filter_expression


class BaseVectorQuery:
    DISTANCE_ID: str = "vector_distance"
    VECTOR_PARAM: str = "vector"


class VectorQuery(BaseVectorQuery, BaseQuery):
    def __init__(
        self,
        vector: Union[List[float], bytes],
        vector_field_name: str,
        return_fields: Optional[List[str]] = None,
```

```
        filter_expression: Optional[Union[str, FilterExpression]] = None,
        dtype: str = "float32",
        num_results: int = 10,
        return_score: bool = True,
        dialect: int = 2,
        sort_by: Optional[str] = None,
        in_order: bool = False,
    ):
        """A query for running a vector search along with an optional filter
        expression.

        Args:
            vector (List[float]): The vector to perform the vector search with.
            vector_field_name (str): The name of the vector field to search
                against in the database.
            return_fields (List[str]): The declared fields to return with search
                results.
            filter_expression (Union[str, FilterExpression], optional): A filter
to apply
                along with the vector search. Defaults to None.
            dtype (str, optional): The dtype of the vector. Defaults to
                "float32".
            num_results (int, optional): The top k results to return from the
                vector search. Defaults to 10.
            return_score (bool, optional): Whether to return the vector
                distance. Defaults to True.
            dialect (int, optional): The RediSearch query dialect.
                Defaults to 2.
            sort_by (Optional[str]): The field to order the results by. Defaults
                to None. Results will be ordered by vector distance.
            in_order (bool): Requires the terms in the field to have
                the same order as the terms in the query filter, regardless of
                the offsets between them. Defaults to False.

        Raises:
            TypeError: If filter_expression is not of type
redisvl.query.FilterExpression

        Note:
            Learn more about vector queries in Redis:
https://redis.io/docs/interact/search-and-query/search/vectors/#knn-search
        """
        self._vector = vector
        self._vector_field_name = vector_field_name
        self._dtype = dtype
        self._num_results = num_results
        self.set_filter(filter_expression)
        query_string = self._build_query_string()

        super().__init__(query_string)

        # Handle query modifiers
        if return_fields:
            self.return_fields(*return_fields)

        self.paging(0, self._num_results).dialect(dialect)

        if return_score:
            self.return_fields(self.DISTANCE_ID)

        if sort_by:
            self.sort_by(sort_by)
        else:
            self.sort_by(self.DISTANCE_ID)
```

```
        if in_order:
            self.in_order()

    def _build_query_string(self) -> str:
        """Build the full query string for vector search with optional filtering."""
        filter_expression = self._filter_expression
        if isinstance(filter_expression, FilterExpression):
            filter_expression = str(filter_expression)
        return f"{filter_expression}=>[KNN {self._num_results}
@{self._vector_field_name} ${self.VECTOR_PARAM} AS {self.DISTANCE_ID}]"

    @property
    def params(self) -> Dict[str, Any]:
        """Return the parameters for the query.

        Returns:
            Dict[str, Any]: The parameters for the query.
        """
        if isinstance(self._vector, bytes):
            vector = self._vector
        else:
            vector = array_to_buffer(self._vector, dtype=self._dtype)

        return {self.VECTOR_PARAM: vector}


class VectorRangeQuery(BaseVectorQuery, BaseQuery):
    DISTANCE_THRESHOLD_PARAM: str = "distance_threshold"

    def __init__(
        self,
        vector: Union[List[float], bytes],
        vector_field_name: str,
        return_fields: Optional[List[str]] = None,
        filter_expression: Optional[Union[str, FilterExpression]] = None,
        dtype: str = "float32",
        distance_threshold: float = 0.2,
        num_results: int = 10,
        return_score: bool = True,
        dialect: int = 2,
        sort_by: Optional[str] = None,
        in_order: bool = False,
    ):
        """A query for running a filtered vector search based on semantic
        distance threshold.

        Args:
            vector (List[float]): The vector to perform the range query with.
            vector_field_name (str): The name of the vector field to search
                against in the database.
            return_fields (List[str]): The declared fields to return with search
                results.
            filter_expression (Union[str, FilterExpression], optional): A filter
to apply
                along with the range query. Defaults to None.
            dtype (str, optional): The dtype of the vector. Defaults to
                "float32".
            distance_threshold (str, float): The threshold for vector distance.
                A smaller threshold indicates a stricter semantic search.
                Defaults to 0.2.
            num_results (int): The MAX number of results to return.
                Defaults to 10.
            return_score (bool, optional): Whether to return the vector
```

```
                distance. Defaults to True.
            dialect (int, optional): The RediSearch query dialect.
                Defaults to 2.
            sort_by (Optional[str]): The field to order the results by. Defaults
                to None. Results will be ordered by vector distance.
            in_order (bool): Requires the terms in the field to have
                the same order as the terms in the query filter, regardless of
                the offsets between them. Defaults to False.

        Raises:
            TypeError: If filter_expression is not of type
redisvl.query.FilterExpression

        Note:
            Learn more about vector range queries:
https://redis.io/docs/interact/search-and-query/search/vectors/#range-query

        """
        self._vector = vector
        self._vector_field_name = vector_field_name
        self._dtype = dtype
        self._num_results = num_results
        self.set_distance_threshold(distance_threshold)
        self.set_filter(filter_expression)
        query_string = self._build_query_string()

        super().__init__(query_string)

        # Handle query modifiers
        if return_fields:
            self.return_fields(*return_fields)

        self.paging(0, self._num_results).dialect(dialect)

        if return_score:
            self.return_fields(self.DISTANCE_ID)

        if sort_by:
            self.sort_by(sort_by)
        else:
            self.sort_by(self.DISTANCE_ID)

        if in_order:
            self.in_order()

    def _build_query_string(self) -> str:
        """Build the full query string for vector range queries with
optional filtering"""
        base_query = f"@{self._vector_field_name}:[VECTOR_RANGE
${self.DISTANCE_THRESHOLD_PARAM} ${self.VECTOR_PARAM}]"

        filter_expression = self._filter_expression
        if isinstance(filter_expression, FilterExpression):
            filter_expression = str(filter_expression)

        if filter_expression == "*":
            return f"{base_query}=>{{$yield_distance_as: {self.DISTANCE_ID}}}"
        return f"({base_query}=>{{$yield_distance_as: {self.DISTANCE_ID}}}
{filter_expression})"

    def set_distance_threshold(self, distance_threshold: float):
        """Set the distance threshold for the query.

        Args:
```

```
            distance_threshold (float): vector distance
        """
        if not isinstance(distance_threshold, (float, int)):
            raise TypeError("distance_threshold must be of type int or float")
        self._distance_threshold = distance_threshold

    @property
    def distance_threshold(self) -> float:
        """Return the distance threshold for the query.

        Returns:
            float: The distance threshold for the query.
        """
        return self._distance_threshold

    @property
    def params(self) -> Dict[str, Any]:
        """Return the parameters for the query.

        Returns:
            Dict[str, Any]: The parameters for the query.
        """
        if isinstance(self._vector, bytes):
            vector_param = self._vector
        else:
            vector_param = array_to_buffer(self._vector, dtype=self._dtype)

        return {
            self.VECTOR_PARAM: vector_param,
            self.DISTANCE_THRESHOLD_PARAM: self._distance_threshold,
        }


class RangeQuery(VectorRangeQuery):
    # keep for backwards compatibility
    pass
}
```

Chapter 2.5.0

redisvl/redis

redisvl/redis/connection.py

```
import os
from typing import Any, Dict, List, Optional, Type

from redis import Redis
from redis.asyncio import Connection as AsyncConnection
from redis.asyncio import ConnectionPool as AsyncConnectionPool
from redis.asyncio import Redis as AsyncRedis
from redis.asyncio import SSLConnection as AsyncSSLConnection
from redis.connection import AbstractConnection, SSLConnection
from redis.exceptions import ResponseError
```

```
from redisvl.exceptions import RedisModuleVersionError
from redisvl.redis.constants import DEFAULT_REQUIRED_MODULES
from redisvl.redis.utils import convert_bytes
from redisvl.version import __version__


def compare_versions(version1, version2):
    """
    Compare two Redis version strings numerically.

    Parameters:
    version1 (str): The first version string (e.g., "7.2.4").
    version2 (str): The second version string (e.g., "6.2.1").

    Returns:
    int: -1 if version1 < version2, 0 if version1 == version2, 1 if version1
> version2.
    """
    v1_parts = list(map(int, version1.split(".")))
    v2_parts = list(map(int, version2.split(".")))

    for v1, v2 in zip(v1_parts, v2_parts):
        if v1 < v2:
            return False
        elif v1 > v2:
            return True

    # If the versions are equal so far, compare the lengths of the version parts
    if len(v1_parts) < len(v2_parts):
        return False
    elif len(v1_parts) > len(v2_parts):
        return True

    return True


def unpack_redis_modules(module_list: List[Dict[str, Any]]) -> Dict[str, Any]:
    """Unpack a list of Redis modules pulled from the MODULES LIST command."""
    return {module["name"]: module["ver"] for module in module_list}


def get_address_from_env() -> str:
    """Get a redis connection from environment variables.

    Returns:
        str: Redis URL
    """
    if "REDIS_URL" not in os.environ:
        raise ValueError("REDIS_URL env var not set")
    return os.environ["REDIS_URL"]


def make_lib_name(*args) -> str:
    """Build the lib name to be reported through the Redis client setinfo
    command.

    Returns:
        str: Redis client library name
    """
    custom_libs = f"redisvl_v{__version__}"
    for arg in args:
        if arg:
            custom_libs += f";{arg}"
```

```
    return f"redis-py({custom_libs})"


def convert_index_info_to_schema(index_info: Dict[str, Any]) -> Dict[str, Any]:
    """Convert the output of FT.INFO into a schema-ready dictionary.

    Args:
        index_info (Dict[str, Any]): Output of the Redis FT.INFO command.

    Returns:
        Dict[str, Any]: Schema dictionary.
    """
    index_name = index_info["index_name"]
    prefixes = index_info["index_definition"][3][0]
    storage_type = index_info["index_definition"][1].lower()

    index_fields = index_info["attributes"]

    def parse_vector_attrs(attrs):
        vector_attrs = {attrs[i].lower(): attrs[i + 1] for i in range(6,
len(attrs), 2)}
        vector_attrs["dims"] = int(vector_attrs.pop("dim"))
        vector_attrs["distance_metric"] = vector_attrs.pop("distance_metric").lower()
        vector_attrs["algorithm"] = vector_attrs.pop("algorithm").lower()
        vector_attrs["datatype"] = vector_attrs.pop("data_type").lower()
        return vector_attrs

    def parse_attrs(attrs):
        return {attrs[i].lower(): attrs[i + 1] for i in range(6, len(attrs), 2)}

    schema_fields = []

    for field_attrs in index_fields:
        # parse field info
        name = field_attrs[1] if storage_type == "hash" else field_attrs[3]
        field = {"name": name, "type": field_attrs[5].lower()}
        if storage_type == "json":
            field["path"] = field_attrs[1]
        # parse field attrs
        if field_attrs[5] == "VECTOR":
            field["attrs"] = parse_vector_attrs(field_attrs)
        else:
            field["attrs"] = parse_attrs(field_attrs)
        # append field
        schema_fields.append(field)

    return {
        "index": {"name": index_name, "prefix": prefixes,
"storage_type": storage_type},
        "fields": schema_fields,
    }


def validate_modules(
    installed_modules: Dict[str, Any],
    required_modules: Optional[List[Dict[str, Any]]] = None,
) -> None:
    """
    Validates if required Redis modules are installed.

    Args:
        installed_modules: List of installed modules.
        required_modules: List of required modules.
```

```
    Raises:
        ValueError: If required Redis modules are not installed.
    """
    required_modules = required_modules or DEFAULT_REQUIRED_MODULES

    for required_module in required_modules:
        if required_module["name"] in installed_modules:
            installed_version = installed_modules[required_module["name"]]  #
type: ignore
            if int(installed_version) >= int(required_module["ver"]):  # type: ignore
                return

    # Build the error message dynamically
    required_modules_str = " OR ".join(
        [f'{module["name"]} >= {module["ver"]}' for module in required_modules]
    )
    error_message = (
        f"Required Redis db module {required_modules_str} not installed. "
        "See Redis Stack docs at
https://redis.io/docs/latest/operate/oss_and_stack/install/install-stack/."
    )

    raise RedisModuleVersionError(error_message)


class RedisConnectionFactory:
    """Builds connections to a Redis database, supporting both synchronous and
    asynchronous clients.

    This class allows for establishing and handling Redis connections using
    either standard Redis or async Redis clients, based on the provided
    configuration.
    """

    @classmethod
    def connect(
        cls, redis_url: Optional[str] = None, use_async: bool = False, **kwargs
    ) -> None:
        """Create a connection to the Redis database based on a URL and some
        connection kwargs.

        This method sets up either a synchronous or asynchronous Redis client
        based on the provided parameters.

        Args:
            redis_url (Optional[str]): The URL of the Redis server to connect
                to. If not provided, the environment variable REDIS_URL is used.
            use_async (bool): If True, an asynchronous client is created.
                Defaults to False.
            **kwargs: Additional keyword arguments to be passed to the Redis
                client constructor.

        Raises:
            ValueError: If redis_url is not provided and REDIS_URL environment
                variable is not set.
        """
        redis_url = redis_url or get_address_from_env()
        connection_func = (
            cls.get_async_redis_connection if use_async else cls.get_redis_connection
        )
        return connection_func(redis_url, **kwargs)  # type: ignore

    @staticmethod
    def get_redis_connection(url: Optional[str] = None, **kwargs) -> Redis:
```

```
        """Creates and returns a synchronous Redis client.

        Args:
            url (Optional[str]): The URL of the Redis server. If not provided,
                the environment variable REDIS_URL is used.
            **kwargs: Additional keyword arguments to be passed to the Redis
                client constructor.

        Returns:
            Redis: A synchronous Redis client instance.

        Raises:
            ValueError: If url is not provided and REDIS_URL environment
                variable is not set.
        """
        if url:
            return Redis.from_url(url, **kwargs)
        # fallback to env var REDIS_URL
        return Redis.from_url(get_address_from_env(), **kwargs)

    @staticmethod
    def get_async_redis_connection(url: Optional[str] = None, **kwargs) -> AsyncRedis:
        """Creates and returns an asynchronous Redis client.

        Args:
            url (Optional[str]): The URL of the Redis server. If not provided,
                the environment variable REDIS_URL is used.
            **kwargs: Additional keyword arguments to be passed to the async
                Redis client constructor.

        Returns:
            AsyncRedis: An asynchronous Redis client instance.

        Raises:
            ValueError: If url is not provided and REDIS_URL environment
                variable is not set.
        """
        if url:
            return AsyncRedis.from_url(url, **kwargs)
        # fallback to env var REDIS_URL
        return AsyncRedis.from_url(get_address_from_env(), **kwargs)

    @staticmethod
    def sync_to_async_redis(redis_client: Redis) -> AsyncRedis:
        # pick the right connection class
        connection_class: Type[AbstractConnection] = (
            AsyncSSLConnection
            if redis_client.connection_pool.connection_class == SSLConnection
            else AsyncConnection
        )
        # make async client
        return AsyncRedis.from_pool(  # type: ignore
            AsyncConnectionPool(
                connection_class=connection_class,
                **redis_client.connection_pool.connection_kwargs,
            )
        )

    @staticmethod
    def get_modules(client: Redis) -> Dict[str, Any]:
        return unpack_redis_modules(convert_bytes(client.module_list()))

    @staticmethod
    async def get_modules_async(client: AsyncRedis) -> Dict[str, Any]:
```

```
        return unpack_redis_modules(convert_bytes(await client.module_list()))

    @staticmethod
    def validate_sync_redis(
        redis_client: Redis,
        lib_name: Optional[str] = None,
        required_modules: Optional[List[Dict[str, Any]]] = None,
    ) -> None:
        """Validates the sync Redis client."""
        # Set client library name
        _lib_name = make_lib_name(lib_name)
        try:
            redis_client.client_setinfo("LIB-NAME", _lib_name)  # type: ignore
        except ResponseError:
            # Fall back to a simple log echo
            redis_client.echo(_lib_name)

        # Get list of modules
        installed_modules = RedisConnectionFactory.get_modules(redis_client)

        # Validate available modules
        validate_modules(installed_modules, required_modules)

    @staticmethod
    async def validate_async_redis(
        redis_client: AsyncRedis,
        lib_name: Optional[str] = None,
        required_modules: Optional[List[Dict[str, Any]]] = None,
    ) -> None:
        """Validates the async Redis client."""
        # Set client library name
        _lib_name = make_lib_name(lib_name)
        try:
            await redis_client.client_setinfo("LIB-NAME", _lib_name)  # type: ignore
        except ResponseError:
            # Fall back to a simple log echo
            await redis_client.echo(_lib_name)

        # Get list of modules
        installed_modules = await
RedisConnectionFactory.get_modules_async(redis_client)

        # Validate available modules
        validate_modules(installed_modules, required_modules)
}
```

redisvl/redis/constants.py

```
# required modules
DEFAULT_REQUIRED_MODULES = [
    {"name": "search", "ver": 20600},
    {"name": "searchlight", "ver": 20600},
]

# default tag separator
REDIS_TAG_SEPARATOR = ","
}
```

redisvl/redis/utils.py

```
import hashlib
from typing import Any, Dict, List, Optional

import numpy as np
from ml_dtypes import bfloat16

from redisvl.schema.fields import VectorDataType


def make_dict(values: List[Any]) -> Dict[Any, Any]:
    """Convert a list of objects into a dictionary"""
    i = 0
    di = {}
    while i < len(values) - 1:
        di[values[i]] = values[i + 1]
        i += 2
    return di


def convert_bytes(data: Any) -> Any:
    """Convert bytes data back to string"""
    if isinstance(data, bytes):
        try:
            return data.decode("utf-8")
        except:
            return data
    if isinstance(data, dict):
        return {convert_bytes(key): convert_bytes(value) for key, value
in data.items()}
    if isinstance(data, list):
        return [convert_bytes(item) for item in data]
    if isinstance(data, tuple):
        return tuple(convert_bytes(item) for item in data)
    return data


def array_to_buffer(array: List[float], dtype: str) -> bytes:
    """Convert a list of floats into a numpy byte string."""
    try:
        VectorDataType(dtype.upper())
    except ValueError:
        raise ValueError(
            f"Invalid data type: {dtype}. Supported types are: {[t.lower() for t
in VectorDataType]}"
        )
    return np.array(array).astype(dtype.lower()).tobytes()


def buffer_to_array(buffer: bytes, dtype: str) -> List[float]:
    """Convert bytes into into a list of floats."""
    try:
        VectorDataType(dtype.upper())
    except ValueError:
        raise ValueError(
            f"Invalid data type: {dtype}. Supported types are: {[t.lower() for t
in VectorDataType]}"
        )
    return np.frombuffer(buffer, dtype=dtype.lower()).tolist()
```

```
def hashify(content: str, extras: Optional[Dict[str, Any]] = None) -> str:
    """Create a secure hash of some arbitrary input text and optional dictionary."""
    if extras:
        extra_string = " ".join([str(k) + str(v) for k, v in sorted(extras.items())])
        content = content + extra_string
    return hashlib.sha256(content.encode("utf-8")).hexdigest()
}
```

Chapter 2.6.0

redisvl/schema

redisvl/schema/__init__.py

```
from redisvl.schema.schema import IndexInfo, IndexSchema, StorageType

__all__ = ["StorageType", "IndexSchema", "IndexInfo"]
}
```

redisvl/schema/fields.py

```
"""
RedisVL Fields, FieldAttributes, and Enums

Reference Redis search source documentation as needed:
https://redis.io/commands/ft.create/
Reference Redis vector search documentation as needed:
https://redis.io/docs/interact/search-and-query/advanced-concepts/vectors/
"""

from enum import Enum
from typing import Any, Dict, Optional, Tuple, Type, Union

from pydantic.v1 import BaseModel, Field, validator
from redis.commands.search.field import Field as RedisField
from redis.commands.search.field import GeoField as RedisGeoField
from redis.commands.search.field import NumericField as RedisNumericField
from redis.commands.search.field import TagField as RedisTagField
from redis.commands.search.field import TextField as RedisTextField
from redis.commands.search.field import VectorField as RedisVectorField

### Attribute Enums ###


class VectorDistanceMetric(str, Enum):
    COSINE = "COSINE"
    L2 = "L2"
    IP = "IP"
```

```
class VectorDataType(str, Enum):
    BFLOAT16 = "BFLOAT16"
    FLOAT16 = "FLOAT16"
    FLOAT32 = "FLOAT32"
    FLOAT64 = "FLOAT64"


class VectorIndexAlgorithm(str, Enum):
    FLAT = "FLAT"
    HNSW = "HNSW"


### Field Attributes ###


class BaseFieldAttributes(BaseModel):
    """Base field attributes shared by other lexical fields"""

    sortable: bool = Field(default=False)
    """Enable faster result sorting on the field at runtime"""


class TextFieldAttributes(BaseFieldAttributes):
    """Full text field attributes"""

    weight: float = Field(default=1)
    """Declares the importance of this field when calculating results"""
    no_stem: bool = Field(default=False)
    """Disable stemming on the text field during indexing"""
    withsuffixtrie: bool = Field(default=False)
    """Keep a suffix trie with all terms which match the suffix to optimize
certain queries"""
    phonetic_matcher: Optional[str] = None
    """Used to perform phonetic matching during search"""


class TagFieldAttributes(BaseFieldAttributes):
    """Tag field attributes"""

    separator: str = Field(default=",")
    """Indicates how the text in the original attribute is split into
individual tags"""
    case_sensitive: bool = Field(default=False)
    """Treat text as case sensitive or not. By default, tag characters are converted
to lowercase"""
    withsuffixtrie: bool = Field(default=False)
    """Keep a suffix trie with all terms which match the suffix to optimize
certain queries"""


class NumericFieldAttributes(BaseFieldAttributes):
    """Numeric field attributes"""

    pass


class GeoFieldAttributes(BaseFieldAttributes):
    """Numeric field attributes"""

    pass


class BaseVectorFieldAttributes(BaseModel):
```

```
    """Base vector field attributes shared by both FLAT and HNSW fields"""

    dims: int
    """Dimensionality of the vector embeddings field"""
    algorithm: VectorIndexAlgorithm
    """The indexing algorithm for the field: HNSW or FLAT"""
    datatype: VectorDataType = Field(default=VectorDataType.FLOAT32)
    """The float datatype for the vector embeddings"""
    distance_metric: VectorDistanceMetric = Field(default=VectorDistanceMetric.COSINE)
    """The distance metric used to measure query relevance"""
    initial_cap: Optional[int] = None
    """Initial vector capacity in the index affecting memory allocation size of
the index"""

    @validator("algorithm", "datatype", "distance_metric", pre=True)
    @classmethod
    def uppercase_strings(cls, v):
        """Validate that provided values are cast to uppercase"""
        return v.upper()

    @property
    def field_data(self) -> Dict[str, Any]:
        """Select attributes required by the Redis API"""
        field_data = {
            "TYPE": self.datatype,
            "DIM": self.dims,
            "DISTANCE_METRIC": self.distance_metric,
        }
        if self.initial_cap is not None:  # Only include it if it's set
            field_data["INITIAL_CAP"] = self.initial_cap
        return field_data


class FlatVectorFieldAttributes(BaseVectorFieldAttributes):
    """FLAT vector field attributes"""

    algorithm: VectorIndexAlgorithm = Field(
        default=VectorIndexAlgorithm.FLAT, const=True
    )
    """The indexing algorithm for the vector field"""
    block_size: Optional[int] = None
    """Block size to hold amount of vectors in a contiguous array. This is useful when
the index is dynamic with respect to addition and deletion"""


class HNSWVectorFieldAttributes(BaseVectorFieldAttributes):
    """HNSW vector field attributes"""

    algorithm: VectorIndexAlgorithm = Field(
        default=VectorIndexAlgorithm.HNSW, const=True
    )
    """The indexing algorithm for the vector field"""
    m: int = Field(default=16)
    """Number of max outgoing edges for each graph node in each layer"""
    ef_construction: int = Field(default=200)
    """Number of max allowed potential outgoing edges candidates for each node in the
graph during build time"""
    ef_runtime: int = Field(default=10)
    """Number of maximum top candidates to hold during KNN search"""
    epsilon: float = Field(default=0.01)
    """Relative factor that sets the boundaries in which a range query may search
for candidates"""
```

```
### Field Classes ###


class BaseField(BaseModel):
    """Base field"""

    name: str
    """Field name"""
    type: str
    """Field type"""
    path: Optional[str] = None
    """Field path (within JSON object)"""
    attrs: Optional[Union[BaseFieldAttributes, BaseVectorFieldAttributes]] = None
    """Specified field attributes"""

    def _handle_names(self) -> Tuple[str, Optional[str]]:
        if self.path:
            return self.path, self.name
        return self.name, None

    def as_redis_field(self) -> RedisField:
        raise NotImplementedError


class TextField(BaseField):
    """Text field supporting a full text search index"""

    type: str = Field(default="text", const=True)
    attrs: TextFieldAttributes = Field(default_factory=TextFieldAttributes)

    def as_redis_field(self) -> RedisField:
        name, as_name = self._handle_names()
        return RedisTextField(
            name,
            as_name=as_name,
            weight=self.attrs.weight,  # type: ignore
            no_stem=self.attrs.no_stem,  # type: ignore
            phonetic_matcher=self.attrs.phonetic_matcher,  # type: ignore
            sortable=self.attrs.sortable,
        )


class TagField(BaseField):
    """Tag field for simple boolean-style filtering"""

    type: str = Field(default="tag", const=True)
    attrs: TagFieldAttributes = Field(default_factory=TagFieldAttributes)

    def as_redis_field(self) -> RedisField:
        name, as_name = self._handle_names()
        return RedisTagField(
            name,
            as_name=as_name,
            separator=self.attrs.separator,  # type: ignore
            case_sensitive=self.attrs.case_sensitive,  # type: ignore
            sortable=self.attrs.sortable,
        )


class NumericField(BaseField):
    """Numeric field for numeric range filtering"""

    type: str = Field(default="numeric", const=True)
    attrs: NumericFieldAttributes = Field(default_factory=NumericFieldAttributes)
```

```
    def as_redis_field(self) -> RedisField:
        name, as_name = self._handle_names()
        return RedisNumericField(
            name,
            as_name=as_name,
            sortable=self.attrs.sortable,
        )


class GeoField(BaseField):
    """Geo field with a geo-spatial index for location based search"""

    type: str = Field(default="geo", const=True)
    attrs: GeoFieldAttributes = Field(default_factory=GeoFieldAttributes)

    def as_redis_field(self) -> RedisField:
        name, as_name = self._handle_names()
        return RedisGeoField(
            name,
            as_name=as_name,
            sortable=self.attrs.sortable,
        )


class FlatVectorField(BaseField):
    "Vector field with a FLAT index (brute force nearest neighbors search)"
    type: str = Field(default="vector", const=True)
    attrs: FlatVectorFieldAttributes

    def as_redis_field(self) -> RedisField:
        # grab base field params and augment with flat-specific fields
        name, as_name = self._handle_names()
        field_data = self.attrs.field_data
        if self.attrs.block_size is not None:
            field_data["BLOCK_SIZE"] = self.attrs.block_size
        return RedisVectorField(name, self.attrs.algorithm,
field_data, as_name=as_name)


class HNSWVectorField(BaseField):
    """Vector field with an HNSW index (approximate nearest neighbors search)"""

    type: str = Field(default="vector", const=True)
    attrs: HNSWVectorFieldAttributes

    def as_redis_field(self) -> RedisField:
        # grab base field params and augment with hnsw-specific fields
        name, as_name = self._handle_names()
        field_data = self.attrs.field_data
        field_data.update(
            {
                "M": self.attrs.m,
                "EF_CONSTRUCTION": self.attrs.ef_construction,
                "EF_RUNTIME": self.attrs.ef_runtime,
                "EPSILON": self.attrs.epsilon,
            }
        )
        return RedisVectorField(name, self.attrs.algorithm,
field_data, as_name=as_name)


class FieldFactory:
    """Factory class to create fields from client data and kwargs."""
```

```
    FIELD_TYPE_MAP = {
        "tag": TagField,
        "text": TextField,
        "numeric": NumericField,
        "geo": GeoField,
    }

    VECTOR_FIELD_TYPE_MAP = {
        "flat": FlatVectorField,
        "hnsw": HNSWVectorField,
    }

    @classmethod
    def pick_vector_field_type(cls, attrs: Dict[str, Any]) -> Type[BaseField]:
        """Get the vector field type from the field data."""
        if "algorithm" not in attrs:
            raise ValueError("Must provide algorithm param for the vector field.")

        if "dims" not in attrs:
            raise ValueError("Must provide dims param for the vector field.")

        algorithm = attrs["algorithm"].lower()
        if algorithm not in cls.VECTOR_FIELD_TYPE_MAP:
            raise ValueError(f"Unknown vector field algorithm: {algorithm}")

        return cls.VECTOR_FIELD_TYPE_MAP[algorithm]  # type: ignore

    @classmethod
    def create_field(
        cls,
        type: str,
        name: str,
        attrs: Dict[str, Any] = {},
        path: Optional[str] = None,
    ) -> BaseField:
        """Create a field of a given type with provided attributes."""

        if type == "vector":
            field_class = cls.pick_vector_field_type(attrs)
        else:
            if type not in cls.FIELD_TYPE_MAP:
                raise ValueError(f"Unknown field type: {type}")
            field_class = cls.FIELD_TYPE_MAP[type]  # type: ignore

        return field_class(name=name, path=path, attrs=attrs)  # type: ignore
}
```

redisvl/schema/schema.py

```
import re
from enum import Enum
from pathlib import Path
from typing import Any, Dict, List

import yaml
from pydantic.v1 import BaseModel, Field, root_validator
from redis.commands.search.field import Field as RedisField

from redisvl.schema.fields import BaseField, FieldFactory
```

```
from redisvl.utils.log import get_logger
from redisvl.utils.utils import model_to_dict

logger = get_logger(__name__)
SCHEMA_VERSION = "0.1.0"


class StorageType(Enum):
    """
    Enumeration for the storage types supported in Redis.

    Attributes:
        HASH (str): Represents the 'hash' storage type in Redis.
        JSON (str): Represents the 'json' storage type in Redis.
    """

    HASH = "hash"
    JSON = "json"


class IndexInfo(BaseModel):
    """Index info includes the essential details regarding index settings,
    such as its name, prefix, key separator, and storage type in Redis.

    In yaml format, the index info section looks like:

    .. code-block:: yaml

        index:
            name: user-index
            prefix: user
            key_separtor: ':'
            storage_type: json

    In dict format, the index info section looks like:

    .. code-block:: python

        {"index": {
            "name": "user-index",
            "prefix": "user",
            "key_separator": ":",
            "storage_type": "json"
        }}

    """

    name: str
    """The unique name of the index."""
    prefix: str = "rvl"
    """The prefix used for Redis keys associated with this index."""
    key_separator: str = ":"
    """The separator character used in designing Redis keys."""
    storage_type: StorageType = StorageType.HASH
    """The storage type used in Redis (e.g., 'hash' or 'json')."""


class IndexSchema(BaseModel):
    """A schema definition for a search index in Redis, used in RedisVL for
    configuring index settings and organizing vector and metadata fields.

    The class offers methods to create an index schema from a YAML file or a
    Python dictionary, supporting flexible schema definitions and easy
    integration into various workflows.
```

An example `schema.yaml` file might look like this:

.. code-block:: yaml

    version: '0.1.0'

    index:
        name: user-index
        prefix: user
        key_separator: ":"
        storage_type: json

    fields:
        - name: user
          type: tag
        - name: credit_score
          type: tag
        - name: embedding
          type: vector
          attrs:
            algorithm: flat
            dims: 3
            distance_metric: cosine
            datatype: float32

Loading the schema for RedisVL from yaml is as simple as:

.. code-block:: python

    from redisvl.schema import IndexSchema

    schema = IndexSchema.from_yaml("schema.yaml")

Loading the schema for RedisVL from dict is as simple as:

.. code-block:: python

    from redisvl.schema import IndexSchema

    schema = IndexSchema.from_dict({
        "index": {
            "name": "user-index",
            "prefix": "user",
            "key_separator": ":",
            "storage_type": "json",
        },
        "fields": [
            {"name": "user", "type": "tag"},
            {"name": "credit_score", "type": "tag"},
            {
                "name": "embedding",
                "type": "vector",
                "attrs": {
                    "algorithm": "flat",
                    "dims": 3,
                    "distance_metric": "cosine",
                    "datatype": "float32"
                }
            }
        ]
    })

Note:

```
        The `fields` attribute in the schema must contain unique field names to ensure
        correct and unambiguous field references.

    """

    index: IndexInfo
    """Details of the basic index configurations."""
    fields: Dict[str, BaseField] = {}
    """Fields associated with the search index and their properties"""
    version: str = Field(default=SCHEMA_VERSION, const=True)
    """Version of the underlying index schema."""

    @staticmethod
    def _make_field(storage_type, **field_inputs) -> BaseField:
        """
        Parse raw field inputs derived from YAML or dict.

        Validates and sets the 'path' attribute for fields when using JSON
storage type.
        """
        # Create field from inputs
        field = FieldFactory.create_field(**field_inputs)
        # Handle field path and storage type
        if storage_type == StorageType.JSON:
            field.path = field.path if field.path else f"$.{field.name}"
        else:
            if field.path is not None:
                logger.warning(
                    f"Path attribute for field '{field.name}' will be ignored for HASH
storage type."
                )
            field.path = None
        return field

    @root_validator(pre=True)
    @classmethod
    def validate_and_create_fields(cls, values):
        """
        Validate uniqueness of field names and create valid field instances.
        """
        # Ensure index is a dictionary for validation
        index = values.get("index")
        if not isinstance(index, IndexInfo):
            index = IndexInfo(**index)

        input_fields = values.get("fields", [])
        prepared_fields: Dict[str, BaseField] = {}
        # Handle old fields format temporarily
        if isinstance(input_fields, dict):
            raise ValueError("New schema format introduced; please update
schema spec.")
        # Process and create fields
        for field_input in input_fields:
            field = cls._make_field(index.storage_type, **field_input)
            if field.name in prepared_fields:
                raise ValueError(
                    f"Duplicate field name: {field.name}. Field names must be unique
across all fields."
                )
            prepared_fields[field.name] = field

        values["fields"] = prepared_fields
        values["index"] = index
        return values
```

```
    @classmethod
    def from_yaml(cls, file_path: str) -> "IndexSchema":
        """Create an IndexSchema from a YAML file.

        Args:
            file_path (str): The path to the YAML file.

        Returns:
            IndexSchema: The index schema.

        .. code-block:: python

            from redisvl.schema import IndexSchema
            schema = IndexSchema.from_yaml("schema.yaml")
        """
        try:
            fp = Path(file_path).resolve()
        except OSError as e:
            raise ValueError(f"Invalid file path: {file_path}") from e

        if not fp.exists():
            raise FileNotFoundError(f"Schema file {file_path} does not exist")

        with open(fp, "r") as f:
            yaml_data = yaml.safe_load(f)
            return cls(**yaml_data)

    @classmethod
    def from_dict(cls, data: Dict[str, Any]) -> "IndexSchema":
        """Create an IndexSchema from a dictionary.

        Args:
            data (Dict[str, Any]): The index schema data.

        Returns:
            IndexSchema: The index schema.

        .. code-block:: python

            from redisvl.schema import IndexSchema

            schema = IndexSchema.from_dict({
                "index": {
                    "name": "docs-index",
                    "prefix": "docs",
                    "storage_type": "hash",
                },
                "fields": [
                    {
                        "name": "doc-id",
                        "type": "tag"
                    },
                    {
                        "name": "doc-embedding",
                        "type": "vector",
                        "attrs": {
                            "algorithm": "flat",
                            "dims": 1536
                        }
                    }
                ]
            })
        """
```

```
        return cls(**data)

    @property
    def field_names(self) -> List[str]:
        """A list of field names associated with the index schema.

        Returns:
            List[str]: A list of field names from the schema.
        """
        return list(self.fields.keys())

    @property
    def redis_fields(self) -> List[RedisField]:
        """A list of core redis-py field definitions based on the
        current schema fields.

        Converts RedisVL field definitions into a format suitable for use with
        redis-py, facilitating the creation and management of index structures in
        the Redis database.

        Returns:
            List[RedisField]: A list of redis-py field definitions.
        """
        redis_fields: List[RedisField] = [
            field.as_redis_field() for _, field in self.fields.items()
        ]
        return redis_fields

    def add_field(self, field_inputs: Dict[str, Any]):
        """Adds a single field to the index schema based on the specified field
        type and attributes.

        This method allows for the addition of individual fields to the schema,
        providing flexibility in defining the structure of the index.

        Args:
            field_inputs (Dict[str, Any]): A field to add.

        Raises:
            ValueError: If the field name or type are not provided or if the name
                already exists within the schema.

        .. code-block:: python

            # Add a tag field
            schema.add_field({"name": "user", "type": "tag})

            # Add a vector field
            schema.add_field({
                "name": "user-embedding",
                "type": "vector",
                "attrs": {
                    "dims": 1024,
                    "algorithm": "flat",
                    "datatype": "float32"
                }
            })
        """
        # Parse field inputs
        field = self._make_field(self.index.storage_type, **field_inputs)
        # Check for duplicates
        if field.name in self.fields:
            raise ValueError(
                f"Duplicate field name: {field.name}. Field names must be unique across
```

```
all fields for this index."
            )
        # Add field
        self.fields[field.name] = field

    def add_fields(self, fields: List[Dict[str, Any]]):
        """Extends the schema with additional fields.

        This method allows dynamically adding new fields to the index schema. It
        processes a list of field definitions.

        Args:
            fields (List[Dict[str, Any]]): A list of fields to add.

        Raises:
            ValueError: If a field with the same name already exists in the
                schema.

        .. code-block:: python

            schema.add_fields([
                {"name": "user", "type": "tag"},
                {"name": "bio", "type": "text"},
                {
                    "name": "user-embedding",
                    "type": "vector",
                    "attrs": {
                        "dims": 1024,
                        "algorithm": "flat",
                        "datatype": "float32"
                    }
                }
            ])
        """
        for field in fields:
            self.add_field(field)

    def remove_field(self, field_name: str):
        """Removes a field from the schema based on the specified name.

        This method is useful for dynamically altering the schema by removing
        existing fields.

        Args:
            field_name (str): The name of the field to be removed.
        """
        if field_name not in self.fields:
            logger.warning(f"Field '{field_name}' does not exist in the schema")
            return
        del self.fields[field_name]

    def generate_fields(
        self,
        data: Dict[str, Any],
        strict: bool = False,
        ignore_fields: List[str] = [],
    ) -> List[Dict[str, Any]]:
        """Generates a list of extracted field specs from a sample data point.

        This method simplifies the process of creating a schema by inferring
        field types and attributes from sample data. It's particularly useful
        during the development process while dealing with datasets containing
        numerous fields, reducing the need for manual specification.
```

```
        Args:
            data (Dict[str, Any]): Sample data used to infer field definitions.
            strict (bool, optional): If True, raises an error on failing to
                infer a field type. Defaults to False.
            ignore_fields (List[str], optional): A list of field names to
                exclude from processing. Defaults to an empty list.

        Returns:
            Dict[str, List[Dict[str, Any]]]: A dictionary with inferred field
                types and attributes.

        Notes:
            - Vector fields are not generated by this method.
            - This method employs heuristics and may not always correctly infer
                field types.
        """
        fields: List[Dict[str, Any]] = []
        for field_name, value in data.items():
            if field_name in ignore_fields:
                continue
            try:
                field_type = TypeInferrer.infer(value)
                fields.append(
                    FieldFactory.create_field(
                        field_type,
                        field_name,
                    ).dict()
                )
            except ValueError as e:
                if strict:
                    raise
                else:
                    logger.warn(
                        message=f"Error inferring field type for {field_name}: {e}"
                    )
        return fields

    def to_dict(self) -> Dict[str, Any]:
        """Serialize the index schema model to a dictionary, handling Enums
        and other special cases properly.

        Returns:
            Dict[str, Any]: The index schema as a dictionary.
        """
        dict_schema = model_to_dict(self)
        # cast fields back to a pure list
        dict_schema["fields"] = [
            field for field_name, field in dict_schema["fields"].items()
        ]
        return dict_schema

    def to_yaml(self, file_path: str, overwrite: bool = True) -> None:
        """Write the index schema to a YAML file.

        Args:
            file_path (str): The path to the YAML file.
            overwrite (bool): Whether to overwrite the file if it already exists.

        Raises:
            FileExistsError: If the file already exists and overwrite is False.
        """
        fp = Path(file_path).resolve()
        if fp.exists() and not overwrite:
            raise FileExistsError(f"Schema file {file_path} already exists.")
```

```
        with open(fp, "w") as f:
            yaml_data = self.to_dict()
            yaml.dump(yaml_data, f, sort_keys=False)


class TypeInferrer:
    """Infers the type of a field based on its value."""

    GEO_PATTERN = re.compile(
        r"^\s*[-+]?([1-8]?\d(\.\d+)?|90(\.0+)?),\s*[-+]?(180(\.0+)?|((1[0-7]\d)|([1-9]?
\d))(\.\d+)?)\s*$"
    )

    TYPE_METHOD_MAP = {
        "numeric": "_is_numeric",
        "geo": "_is_geographic",
        "tag": "_is_tag",
        "text": "_is_text",
    }

    @classmethod
    def infer(cls, value: Any) -> str:
        """Infers the field type for a given value.

        Args:
            value: The value to infer the type of.

        Returns:
            The inferred field type as a string.

        Raises:
            ValueError: If the type cannot be inferred.
        """
        for type_name, method_name in cls.TYPE_METHOD_MAP.items():
            if getattr(cls, method_name)(value):
                return type_name
        raise ValueError(f"Unable to infer type for value: {value}")

    @classmethod
    def _is_numeric(cls, value: Any) -> bool:
        """Check if the value is numeric."""
        if not isinstance(value, (int, float, str)):
            return False
        try:
            float(value)
            return True
        except (ValueError, TypeError):
            return False

    @classmethod
    def _is_tag(cls, value: Any) -> bool:
        """Check if the value is a tag."""
        return isinstance(value, (list, set, tuple)) and all(
            isinstance(v, str) for v in value
        )

    @classmethod
    def _is_text(cls, value: Any) -> bool:
        """Check if the value is text."""
        return isinstance(value, str)

    @classmethod
    def _is_geographic(cls, value: Any) -> bool:
```

```
        """Check if the value is a geographic coordinate."""
        return isinstance(value, str) and cls.GEO_PATTERN.match(value) is not None
}
```

Chapter 2.7.0

redisvl/utils

redisvl/utils/log.py

```
import logging
import sys

import coloredlogs

# constants for logging
coloredlogs.DEFAULT_DATE_FORMAT = "%H:%M:%S"
coloredlogs.DEFAULT_LOG_FORMAT = "%(asctime)s %(name)s %(levelname)s   %(message)s"


def get_logger(name, log_level="info", fmt=None):
    """Return a logger instance."""

    # Use file name if logger is in debug mode
    name = "RedisVL" if log_level == "debug" else name

    logger = logging.getLogger(name)
    coloredlogs.install(level=log_level, logger=logger, fmt=fmt, stream=sys.stdout)
    return logger
}
```

Chapter 2.7.1

redisvl/utils/rerank

redisvl/utils/rerank/__init__.py

```
from redisvl.utils.rerank.base import BaseReranker
from redisvl.utils.rerank.cohere import CohereReranker
from redisvl.utils.rerank.hf_cross_encoder import HFCrossEncoderReranker
```

```
__all__ = ["BaseReranker", "CohereReranker", "HFCrossEncoderReranker"]
}
```

redisvl/utils/rerank/base.py

```
from abc import ABC, abstractmethod
from typing import Any, Dict, List, Optional, Tuple, Union

from pydantic.v1 import BaseModel, validator


class BaseReranker(BaseModel, ABC):
    model: str
    rank_by: Optional[List[str]] = None
    limit: int
    return_score: bool

    @validator("limit")
    @classmethod
    def check_limit(cls, value):
        """Ensures the limit is a positive integer."""
        if value <= 0:
            raise ValueError("Limit must be a positive integer.")
        return value

    @validator("rank_by")
    @classmethod
    def check_rank_by(cls, value):
        """Ensures that rank_by is a list of strings if provided."""
        if value is not None and (
            not isinstance(value, list)
            or any(not isinstance(item, str) for item in value)
        ):
            raise ValueError("rank_by must be a list of strings.")
        return value

    @abstractmethod
    def rank(
        self, query: str, docs: Union[List[Dict[str, Any]], List[str]], **kwargs
    ) -> Union[Tuple[List[Dict[str, Any]], List[float]], List[Dict[str, Any]]]:
        """
        Synchronously rerank the docs based on the provided query.
        """
        pass

    @abstractmethod
    async def arank(
        self, query: str, docs: Union[List[Dict[str, Any]], List[str]], **kwargs
    ) -> Union[Tuple[List[Dict[str, Any]], List[float]], List[Dict[str, Any]]]:
        """
        Asynchronously rerank the docs based on the provided query.
        """
        pass
}
```

redisvl/utils/rerank/cohere.py

```
import os
from typing import Any, Dict, List, Optional, Tuple, Union

from pydantic.v1 import PrivateAttr

from redisvl.utils.rerank.base import BaseReranker


class CohereReranker(BaseReranker):
    """
    The CohereReranker class uses Cohere's API to rerank documents based on an
    input query.

    This reranker is designed to interact with Cohere's /rerank API,
    requiring an API key for authentication. The key can be provided
    directly in the `api_config` dictionary or through the `COHERE_API_KEY`
    environment variable. User must obtain an API key from Cohere's website
    (https://dashboard.cohere.com/). Additionally, the `cohere` python
    client must be installed with `pip install cohere`.

    .. code-block:: python

        from redisvl.utils.rerank import CohereReranker

        # set up the Cohere reranker with some configuration
        reranker = CohereReranker(rank_by=["content"], limit=2)
        # rerank raw search results based on user input/query
        results = reranker.rank(
            query="your input query text here",
            docs=[
                {"content": "document 1"},
                {"content": "document 2"},
                {"content": "document 3"}
            ]
        )
    """

    _client: Any = PrivateAttr()
    _aclient: Any = PrivateAttr()

    def __init__(
        self,
        model: str = "rerank-english-v3.0",
        rank_by: Optional[List[str]] = None,
        limit: int = 5,
        return_score: bool = True,
        api_config: Optional[Dict] = None,
    ) -> None:
        """
        Initialize the CohereReranker with specified model, ranking criteria,
        and API configuration.

        Parameters:
            model (str): The identifier for the Cohere model used for reranking.
                Defaults to 'rerank-english-v3.0'.
            rank_by (Optional[List[str]]): Optional list of keys specifying the
                attributes in the documents that should be considered for
                ranking. None means ranking will rely on the model's default
                behavior.
            limit (int): The maximum number of results to return after
                reranking. Must be a positive integer.
            return_score (bool): Whether to return scores alongside the
```

```
                reranked results.
            api_config (Optional[Dict], optional): Dictionary containing the API key.
                Defaults to None.

        Raises:
            ImportError: If the cohere library is not installed.
            ValueError: If the API key is not provided.
        """
        super().__init__(
            model=model, rank_by=rank_by, limit=limit, return_score=return_score
        )
        self._initialize_clients(api_config)

    def _initialize_clients(self, api_config: Optional[Dict]):
        """
        Setup the Cohere clients using the provided API key or an
        environment variable.
        """
        # Dynamic import of the cohere module
        try:
            from cohere import AsyncClient, Client
        except ImportError:
            raise ImportError(
                "Cohere reranker requires the cohere library. \
                    Please install with `pip install cohere`"
            )

        # Fetch the API key from api_config or environment variable
        api_key = (
            api_config.get("api_key") if api_config else os.getenv("COHERE_API_KEY")
        )
        if not api_key:
            raise ValueError(
                "Cohere API key is required. "
                "Provide it in api_config or set the COHERE_API_KEY
environment variable."
            )
        self._client = Client(api_key=api_key, client_name="redisvl")
        self._aclient = AsyncClient(api_key=api_key, client_name="redisvl")

    def _preprocess(
        self, query: str, docs: Union[List[Dict[str, Any]], List[str]], **kwargs
    ):
        """
        Prepare and validate reranking config based on provided input and
        optional overrides.
        """
        limit = kwargs.get("limit", self.limit)
        return_score = kwargs.get("return_score", self.return_score)
        max_chunks_per_doc = kwargs.get("max_chunks_per_doc")
        rank_by = kwargs.get("rank_by", self.rank_by) or []
        rank_by = [rank_by] if isinstance(rank_by, str) else rank_by

        reranker_kwargs = {
            "model": self.model,
            "query": query,
            "top_n": limit,
            "documents": docs,
            "max_chunks_per_doc": max_chunks_per_doc,
        }
        # if we are working with list of dicts
        if all(isinstance(doc, dict) for doc in docs):
            if rank_by:
                reranker_kwargs["rank_fields"] = rank_by
```

```
            else:
                raise ValueError(
                    "If reranking dictionary-like docs, "
                    "you must provide a list of rank_by fields"
                )

        return reranker_kwargs, return_score

    @staticmethod
    def _postprocess(
        docs: Union[List[Dict[str, Any]], List[str]],
        rankings: List[Any],
    ) -> Tuple[List[Any], List[float]]:
        """
        Post-process the initial list of documents to include ranking scores,
        if specified.
        """
        reranked_docs, scores = [], []
        for item in rankings.results:  # type: ignore
            scores.append(item.relevance_score)
            reranked_docs.append(docs[item.index])
        return reranked_docs, scores

    def rank(
        self, query: str, docs: Union[List[Dict[str, Any]], List[str]], **kwargs
    ) -> Union[Tuple[List[Dict[str, Any]], List[float]], List[Dict[str, Any]]]:
        """
        Rerank documents based on the provided query using the Cohere rerank API.

        This method processes the user's query and the provided documents to
        rerank them in a manner that is potentially more relevant to the
        query's context.

        Parameters:
            query (str): The user's search query.
            docs (Union[List[Dict[str, Any]], List[str]]): The list of documents
                to be ranked, either as dictionaries or strings.

        Returns:
            Union[Tuple[Union[List[Dict[str, Any]], List[str]], float], List[Dict[str,
Any]]]: The reranked list of documents and optionally associated scores.
        """
        reranker_kwargs, return_score = self._preprocess(query, docs, **kwargs)
        rankings = self._client.rerank(**reranker_kwargs)
        reranked_docs, scores = self._postprocess(docs, rankings)
        if return_score:
            return reranked_docs, scores
        return reranked_docs

    async def arank(
        self, query: str, docs: Union[List[Dict[str, Any]], List[str]], **kwargs
    ) -> Union[Tuple[List[Dict[str, Any]], List[float]], List[Dict[str, Any]]]:
        """
        Rerank documents based on the provided query using the Cohere rerank API.

        This method processes the user's query and the provided documents to
        rerank them in a manner that is potentially more relevant to the
        query's context.

        Parameters:
            query (str): The user's search query.
            docs (Union[List[Dict[str, Any]], List[str]]): The list of documents
                to be ranked, either as dictionaries or strings.
```

```
        Returns:
            Union[Tuple[Union[List[Dict[str, Any]], List[str]], float], List[Dict[str,
Any]]]: The reranked list of documents and optionally associated scores.
        """
        reranker_kwargs, return_score = self._preprocess(query, docs, **kwargs)
        rankings = await self._aclient.rerank(**reranker_kwargs)
        reranked_docs, scores = self._postprocess(docs, rankings)
        if return_score:
            return reranked_docs, scores
        return reranked_docs
}
```

redisvl/utils/rerank/hf_cross_encoder.py

```
from typing import Any, Dict, List, Optional, Tuple, Union

from pydantic.v1 import PrivateAttr

from redisvl.utils.rerank.base import BaseReranker


class HFCrossEncoderReranker(BaseReranker):
    """
    The HFCrossEncoderReranker class uses a cross-encoder models from Hugging Face
    to rerank documents based on an input query.

    This reranker loads a cross-encoder model using the `CrossEncoder` class
    from the `sentence_transformers` library. It requires the
    `sentence_transformers` library to be installed.

    .. code-block:: python

        from redisvl.utils.rerank import HFCrossEncoderReranker

        # set up the HFCrossEncoderReranker with a specific model
        reranker = HFCrossEncoderReranker(model_name="cross-encoder/ms-marco-MiniLM-L-
6-v2", limit=3)
        # rerank raw search results based on user input/query
        results = reranker.rank(
            query="your input query text here",
            docs=[
                {"content": "document 1"},
                {"content": "document 2"},
                {"content": "document 3"}
            ]
        )
    """

    _client: Any = PrivateAttr()

    def __init__(
        self,
        model: str = "cross-encoder/ms-marco-MiniLM-L-6-v2",
        limit: int = 3,
        return_score: bool = True,
        **kwargs,
    ) -> None:
        """
        Initialize the HFCrossEncoderReranker with a specified model and
ranking criteria.
```

```
        Parameters:
            model (str): The name or path of the cross-encoder model to use
for reranking.
                Defaults to 'cross-encoder/ms-marco-MiniLM-L-6-v2'.
            limit (int): The maximum number of results to return after reranking. Must
be a positive integer.
            return_score (bool): Whether to return scores alongside the
reranked results.
        """
        model = model or kwargs.pop("model_name", None)
        super().__init__(
            model=model, rank_by=None, limit=limit, return_score=return_score
        )
        self._initialize_client(**kwargs)

    def _initialize_client(self, **kwargs):
        """
        Setup the huggingface cross-encoder client using optional kwargs.
        """
        # Dynamic import of the sentence-transformers module
        try:
            from sentence_transformers import CrossEncoder
        except ImportError:
            raise ImportError(
                "HFCrossEncoder reranker requires the sentence-transformers library. \
                    Please install with `pip install sentence-transformers`"
            )

        self._client = CrossEncoder(self.model, **kwargs)

    def rank(
        self, query: str, docs: Union[List[Dict[str, Any]], List[str]], **kwargs
    ) -> Union[Tuple[List[Dict[str, Any]], List[float]], List[Dict[str, Any]]]:
        """
        Rerank documents based on the provided query using the loaded cross-
encoder model.

        This method processes the user's query and the provided documents to
rerank them
        in a manner that is potentially more relevant to the query's context.

        Parameters:
            query (str): The user's search query.
            docs (Union[List[Dict[str, Any]], List[str]]): The list of documents to
be ranked,
                either as dictionaries or strings.

        Returns:
            Union[Tuple[List[Dict[str, Any]], List[float]], List[Dict[str, Any]]]:
                The reranked list of documents and optionally associated scores.
        """
        limit = kwargs.get("limit", self.limit)
        return_score = kwargs.get("return_score", self.return_score)

        if not query:
            raise ValueError("query cannot be empty")

        if not isinstance(query, str):
            raise TypeError("query must be a string")

        if not isinstance(docs, list):
            raise TypeError("docs must be a list")
```

```
        if not docs:
            return [] if not return_score else ([], [])

        if all(isinstance(doc, dict) for doc in docs):
            texts = [
                str(doc["content"])
                for doc in docs
                if isinstance(doc, dict) and "content" in doc
            ]
            doc_subset = [
                doc for doc in docs if isinstance(doc, dict) and "content" in doc
            ]
        else:
            texts = [str(doc) for doc in docs]
            doc_subset = [{"content": doc} for doc in docs]

        scores = self._client.predict([(query, text) for text in texts])
        scores = [float(score) for score in scores]
        docs_with_scores = list(zip(doc_subset, scores))
        docs_with_scores.sort(key=lambda x: x[1], reverse=True)
        reranked_docs = [doc for doc, _ in docs_with_scores[:limit]]
        scores = scores[:limit]

        if return_score:
            return reranked_docs, scores
        return reranked_docs

    async def arank(
        self, query: str, docs: Union[List[Dict[str, Any]], List[str]], **kwargs
    ) -> Union[Tuple[List[Dict[str, Any]], List[float]], List[Dict[str, Any]]]:
        """
        Asynchronously rerank documents based on the provided query using the loaded
cross-encoder model.

        This method processes the user's query and the provided documents to
rerank them
        in a manner that is potentially more relevant to the query's context.

        Parameters:
            query (str): The user's search query.
            docs (Union[List[Dict[str, Any]], List[str]]): The list of documents to
be ranked,
                either as dictionaries or strings.

        Returns:
            Union[Tuple[List[Dict[str, Any]], List[float]], List[Dict[str, Any]]]:
                The reranked list of documents and optionally associated scores.
        """
        return self.rank(query, docs, **kwargs)
}
```

redisvl/utils/token_escaper.py

```
import re
from typing import Optional, Pattern


class TokenEscaper:
    """Escape punctuation within an input string.
```

```
    Adapted from RedisOM Python.
    """

    # Characters that RediSearch requires us to escape during queries.
    # Source: https://redis.io/docs/stack/search/reference/escaping/#the-rules-of-text-
field-tokenization
    DEFAULT_ESCAPED_CHARS = r"[,.<>{}\[\]\\\"\':;!@#$%^&*()\-+=~\/ ]"

    def __init__(self, escape_chars_re: Optional[Pattern] = None):
        if escape_chars_re:
            self.escaped_chars_re = escape_chars_re
        else:
            self.escaped_chars_re = re.compile(self.DEFAULT_ESCAPED_CHARS)

    def escape(self, value: str) -> str:
        if not isinstance(value, str):
            raise TypeError(
                f"Value must be a string object for token escaping, got
type {type(value)}"
            )

        def escape_symbol(match):
            value = match.group(0)
            return f"\\{value}"

        return self.escaped_chars_re.sub(escape_symbol, value)
}
```

redisvl/utils/utils.py

```
import json
from enum import Enum
from time import time
from typing import Any, Dict
from uuid import uuid4

from pydantic.v1 import BaseModel


def create_uuid() -> str:
    """Generate a unique indentifier to group related Redis documents."""
    return str(uuid4())


def current_timestamp() -> float:
    """Generate a unix epoch timestamp to assign to Redis documents."""
    return time()


def model_to_dict(model: BaseModel) -> Dict[str, Any]:
    """
    Custom serialization function that converts a Pydantic model to a dict,
    serializing Enum fields to their values, and handling nested models and lists.
    """

    def serialize_item(item):
        if isinstance(item, Enum):
            return item.value.lower()
        elif isinstance(item, dict):
            return {key: serialize_item(value) for key, value in item.items()}
```

```
        elif isinstance(item, list):
            return [serialize_item(element) for element in item]
        else:
            return item

    serialized_data = model.dict(exclude_none=True)
    for key, value in serialized_data.items():
        serialized_data[key] = serialize_item(value)
    return serialized_data


def validate_vector_dims(v1: int, v2: int) -> None:
    """Check the equality of vector dimensions."""
    if v1 != v2:
        raise ValueError(
            "Invalid vector dimensions! " f"Vector has dims defined as {v1}",
            f"Vector field has dims defined as {v2}",
            "Vector dims must be equal in order to perform similarity search.",
        )


def serialize(data: Dict[str, Any]) -> str:
    """Serlize the input into a string."""
    return json.dumps(data)


def deserialize(data: str) -> Dict[str, Any]:
    """Deserialize the input from a string."""
    return json.loads(data)
}
```

Chapter 2.7.2

redisvl/utils/vectorize

redisvl/utils/vectorize/__init__.py

```
from redisvl.utils.vectorize.base import BaseVectorizer, Vectorizers
from redisvl.utils.vectorize.text.azureopenai import AzureOpenAITextVectorizer
from redisvl.utils.vectorize.text.cohere import CohereTextVectorizer
from redisvl.utils.vectorize.text.custom import CustomTextVectorizer
from redisvl.utils.vectorize.text.huggingface import HFTextVectorizer
from redisvl.utils.vectorize.text.mistral import MistralAITextVectorizer
from redisvl.utils.vectorize.text.openai import OpenAITextVectorizer
from redisvl.utils.vectorize.text.vertexai import VertexAITextVectorizer

__all__ = [
    "BaseVectrorizer",
    "CohereTextVectorizer",
    "HFTextVectorizer",
    "OpenAITextVectorizer",
    "VertexAITextVectorizer",
    "AzureOpenAITextVectorizer",
    "MistralAITextVectorizer",
```

```
    "CustomTextVectorizer",
]


def vectorizer_from_dict(vectorizer: dict) -> BaseVectorizer:
    vectorizer_type = Vectorizers(vectorizer["type"])
    model = vectorizer["model"]
    if vectorizer_type == Vectorizers.cohere:
        return CohereTextVectorizer(model)
    elif vectorizer_type == Vectorizers.openai:
        return OpenAITextVectorizer(model)
    elif vectorizer_type == Vectorizers.azure_openai:
        return AzureOpenAITextVectorizer(model)
    elif vectorizer_type == Vectorizers.hf:
        return HFTextVectorizer(model)
    elif vectorizer_type == Vectorizers.mistral:
        return MistralAITextVectorizer(model)
    elif vectorizer_type == Vectorizers.vertexai:
        return VertexAITextVectorizer(model)
}
```

redisvl/utils/vectorize/base.py

```
from abc import ABC, abstractmethod
from enum import Enum
from typing import Callable, List, Optional

from pydantic.v1 import BaseModel, validator

from redisvl.redis.utils import array_to_buffer


class Vectorizers(Enum):
    azure_openai = "azure_openai"
    openai = "openai"
    cohere = "cohere"
    mistral = "mistral"
    vertexai = "vertexai"
    hf = "hf"


class BaseVectorizer(BaseModel, ABC):
    model: str
    dims: int

    @property
    def type(self) -> str:
        return "base"

    @validator("dims")
    @classmethod
    def check_dims(cls, value):
        """Ensures the dims are a positive integer."""
        if value <= 0:
            raise ValueError("Dims must be a positive integer.")
        return value

    @abstractmethod
    def embed_many(
        self,
```

```
        texts: List[str],
        preprocess: Optional[Callable] = None,
        batch_size: int = 1000,
        as_buffer: bool = False,
        **kwargs,
    ) -> List[List[float]]:
        raise NotImplementedError

    @abstractmethod
    def embed(
        self,
        text: str,
        preprocess: Optional[Callable] = None,
        as_buffer: bool = False,
        **kwargs,
    ) -> List[float]:
        raise NotImplementedError

    async def aembed_many(
        self,
        texts: List[str],
        preprocess: Optional[Callable] = None,
        batch_size: int = 1000,
        as_buffer: bool = False,
        **kwargs,
    ) -> List[List[float]]:
        # Fallback to standard embedding call if no async support
        return self.embed_many(texts, preprocess, batch_size, as_buffer, **kwargs)

    async def aembed(
        self,
        text: str,
        preprocess: Optional[Callable] = None,
        as_buffer: bool = False,
        **kwargs,
    ) -> List[float]:
        # Fallback to standard embedding call if no async support
        return self.embed(text, preprocess, as_buffer, **kwargs)

    def batchify(self, seq: list, size: int, preprocess: Optional[Callable] = None):
        for pos in range(0, len(seq), size):
            if preprocess is not None:
                yield [preprocess(chunk) for chunk in seq[pos : pos + size]]
            else:
                yield seq[pos : pos + size]

    def _process_embedding(self, embedding: List[float], as_buffer: bool, **kwargs):
        if as_buffer:
            if "dtype" not in kwargs:
                raise RuntimeError(
                    "dtype is required if converting from float to byte string."
                )
            return array_to_buffer(embedding, kwargs["dtype"])
        return embedding
}
```

Chapter 2.7.3

redisvl/utils/vectorize/text

redisvl/utils/vectorize/text/azureopenai.py

```
import os
from typing import Any, Callable, Dict, List, Optional

from pydantic.v1 import PrivateAttr
from tenacity import retry, stop_after_attempt, wait_random_exponential
from tenacity.retry import retry_if_not_exception_type

from redisvl.utils.vectorize.base import BaseVectorizer

# ignore that openai isn't imported
# mypy: disable-error-code="name-defined"


class AzureOpenAITextVectorizer(BaseVectorizer):
    """The AzureOpenAITextVectorizer class utilizes AzureOpenAI's API to generate
    embeddings for text data.

    This vectorizer is designed to interact with AzureOpenAI's embeddings API,
    requiring an API key, an AzureOpenAI deployment endpoint and API version.
    These values can be provided directly in the `api_config` dictionary with
    the parameters 'azure_endpoint', 'api_version' and 'api_key' or through the
    environment variables 'AZURE_OPENAI_ENDPOINT', 'OPENAI_API_VERSION',
and 'AZURE_OPENAI_API_KEY'.
    Users must obtain these values from the 'Keys and Endpoints' section in their Azure
OpenAI service.
    Additionally, the `openai` python client must be installed with `pip
install openai>=1.13.0`.

    The vectorizer supports both synchronous and asynchronous operations,
    allowing for batch processing of texts and flexibility in handling
    preprocessing tasks.

    .. code-block:: python

        # Synchronous embedding of a single text
        vectorizer = AzureOpenAITextVectorizer(
            model="text-embedding-ada-002",
            api_config={
                "api_key": "your_api_key", # OR set AZURE_OPENAI_API_KEY in your env
                "api_version": "your_api_version", # OR set OPENAI_API_VERSION in
your env
                "azure_endpoint": "your_azure_endpoint", # OR set AZURE_OPENAI_ENDPOINT
in your env
            }
        )
        embedding = vectorizer.embed("Hello, world!")

        # Asynchronous batch embedding of multiple texts
        embeddings = await vectorizer.aembed_many(
            ["Hello, world!", "How are you?"],
            batch_size=2
        )

    """
```

```
    _client: Any = PrivateAttr()
    _aclient: Any = PrivateAttr()

    def __init__(
        self, model: str = "text-embedding-ada-002", api_config: Optional[Dict] = None
    ):
        """Initialize the AzureOpenAI vectorizer.

        Args:
            model (str): Deployment to use for embedding. Must be the
                'Deployment name' not the 'Model name'. Defaults to
                'text-embedding-ada-002'.
            api_config (Optional[Dict], optional): Dictionary containing the
                API key, API version, Azure endpoint, and any other API options.
                Defaults to None.

        Raises:
            ImportError: If the openai library is not installed.
            ValueError: If the AzureOpenAI API key, version, or endpoint are
not provided.
        """
        self._initialize_clients(api_config)
        super().__init__(model=model, dims=self._set_model_dims(model))

    def _initialize_clients(self, api_config: Optional[Dict]):
        """
        Setup the OpenAI clients using the provided API key or an
        environment variable.
        """
        if api_config is None:
            api_config = {}

        # Dynamic import of the openai module
        try:
            from openai import AsyncAzureOpenAI, AzureOpenAI
        except ImportError:
            raise ImportError(
                "AzureOpenAI vectorizer requires the openai library. \
                    Please install with `pip install openai`"
            )

        # Fetch the API key, version and endpoint from api_config or
environment variable
        azure_endpoint = (
            api_config.pop("azure_endpoint")
            if api_config
            else os.getenv("AZURE_OPENAI_ENDPOINT")
        )

        if not azure_endpoint:
            raise ValueError(
                "AzureOpenAI API endpoint is required. "
                "Provide it in api_config or set the AZURE_OPENAI_ENDPOINT\
                    environment variable."
            )

        api_version = (
            api_config.pop("api_version")
            if api_config
            else os.getenv("OPENAI_API_VERSION")
        )

        if not api_version:
            raise ValueError(
```

```
                "AzureOpenAI API version is required. "
                "Provide it in api_config or set the OPENAI_API_VERSION\
                    environment variable."
            )

        api_key = (
            api_config.pop("api_key")
            if api_config
            else os.getenv("AZURE_OPENAI_API_KEY")
        )

        if not api_key:
            raise ValueError(
                "AzureOpenAI API key is required. "
                "Provide it in api_config or set the AZURE_OPENAI_API_KEY\
                    environment variable."
            )

        self._client = AzureOpenAI(
            api_key=api_key,
            api_version=api_version,
            azure_endpoint=azure_endpoint,
            **api_config,
        )
        self._aclient = AsyncAzureOpenAI(
            api_key=api_key,
            api_version=api_version,
            azure_endpoint=azure_endpoint,
            **api_config,
        )

    def _set_model_dims(self, model) -> int:
        try:
            embedding = (
                self._client.embeddings.create(input=["dimension test"], model=model)
                .data[0]
                .embedding
            )
        except (KeyError, IndexError) as ke:
            raise ValueError(f"Unexpected response from the AzureOpenAI
API: {str(ke)}")
        except Exception as e:  # pylint: disable=broad-except
            # fall back (TODO get more specific)
            raise ValueError(f"Error setting embedding model dimensions: {str(e)}")
        return len(embedding)

    @retry(
        wait=wait_random_exponential(min=1, max=60),
        stop=stop_after_attempt(6),
        retry=retry_if_not_exception_type(TypeError),
    )
    def embed_many(
        self,
        texts: List[str],
        preprocess: Optional[Callable] = None,
        batch_size: int = 10,
        as_buffer: bool = False,
        **kwargs,
    ) -> List[List[float]]:
        """Embed many chunks of texts using the AzureOpenAI API.

        Args:
            texts (List[str]): List of text chunks to embed.
            preprocess (Optional[Callable], optional): Optional preprocessing
```

```
                callable to perform before vectorization. Defaults to None.
            batch_size (int, optional): Batch size of texts to use when creating
                embeddings. Defaults to 10.
            as_buffer (bool, optional): Whether to convert the raw embedding
                to a byte string. Defaults to False.

        Returns:
            List[List[float]]: List of embeddings.

        Raises:
            TypeError: If the wrong input type is passed in for the test.
        """
        if not isinstance(texts, list):
            raise TypeError("Must pass in a list of str values to embed.")
        if len(texts) > 0 and not isinstance(texts[0], str):
            raise TypeError("Must pass in a list of str values to embed.")

        embeddings: List = []
        for batch in self.batchify(texts, batch_size, preprocess):
            response = self._client.embeddings.create(input=batch, model=self.model)
            embeddings += [
                self._process_embedding(r.embedding, as_buffer, **kwargs)
                for r in response.data
            ]
        return embeddings

    @retry(
        wait=wait_random_exponential(min=1, max=60),
        stop=stop_after_attempt(6),
        retry=retry_if_not_exception_type(TypeError),
    )
    def embed(
        self,
        text: str,
        preprocess: Optional[Callable] = None,
        as_buffer: bool = False,
        **kwargs,
    ) -> List[float]:
        """Embed a chunk of text using the AzureOpenAI API.

        Args:
            text (str): Chunk of text to embed.
            preprocess (Optional[Callable], optional): Optional preprocessing
callable to
                perform before vectorization. Defaults to None.
            as_buffer (bool, optional): Whether to convert the raw embedding
                to a byte string. Defaults to False.

        Returns:
            List[float]: Embedding.

        Raises:
            TypeError: If the wrong input type is passed in for the test.
        """
        if not isinstance(text, str):
            raise TypeError("Must pass in a str value to embed.")

        if preprocess:
            text = preprocess(text)
        result = self._client.embeddings.create(input=[text], model=self.model)
        return self._process_embedding(result.data[0].embedding, as_buffer, **kwargs)

    @retry(
        wait=wait_random_exponential(min=1, max=60),
```

```
        stop=stop_after_attempt(6),
        retry=retry_if_not_exception_type(TypeError),
    )
    async def aembed_many(
        self,
        texts: List[str],
        preprocess: Optional[Callable] = None,
        batch_size: int = 1000,
        as_buffer: bool = False,
        **kwargs,
    ) -> List[List[float]]:
        """Asynchronously embed many chunks of texts using the AzureOpenAI API.

        Args:
            texts (List[str]): List of text chunks to embed.
            preprocess (Optional[Callable], optional): Optional preprocessing
callable to
                perform before vectorization. Defaults to None.
            batch_size (int, optional): Batch size of texts to use when creating
                embeddings. Defaults to 10.
            as_buffer (bool, optional): Whether to convert the raw embedding
                to a byte string. Defaults to False.

        Returns:
            List[List[float]]: List of embeddings.

        Raises:
            TypeError: If the wrong input type is passed in for the test.
        """
        if not isinstance(texts, list):
            raise TypeError("Must pass in a list of str values to embed.")
        if len(texts) > 0 and not isinstance(texts[0], str):
            raise TypeError("Must pass in a list of str values to embed.")

        embeddings: List = []
        for batch in self.batchify(texts, batch_size, preprocess):
            response = await self._aclient.embeddings.create(
                input=batch, model=self.model
            )
            embeddings += [
                self._process_embedding(r.embedding, as_buffer, **kwargs)
                for r in response.data
            ]
        return embeddings

    @retry(
        wait=wait_random_exponential(min=1, max=60),
        stop=stop_after_attempt(6),
        retry=retry_if_not_exception_type(TypeError),
    )
    async def aembed(
        self,
        text: str,
        preprocess: Optional[Callable] = None,
        as_buffer: bool = False,
        **kwargs,
    ) -> List[float]:
        """Asynchronously embed a chunk of text using the OpenAI API.

        Args:
            text (str): Chunk of text to embed.
            preprocess (Optional[Callable], optional): Optional preprocessing
callable to
                perform before vectorization. Defaults to None.
```

```
            as_buffer (bool, optional): Whether to convert the raw embedding
                to a byte string. Defaults to False.

        Returns:
            List[float]: Embedding.

        Raises:
            TypeError: If the wrong input type is passed in for the test.
        """
        if not isinstance(text, str):
            raise TypeError("Must pass in a str value to embed.")

        if preprocess:
            text = preprocess(text)
        result = await self._aclient.embeddings.create(input=[text], model=self.model)
        return self._process_embedding(result.data[0].embedding, as_buffer, **kwargs)

    @property
    def type(self) -> str:
        return "azure_openai"
}
```

redisvl/utils/vectorize/text/cohere.py

```
import os
from typing import Any, Callable, Dict, List, Optional

from pydantic.v1 import PrivateAttr
from tenacity import retry, stop_after_attempt, wait_random_exponential
from tenacity.retry import retry_if_not_exception_type

from redisvl.utils.vectorize.base import BaseVectorizer

# ignore that cohere isn't imported
# mypy: disable-error-code="name-defined"


class CohereTextVectorizer(BaseVectorizer):
    """The CohereTextVectorizer class utilizes Cohere's API to generate
    embeddings for text data.

    This vectorizer is designed to interact with Cohere's /embed API,
    requiring an API key for authentication. The key can be provided
    directly in the `api_config` dictionary or through the `COHERE_API_KEY`
    environment variable. User must obtain an API key from Cohere's website
    (https://dashboard.cohere.com/). Additionally, the `cohere` python
    client must be installed with `pip install cohere`.

    The vectorizer supports only synchronous operations, allows for batch
    processing of texts and flexibility in handling preprocessing tasks.

    .. code-block:: python

        from redisvl.utils.vectorize import CohereTextVectorizer

        vectorizer = CohereTextVectorizer(
            model="embed-english-v3.0",
            api_config={"api_key": "your-cohere-api-key"} # OR set COHERE_API_KEY in
your env
        )
```

```
        query_embedding = vectorizer.embed(
            text="your input query text here",
            input_type="search_query"
        )
        doc_embeddings = cohere.embed_many(
            texts=["your document text", "more document text"],
            input_type="search_document"
        )

    """

    _client: Any = PrivateAttr()

    def __init__(
        self, model: str = "embed-english-v3.0", api_config: Optional[Dict] = None
    ):
        """Initialize the Cohere vectorizer.

        Visit https://cohere.ai/embed to learn about embeddings.

        Args:
            model (str): Model to use for embedding. Defaults to 'embed-english-v3.0'.
            api_config (Optional[Dict], optional): Dictionary containing the API key.
                Defaults to None.

        Raises:
            ImportError: If the cohere library is not installed.
            ValueError: If the API key is not provided.

        """
        self._initialize_client(api_config)
        super().__init__(model=model, dims=self._set_model_dims(model))

    def _initialize_client(self, api_config: Optional[Dict]):
        """
        Setup the Cohere clients using the provided API key or an
        environment variable.
        """
        # Dynamic import of the cohere module
        try:
            from cohere import AsyncClient, Client
        except ImportError:
            raise ImportError(
                "Cohere vectorizer requires the cohere library. \
                    Please install with `pip install cohere`"
            )

        # Fetch the API key from api_config or environment variable
        api_key = (
            api_config.get("api_key") if api_config else os.getenv("COHERE_API_KEY")
        )
        if not api_key:
            raise ValueError(
                "Cohere API key is required. "
                "Provide it in api_config or set the COHERE_API_KEY
environment variable."
            )
        self._client = Client(api_key=api_key, client_name="redisvl")

    def _set_model_dims(self, model) -> int:
        try:
            embedding = self._client.embed(
                texts=["dimension test"],
                model=model,
```

```
                input_type="search_document",
            ).embeddings[0]
        except (KeyError, IndexError) as ke:
            raise ValueError(f"Unexpected response from the Cohere API: {str(ke)}")
        except Exception as e:  # pylint: disable=broad-except
            # fall back (TODO get more specific)
            raise ValueError(f"Error setting embedding model dimensions: {str(e)}")
        return len(embedding)

    def embed(
        self,
        text: str,
        preprocess: Optional[Callable] = None,
        as_buffer: bool = False,
        **kwargs,
    ) -> List[float]:
        """Embed a chunk of text using the Cohere Embeddings API.

        Must provide the embedding `input_type` as a `kwarg` to this method
        that specifies the type of input you're giving to the model.

        Supported input types:
            - ``search_document``: Used for embeddings stored in a vector database for
search use-cases.
            - ``search_query``: Used for embeddings of search queries run against a
vector DB to find relevant documents.
            - ``classification``: Used for embeddings passed through a text classifier
            - ``clustering``: Used for the embeddings run through a
clustering algorithm.

        When hydrating your Redis DB, the documents you want to search over
        should be embedded with input_type= "search_document" and when you are
        querying the database, you should set the input_type = "search query".
        If you want to use the embeddings for a classification or clustering
        task downstream, you should set input_type= "classification" or
        "clustering".

        Args:
            text (str): Chunk of text to embed.
            preprocess (Optional[Callable], optional): Optional preprocessing
callable to
                perform before vectorization. Defaults to None.
            as_buffer (bool, optional): Whether to convert the raw embedding
                to a byte string. Defaults to False.
            input_type (str): Specifies the type of input passed to the model.
                Required for embedding models v3 and higher.

        Returns:
            List[float]: Embedding.

        Raises:
            TypeError: In an invalid input_type is provided.

        """
        input_type = kwargs.get("input_type")

        if not isinstance(text, str):
            raise TypeError("Must pass in a str value to embed.")
        if not isinstance(input_type, str):
            raise TypeError(
                "Must pass in a str value for cohere embedding input_type. \
                    See https://docs.cohere.com/reference/embed."
            )
        if preprocess:
```

```
            text = preprocess(text)
        embedding = self._client.embed(
            texts=[text], model=self.model, input_type=input_type
        ).embeddings[0]
        return self._process_embedding(embedding, as_buffer, **kwargs)

    @retry(
        wait=wait_random_exponential(min=1, max=60),
        stop=stop_after_attempt(6),
        retry=retry_if_not_exception_type(TypeError),
    )
    def embed_many(
        self,
        texts: List[str],
        preprocess: Optional[Callable] = None,
        batch_size: int = 10,
        as_buffer: bool = False,
        **kwargs,
    ) -> List[List[float]]:
        """Embed many chunks of text using the Cohere Embeddings API.

        Must provide the embedding `input_type` as a `kwarg` to this method
        that specifies the type of input you're giving to the model.

        Supported input types:
            - ``search_document``: Used for embeddings stored in a vector database for
search use-cases.
            - ``search_query``: Used for embeddings of search queries run against a
vector DB to find relevant documents.
            - ``classification``: Used for embeddings passed through a text classifier
            - ``clustering``: Used for the embeddings run through a
clustering algorithm.


        When hydrating your Redis DB, the documents you want to search over
        should be embedded with input_type= "search_document" and when you are
        querying the database, you should set the input_type = "search query".
        If you want to use the embeddings for a classification or clustering
        task downstream, you should set input_type= "classification" or
        "clustering".

        Args:
            texts (List[str]): List of text chunks to embed.
            preprocess (Optional[Callable], optional): Optional preprocessing
callable to
                perform before vectorization. Defaults to None.
            batch_size (int, optional): Batch size of texts to use when creating
                embeddings. Defaults to 10.
            as_buffer (bool, optional): Whether to convert the raw embedding
                to a byte string. Defaults to False.
            input_type (str): Specifies the type of input passed to the model.
                Required for embedding models v3 and higher.

        Returns:
            List[List[float]]: List of embeddings.

        Raises:
            TypeError: In an invalid input_type is provided.

        """
        input_type = kwargs.get("input_type")

        if not isinstance(texts, list):
            raise TypeError("Must pass in a list of str values to embed.")
```

```
        if len(texts) > 0 and not isinstance(texts[0], str):
            raise TypeError("Must pass in a list of str values to embed.")
        if not isinstance(input_type, str):
            raise TypeError(
                "Must pass in a str value for cohere embedding input_type.\
                    See https://docs.cohere.com/reference/embed."
            )

        embeddings: List = []
        for batch in self.batchify(texts, batch_size, preprocess):
            response = self._client.embed(
                texts=batch, model=self.model, input_type=input_type
            )
            embeddings += [
                self._process_embedding(embedding, as_buffer, **kwargs)
                for embedding in response.embeddings
            ]
        return embeddings

    @property
    def type(self) -> str:
        return "cohere"
}
```

redisvl/utils/vectorize/text/custom.py

```
import os
from typing import Any, Callable, Dict, List, Optional

from pydantic.v1 import PrivateAttr

from redisvl.utils.vectorize.base import BaseVectorizer


class CustomTextVectorizer(BaseVectorizer):
    """The CustomTextVectorizer class wraps user-defined embeding methods to create
    embeddings for text data.

    This vectorizer is designed to accept a provided callable text vectorizer and
    provides a class definition to allow for compatibility with RedisVL.

    The vectorizer may support both synchronous and asynchronous operations which
    allows for batch processing of texts, but at a minimum only syncronous embedding
    is required to satisfy the 'embed()' method.

    .. code-block:: python

        # Synchronous embedding of a single text
        vectorizer = CustomTextVectorizer(
            embed = my_vectorizer.generate_embedding
        )
        embedding = vectorizer.embed("Hello, world!")

        # Asynchronous batch embedding of multiple texts
        embeddings = await vectorizer.aembed_many(
            ["Hello, world!", "How are you?"],
            batch_size=2
        )

    """
```

```
    _embed_func: Callable = PrivateAttr()
    _embed_many_func: Optional[Callable] = PrivateAttr()
    _aembed_func: Optional[Callable] = PrivateAttr()
    _aembed_many_func: Optional[Callable] = PrivateAttr()

    def __init__(
        self,
        embed: Callable,
        embed_many: Optional[Callable] = None,
        aembed: Optional[Callable] = None,
        aembed_many: Optional[Callable] = None,
    ):
        """Initialize the Custom vectorizer.

        Args:
            embed (Callable): a Callable function that accepts a string object and
returns a list of floats.
            embed_many (Optional[Callable)]: a Callable function that accepts a list of
string objects and returns a list containing lists of floats. Defaults to None.
            aembed (Optional[Callable]): an asyncronous Callable function that accepts
a string object and returns a lists of floats. Defaults to None.
            aembed_many (Optional[Callable]):  an asyncronous Callable function that
accepts a list of string objects and returns a list containing lists of floats.
Defaults to None.

        Raises:
            ValueError if any of the provided functions accept or return
incorrect types.
            TypeError if any of the provided functions are not Callable objects.
        """

        self._validate_embed(embed)
        self._embed_func = embed
        if embed_many:
            self._validate_embed_many(embed_many)
            self._embed_many_func = embed_many

        if aembed:
            self._validate_aembed(aembed)
            self._aembed_func = aembed
        if aembed_many:
            self._validate_aembed_many(aembed_many)
            self._aembed_many_func = aembed_many

        super().__init__(model=self.type, dims=self._set_model_dims())

    def _validate_embed(self, func: Callable):
        """calls the func with dummy input and validates that it returns a vector"""
        try:
            test_str = "this is a test sentence"
            candidate_vector = func(test_str)
            if type(candidate_vector) != list or type(candidate_vector[0]) != float:
                raise ValueError(
                    f"Candidate function for embed() does not have the correct return
type. Please provide a function with with return type List[float]"
                )
        except TypeError:
            raise TypeError(f"{func} is not a callable object")

    def _validate_embed_many(self, func: Callable):
        """calls the func with dummy input and validates that it returns a list
of vectors"""
        try:
```

```
            test_strs = ["first test sentence", "second test sentence"]
            candidate_vectors = func(test_strs)
            if (
                type(candidate_vectors) != list
                or type(candidate_vectors[0]) != list
                or type(candidate_vectors[0][0]) != float
            ):
                raise ValueError(
                    f"Candidate function for embed_many does not have the correct
return type. Please provide a function with with return type List[List[float]]"
                )
        except TypeError:
            raise TypeError(f"{func} is not a callable object")

    def _validate_aembed(self, func: Callable):
        """calls the func with dummy input and validates that it returns a vector"""
        import asyncio

        try:
            test_str = "this is a test sentence"
            loop = asyncio.get_event_loop()
            candidate_vector = loop.run_until_complete(func(test_str))
            if type(candidate_vector) != list or type(candidate_vector[0]) != float:
                raise ValueError(
                    f"Candidate function for aembed() does not have the correct return
type. Please provide a function with with return type List[float]"
                )
        except TypeError:
            raise TypeError(f"{func} is not a callable object")

    def _validate_aembed_many(self, func: Callable):
        """calls the func with dummy input and validates that it returns a list
of vectors"""
        import asyncio

        try:
            test_strs = ["first test sentence", "second test sentence"]
            loop = asyncio.get_event_loop()
            candidate_vectors = loop.run_until_complete(func(test_strs))
            if (
                type(candidate_vectors) != list
                or type(candidate_vectors[0]) != list
                or type(candidate_vectors[0][0]) != float
            ):
                raise ValueError(
                    f"Candidate function for aembed_many does not have the correct
return type. Please provide a function with with return type List[List[float]]"
                )
        except TypeError:
            raise TypeError(f"{func} is not a callable object")

    def _set_model_dims(self) -> int:
        try:
            test_string = "dimension test"
            embedding = self._embed_func(test_string)
        except Exception as e:  # pylint: disable=broad-except
            raise ValueError(
                f"Error in checking model dimensions. Attempted to embed
'{test_string}'. :{str(e)}"
            )
        return len(embedding)

    def embed(
        self,
```

```
        text: str,
        preprocess: Optional[Callable] = None,
        as_buffer: bool = False,
        **kwargs,
    ) -> List[float]:
        """Embed a chunk of text using the provided function.

        Args:
            text (str): Chunk of text to embed.
            preprocess (Optional[Callable], optional): Optional preprocessing
callable to
                perform before vectorization. Defaults to None.
            as_buffer (bool, optional): Whether to convert the raw embedding
                to a byte string. Defaults to False.

        Returns:
            List[float]: Embedding.

        Raises:
            TypeError: If the wrong input type is passed in for the text.
        """
        if not isinstance(text, str):
            raise TypeError("Must pass in a str value to embed.")

        if preprocess:
            text = preprocess(text)
        else:
            result = self._embed_func(text, **kwargs)
        return self._process_embedding(result, as_buffer, **kwargs)

    def embed_many(
        self,
        texts: List[str],
        preprocess: Optional[Callable] = None,
        batch_size: int = 10,
        as_buffer: bool = False,
        **kwargs,
    ) -> List[List[float]]:
        """Embed many chunks of texts using the provided function.

        Args:
            texts (List[str]): List of text chunks to embed.
            preprocess (Optional[Callable], optional): Optional preprocessing
                callable to perform before vectorization. Defaults to None.
            batch_size (int, optional): Batch size of texts to use when creating
                embeddings. Defaults to 10.
            as_buffer (bool, optional): Whether to convert the raw embedding
                to a byte string. Defaults to False.

        Returns:
            List[List[float]]: List of embeddings.

        Raises:
            TypeError: If the wrong input type is passed in for the text.
            NotImplementedError: if embed_many was not passed to constructor.
        """
        if not isinstance(texts, list):
            raise TypeError("Must pass in a list of str values to embed.")
        if len(texts) > 0 and not isinstance(texts[0], str):
            raise TypeError("Must pass in a list of str values to embed.")

        if not self._embed_many_func:
            raise NotImplementedError
```

```
        embeddings: List = []
        for batch in self.batchify(texts, batch_size, preprocess):
            results = self._embed_many_func(batch, **kwargs)
            embeddings += [
                self._process_embedding(r, as_buffer, **kwargs) for r in results
            ]
        return embeddings

    async def aembed(
        self,
        text: str,
        preprocess: Optional[Callable] = None,
        as_buffer: bool = False,
        **kwargs,
    ) -> List[float]:
        """Asynchronously embed a chunk of text.

        Args:
            text (str): Chunk of text to embed.
            preprocess (Optional[Callable], optional): Optional preprocessing
callable to
                perform before vectorization. Defaults to None.
            as_buffer (bool, optional): Whether to convert the raw embedding
                to a byte string. Defaults to False.

        Returns:
            List[float]: Embedding.

        Raises:
            TypeError: If the wrong input type is passed in for the text.
            NotImplementedError: if aembed was not passed to constructor.
        """
        if not isinstance(text, str):
            raise TypeError("Must pass in a str value to embed.")

        if not self._aembed_func:
            raise NotImplementedError

        if preprocess:
            text = preprocess(text)
        else:
            result = await self._aembed_func(text, **kwargs)
        return self._process_embedding(result, as_buffer, **kwargs)

    async def aembed_many(
        self,
        texts: List[str],
        preprocess: Optional[Callable] = None,
        batch_size: int = 1000,
        as_buffer: bool = False,
        **kwargs,
    ) -> List[List[float]]:
        """Asynchronously embed many chunks of texts.

        Args:
            texts (List[str]): List of text chunks to embed.
            preprocess (Optional[Callable], optional): Optional preprocessing
callable to
                perform before vectorization. Defaults to None.
            batch_size (int, optional): Batch size of texts to use when creating
                embeddings. Defaults to 10.
            as_buffer (bool, optional): Whether to convert the raw embedding
                to a byte string. Defaults to False.
```

```
        Returns:
            List[List[float]]: List of embeddings.

        Raises:
            TypeError: If the wrong input type is passed in for the text.
            NotImplementedError: If aembed_many was not passed to constructor.
        """
        if not isinstance(texts, list):
            raise TypeError("Must pass in a list of str values to embed.")
        if len(texts) > 0 and not isinstance(texts[0], str):
            raise TypeError("Must pass in a list of str values to embed.")

        if not self._aembed_many_func:
            raise NotImplementedError

        embeddings: List = []
        for batch in self.batchify(texts, batch_size, preprocess):
            results = await self._aembed_many_func(batch, **kwargs)
            embeddings += [
                self._process_embedding(r, as_buffer, **kwargs) for r in results
            ]
        return embeddings

    @property
    def type(self) -> str:
        return "custom"
}
```

redisvl/utils/vectorize/text/huggingface.py

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

from pydantic.v1 import PrivateAttr

from redisvl.utils.vectorize.base import BaseVectorizer


class HFTextVectorizer(BaseVectorizer):
    """The HFTextVectorizer class is designed to leverage the power of Hugging
    Face's Sentence Transformers for generating text embeddings. This vectorizer
    is particularly useful in scenarios where advanced natural language
    processing and understanding are required, and ideal for running on your own
    hardware (for free).

    Utilizing this vectorizer involves specifying a pre-trained model from
    Hugging Face's vast collection of Sentence Transformers. These models are
    trained on a variety of datasets and tasks, ensuring versatility and
    robust performance across different text embedding needs. Additionally,
    make sure the `sentence-transformers` library is installed with
    `pip install sentence-transformers==2.2.2`.

    .. code-block:: python

        # Embedding a single text
        vectorizer = HFTextVectorizer(model="sentence-transformers/all-mpnet-base-v2")
        embedding = vectorizer.embed("Hello, world!")

        # Embedding a batch of texts
        embeddings = vectorizer.embed_many(["Hello, world!", "How are
you?"], batch_size=2)
```

```
    """

    _client: Any = PrivateAttr()

    def __init__(
        self, model: str = "sentence-transformers/all-mpnet-base-v2", **kwargs
    ):
        """Initialize the Hugging Face text vectorizer.

        Args:
            model (str): The pre-trained model from Hugging Face's Sentence
                Transformers to be used for embedding. Defaults to
                'sentence-transformers/all-mpnet-base-v2'.

        Raises:
            ImportError: If the sentence-transformers library is not installed.
            ValueError: If there is an error setting the embedding model dimensions.
        """
        self._initialize_client(model)
        super().__init__(model=model, dims=self._set_model_dims())

    def _initialize_client(self, model: str):
        """Setup the HuggingFace client"""
        # Dynamic import of the cohere module\
        try:
            from sentence_transformers import SentenceTransformer
        except ImportError:
            raise ImportError(
                "HFTextVectorizer requires the sentence-transformers library. "
                "Please install with `pip install sentence-transformers`"
            )

        self._client = SentenceTransformer(model)

    def _set_model_dims(self):
        try:
            embedding = self._client.encode(["dimension check"])[0]
        except (KeyError, IndexError) as ke:
            raise ValueError(f"Empty response from the embedding model: {str(ke)}")
        except Exception as e:  # pylint: disable=broad-except
            # fall back (TODO get more specific)
            raise ValueError(f"Error setting embedding model dimensions: {str(e)}")
        return len(embedding)

    def embed(
        self,
        text: str,
        preprocess: Optional[Callable] = None,
        as_buffer: bool = False,
        **kwargs,
    ) -> List[float]:
        """Embed a chunk of text using the Hugging Face sentence transformer.

        Args:
            text (str): Chunk of text to embed.
            preprocess (Optional[Callable], optional): Optional preprocessing
                callable to perform before vectorization. Defaults to None.
            as_buffer (bool, optional): Whether to convert the raw embedding
                to a byte string. Defaults to False.

        Returns:
            List[float]: Embedding.
```

```
        Raises:
            TypeError: If the wrong input type is passed in for the text.
        """
        if not isinstance(text, str):
            raise TypeError("Must pass in a str value to embed.")

        if preprocess:
            text = preprocess(text)
        embedding = self._client.encode([text])[0]
        return self._process_embedding(embedding.tolist(), as_buffer, **kwargs)

    def embed_many(
        self,
        texts: List[str],
        preprocess: Optional[Callable] = None,
        batch_size: int = 1000,
        as_buffer: bool = False,
        **kwargs,
    ) -> List[List[float]]:
        """Asynchronously embed many chunks of texts using the Hugging Face
        sentence transformer.

        Args:
            texts (List[str]): List of text chunks to embed.
            preprocess (Optional[Callable], optional): Optional preprocessing
                callable to perform before vectorization. Defaults to None.
            batch_size (int, optional): Batch size of texts to use when creating
                embeddings. Defaults to 10.
            as_buffer (bool, optional): Whether to convert the raw embedding
                to a byte string. Defaults to False.

        Returns:
            List[List[float]]: List of embeddings.

        Raises:
            TypeError: If the wrong input type is passed in for the test.
        """
        if not isinstance(texts, list):
            raise TypeError("Must pass in a list of str values to embed.")
        if len(texts) > 0 and not isinstance(texts[0], str):
            raise TypeError("Must pass in a list of str values to embed.")

        embeddings: List = []
        for batch in self.batchify(texts, batch_size, preprocess):
            batch_embeddings = self._client.encode(batch)
            embeddings.extend(
                [
                    self._process_embedding(embedding.tolist(), as_buffer, **kwargs)
                    for embedding in batch_embeddings
                ]
            )
        return embeddings

    @property
    def type(self) -> str:
        return "hf"
}
```

redisvl/utils/vectorize/text/mistral.py

```
import os
from typing import Any, Callable, Dict, List, Optional

from pydantic.v1 import PrivateAttr
from tenacity import retry, stop_after_attempt, wait_random_exponential
from tenacity.retry import retry_if_not_exception_type

from redisvl.utils.vectorize.base import BaseVectorizer

# ignore that mistralai isn't imported
# mypy: disable-error-code="name-defined"


class MistralAITextVectorizer(BaseVectorizer):
    """The MistralAITextVectorizer class utilizes MistralAI's API to generate
    embeddings for text data.

    This vectorizer is designed to interact with Mistral's embeddings API,
    requiring an API key for authentication. The key can be provided directly
    in the `api_config` dictionary or through the `MISTRAL_API_KEY` environment
    variable. Users must obtain an API key from Mistral's website
    (https://console.mistral.ai/). Additionally, the `mistralai` python client
    must be installed with `pip install mistralai`.

    The vectorizer supports both synchronous and asynchronous operations,
    allowing for batch processing of texts and flexibility in handling
    preprocessing tasks.

    .. code-block:: python

        # Synchronous embedding of a single text
        vectorizer = MistralAITextVectorizer(
            model="mistral-embed"
            api_config={"api_key": "your_api_key"} # OR set MISTRAL_API_KEY in your env
        )
        embedding = vectorizer.embed("Hello, world!")

        # Asynchronous batch embedding of multiple texts
        embeddings = await vectorizer.aembed_many(
            ["Hello, world!", "How are you?"],
            batch_size=2
        )

    """

    _client: Any = PrivateAttr()
    _aclient: Any = PrivateAttr()

    def __init__(self, model: str = "mistral-embed", api_config: Optional[Dict]
= None):
        """Initialize the MistralAI vectorizer.

        Args:
            model (str): Model to use for embedding. Defaults to
                'text-embedding-ada-002'.
            api_config (Optional[Dict], optional): Dictionary containing the
                API key. Defaults to None.

        Raises:
            ImportError: If the mistralai library is not installed.
            ValueError: If the Mistral API key is not provided.
        """
        self._initialize_clients(api_config)
        super().__init__(model=model, dims=self._set_model_dims(model))
```

```
    def _initialize_clients(self, api_config: Optional[Dict]):
        """
        Setup the Mistral clients using the provided API key or an
        environment variable.
        """
        # Dynamic import of the mistralai module
        try:
            from mistralai.async_client import MistralAsyncClient
            from mistralai.client import MistralClient
        except ImportError:
            raise ImportError(
                "MistralAI vectorizer requires the mistralai library. \
                    Please install with `pip install mistralai`"
            )

        # Fetch the API key from api_config or environment variable
        api_key = (
            api_config.get("api_key") if api_config else os.getenv("MISTRAL_API_KEY")
        )
        if not api_key:
            raise ValueError(
                "MISTRAL API key is required. "
                "Provide it in api_config or set the MISTRAL_API_KEY\
                    environment variable."
            )

        self._client = MistralClient(api_key=api_key)
        self._aclient = MistralAsyncClient(api_key=api_key)

    def _set_model_dims(self, model) -> int:
        try:
            embedding = (
                self._client.embeddings(model=model, input=["dimension test"])
                .data[0]
                .embedding
            )
        except (KeyError, IndexError) as ke:
            raise ValueError(f"Unexpected response from the MISTRAL API: {str(ke)}")
        except Exception as e:  # pylint: disable=broad-except
            # fall back (TODO get more specific)
            raise ValueError(f"Error setting embedding model dimensions: {str(e)}")
        return len(embedding)

    @retry(
        wait=wait_random_exponential(min=1, max=60),
        stop=stop_after_attempt(6),
        retry=retry_if_not_exception_type(TypeError),
    )
    def embed_many(
        self,
        texts: List[str],
        preprocess: Optional[Callable] = None,
        batch_size: int = 10,
        as_buffer: bool = False,
        **kwargs,
    ) -> List[List[float]]:
        """Embed many chunks of texts using the Mistral API.

        Args:
            texts (List[str]): List of text chunks to embed.
            preprocess (Optional[Callable], optional): Optional preprocessing
                callable to perform before vectorization. Defaults to None.
            batch_size (int, optional): Batch size of texts to use when creating
```

```
                embeddings. Defaults to 10.
            as_buffer (bool, optional): Whether to convert the raw embedding
                to a byte string. Defaults to False.

        Returns:
            List[List[float]]: List of embeddings.

        Raises:
            TypeError: If the wrong input type is passed in for the test.
        """
        if not isinstance(texts, list):
            raise TypeError("Must pass in a list of str values to embed.")
        if len(texts) > 0 and not isinstance(texts[0], str):
            raise TypeError("Must pass in a list of str values to embed.")

        embeddings: List = []
        for batch in self.batchify(texts, batch_size, preprocess):
            response = self._client.embeddings(model=self.model, input=batch)
            embeddings += [
                self._process_embedding(r.embedding, as_buffer, **kwargs)
                for r in response.data
            ]
        return embeddings

    @retry(
        wait=wait_random_exponential(min=1, max=60),
        stop=stop_after_attempt(6),
        retry=retry_if_not_exception_type(TypeError),
    )
    def embed(
        self,
        text: str,
        preprocess: Optional[Callable] = None,
        as_buffer: bool = False,
        **kwargs,
    ) -> List[float]:
        """Embed a chunk of text using the Mistral API.

        Args:
            text (str): Chunk of text to embed.
            preprocess (Optional[Callable], optional): Optional preprocessing
callable to
                perform before vectorization. Defaults to None.
            as_buffer (bool, optional): Whether to convert the raw embedding
                to a byte string. Defaults to False.

        Returns:
            List[float]: Embedding.

        Raises:
            TypeError: If the wrong input type is passed in for the test.
        """
        if not isinstance(text, str):
            raise TypeError("Must pass in a str value to embed.")

        if preprocess:
            text = preprocess(text)
        result = self._client.embeddings(model=self.model, input=[text])
        return self._process_embedding(result.data[0].embedding, as_buffer, **kwargs)

    @retry(
        wait=wait_random_exponential(min=1, max=60),
        stop=stop_after_attempt(6),
        retry=retry_if_not_exception_type(TypeError),
```

```
    )
    async def aembed_many(
        self,
        texts: List[str],
        preprocess: Optional[Callable] = None,
        batch_size: int = 1000,
        as_buffer: bool = False,
        **kwargs,
    ) -> List[List[float]]:
        """Asynchronously embed many chunks of texts using the Mistral API.

        Args:
            texts (List[str]): List of text chunks to embed.
            preprocess (Optional[Callable], optional): Optional preprocessing
callable to
                perform before vectorization. Defaults to None.
            batch_size (int, optional): Batch size of texts to use when creating
                embeddings. Defaults to 10.
            as_buffer (bool, optional): Whether to convert the raw embedding
                to a byte string. Defaults to False.

        Returns:
            List[List[float]]: List of embeddings.

        Raises:
            TypeError: If the wrong input type is passed in for the test.
        """
        if not isinstance(texts, list):
            raise TypeError("Must pass in a list of str values to embed.")
        if len(texts) > 0 and not isinstance(texts[0], str):
            raise TypeError("Must pass in a list of str values to embed.")

        embeddings: List = []
        for batch in self.batchify(texts, batch_size, preprocess):
            response = await self._aclient.embeddings(model=self.model, input=batch)
            embeddings += [
                self._process_embedding(r.embedding, as_buffer, **kwargs)
                for r in response.data
            ]
        return embeddings

    @retry(
        wait=wait_random_exponential(min=1, max=60),
        stop=stop_after_attempt(6),
        retry=retry_if_not_exception_type(TypeError),
    )
    async def aembed(
        self,
        text: str,
        preprocess: Optional[Callable] = None,
        as_buffer: bool = False,
        **kwargs,
    ) -> List[float]:
        """Asynchronously embed a chunk of text using the MistralAPI.

        Args:
            text (str): Chunk of text to embed.
            preprocess (Optional[Callable], optional): Optional preprocessing
callable to
                perform before vectorization. Defaults to None.
            as_buffer (bool, optional): Whether to convert the raw embedding
                to a byte string. Defaults to False.

        Returns:
```

```
            List[float]: Embedding.

        Raises:
            TypeError: If the wrong input type is passed in for the test.
        """
        if not isinstance(text, str):
            raise TypeError("Must pass in a str value to embed.")

        if preprocess:
            text = preprocess(text)
        result = await self._aclient.embeddings(model=self.model, input=[text])
        return self._process_embedding(result.data[0].embedding, as_buffer, **kwargs)

    @property
    def type(self) -> str:
        return "mistral"
}
```

redisvl/utils/vectorize/text/openai.py

```
import os
from typing import Any, Callable, Dict, List, Optional

from pydantic.v1 import PrivateAttr
from tenacity import retry, stop_after_attempt, wait_random_exponential
from tenacity.retry import retry_if_not_exception_type

from redisvl.utils.vectorize.base import BaseVectorizer

# ignore that openai isn't imported
# mypy: disable-error-code="name-defined"


class OpenAITextVectorizer(BaseVectorizer):
    """The OpenAITextVectorizer class utilizes OpenAI's API to generate
    embeddings for text data.

    This vectorizer is designed to interact with OpenAI's embeddings API,
    requiring an API key for authentication. The key can be provided directly
    in the `api_config` dictionary or through the `OPENAI_API_KEY` environment
    variable. Users must obtain an API key from OpenAI's website
    (https://api.openai.com/). Additionally, the `openai` python client must be
    installed with `pip install openai>=1.13.0`.

    The vectorizer supports both synchronous and asynchronous operations,
    allowing for batch processing of texts and flexibility in handling
    preprocessing tasks.

    .. code-block:: python

        # Synchronous embedding of a single text
        vectorizer = OpenAITextVectorizer(
            model="text-embedding-ada-002",
            api_config={"api_key": "your_api_key"} # OR set OPENAI_API_KEY in your env
        )
        embedding = vectorizer.embed("Hello, world!")

        # Asynchronous batch embedding of multiple texts
        embeddings = await vectorizer.aembed_many(
            ["Hello, world!", "How are you?"],
```

```
            batch_size=2
        )

    """

    _client: Any = PrivateAttr()
    _aclient: Any = PrivateAttr()

    def __init__(
        self, model: str = "text-embedding-ada-002", api_config: Optional[Dict] = None
    ):
        """Initialize the OpenAI vectorizer.

        Args:
            model (str): Model to use for embedding. Defaults to
                'text-embedding-ada-002'.
            api_config (Optional[Dict], optional): Dictionary containing the
                API key and any additional OpenAI API options. Defaults to None.

        Raises:
            ImportError: If the openai library is not installed.
            ValueError: If the OpenAI API key is not provided.
        """
        self._initialize_clients(api_config)
        super().__init__(model=model, dims=self._set_model_dims(model))

    def _initialize_clients(self, api_config: Optional[Dict]):
        """
        Setup the OpenAI clients using the provided API key or an
        environment variable.
        """
        if api_config is None:
            api_config = {}

        # Dynamic import of the openai module
        try:
            from openai import AsyncOpenAI, OpenAI
        except ImportError:
            raise ImportError(
                "OpenAI vectorizer requires the openai library. \
                    Please install with `pip install openai`"
            )

        # Pull the API key from api_config or environment variable
        api_key = (
            api_config.pop("api_key") if api_config else os.getenv("OPENAI_API_KEY")
        )
        if not api_key:
            raise ValueError(
                "OpenAI API key is required. "
                "Provide it in api_config or set the OPENAI_API_KEY\
                    environment variable."
            )

        self._client = OpenAI(api_key=api_key, **api_config)
        self._aclient = AsyncOpenAI(api_key=api_key, **api_config)

    def _set_model_dims(self, model) -> int:
        try:
            embedding = (
                self._client.embeddings.create(input=["dimension test"], model=model)
                .data[0]
                .embedding
            )
```

```
        except (KeyError, IndexError) as ke:
            raise ValueError(f"Unexpected response from the OpenAI API: {str(ke)}")
        except Exception as e:  # pylint: disable=broad-except
            # fall back (TODO get more specific)
            raise ValueError(f"Error setting embedding model dimensions: {str(e)}")
        return len(embedding)

    @retry(
        wait=wait_random_exponential(min=1, max=60),
        stop=stop_after_attempt(6),
        retry=retry_if_not_exception_type(TypeError),
    )
    def embed_many(
        self,
        texts: List[str],
        preprocess: Optional[Callable] = None,
        batch_size: int = 10,
        as_buffer: bool = False,
        **kwargs,
    ) -> List[List[float]]:
        """Embed many chunks of texts using the OpenAI API.

        Args:
            texts (List[str]): List of text chunks to embed.
            preprocess (Optional[Callable], optional): Optional preprocessing
                callable to perform before vectorization. Defaults to None.
            batch_size (int, optional): Batch size of texts to use when creating
                embeddings. Defaults to 10.
            as_buffer (bool, optional): Whether to convert the raw embedding
                to a byte string. Defaults to False.

        Returns:
            List[List[float]]: List of embeddings.

        Raises:
            TypeError: If the wrong input type is passed in for the text.
        """
        if not isinstance(texts, list):
            raise TypeError("Must pass in a list of str values to embed.")
        if len(texts) > 0 and not isinstance(texts[0], str):
            raise TypeError("Must pass in a list of str values to embed.")

        embeddings: List = []
        for batch in self.batchify(texts, batch_size, preprocess):
            response = self._client.embeddings.create(input=batch, model=self.model)
            embeddings += [
                self._process_embedding(r.embedding, as_buffer, **kwargs)
                for r in response.data
            ]
        return embeddings

    @retry(
        wait=wait_random_exponential(min=1, max=60),
        stop=stop_after_attempt(6),
        retry=retry_if_not_exception_type(TypeError),
    )
    def embed(
        self,
        text: str,
        preprocess: Optional[Callable] = None,
        as_buffer: bool = False,
        **kwargs,
    ) -> List[float]:
        """Embed a chunk of text using the OpenAI API.
```

```
        Args:
            text (str): Chunk of text to embed.
            preprocess (Optional[Callable], optional): Optional preprocessing
callable to
                perform before vectorization. Defaults to None.
            as_buffer (bool, optional): Whether to convert the raw embedding
                to a byte string. Defaults to False.

        Returns:
            List[float]: Embedding.

        Raises:
            TypeError: If the wrong input type is passed in for the text.
        """
        if not isinstance(text, str):
            raise TypeError("Must pass in a str value to embed.")

        if preprocess:
            text = preprocess(text)
        result = self._client.embeddings.create(input=[text], model=self.model)
        return self._process_embedding(result.data[0].embedding, as_buffer, **kwargs)

    @retry(
        wait=wait_random_exponential(min=1, max=60),
        stop=stop_after_attempt(6),
        retry=retry_if_not_exception_type(TypeError),
    )
    async def aembed_many(
        self,
        texts: List[str],
        preprocess: Optional[Callable] = None,
        batch_size: int = 1000,
        as_buffer: bool = False,
        **kwargs,
    ) -> List[List[float]]:
        """Asynchronously embed many chunks of texts using the OpenAI API.

        Args:
            texts (List[str]): List of text chunks to embed.
            preprocess (Optional[Callable], optional): Optional preprocessing
callable to
                perform before vectorization. Defaults to None.
            batch_size (int, optional): Batch size of texts to use when creating
                embeddings. Defaults to 10.
            as_buffer (bool, optional): Whether to convert the raw embedding
                to a byte string. Defaults to False.

        Returns:
            List[List[float]]: List of embeddings.

        Raises:
            TypeError: If the wrong input type is passed in for the text.
        """
        if not isinstance(texts, list):
            raise TypeError("Must pass in a list of str values to embed.")
        if len(texts) > 0 and not isinstance(texts[0], str):
            raise TypeError("Must pass in a list of str values to embed.")

        embeddings: List = []
        for batch in self.batchify(texts, batch_size, preprocess):
            response = await self._aclient.embeddings.create(
                input=batch, model=self.model
            )
```

```
            embeddings += [
                self._process_embedding(r.embedding, as_buffer, **kwargs)
                for r in response.data
            ]
        return embeddings

    @retry(
        wait=wait_random_exponential(min=1, max=60),
        stop=stop_after_attempt(6),
        retry=retry_if_not_exception_type(TypeError),
    )
    async def aembed(
        self,
        text: str,
        preprocess: Optional[Callable] = None,
        as_buffer: bool = False,
        **kwargs,
    ) -> List[float]:
        """Asynchronously embed a chunk of text using the OpenAI API.

        Args:
            text (str): Chunk of text to embed.
            preprocess (Optional[Callable], optional): Optional preprocessing
callable to
                perform before vectorization. Defaults to None.
            as_buffer (bool, optional): Whether to convert the raw embedding
                to a byte string. Defaults to False.

        Returns:
            List[float]: Embedding.

        Raises:
            TypeError: If the wrong input type is passed in for the text.
        """
        if not isinstance(text, str):
            raise TypeError("Must pass in a str value to embed.")

        if preprocess:
            text = preprocess(text)
        result = await self._aclient.embeddings.create(input=[text], model=self.model)
        return self._process_embedding(result.data[0].embedding, as_buffer, **kwargs)

    @property
    def type(self) -> str:
        return "openai"
}
```

redisvl/utils/vectorize/text/vertexai.py

```
import os
from typing import Any, Callable, Dict, List, Optional

from pydantic.v1 import PrivateAttr
from tenacity import retry, stop_after_attempt, wait_random_exponential
from tenacity.retry import retry_if_not_exception_type

from redisvl.utils.vectorize.base import BaseVectorizer


class VertexAITextVectorizer(BaseVectorizer):
```

```
    """The VertexAITextVectorizer uses Google's VertexAI Palm 2 embedding model
    API to create text embeddings.

    This vectorizer is tailored for use in
    environments where integration with Google Cloud Platform (GCP) services is
    a key requirement.

    Utilizing this vectorizer requires an active GCP project and location
    (region), along with appropriate application credentials. These can be
    provided through the `api_config` dictionary or set the
GOOGLE_APPLICATION_CREDENTIALS
    env var. Additionally, the vertexai python client must be
    installed with `pip install google-cloud-aiplatform>=1.26`.

    .. code-block:: python

        # Synchronous embedding of a single text
        vectorizer = VertexAITextVectorizer(
            model="textembedding-gecko",
            api_config={
                "project_id": "your_gcp_project_id", # OR set GCP_PROJECT_ID
                "location": "your_gcp_location",     # OR set GCP_LOCATION
            })
        embedding = vectorizer.embed("Hello, world!")

        # Asynchronous batch embedding of multiple texts
        embeddings = await vectorizer.embed_many(
            ["Hello, world!", "Goodbye, world!"],
            batch_size=2
        )

    """

    _client: Any = PrivateAttr()

    def __init__(
        self, model: str = "textembedding-gecko", api_config: Optional[Dict] = None
    ):
        """Initialize the VertexAI vectorizer.

        Args:
            model (str): Model to use for embedding. Defaults to
                'textembedding-gecko'.
            api_config (Optional[Dict], optional): Dictionary containing the
                API config details. Defaults to None.

        Raises:
            ImportError: If the google-cloud-aiplatform library is not installed.
            ValueError: If the API key is not provided.
        """
        self._initialize_client(model, api_config)
        super().__init__(model=model, dims=self._set_model_dims())

    def _initialize_client(self, model: str, api_config: Optional[Dict]):
        """
        Setup the VertexAI clients using the provided API key or an
        environment variable.
        """
        # Fetch the project_id and location from api_config or environment variables
        project_id = (
            api_config.get("project_id") if api_config else os.getenv("GCP_PROJECT_ID")
        )
        location = (
            api_config.get("location") if api_config else os.getenv("GCP_LOCATION")
```

```
        )

        if not project_id:
            raise ValueError(
                "Missing project_id. "
                "Provide the id in the api_config with key 'project_id' "
                "or set the GCP_PROJECT_ID environment variable."
            )

        if not location:
            raise ValueError(
                "Missing location. "
                "Provide the location (region) in the api_config with key 'location' "
                "or set the GCP_LOCATION environment variable."
            )

        # Check for credentials
        credentials = api_config.get("credentials") if api_config else None

        try:
            import vertexai
            from vertexai.language_models import TextEmbeddingModel

            vertexai.init(
                project=project_id, location=location, credentials=credentials
            )
        except ImportError:
            raise ImportError(
                "VertexAI vectorizer requires the google-cloud-aiplatform library. "
                "Please install with `pip install google-cloud-aiplatform>=1.26`"
            )

        self._client = TextEmbeddingModel.from_pretrained(model)

    def _set_model_dims(self) -> int:
        try:
            embedding = self._client.get_embeddings(["dimension test"])[0].values
        except (KeyError, IndexError) as ke:
            raise ValueError(f"Unexpected response from the VertexAI API: {str(ke)}")
        except Exception as e:  # pylint: disable=broad-except
            # fall back (TODO get more specific)
            raise ValueError(f"Error setting embedding model dimensions: {str(e)}")
        return len(embedding)

    @retry(
        wait=wait_random_exponential(min=1, max=60),
        stop=stop_after_attempt(6),
        retry=retry_if_not_exception_type(TypeError),
    )
    def embed_many(
        self,
        texts: List[str],
        preprocess: Optional[Callable] = None,
        batch_size: int = 10,
        as_buffer: bool = False,
        **kwargs,
    ) -> List[List[float]]:
        """Embed many chunks of texts using the VertexAI API.

        Args:
            texts (List[str]): List of text chunks to embed.
            preprocess (Optional[Callable], optional): Optional preprocessing
callable to
                perform before vectorization. Defaults to None.
```

```
            batch_size (int, optional): Batch size of texts to use when creating
                embeddings. Defaults to 10.
            as_buffer (bool, optional): Whether to convert the raw embedding
                to a byte string. Defaults to False.

        Returns:
            List[List[float]]: List of embeddings.

        Raises:
            TypeError: If the wrong input type is passed in for the test.
        """
        if not isinstance(texts, list):
            raise TypeError("Must pass in a list of str values to embed.")
        if len(texts) > 0 and not isinstance(texts[0], str):
            raise TypeError("Must pass in a list of str values to embed.")

        embeddings: List = []
        for batch in self.batchify(texts, batch_size, preprocess):
            response = self._client.get_embeddings(batch)
            embeddings += [
                self._process_embedding(r.values, as_buffer, **kwargs) for r
in response
            ]
        return embeddings

    @retry(
        wait=wait_random_exponential(min=1, max=60),
        stop=stop_after_attempt(6),
        retry=retry_if_not_exception_type(TypeError),
    )
    def embed(
        self,
        text: str,
        preprocess: Optional[Callable] = None,
        as_buffer: bool = False,
        **kwargs,
    ) -> List[float]:
        """Embed a chunk of text using the VertexAI API.

        Args:
            text (str): Chunk of text to embed.
            preprocess (Optional[Callable], optional): Optional preprocessing
callable to
                perform before vectorization. Defaults to None.
            as_buffer (bool, optional): Whether to convert the raw embedding
                to a byte string. Defaults to False.

        Returns:
            List[float]: Embedding.

        Raises:
            TypeError: If the wrong input type is passed in for the test.
        """
        if not isinstance(text, str):
            raise TypeError("Must pass in a str value to embed.")

        if preprocess:
            text = preprocess(text)
        result = self._client.get_embeddings([text])
        return self._process_embedding(result[0].values, as_buffer, **kwargs)

    @property
    def type(self) -> str:
```

```
        return "vertexai"
}
```

redisvl/version.py

```
__version__ = "0.3.5"
}
```

Chapter 3.0.0

schemas

schemas/schema.yaml

```
version: '0.1.0'

index:
  name: user-idx
  prefix: user
  storage_type: json

fields:
  - name: user
    type: tag
  - name: credit_score
    type: tag
  - name: embedding
    type: vector
    attrs:
      algorithm: flat
      dims: 4
      distance_metric: cosine
      datatype: float32
}
```

schemas/semantic_router.yaml

```
name: test-router
routes:
- name: greeting
  references:
  - hello
  - hi
  metadata:
```

```
    type: greeting
  distance_threshold: 0.3
- name: farewell
  references:
  - bye
  - goodbye
  metadata:
    type: farewell
  distance_threshold: 0.3
vectorizer:
  type: hf
  model: sentence-transformers/all-mpnet-base-v2
routing_config:
  distance_threshold: 0.3
  max_k: 2
  aggregation_method: avg
}
```

schemas/test_hash_schema.yaml

```
version: '0.1.0'

index:
    name: hash-test
    prefix: hash
    storage_type: hash

fields:
  - name: sentence
    type: text
  - name: embedding
    type: vector
    attrs:
      dims: 768
      algorithm: flat
      distance_metric: cosine}
```

schemas/test_json_schema.yaml

```
version: '0.1.0'

index:
    name: json-test
    prefix: json
    storage_type: json

fields:
  - name: sentence
    type: text
  - name: embedding
    type: vector
    attrs:
      dims: 768
```

```
      algorithm: flat
      distance_metric: cosine}
```

scripts.py

```
import subprocess

def format():
    subprocess.run(["isort", "./redisvl", "./tests/", "--profile",
"black"], check=True)
    subprocess.run(["black", "./redisvl", "./tests/"], check=True)

def check_format():
    subprocess.run(["black", "--check", "./redisvl"], check=True)

def sort_imports():
    subprocess.run(["isort", "./redisvl", "./tests/", "--profile",
"black"], check=True)

def check_sort_imports():
    subprocess.run(["isort", "./redisvl", "--check-only", "--profile",
"black"], check=True)

def check_lint():
    subprocess.run(["pylint", "--rcfile=.pylintrc", "./redisvl"], check=True)

def mypy():
    subprocess.run(["python", "-m", "mypy", "./redisvl"], check=True)

def test():
    subprocess.run(["python", "-m", "pytest", "--log-level=CRITICAL"], check=True)

def test_verbose():
    subprocess.run(["python", "-m", "pytest", "-vv", "-s", "--log-
level=CRITICAL"], check=True)

def test_cov():
    subprocess.run(["python", "-m", "pytest", "-vv", "--cov=./redisvl", "--cov-
report=xml", "--log-level=CRITICAL"], check=True)

def cov():
    subprocess.run(["coverage", "html"], check=True)
    print("If data was present, coverage report is in ./htmlcov/index.html")

def test_notebooks():
    subprocess.run(["cd", "docs/", "&&", "poetry run treon", "-v"], check=True)

def build_docs():
    subprocess.run("cd docs/ && make html", shell=True)

def serve_docs():
    subprocess.run("cd docs/_build/html && python -m http.server", shell=True)
}
```

Chapter 4.0.0

tests

tests/docker-compose.yml

```
version: "3.9"
services:
  redis:
    image: "redis/redis-stack:${REDIS_VERSION}"
    ports:
      - "6379"
    environment:
      - "REDIS_ARGS=--save '' --appendonly no"
    deploy:
      replicas: 1
      restart_policy:
        condition: on-failure
    labels:
      - "com.docker.compose.publishers=redis,6379,6379"}
```

Chapter 4.1.0

tests/integration

tests/integration/test_async_search_index.py

```
import pytest

from redisvl.exceptions import RedisSearchError
from redisvl.index import AsyncSearchIndex
from redisvl.query import VectorQuery
from redisvl.redis.utils import convert_bytes
from redisvl.schema import IndexSchema, StorageType

fields = [{"name": "test", "type": "tag"}]


@pytest.fixture
def index_schema():
    return IndexSchema.from_dict({"index": {"name": "my_index"}, "fields": fields})


@pytest.fixture
def async_index(index_schema):
    return AsyncSearchIndex(schema=index_schema)


@pytest.fixture
```

```
def async_index_from_dict():
    return AsyncSearchIndex.from_dict({"index": {"name": "my_index"},
"fields": fields})


@pytest.fixture
def async_index_from_yaml():
    return AsyncSearchIndex.from_yaml("schemas/test_json_schema.yaml")


def test_search_index_properties(index_schema, async_index):
    assert async_index.schema == index_schema
    # custom settings
    assert async_index.name == index_schema.index.name == "my_index"
    assert async_index.client == None
    # default settings
    assert async_index.prefix == index_schema.index.prefix == "rvl"
    assert async_index.key_separator == index_schema.index.key_separator == ":"
    assert (
        async_index.storage_type == index_schema.index.storage_type == StorageType.HASH
    )
    assert async_index.key("foo").startswith(async_index.prefix)


def test_search_index_from_yaml(async_index_from_yaml):
    assert async_index_from_yaml.name == "json-test"
    assert async_index_from_yaml.client == None
    assert async_index_from_yaml.prefix == "json"
    assert async_index_from_yaml.key_separator == ":"
    assert async_index_from_yaml.storage_type == StorageType.JSON
    assert async_index_from_yaml.key("foo").startswith(async_index_from_yaml.prefix)


def test_search_index_from_dict(async_index_from_dict):
    assert async_index_from_dict.name == "my_index"
    assert async_index_from_dict.client == None
    assert async_index_from_dict.prefix == "rvl"
    assert async_index_from_dict.key_separator == ":"
    assert async_index_from_dict.storage_type == StorageType.HASH
    assert len(async_index_from_dict.schema.fields) == len(fields)
    assert async_index_from_dict.key("foo").startswith(async_index_from_dict.prefix)


@pytest.mark.asyncio
async def test_search_index_from_existing(async_client, async_index):
    await async_index.set_client(async_client)
    await async_index.create(overwrite=True)

    try:
        async_index2 = await AsyncSearchIndex.from_existing(
            async_index.name, redis_client=async_client
        )
    except Exception as e:
        pytest.skip(str(e))

    assert async_index2.schema == async_index.schema


@pytest.mark.asyncio
async def test_search_index_from_existing_complex(async_client):
    schema = {
        "index": {
            "name": "test",
            "prefix": "test",
```

```
            "storage_type": "json",
        },
        "fields": [
            {"name": "user", "type": "tag", "path": "$.user"},
            {"name": "credit_score", "type": "tag", "path": "$.metadata.credit_score"},
            {"name": "job", "type": "text", "path": "$.metadata.job"},
            {
                "name": "age",
                "type": "numeric",
                "path": "$.metadata.age",
                "attrs": {"sortable": False},
            },
            {
                "name": "user_embedding",
                "type": "vector",
                "attrs": {
                    "dims": 3,
                    "distance_metric": "cosine",
                    "algorithm": "flat",
                    "datatype": "float32",
                },
            },
        ],
    }
    async_index = await AsyncSearchIndex.from_dict(schema).set_client(
        redis_client=async_client
    )
    await async_index.create(overwrite=True)

    try:
        async_index2 = await AsyncSearchIndex.from_existing(
            async_index.name, redis_client=async_client
        )
    except Exception as e:
        pytest.skip(str(e))

    assert async_index2.schema == async_index.schema


def test_search_index_no_prefix(index_schema):
    # specify an explicitly empty prefix...
    index_schema.index.prefix = ""
    async_index = AsyncSearchIndex(schema=index_schema)
    assert async_index.prefix == ""
    assert async_index.key("foo") == "foo"


@pytest.mark.asyncio
async def test_search_index_redis_url(redis_url, index_schema):
    async_index = await AsyncSearchIndex(schema=index_schema).connect(
        redis_url=redis_url
    )
    assert async_index.client

    async_index.disconnect()
    assert async_index.client == None


@pytest.mark.asyncio
async def test_search_index_client(async_client, index_schema):
    async_index = await AsyncSearchIndex(schema=index_schema).set_client(
        redis_client=async_client
    )
    assert async_index.client == async_client
```

```
@pytest.mark.asyncio
async def test_search_index_set_client(async_client, client, async_index):
    await async_index.set_client(async_client)
    assert async_index.client == async_client
    await async_index.set_client(client)

    async_index.disconnect()
    assert async_index.client == None


@pytest.mark.asyncio
async def test_search_index_create(async_client, async_index):
    await async_index.set_client(async_client)
    await async_index.create(overwrite=True, drop=True)
    assert await async_index.exists()
    assert async_index.name in convert_bytes(
        await async_index.client.execute_command("FT._LIST")
    )


@pytest.mark.asyncio
async def test_search_index_delete(async_client, async_index):
    await async_index.set_client(async_client)
    await async_index.create(overwrite=True, drop=True)
    await async_index.delete(drop=True)
    assert not await async_index.exists()
    assert async_index.name not in convert_bytes(
        await async_index.client.execute_command("FT._LIST")
    )


@pytest.mark.asyncio
async def test_search_index_clear(async_client, async_index):
    await async_index.set_client(async_client)
    await async_index.create(overwrite=True, drop=True)
    data = [{"id": "1", "test": "foo"}]
    await async_index.load(data, id_field="id")

    count = await async_index.clear()
    assert count == len(data)
    assert await async_index.exists()


@pytest.mark.asyncio
async def test_search_index_drop_key(async_client, async_index):
    await async_index.set_client(async_client)
    await async_index.create(overwrite=True, drop=True)
    data = [{"id": "1", "test": "foo"}, {"id": "2", "test": "bar"}]
    keys = await async_index.load(data, id_field="id")

    dropped = await async_index.drop_keys(keys[0])
    assert dropped == 1
    assert not await async_index.fetch(keys[0])
    assert await async_index.fetch(keys[1]) is not None


@pytest.mark.asyncio
async def test_search_index_drop_keys(async_client, async_index):
    await async_index.set_client(async_client)
    await async_index.create(overwrite=True, drop=True)
    data = [
        {"id": "1", "test": "foo"},
```

```
        {"id": "2", "test": "bar"},
        {"id": "3", "test": "baz"},
    ]
    keys = await async_index.load(data, id_field="id")

    dropped = await async_index.drop_keys(keys[0:2])
    assert dropped == 2
    assert not await async_index.fetch(keys[0])
    assert not await async_index.fetch(keys[1])
    assert await async_index.fetch(keys[2]) is not None

    assert await async_index.exists()


@pytest.mark.asyncio
async def test_search_index_load_and_fetch(async_client, async_index):
    await async_index.set_client(async_client)
    await async_index.create(overwrite=True, drop=True)
    data = [{"id": "1", "test": "foo"}]
    await async_index.load(data, id_field="id")

    res = await async_index.fetch("1")
    assert (
        res["test"]
        == convert_bytes(await async_index.client.hget("rvl:1", "test"))
        == "foo"
    )

    await async_index.delete(drop=True)
    assert not await async_index.exists()
    assert not await async_index.fetch("1")


@pytest.mark.asyncio
async def test_search_index_load_preprocess(async_client, async_index):
    await async_index.set_client(async_client)
    await async_index.create(overwrite=True, drop=True)
    data = [{"id": "1", "test": "foo"}]

    async def preprocess(record):
        record["test"] = "bar"
        return record

    await async_index.load(data, id_field="id", preprocess=preprocess)
    res = await async_index.fetch("1")
    assert (
        res["test"]
        == convert_bytes(await async_index.client.hget("rvl:1", "test"))
        == "bar"
    )

    async def bad_preprocess(record):
        return 1

    with pytest.raises(TypeError):
        await async_index.load(data, id_field="id", preprocess=bad_preprocess)


@pytest.mark.asyncio
async def test_search_index_load_empty(async_client, async_index):
    await async_index.set_client(async_client)
    await async_index.create(overwrite=True, drop=True)
    await async_index.load([])
```

```
@pytest.mark.asyncio
async def test_no_id_field(async_client, async_index):
    await async_index.set_client(async_client)
    await async_index.create(overwrite=True, drop=True)
    bad_data = [{"wrong_key": "1", "value": "test"}]

    # catch missing / invalid id_field
    with pytest.raises(ValueError):
        await async_index.load(bad_data, id_field="key")


@pytest.mark.asyncio
async def test_check_index_exists_before_delete(async_client, async_index):
    await async_index.set_client(async_client)
    await async_index.create(overwrite=True, drop=True)
    await async_index.delete(drop=True)
    with pytest.raises(RedisSearchError):
        await async_index.delete()


@pytest.mark.asyncio
async def test_check_index_exists_before_search(async_client, async_index):
    await async_index.set_client(async_client)
    await async_index.create(overwrite=True, drop=True)
    await async_index.delete(drop=True)

    query = VectorQuery(
        [0.1, 0.1, 0.5],
        "user_embedding",
        return_fields=["user", "credit_score", "age", "job", "location"],
        num_results=7,
    )
    with pytest.raises(RedisSearchError):
        await async_index.search(query.query, query_params=query.params)


@pytest.mark.asyncio
async def test_check_index_exists_before_info(async_client, async_index):
    await async_index.set_client(async_client)
    await async_index.create(overwrite=True, drop=True)
    await async_index.delete(drop=True)

    with pytest.raises(RedisSearchError):
        await async_index.info()
}
```

tests/integration/test_connection.py

```
import os

import pytest
from redis import Redis
from redis.asyncio import Redis as AsyncRedis
from redis.exceptions import ConnectionError

from redisvl.exceptions import RedisModuleVersionError
from redisvl.redis.connection import (
    RedisConnectionFactory,
    compare_versions,
```

```
    convert_index_info_to_schema,
    get_address_from_env,
    unpack_redis_modules,
    validate_modules,
)
from redisvl.schema import IndexSchema
from redisvl.version import __version__

EXPECTED_LIB_NAME = f"redis-py(redisvl_v{__version__})"


def test_get_address_from_env(redis_url):
    assert get_address_from_env() == redis_url


def test_unpack_redis_modules():
    module_list = [
        {
            "name": "search",
            "ver": 20811,
            "path": "/opt/redis-stack/lib/redisearch.so",
            "args": [],
        },
        {
            "name": "ReJSON",
            "ver": 20609,
            "path": "/opt/redis-stack/lib/rejson.so",
            "args": [],
        },
    ]
    installed_modules = unpack_redis_modules(module_list)
    assert installed_modules == {"search": 20811, "ReJSON": 20609}


def test_convert_index_info_to_schema():
    index_info = {
        "index_name": "image_summaries",
        "index_options": [],
        "index_definition": [
            "key_type",
            "HASH",
            "prefixes",
            ["summary"],
            "default_score",
            "1",
        ],
        "attributes": [
            [
                "identifier",
                "content",
                "attribute",
                "content",
                "type",
                "TEXT",
                "WEIGHT",
                "1",
            ],
            [
                "identifier",
                "doc_id",
                "attribute",
                "doc_id",
                "type",
                "TAG",
```

```
                "SEPARATOR",
                ",",
            ],
            [
                "identifier",
                "content_vector",
                "attribute",
                "content_vector",
                "type",
                "VECTOR",
                "algorithm",
                "FLAT",
                "data_type",
                "FLOAT32",
                "dim",
                1536,
                "distance_metric",
                "COSINE",
            ],
        ],
    }
    schema_dict = convert_index_info_to_schema(index_info)
    assert "index" in schema_dict
    assert "fields" in schema_dict
    assert len(schema_dict["fields"]) == len(index_info["attributes"])

    schema = IndexSchema.from_dict(schema_dict)
    assert schema.index.name == index_info["index_name"]


def test_validate_modules_exist_search():
    validate_modules(
        installed_modules={"search": 20811},
        required_modules=[
            {"name": "search", "ver": 20600},
            {"name": "searchlight", "ver": 20600},
        ],
    )


def test_validate_modules_exist_searchlight():
    validate_modules(
        installed_modules={"searchlight": 20819},
        required_modules=[
            {"name": "search", "ver": 20810},
            {"name": "searchlight", "ver": 20810},
        ],
    )


def test_validate_modules_not_exist():
    with pytest.raises(RedisModuleVersionError):
        validate_modules(
            installed_modules={"search": 20811},
            required_modules=[
                {"name": "ReJSON", "ver": 20600},
            ],
        )


def test_sync_redis_connect(redis_url):
    client = RedisConnectionFactory.connect(redis_url)
    assert client is not None
    assert isinstance(client, Redis)
```

```
    # Perform a simple operation
    assert client.ping()


@pytest.mark.asyncio
async def test_async_redis_connect(redis_url):
    client = RedisConnectionFactory.connect(redis_url, use_async=True)
    assert client is not None
    assert isinstance(client, AsyncRedis)
    # Perform a simple operation
    assert await client.ping()


def test_missing_env_var():
    redis_url = os.getenv("REDIS_URL")
    if redis_url:
        del os.environ["REDIS_URL"]
        with pytest.raises(ValueError):
            RedisConnectionFactory.connect()
        os.environ["REDIS_URL"] = redis_url


def test_invalid_url_format():
    with pytest.raises(ValueError):
        RedisConnectionFactory.connect(redis_url="invalid_url_format")


def test_unknown_redis():
    bad_client = RedisConnectionFactory.connect(redis_url="redis://fake:1234")
    with pytest.raises(ConnectionError):
        bad_client.ping()


def test_validate_redis(client):
    redis_version = client.info()["redis_version"]
    if not compare_versions(redis_version, "7.2.0"):
        pytest.skip("Not using a late enough version of Redis")
    RedisConnectionFactory.validate_sync_redis(client)
    lib_name = client.client_info()
    assert lib_name["lib-name"] == EXPECTED_LIB_NAME


@pytest.mark.asyncio
async def test_validate_async_redis(async_client):
    redis_version = (await async_client.info())["redis_version"]
    if not compare_versions(redis_version, "7.2.0"):
        pytest.skip("Not using a late enough version of Redis")
    await RedisConnectionFactory.validate_async_redis(async_client)
    lib_name = await async_client.client_info()
    assert lib_name["lib-name"] == EXPECTED_LIB_NAME


def test_validate_redis_custom_lib_name(client):
    redis_version = client.info()["redis_version"]
    if not compare_versions(redis_version, "7.2.0"):
        pytest.skip("Not using a late enough version of Redis")
    RedisConnectionFactory.validate_sync_redis(client, "langchain_v0.1.0")
    lib_name = client.client_info()
    assert lib_name["lib-name"] == f"redis-py(redisvl_v{__version__};langchain_v0.1.0)"


@pytest.mark.asyncio
async def test_validate_async_redis_custom_lib_name(async_client):
    redis_version = (await async_client.info())["redis_version"]
```

```
    if not compare_versions(redis_version, "7.2.0"):
        pytest.skip("Not using a late enough version of Redis")
    await RedisConnectionFactory.validate_async_redis(async_client, "langchain_v0.1.0")
    lib_name = await async_client.client_info()
    assert lib_name["lib-name"] == f"redis-py(redisvl_v{__version__};langchain_v0.1.0)"
}
```

tests/integration/test_flow.py

```
import pytest

from redisvl.index import SearchIndex
from redisvl.query import VectorQuery
from redisvl.redis.utils import array_to_buffer
from redisvl.schema import StorageType

fields_spec = [
    {"name": "credit_score", "type": "tag"},
    {"name": "user", "type": "tag"},
    {"name": "job", "type": "text"},
    {"name": "age", "type": "numeric"},
    {
        "name": "user_embedding",
        "type": "vector",
        "attrs": {
            "dims": 3,
            "distance_metric": "cosine",
            "algorithm": "flat",
            "datatype": "float32",
        },
    },
]

hash_schema = {
    "index": {
        "name": "user_index_hash",
        "prefix": "users_hash",
        "storage_type": "hash",
    },
    "fields": fields_spec,
}

json_schema = {
    "index": {
        "name": "user_index_json",
        "prefix": "users_json",
        "storage_type": "json",
    },
    "fields": fields_spec,
}


@pytest.mark.parametrize("schema", [hash_schema, json_schema])
def test_simple(client, schema, sample_data):
    index = SearchIndex.from_dict(schema)
    # assign client (only for testing)
    index.set_client(client)
    # create the index
    index.create(overwrite=True, drop=True)
```

```
    # Prepare and load the data based on storage type
    def hash_preprocess(item: dict) -> dict:
        return {
            **item,
            "user_embedding": array_to_buffer(item["user_embedding"], "float32"),
        }

    if index.storage_type == StorageType.HASH:
        index.load(sample_data, preprocess=hash_preprocess, id_field="user")
    else:
        index.load(sample_data, id_field="user")

    assert index.fetch("john")

    return_fields = ["user", "age", "job", "credit_score"]
    query = VectorQuery(
        vector=[0.1, 0.1, 0.5],
        vector_field_name="user_embedding",
        return_fields=return_fields,
        num_results=3,
    )

    results = index.search(query.query, query_params=query.params)
    results_2 = index.query(query)
    assert len(results.docs) == len(results_2)

    # make sure correct users returned
    # users = list(results.docs)
    # print(len(users))
    users = [doc for doc in results.docs]
    assert users[0].user in ["john", "mary"]
    assert users[1].user in ["john", "mary"]

    # make sure vector scores are correct
    # query vector and first two are the same vector.
    # third is different (hence should be positive difference)
    assert float(users[0].vector_distance) == 0.0
    assert float(users[1].vector_distance) == 0.0
    assert float(users[2].vector_distance) > 0

    for doc1, doc2 in zip(results.docs, results_2):
        for field in return_fields:
            assert getattr(doc1, field) == doc2[field]

    count_deleted_keys = index.clear()
    assert count_deleted_keys == len(sample_data)

    assert index.exists() == True

    index.delete()

    assert index.exists() == False
}
```

tests/integration/test_flow_async.py

```
import asyncio
import time

import pytest
```

```
from redisvl.index import AsyncSearchIndex
from redisvl.query import VectorQuery
from redisvl.redis.utils import array_to_buffer
from redisvl.schema import StorageType

fields_spec = [
    {"name": "credit_score", "type": "tag"},
    {"name": "user", "type": "tag"},
    {"name": "job", "type": "text"},
    {"name": "age", "type": "numeric"},
    {
        "name": "user_embedding",
        "type": "vector",
        "attrs": {
            "dims": 3,
            "distance_metric": "cosine",
            "algorithm": "flat",
            "datatype": "float32",
        },
    },
]

hash_schema = {
    "index": {
        "name": "user_index_hash",
        "prefix": "users_hash",
        "storage_type": "hash",
    },
    "fields": fields_spec,
}

json_schema = {
    "index": {
        "name": "user_index_json",
        "prefix": "users_json",
        "storage_type": "json",
    },
    "fields": fields_spec,
}


@pytest.mark.asyncio
@pytest.mark.parametrize("schema", [hash_schema, json_schema])
async def test_simple(async_client, schema, sample_data):
    index = AsyncSearchIndex.from_dict(schema)
    # assign client (only for testing)
    await index.set_client(async_client)
    # create the index
    await index.create(overwrite=True, drop=True)

    # Prepare and load the data based on storage type
    async def hash_preprocess(item: dict) -> dict:
        return {
            **item,
            "user_embedding": array_to_buffer(item["user_embedding"], "float32"),
        }

    if index.storage_type == StorageType.HASH:
        await index.load(sample_data, preprocess=hash_preprocess, id_field="user")
    else:
        await index.load(sample_data, id_field="user")

    assert await index.fetch("john")
```

```
    # wait for async index to create
    await asyncio.sleep(1)

    return_fields = ["user", "age", "job", "credit_score"]
    query = VectorQuery(
        vector=[0.1, 0.1, 0.5],
        vector_field_name="user_embedding",
        return_fields=return_fields,
        num_results=3,
    )

    results = await index.search(query.query, query_params=query.params)
    results_2 = await index.query(query)
    assert len(results.docs) == len(results_2)

    # make sure correct users returned
    users = [doc for doc in results.docs]
    assert users[0].user in ["john", "mary"]
    assert users[1].user in ["john", "mary"]

    # make sure vector scores are correct
    assert float(users[0].vector_distance) == 0.0
    assert float(users[1].vector_distance) == 0.0
    assert float(users[2].vector_distance) > 0

    for doc1, doc2 in zip(results.docs, results_2):
        for field in return_fields:
            assert getattr(doc1, field) == doc2[field]

    count_deleted_keys = await index.clear()
    assert count_deleted_keys == len(sample_data)

    assert await index.exists() == True

    await index.delete()

    assert await index.exists() == False
}
```

tests/integration/test_llmcache.py

```
import asyncio
import os
from collections import namedtuple
from time import sleep, time

import pytest
from pydantic.v1 import ValidationError
from redis.exceptions import ConnectionError

from redisvl.exceptions import RedisModuleVersionError
from redisvl.extensions.llmcache import SemanticCache
from redisvl.index.index import AsyncSearchIndex, SearchIndex
from redisvl.query.filter import Num, Tag, Text
from redisvl.utils.vectorize import HFTextVectorizer


@pytest.fixture
def vectorizer():
```

```
    return HFTextVectorizer("sentence-transformers/all-mpnet-base-v2")


@pytest.fixture
def cache(vectorizer, redis_url):
    cache_instance = SemanticCache(
        vectorizer=vectorizer,
        distance_threshold=0.2,
        redis_url=redis_url,
    )
    yield cache_instance
    cache_instance._index.delete(True)  # Clean up index


@pytest.fixture
def cache_with_filters(vectorizer, redis_url):
    cache_instance = SemanticCache(
        vectorizer=vectorizer,
        distance_threshold=0.2,
        filterable_fields=[{"name": "label", "type": "tag"}],
        redis_url=redis_url,
    )
    yield cache_instance
    cache_instance._index.delete(True)  # Clean up index


@pytest.fixture
def cache_no_cleanup(vectorizer, redis_url):
    cache_instance = SemanticCache(
        vectorizer=vectorizer, distance_threshold=0.2, redis_url=redis_url
    )
    yield cache_instance


@pytest.fixture
def cache_with_ttl(vectorizer, redis_url):
    cache_instance = SemanticCache(
        vectorizer=vectorizer, distance_threshold=0.2, ttl=2, redis_url=redis_url
    )
    yield cache_instance
    cache_instance._index.delete(True)  # Clean up index


@pytest.fixture
def cache_with_redis_client(vectorizer, client):
    cache_instance = SemanticCache(
        vectorizer=vectorizer,
        redis_client=client,
        distance_threshold=0.2,
    )
    yield cache_instance
    cache_instance.clear()  # Clear cache after each test
    cache_instance._index.delete(True)  # Clean up index


def test_bad_ttl(cache):
    with pytest.raises(ValueError):
        cache.set_ttl(2.5)


def test_cache_ttl(cache_with_ttl):
    assert cache_with_ttl.ttl == 2
    cache_with_ttl.set_ttl(5)
    assert cache_with_ttl.ttl == 5
```

```
def test_set_ttl(cache):
    assert cache.ttl == None
    cache.set_ttl(5)
    assert cache.ttl == 5


def test_reset_ttl(cache):
    cache.set_ttl(4)
    cache.set_ttl()
    assert cache.ttl is None


def test_get_index(cache):
    assert isinstance(cache.index, SearchIndex)


@pytest.mark.asyncio
async def test_get_async_index(cache):
    aindex = await cache._get_async_index()
    assert isinstance(aindex, AsyncSearchIndex)


@pytest.mark.asyncio
async def test_get_async_index_from_provided_client(cache_with_redis_client):
    aindex = await cache_with_redis_client._get_async_index()
    assert isinstance(aindex, AsyncSearchIndex)
    assert aindex == cache_with_redis_client.aindex


def test_delete(cache_no_cleanup):
    cache_no_cleanup.delete()
    assert not cache_no_cleanup.index.exists()


@pytest.mark.asyncio
async def test_async_delete(cache_no_cleanup):
    await cache_no_cleanup.adelete()
    assert not cache_no_cleanup.index.exists()


def test_store_and_check(cache, vectorizer):
    prompt = "This is a test prompt."
    response = "This is a test response."
    vector = vectorizer.embed(prompt)

    cache.store(prompt, response, vector=vector)
    check_result = cache.check(vector=vector, distance_threshold=0.4)

    assert len(check_result) == 1
    print(check_result, flush=True)
    assert response == check_result[0]["response"]
    assert "metadata" not in check_result[0]


@pytest.mark.asyncio
async def test_async_store_and_check(cache, vectorizer):
    prompt = "This is a test prompt."
    response = "This is a test response."
    vector = vectorizer.embed(prompt)

    await cache.astore(prompt, response, vector=vector)
    check_result = await cache.acheck(vector=vector, distance_threshold=0.4)
```

```
    assert len(check_result) == 1
    print(check_result, flush=True)
    assert response == check_result[0]["response"]
    assert "metadata" not in check_result[0]


def test_return_fields(cache, vectorizer):
    prompt = "This is a test prompt."
    response = "This is a test response."
    vector = vectorizer.embed(prompt)

    cache.store(prompt, response, vector=vector)

    # check default return fields
    check_result = cache.check(vector=vector)
    assert set(check_result[0].keys()) == {
        "key",
        "entry_id",
        "prompt",
        "response",
        "vector_distance",
        "inserted_at",
        "updated_at",
    }

    # check specific return fields
    fields = [
        "key",
        "entry_id",
        "prompt",
        "response",
        "vector_distance",
    ]
    check_result = cache.check(vector=vector, return_fields=fields)
    assert set(check_result[0].keys()) == set(fields)

    # check only some return fields
    fields = ["inserted_at", "updated_at"]
    check_result = cache.check(vector=vector, return_fields=fields)
    fields.append("key")
    assert set(check_result[0].keys()) == set(fields)


@pytest.mark.asyncio
async def test_async_return_fields(cache, vectorizer):
    prompt = "This is a test prompt."
    response = "This is a test response."
    vector = vectorizer.embed(prompt)

    await cache.astore(prompt, response, vector=vector)

    # check default return fields
    check_result = await cache.acheck(vector=vector)
    assert set(check_result[0].keys()) == {
        "key",
        "entry_id",
        "prompt",
        "response",
        "vector_distance",
        "inserted_at",
        "updated_at",
    }
```

```
    # check specific return fields
    fields = [
        "key",
        "entry_id",
        "prompt",
        "response",
        "vector_distance",
    ]
    check_result = await cache.acheck(vector=vector, return_fields=fields)
    assert set(check_result[0].keys()) == set(fields)

    # check only some return fields
    fields = ["inserted_at", "updated_at"]
    check_result = await cache.acheck(vector=vector, return_fields=fields)
    fields.append("key")
    assert set(check_result[0].keys()) == set(fields)


# Test clearing the cache
def test_clear(cache, vectorizer):
    prompt = "This is a test prompt."
    response = "This is a test response."
    vector = vectorizer.embed(prompt)

    cache.store(prompt, response, vector=vector)
    cache.clear()
    check_result = cache.check(vector=vector)

    assert len(check_result) == 0


@pytest.mark.asyncio
async def test_async_clear(cache, vectorizer):
    prompt = "This is a test prompt."
    response = "This is a test response."
    vector = vectorizer.embed(prompt)

    await cache.astore(prompt, response, vector=vector)
    await cache.aclear()
    check_result = await cache.acheck(vector=vector)

    assert len(check_result) == 0


# Test TTL functionality
def test_ttl_expiration(cache_with_ttl, vectorizer):
    prompt = "This is a test prompt."
    response = "This is a test response."
    vector = vectorizer.embed(prompt)

    cache_with_ttl.store(prompt, response, vector=vector)
    sleep(3)

    check_result = cache_with_ttl.check(vector=vector)
    assert len(check_result) == 0


@pytest.mark.asyncio
async def test_async_ttl_expiration(cache_with_ttl, vectorizer):
    prompt = "This is a test prompt."
    response = "This is a test response."
    vector = vectorizer.embed(prompt)

    await cache_with_ttl.astore(prompt, response, vector=vector)
```

```
    sleep(3)

    check_result = await cache_with_ttl.acheck(vector=vector)
    assert len(check_result) == 0


def test_custom_ttl(cache_with_ttl, vectorizer):
    prompt = "This is a test prompt."
    response = "This is a test response."
    vector = vectorizer.embed(prompt)

    cache_with_ttl.store(prompt, response, vector=vector, ttl=5)
    sleep(3)

    check_result = cache_with_ttl.check(vector=vector)
    assert len(check_result) != 0
    assert cache_with_ttl.ttl == 2


@pytest.mark.asyncio
async def test_async_custom_ttl(cache_with_ttl, vectorizer):
    prompt = "This is a test prompt."
    response = "This is a test response."
    vector = vectorizer.embed(prompt)

    await cache_with_ttl.astore(prompt, response, vector=vector, ttl=5)
    await asyncio.sleep(3)

    check_result = await cache_with_ttl.acheck(vector=vector)
    assert len(check_result) != 0
    assert cache_with_ttl.ttl == 2


def test_ttl_refresh(cache_with_ttl, vectorizer):
    prompt = "This is a test prompt."
    response = "This is a test response."
    vector = vectorizer.embed(prompt)

    cache_with_ttl.store(prompt, response, vector=vector)

    for _ in range(3):
        sleep(1)
        check_result = cache_with_ttl.check(vector=vector)

    assert len(check_result) == 1


@pytest.mark.asyncio
async def test_async_ttl_refresh(cache_with_ttl, vectorizer):
    prompt = "This is a test prompt."
    response = "This is a test response."
    vector = vectorizer.embed(prompt)

    await cache_with_ttl.astore(prompt, response, vector=vector)

    for _ in range(3):
        await asyncio.sleep(1)
        check_result = await cache_with_ttl.acheck(vector=vector)

    assert len(check_result) == 1


# Test manual expiration of single document
def test_drop_document(cache, vectorizer):
```

```
    prompt = "This is a test prompt."
    response = "This is a test response."
    vector = vectorizer.embed(prompt)

    cache.store(prompt, response, vector=vector)
    check_result = cache.check(vector=vector)

    cache.drop(ids=[check_result[0]["entry_id"]])
    recheck_result = cache.check(vector=vector)
    assert len(recheck_result) == 0


@pytest.mark.asyncio
async def test_async_drop_document(cache, vectorizer):
    prompt = "This is a test prompt."
    response = "This is a test response."
    vector = vectorizer.embed(prompt)

    await cache.astore(prompt, response, vector=vector)
    check_result = await cache.acheck(vector=vector)

    await cache.adrop(ids=[check_result[0]["entry_id"]])
    recheck_result = await cache.acheck(vector=vector)
    assert len(recheck_result) == 0


# Test manual expiration of multiple documents
def test_drop_documents(cache, vectorizer):
    prompts = [
        "This is a test prompt.",
        "This is also test prompt.",
        "This is another test prompt.",
    ]
    responses = [
        "This is a test response.",
        "This is also test response.",
        "This is a another test response.",
    ]
    for prompt, response in zip(prompts, responses):
        vector = vectorizer.embed(prompt)
        cache.store(prompt, response, vector=vector)

    check_result = cache.check(vector=vector, num_results=3)
    print(check_result, flush=True)
    ids = [r["entry_id"] for r in check_result[0:2]]  # drop first 2 entries
    cache.drop(ids=ids)

    recheck_result = cache.check(vector=vector, num_results=3)
    assert len(recheck_result) == 1


@pytest.mark.asyncio
async def test_async_drop_documents(cache, vectorizer):
    prompts = [
        "This is a test prompt.",
        "This is also test prompt.",
        "This is another test prompt.",
    ]
    responses = [
        "This is a test response.",
        "This is also test response.",
        "This is a another test response.",
    ]
    for prompt, response in zip(prompts, responses):
```

```
        vector = vectorizer.embed(prompt)
        await cache.astore(prompt, response, vector=vector)

    check_result = await cache.acheck(vector=vector, num_results=3)
    print(check_result, flush=True)
    ids = [r["entry_id"] for r in check_result[0:2]]  # drop first 2 entries
    await cache.adrop(ids=ids)

    recheck_result = await cache.acheck(vector=vector, num_results=3)
    assert len(recheck_result) == 1


# Test updating document fields
def test_updating_document(cache):
    prompt = "This is a test prompt."
    response = "This is a test response."
    cache.store(prompt=prompt, response=response)

    check_result = cache.check(prompt=prompt, return_fields=["updated_at"])
    key = check_result[0]["key"]

    sleep(1)

    metadata = {"foo": "bar"}
    cache.update(key=key, metadata=metadata)

    updated_result = cache.check(
        prompt=prompt, return_fields=["updated_at", "metadata"]
    )
    assert updated_result[0]["metadata"] == metadata
    assert updated_result[0]["updated_at"] > check_result[0]["updated_at"]


@pytest.mark.asyncio
async def test_async_updating_document(cache):
    prompt = "This is a test prompt."
    response = "This is a test response."
    await cache.astore(prompt=prompt, response=response)

    check_result = await cache.acheck(prompt=prompt, return_fields=["updated_at"])
    key = check_result[0]["key"]

    sleep(1)

    metadata = {"foo": "bar"}
    await cache.aupdate(key=key, metadata=metadata)

    updated_result = await cache.acheck(
        prompt=prompt, return_fields=["updated_at", "metadata"]
    )
    assert updated_result[0]["metadata"] == metadata
    assert updated_result[0]["updated_at"] > check_result[0]["updated_at"]


def test_ttl_expiration_after_update(cache_with_ttl, vectorizer):
    prompt = "This is a test prompt."
    response = "This is a test response."
    vector = vectorizer.embed(prompt)
    cache_with_ttl.set_ttl(4)

    assert cache_with_ttl.ttl == 4

    cache_with_ttl.store(prompt, response, vector=vector)
    sleep(5)
```

```
    check_result = cache_with_ttl.check(vector=vector)
    assert len(check_result) == 0


@pytest.mark.asyncio
async def test_async_ttl_expiration_after_update(cache_with_ttl, vectorizer):
    prompt = "This is a test prompt."
    response = "This is a test response."
    vector = vectorizer.embed(prompt)
    cache_with_ttl.set_ttl(4)

    assert cache_with_ttl.ttl == 4

    await cache_with_ttl.astore(prompt, response, vector=vector)
    sleep(5)

    check_result = await cache_with_ttl.acheck(vector=vector)
    assert len(check_result) == 0


# Test check behavior with no match
def test_check_no_match(cache, vectorizer):
    vector = vectorizer.embed("Some random sentence.")
    check_result = cache.check(vector=vector)
    assert len(check_result) == 0


def test_check_invalid_input(cache):
    with pytest.raises(ValueError):
        cache.check()

    with pytest.raises(TypeError):
        cache.check(prompt="test", return_fields="bad value")


@pytest.mark.asyncio
async def test_async_check_invalid_input(cache):
    with pytest.raises(ValueError):
        await cache.acheck()

    with pytest.raises(TypeError):
        await cache.acheck(prompt="test", return_fields="bad value")


def test_bad_connection_info(vectorizer):
    with pytest.raises(ConnectionError):
        SemanticCache(
            vectorizer=vectorizer,
            distance_threshold=0.2,
            redis_url="redis://localhost:6389",
        )


def test_store_with_metadata(cache, vectorizer):
    prompt = "This is another test prompt."
    response = "This is another test response."
    metadata = {"source": "test"}
    vector = vectorizer.embed(prompt)

    cache.store(prompt, response, vector=vector, metadata=metadata)
    check_result = cache.check(vector=vector, num_results=1)

    assert len(check_result) == 1
```

```
    print(check_result, flush=True)
    assert check_result[0]["response"] == response
    assert check_result[0]["metadata"] == metadata
    assert check_result[0]["prompt"] == prompt


def test_store_with_empty_metadata(cache, vectorizer):
    prompt = "This is another test prompt."
    response = "This is another test response."
    metadata = {}
    vector = vectorizer.embed(prompt)

    cache.store(prompt, response, vector=vector, metadata=metadata)
    check_result = cache.check(vector=vector, num_results=1)

    assert len(check_result) == 1
    print(check_result, flush=True)
    assert check_result[0]["response"] == response
    assert check_result[0]["metadata"] == metadata
    assert check_result[0]["prompt"] == prompt


def test_store_with_invalid_metadata(cache, vectorizer):
    prompt = "This is another test prompt."
    response = "This is another test response."
    metadata = namedtuple("metadata", "source")(**{"source": "test"})

    vector = vectorizer.embed(prompt)

    with pytest.raises(ValidationError):
        cache.store(prompt, response, vector=vector, metadata=metadata)


def test_distance_threshold(cache):
    initial_threshold = cache.distance_threshold
    new_threshold = 0.1

    cache.set_threshold(new_threshold)
    assert cache.distance_threshold == new_threshold
    assert cache.distance_threshold != initial_threshold


def test_distance_threshold_out_of_range(cache):
    out_of_range_threshold = -1
    with pytest.raises(ValueError):
        cache.set_threshold(out_of_range_threshold)


def test_multiple_items(cache, vectorizer):
    prompts_responses = {
        "prompt1": "response1",
        "prompt2": "response2",
        "prompt3": "response3",
    }

    for prompt, response in prompts_responses.items():
        vector = vectorizer.embed(prompt)
        cache.store(prompt, response, vector=vector)

    for prompt, expected_response in prompts_responses.items():
        vector = vectorizer.embed(prompt)
        check_result = cache.check(vector=vector)
        assert len(check_result) == 1
        print(check_result, flush=True)
```

```
        assert check_result[0]["response"] == expected_response
        assert "metadata" not in check_result[0]


def test_store_and_check_with_provided_client(cache_with_redis_client, vectorizer):
    prompt = "This is a test prompt."
    response = "This is a test response."
    vector = vectorizer.embed(prompt)

    cache_with_redis_client.store(prompt, response, vector=vector)
    check_result = cache_with_redis_client.check(vector=vector)

    assert len(check_result) == 1
    print(check_result, flush=True)
    assert response == check_result[0]["response"]
    assert "metadata" not in check_result[0]


@pytest.mark.asyncio
async def test_async_store_and_check_with_provided_client(
    cache_with_redis_client, vectorizer
):
    prompt = "This is a test prompt."
    response = "This is a test response."
    vector = vectorizer.embed(prompt)

    await cache_with_redis_client.astore(prompt, response, vector=vector)
    check_result = await cache_with_redis_client.acheck(vector=vector)

    assert len(check_result) == 1
    print(check_result, flush=True)
    assert response == check_result[0]["response"]
    assert "metadata" not in check_result[0]


def test_vector_size(cache, vectorizer):
    prompt = "This is test prompt."
    response = "This is a test response."

    vector = vectorizer.embed(prompt)
    cache.store(prompt=prompt, response=response, vector=vector)

    # Test we can query with modified embeddings of correct size
    vector_2 = [v * 0.99 for v in vector]  # same dimensions
    check_result = cache.check(vector=vector_2)
    assert check_result[0]["prompt"] == prompt

    # Test that error is raised when we try to load wrong size vectors
    with pytest.raises(ValueError):
        cache.store(prompt=prompt, response=response, vector=vector[0:-1])

    with pytest.raises(ValueError):
        cache.store(prompt=prompt, response=response, vector=[1, 2, 3])

    # Test that error is raised when we try to query with wrong size vector
    with pytest.raises(ValueError):
        cache.check(vector=vector[0:-1])

    with pytest.raises(ValueError):
        cache.check(vector=[1, 2, 3])


def test_cache_with_filters(cache_with_filters):
    assert "label" in cache_with_filters._index.schema.fields
```

```
def test_cache_filtering(cache_with_filters):
    tag_1 = "group 0"
    tag_2 = "group 1"
    tag_3 = "group 2"
    tag_4 = "group 3"
    tags = [tag_1, tag_2, tag_3, tag_4]

    filter_1 = Tag("label") == tag_1
    filter_2 = Tag("label") == tag_2
    filter_3 = Tag("label") == tag_3

    for i in range(4):
        prompt = f"test prompt {i}"
        response = f"test response {i}"
        cache_with_filters.store(prompt, response, filters={"label": tags[i]})

    # test we can specify one specific tag
    results = cache_with_filters.check(
        "test prompt 1", filter_expression=filter_1, num_results=5
    )
    assert len(results) == 1
    assert results[0]["prompt"] == "test prompt 0"

    # test we can pass a list of tags
    combined_filter = filter_1 | filter_2 | filter_3
    results = cache_with_filters.check(
        "test prompt 1", filter_expression=combined_filter, num_results=5
    )
    assert len(results) == 3

    # test that default tag param searches full cache
    results = cache_with_filters.check("test prompt 1", num_results=5)
    assert len(results) == 4

    # test no results are returned if we pass a nonexistant tag
    bad_filter = Tag("label") == "bad tag"
    results = cache_with_filters.check(
        "test prompt 1", filter_expression=bad_filter, num_results=5
    )
    assert len(results) == 0


def test_cache_bad_filters(vectorizer, redis_url):
    with pytest.raises(ValueError):
        cache_instance = SemanticCache(
            vectorizer=vectorizer,
            distance_threshold=0.2,
            # invalid field type
            filterable_fields=[
                {"name": "label", "type": "tag"},
                {"name": "test", "type": "nothing"},
            ],
            redis_url=redis_url,
        )

    with pytest.raises(ValueError):
        cache_instance = SemanticCache(
            vectorizer=vectorizer,
            distance_threshold=0.2,
            # duplicate field type
            filterable_fields=[
                {"name": "label", "type": "tag"},
```

```
                {"name": "label", "type": "tag"},
            ],
            redis_url=redis_url,
        )

    with pytest.raises(ValueError):
        cache_instance = SemanticCache(
            vectorizer=vectorizer,
            distance_threshold=0.2,
            # reserved field name
            filterable_fields=[
                {"name": "label", "type": "tag"},
                {"name": "metadata", "type": "tag"},
            ],
            redis_url=redis_url,
        )


def test_complex_filters(cache_with_filters):
    cache_with_filters.store(prompt="prompt 1", response="response 1")
    cache_with_filters.store(prompt="prompt 2", response="response 2")
    sleep(1)
    current_timestamp = time()
    cache_with_filters.store(prompt="prompt 3", response="response 3")

    # test we can do range filters on inserted_at and updated_at fields
    range_filter = Num("inserted_at") < current_timestamp
    results = cache_with_filters.check(
        "prompt 1", filter_expression=range_filter, num_results=5
    )
    assert len(results) == 2

    # test we can combine range filters and text filters
    prompt_filter = Text("prompt") % "*pt 1"
    combined_filter = prompt_filter & range_filter

    results = cache_with_filters.check(
        "prompt 1", filter_expression=combined_filter, num_results=5
    )
    assert len(results) == 1


def test_index_updating(redis_url):
    cache_no_tags = SemanticCache(
        name="test_cache",
        redis_url=redis_url,
    )

    cache_no_tags.store(
        prompt="this prompt has tags",
        response="this response has tags",
        filters={"some_tag": "abc"},
    )

    # filterable_fields not defined in schema, so no tags will match
    tag_filter = Tag("some_tag") == "abc"

    response = cache_no_tags.check(
        prompt="this prompt has a tag",
        filter_expression=tag_filter,
    )
    assert response == []

    with pytest.raises((RedisModuleVersionError, ValueError)):
```

```
        cache_with_tags = SemanticCache(
            name="test_cache",
            redis_url=redis_url,
            filterable_fields=[{"name": "some_tag", "type": "tag"}],
        )

    cache_overwrite = SemanticCache(
        name="test_cache",
        redis_url=redis_url,
        filterable_fields=[{"name": "some_tag", "type": "tag"}],
        overwrite=True,
    )

    response = cache_overwrite.check(
        prompt="this prompt has a tag",
        filter_expression=tag_filter,
    )
    assert len(response) == 1


def test_no_key_collision_on_identical_prompts(redis_url):
    private_cache = SemanticCache(
        name="private_cache",
        redis_url=redis_url,
        filterable_fields=[
            {"name": "user_id", "type": "tag"},
            {"name": "zip_code", "type": "numeric"},
        ],
    )

    private_cache.store(
        prompt="What is the phone number linked to my account?",
        response="The number on file is 123-555-0000",
        filters={"user_id": "gabs"},
    )

    private_cache.store(
        prompt="What's the phone number linked in my account?",
        response="The number on file is 123-555-9999",
        filters={"user_id": "cerioni", "zip_code": 90210},
    )

    private_cache.store(
        prompt="What's the phone number linked in my account?",
        response="The number on file is 123-555-1111",
        filters={"user_id": "bart"},
    )

    results = private_cache.check(
        "What's the phone number linked in my account?", num_results=5
    )
    assert len(results) == 3

    zip_code_filter = Num("zip_code") != 90210
    filtered_results = private_cache.check(
        "what's the phone number linked in my account?",
        num_results=5,
        filter_expression=zip_code_filter,
    )
    assert len(filtered_results) == 2


def test_create_cache_with_different_vector_types():
```

```
    try:
        bfloat_cache = SemanticCache(name="bfloat_cache", dtype="bfloat16")
        bfloat_cache.store("bfloat16 prompt", "bfloat16 response")

        float16_cache = SemanticCache(name="float16_cache", dtype="float16")
        float16_cache.store("float16 prompt", "float16 response")

        float32_cache = SemanticCache(name="float32_cache", dtype="float32")
        float32_cache.store("float32 prompt", "float32 response")

        float64_cache = SemanticCache(name="float64_cache", dtype="float64")
        float64_cache.store("float64 prompt", "float64 response")

        for cache in [bfloat_cache, float16_cache, float32_cache, float64_cache]:
            cache.set_threshold(0.6)
            assert len(cache.check("float prompt", num_results=5)) == 1
    except:
        pytest.skip("Not using a late enough version of Redis")


def test_bad_dtype_connecting_to_existing_cache():
    try:
        cache = SemanticCache(name="float64_cache", dtype="float64")
        same_type = SemanticCache(name="float64_cache", dtype="float64")
        # under the hood uses from_existing
    except RedisModuleVersionError:
        pytest.skip("Not using a late enough version of Redis")

    with pytest.raises(ValueError):
        bad_type = SemanticCache(name="float64_cache", dtype="float16")
}
```

tests/integration/test_query.py

```
import pytest
from redis.commands.search.result import Result

from redisvl.index import SearchIndex
from redisvl.query import CountQuery, FilterQuery, RangeQuery, VectorQuery
from redisvl.query.filter import FilterExpression, Geo, GeoRadius, Num, Tag, Text
from redisvl.redis.utils import array_to_buffer

# TODO expand to multiple schema types and sync + async


@pytest.fixture
def vector_query():
    return VectorQuery(
        vector=[0.1, 0.1, 0.5],
        vector_field_name="user_embedding",
        return_fields=["user", "credit_score", "age", "job", "location"],
    )


@pytest.fixture
def sorted_vector_query():
    return VectorQuery(
        vector=[0.1, 0.1, 0.5],
        vector_field_name="user_embedding",
        return_fields=["user", "credit_score", "age", "job", "location"],
```

```
        sort_by="age",
    )


@pytest.fixture
def filter_query():
    return FilterQuery(
        return_fields=["user", "credit_score", "age", "job", "location"],
        filter_expression=Tag("credit_score") == "high",
    )


@pytest.fixture
def sorted_filter_query():
    return FilterQuery(
        return_fields=["user", "credit_score", "age", "job", "location"],
        filter_expression=Tag("credit_score") == "high",
        sort_by="age",
    )


@pytest.fixture
def range_query():
    return RangeQuery(
        vector=[0.1, 0.1, 0.5],
        vector_field_name="user_embedding",
        return_fields=["user", "credit_score", "age", "job", "location"],
        distance_threshold=0.2,
    )


@pytest.fixture
def sorted_range_query():
    return RangeQuery(
        vector=[0.1, 0.1, 0.5],
        vector_field_name="user_embedding",
        return_fields=["user", "credit_score", "age", "job", "location"],
        distance_threshold=0.2,
        sort_by="age",
    )


@pytest.fixture
def index(sample_data, redis_url):
    # construct a search index from the schema
    index = SearchIndex.from_dict(
        {
            "index": {
                "name": "user_index",
                "prefix": "v1",
                "storage_type": "hash",
            },
            "fields": [
                {"name": "credit_score", "type": "tag"},
                {"name": "job", "type": "text"},
                {"name": "age", "type": "numeric"},
                {"name": "location", "type": "geo"},
                {
                    "name": "user_embedding",
                    "type": "vector",
                    "attrs": {
                        "dims": 3,
                        "distance_metric": "cosine",
                        "algorithm": "flat",
```

```
                    "datatype": "float32",
                },
            },
        ],
    }
)

# connect to local redis instance
index.connect(redis_url)

# create the index (no data yet)
index.create(overwrite=True)

# Prepare and load the data
def hash_preprocess(item: dict) -> dict:
    return {
        **item,
        "user_embedding": array_to_buffer(item["user_embedding"], "float32"),
    }

index.load(sample_data, preprocess=hash_preprocess)

# run the test
yield index

# clean up
index.delete(drop=True)


def test_search_and_query(index):
    # *=>[KNN 7 @user_embedding $vector AS vector_distance]
    v = VectorQuery(
        [0.1, 0.1, 0.5],
        "user_embedding",
        return_fields=["user", "credit_score", "age", "job", "location"],
        num_results=7,
    )
    results = index.search(v.query, query_params=v.params)
    assert isinstance(results, Result)
    assert len(results.docs) == 7
    for doc in results.docs:
        # ensure all return fields present
        assert doc.user in [
            "john",
            "derrick",
            "nancy",
            "tyler",
            "tim",
            "taimur",
            "joe",
            "mary",
        ]
        assert int(doc.age) in [18, 14, 94, 100, 12, 15, 35]
        assert doc.job in ["engineer", "doctor", "dermatologist", "CEO", "dentist"]
        assert doc.credit_score in ["high", "low", "medium"]

    processed_results = index.query(v)
    assert len(processed_results) == 7
    assert isinstance(processed_results[0], dict)
    result = results.docs[0].__dict__
    result.pop("payload")
    assert processed_results[0] == results.docs[0].__dict__
```

```
def test_range_query(index):
    r = RangeQuery(
        vector=[0.1, 0.1, 0.5],
        vector_field_name="user_embedding",
        return_fields=["user", "credit_score", "age", "job"],
        distance_threshold=0.2,
        num_results=7,
    )
    results = index.query(r)
    for result in results:
        assert float(result["vector_distance"]) <= 0.2
    assert len(results) == 4
    assert r.distance_threshold == 0.2

    r.set_distance_threshold(0.1)
    assert r.distance_threshold == 0.1
    results = index.query(r)
    for result in results:
        assert float(result["vector_distance"]) <= 0.1
    assert len(results) == 2


def test_count_query(index, sample_data):
    c = CountQuery(FilterExpression("*"))
    results = index.query(c)
    assert results == len(sample_data)

    c = CountQuery(Tag("credit_score") == "high")
    results = index.query(c)
    assert results == 4


def search(
    query,
    index,
    _filter,
    expected_count,
    credit_check=None,
    age_range=None,
    location=None,
    distance_threshold=0.2,
    sort=False,
):
    """Utility function to test filters."""

    # set the new filter
    query.set_filter(_filter)
    print(str(query))

    results = index.search(query.query, query_params=query.params)

    # check for tag filter correctness
    if credit_check:
        for doc in results.docs:
            assert doc.credit_score == credit_check

    # check for numeric filter correctness
    if age_range:
        for doc in results.docs:
            if len(age_range) == 3:
                assert int(doc.age) != age_range[2]
            elif age_range[1] < age_range[0]:
                assert (int(doc.age) <= age_range[0]) or (int(doc.age) >= age_range[1])
            else:
```

```
                assert age_range[0] <= int(doc.age) <= age_range[1]

    # check for geographic filter correctness
    if location:
        for doc in results.docs:
            assert doc.location == location

    # if range query, test results by distance threshold
    if isinstance(query, RangeQuery):
        for doc in results.docs:
            print(doc.vector_distance)
            assert float(doc.vector_distance) <= distance_threshold

    # otherwise check by expected count.
    else:
        assert len(results.docs) == expected_count

    # check results are in sorted order
    if sort:
        if isinstance(query, RangeQuery):
            assert [int(doc.age) for doc in results.docs] == [12, 14, 18, 100]
        else:
            assert [int(doc.age) for doc in results.docs] == [
                12,
                14,
                15,
                18,
                35,
                94,
                100,
            ]


@pytest.fixture(
    params=["vector_query", "filter_query", "range_query"],
    ids=["VectorQuery", "FilterQuery", "RangeQuery"],
)
def query(request):
    return request.getfixturevalue(request.param)


def test_filters(index, query):
    # Simple Tag Filter
    t = Tag("credit_score") == "high"
    search(query, index, t, 4, credit_check="high")

    # Multiple Tags
    t = Tag("credit_score") == ["high", "low"]
    search(query, index, t, 6)

    # Empty tag filter
    t = Tag("credit_score") == []
    search(query, index, t, 7)

    # Simple Numeric Filter
    n1 = Num("age") >= 18
    search(query, index, n1, 4, age_range=(18, 100))

    # intersection of rules
    n2 = (Num("age") >= 18) & (Num("age") < 100)
    search(query, index, n2, 3, age_range=(18, 99))

    # union
    n3 = (Num("age") < 18) | (Num("age") > 94)
```

```
    search(query, index, n3, 4, age_range=(95, 17))

    n4 = Num("age") != 18
    search(query, index, n4, 6, age_range=(0, 0, 18))

    # Geographic filters
    g = Geo("location") == GeoRadius(-122.4194, 37.7749, 1, unit="m")
    search(query, index, g, 3, location="-122.4194,37.7749")

    g = Geo("location") != GeoRadius(-122.4194, 37.7749, 1, unit="m")
    search(query, index, g, 4, location="-110.0839,37.3861")

    # Text filters
    t = Text("job") == "engineer"
    search(query, index, t, 2)

    t = Text("job") != "engineer"
    search(query, index, t, 5)

    t = Text("job") % "enginee*"
    search(query, index, t, 2)

    t = Text("job") % "engine*|doctor"
    search(query, index, t, 4)

    t = Text("job") % "%%engine%%"
    search(query, index, t, 2)

    # Test empty filters
    t = Text("job") % ""
    search(query, index, t, 7)


def test_manual_string_filters(index, query):
    # Simple Tag Filter
    t = "@credit_score:{high}"
    search(query, index, t, 4, credit_check="high")

    # Multiple Tags
    t = "@credit_score:{high|low}"
    search(query, index, t, 6)

    # Simple Numeric Filter
    n1 = "@age:[18 +inf]"
    search(query, index, n1, 4, age_range=(18, 100))

    # intersection of rules
    n2 = "@age:[18 (100]"
    search(query, index, n2, 3, age_range=(18, 99))

    n3 = "(@age:[-inf (18] | @age:[(94 +inf])"
    search(query, index, n3, 4, age_range=(95, 17))

    n4 = "(-@age:[18 18])"
    search(query, index, n4, 6, age_range=(0, 0, 18))

    # Geographic filters
    g = "@location:[-122.4194 37.7749 1 m]"
    search(query, index, g, 3, location="-122.4194,37.7749")

    g = "(-@location:[-122.4194 37.7749 1 m])"
    search(query, index, g, 4, location="-110.0839,37.3861")

    # Text filters
```

```
    t = "@job:engineer"
    search(query, index, t, 2)

    t = "(-@job:engineer)"
    search(query, index, t, 5)

    t = "@job:enginee*"
    search(query, index, t, 2)

    t = "@job:(engine*|doctor)"
    search(query, index, t, 4)

    t = "@job:*engine*"
    search(query, index, t, 2)


def test_filter_combinations(index, query):
    # test combinations
    # intersection
    t = Tag("credit_score") == "high"
    text = Text("job") == "engineer"
    search(query, index, t & text, 2, credit_check="high")

    # union
    t = Tag("credit_score") == "high"
    text = Text("job") == "engineer"
    search(query, index, t | text, 4, credit_check="high")

    # union of negated expressions
    _filter = (Tag("credit_score") != "high") & (Text("job") != "engineer")
    search(query, index, _filter, 3)

    # geo + text
    g = Geo("location") == GeoRadius(-122.4194, 37.7749, 1, unit="m")
    text = Text("job") == "engineer"
    search(query, index, g & text, 1, location="-122.4194,37.7749")

    # geo + text
    g = Geo("location") != GeoRadius(-122.4194, 37.7749, 1, unit="m")
    text = Text("job") == "engineer"
    search(query, index, g & text, 1, location="-110.0839,37.3861")

    # num + text + geo
    n = (Num("age") >= 18) & (Num("age") < 100)
    t = Text("job") != "engineer"
    g = Geo("location") == GeoRadius(-122.4194, 37.7749, 1, unit="m")
    search(query, index, n & t & g, 1, age_range=(18,
99), location="-122.4194,37.7749")


def test_paginate_vector_query(index, vector_query, sample_data):
    batch_size = 2
    all_results = []
    for i, batch in enumerate(index.paginate(vector_query, batch_size), start=1):
        all_results.extend(batch)
        assert len(batch) <= batch_size

    expected_total_results = len(sample_data)
    expected_iterations = -(-expected_total_results // batch_size)  # Ceiling division
    assert len(all_results) == expected_total_results
    assert i == expected_iterations


def test_paginate_filter_query(index, filter_query):
```

```
    batch_size = 3
    all_results = []
    for i, batch in enumerate(index.paginate(filter_query, batch_size), start=1):
        all_results.extend(batch)
        assert len(batch) <= batch_size

    expected_count = 4  # Adjust based on your filter
    expected_iterations = -(-expected_count // batch_size)  # Ceiling division
    assert len(all_results) == expected_count
    assert i == expected_iterations
    assert all(item["credit_score"] == "high" for item in all_results)


def test_paginate_range_query(index, range_query):
    batch_size = 1
    all_results = []
    for i, batch in enumerate(index.paginate(range_query, batch_size), start=1):
        all_results.extend(batch)
        assert len(batch) <= batch_size

    expected_count = 4  # Adjust based on your range query
    expected_iterations = -(-expected_count // batch_size)  # Ceiling division
    assert len(all_results) == expected_count
    assert i == expected_iterations
    assert all(float(item["vector_distance"]) <= 0.2 for item in all_results)


def test_sort_filter_query(index, sorted_filter_query):
    t = Text("job") % ""
    search(sorted_filter_query, index, t, 7, sort=True)


def test_sort_vector_query(index, sorted_vector_query):
    t = Text("job") % ""
    search(sorted_vector_query, index, t, 7, sort=True)


def test_sort_range_query(index, sorted_range_query):
    t = Text("job") % ""
    search(sorted_range_query, index, t, 7, sort=True)


def test_query_with_chunk_number_zero():
    doc_base_id = "8675309"
    file_id = "e9ffbac9ff6f67cc"
    chunk_num = 0

    filter_conditions = (
        (Tag("doc_base_id") == doc_base_id)
        & (Tag("file_id") == file_id)
        & (Num("chunk_number") == chunk_num)
    )

    expected_query_str = (
        "((@doc_base_id:{8675309} @file_id:{e9ffbac9ff6f67cc}) @chunk_number:[0 0])"
    )
    assert (
        str(filter_conditions) == expected_query_str
    ), "Query with chunk_number zero is incorrect"
}
```

tests/integration/test_rerankers.py

```
import os

import pytest

from redisvl.utils.rerank import CohereReranker, HFCrossEncoderReranker


# Fixture for the reranker instance
@pytest.fixture
def cohereReranker():
    skip_reranker = os.getenv("SKIP_RERANKERS", "False").lower() == "true"
    if skip_reranker:
        pytest.skip("Skipping reranker instantiation...")
    return CohereReranker()


@pytest.fixture
def hfCrossEncoderReranker():
    return HFCrossEncoderReranker()


@pytest.fixture
def hfCrossEncoderRerankerWithCustomModel():
    return HFCrossEncoderReranker("cross-encoder/stsb-distilroberta-base")


# Test for basic ranking functionality
def test_rank_documents_cohere(cohereReranker):
    docs = ["document one", "document two", "document three"]
    query = "search query"

    reranked_docs, scores = cohereReranker.rank(query, docs)

    assert isinstance(reranked_docs, list)
    assert len(reranked_docs) == len(docs)  # Ensure we get back as many docs as
we sent
    assert all(isinstance(score, float) for score in scores)  # Scores should be floats


# Test for asynchronous ranking functionality
@pytest.mark.asyncio
async def test_async_rank_documents_cohere(cohereReranker):
    docs = ["document one", "document two", "document three"]
    query = "search query"

    reranked_docs, scores = await cohereReranker.arank(query, docs)

    assert isinstance(reranked_docs, list)
    assert len(reranked_docs) == len(docs)  # Ensure we get back as many docs as
we sent
    assert all(isinstance(score, float) for score in scores)  # Scores should be floats


# Test handling of bad input
def test_bad_input_cohere(cohereReranker):
    with pytest.raises(Exception):
        cohereReranker.rank("", [])  # Empty query or documents

    with pytest.raises(Exception):
        cohereReranker.rank(123, ["valid document"])  # Invalid type for query
```

```
    with pytest.raises(Exception):
        cohereReranker.rank("valid query", "not a list")  # Invalid type for documents

    with pytest.raises(Exception):
        cohereReranker.rank(
            "valid query", [{"field": "valid document"}], rank_by=["invalid_field"]
        )  # Invalid rank_by field


def test_rank_documents_cross_encoder(hfCrossEncoderReranker):
    query = "I love you"
    texts = ["I love you", "I like you", "I don't like you", "I hate you"]
    reranked_docs, scores = hfCrossEncoderReranker.rank(query, texts)

    for i in range(min(len(texts), hfCrossEncoderReranker.limit) - 1):
        assert scores[i] > scores[i + 1]


def test_rank_documents_cross_encoder_custom_model(
    hfCrossEncoderRerankerWithCustomModel,
):
    query = "I love you"
    texts = ["I love you", "I like you", "I don't like you", "I hate you"]
    reranked_docs, scores = hfCrossEncoderRerankerWithCustomModel.rank(query, texts)

    for i in range(min(len(texts), hfCrossEncoderRerankerWithCustomModel.limit) - 1):
        assert scores[i] > scores[i + 1]


@pytest.mark.asyncio
async def test_async_rank_cross_encoder(hfCrossEncoderReranker):
    docs = ["document one", "document two", "document three"]
    query = "search query"

    reranked_docs, scores = await hfCrossEncoderReranker.arank(query, docs)

    assert isinstance(reranked_docs, list)
    assert len(reranked_docs) == len(docs)  # Ensure we get back as many docs as
we sent
    assert all(isinstance(score, float) for score in scores)  # Scores should be floats
}
```

tests/integration/test_search_index.py

```
import pytest

from redisvl.exceptions import RedisSearchError
from redisvl.index import SearchIndex
from redisvl.query import VectorQuery
from redisvl.redis.connection import RedisConnectionFactory, validate_modules
from redisvl.redis.utils import convert_bytes
from redisvl.schema import IndexSchema, StorageType

fields = [{"name": "test", "type": "tag"}]


@pytest.fixture
def index_schema():
    return IndexSchema.from_dict({"index": {"name": "my_index"}, "fields": fields})
```

```
@pytest.fixture
def index(index_schema):
    return SearchIndex(schema=index_schema)


@pytest.fixture
def index_from_dict():
    return SearchIndex.from_dict({"index": {"name": "my_index"}, "fields": fields})


@pytest.fixture
def index_from_yaml():
    return SearchIndex.from_yaml("schemas/test_json_schema.yaml")


def test_search_index_properties(index_schema, index):
    assert index.schema == index_schema
    # custom settings
    assert index.name == index_schema.index.name == "my_index"
    assert index.client == None
    # default settings
    assert index.prefix == index_schema.index.prefix == "rvl"
    assert index.key_separator == index_schema.index.key_separator == ":"
    assert index.storage_type == index_schema.index.storage_type == StorageType.HASH
    assert index.key("foo").startswith(index.prefix)


def test_search_index_from_yaml(index_from_yaml):
    assert index_from_yaml.name == "json-test"
    assert index_from_yaml.client == None
    assert index_from_yaml.prefix == "json"
    assert index_from_yaml.key_separator == ":"
    assert index_from_yaml.storage_type == StorageType.JSON
    assert index_from_yaml.key("foo").startswith(index_from_yaml.prefix)


def test_search_index_from_dict(index_from_dict):
    assert index_from_dict.name == "my_index"
    assert index_from_dict.client == None
    assert index_from_dict.prefix == "rvl"
    assert index_from_dict.key_separator == ":"
    assert index_from_dict.storage_type == StorageType.HASH
    assert len(index_from_dict.schema.fields) == len(fields)
    assert index_from_dict.key("foo").startswith(index_from_dict.prefix)


def test_search_index_from_existing(client, index):
    index.set_client(client)
    index.create(overwrite=True)

    try:
        index2 = SearchIndex.from_existing(index.name, redis_client=client)
    except Exception as e:
        pytest.skip(str(e))

    assert index2.schema == index.schema


def test_search_index_from_existing_complex(client):
    schema = {
        "index": {
            "name": "test",
            "prefix": "test",
```

```
            "storage_type": "json",
        },
        "fields": [
            {"name": "user", "type": "tag", "path": "$.user"},
            {"name": "credit_score", "type": "tag", "path": "$.metadata.credit_score"},
            {"name": "job", "type": "text", "path": "$.metadata.job"},
            {
                "name": "age",
                "type": "numeric",
                "path": "$.metadata.age",
                "attrs": {"sortable": False},
            },
            {
                "name": "user_embedding",
                "type": "vector",
                "attrs": {
                    "dims": 3,
                    "distance_metric": "cosine",
                    "algorithm": "flat",
                    "datatype": "float32",
                },
            },
        ],
    }
    index = SearchIndex.from_dict(schema, redis_client=client)
    index.create(overwrite=True)

    try:
        index2 = SearchIndex.from_existing(index.name, redis_client=client)
    except Exception as e:
        pytest.skip(str(e))

    assert index.schema == index2.schema


def test_search_index_no_prefix(index_schema):
    # specify an explicitly empty prefix...
    index_schema.index.prefix = ""
    index = SearchIndex(schema=index_schema)
    assert index.prefix == ""
    assert index.key("foo") == "foo"


def test_search_index_redis_url(redis_url, index_schema):
    index = SearchIndex(schema=index_schema, redis_url=redis_url)
    assert index.client

    index.disconnect()
    assert index.client == None


def test_search_index_client(client, index_schema):
    index = SearchIndex(schema=index_schema, redis_client=client)
    assert index.client == client


def test_search_index_set_client(async_client, client, index):
    index.set_client(client)
    assert index.client == client
    # should not be able to set the sync client here
    with pytest.raises(TypeError):
        index.set_client(async_client)

    index.disconnect()
```

```
    assert index.client == None


def test_search_index_create(client, index):
    index.set_client(client)
    index.create(overwrite=True, drop=True)
    assert index.exists()
    assert index.name in convert_bytes(index.client.execute_command("FT._LIST"))


def test_search_index_delete(client, index):
    index.set_client(client)
    index.create(overwrite=True, drop=True)
    index.delete(drop=True)
    assert not index.exists()
    assert index.name not in convert_bytes(index.client.execute_command("FT._LIST"))


def test_search_index_clear(client, index):
    index.set_client(client)
    index.create(overwrite=True, drop=True)
    data = [{"id": "1", "test": "foo"}]
    index.load(data, id_field="id")

    count = index.clear()
    assert count == len(data)
    assert index.exists()


def test_search_index_drop_key(client, index):
    index.set_client(client)
    index.create(overwrite=True, drop=True)
    data = [{"id": "1", "test": "foo"}, {"id": "2", "test": "bar"}]
    keys = index.load(data, id_field="id")

    # test passing a single string key removes only that key
    dropped = index.drop_keys(keys[0])
    assert dropped == 1
    assert not index.fetch(keys[0])
    assert index.fetch(keys[1]) is not None  # still have all other entries


def test_search_index_drop_keys(client, index):
    index.set_client(client)
    index.create(overwrite=True, drop=True)
    data = [
        {"id": "1", "test": "foo"},
        {"id": "2", "test": "bar"},
        {"id": "3", "test": "baz"},
    ]
    keys = index.load(data, id_field="id")

    # test passing a list of keys selectively removes only those keys
    dropped = index.drop_keys(keys[0:2])
    assert dropped == 2
    assert not index.fetch(keys[0])
    assert not index.fetch(keys[1])
    assert index.fetch(keys[2]) is not None

    assert index.exists()


def test_search_index_load_and_fetch(client, index):
    index.set_client(client)
```

```
    index.create(overwrite=True, drop=True)
    data = [{"id": "1", "test": "foo"}]
    index.load(data, id_field="id")

    res = index.fetch("1")
    assert res["test"] == convert_bytes(client.hget("rvl:1", "test")) == "foo"

    index.delete(drop=True)
    assert not index.exists()
    assert not index.fetch("1")


def test_search_index_load_preprocess(client, index):
    index.set_client(client)
    index.create(overwrite=True, drop=True)
    data = [{"id": "1", "test": "foo"}]

    def preprocess(record):
        record["test"] = "bar"
        return record

    index.load(data, id_field="id", preprocess=preprocess)
    res = index.fetch("1")
    assert res["test"] == convert_bytes(client.hget("rvl:1", "test")) == "bar"

    def bad_preprocess(record):
        return 1

    with pytest.raises(TypeError):
        index.load(data, id_field="id", preprocess=bad_preprocess)


def test_no_id_field(client, index):
    index.set_client(client)
    index.create(overwrite=True, drop=True)
    bad_data = [{"wrong_key": "1", "value": "test"}]

    # catch missing / invalid id_field
    with pytest.raises(ValueError):
        index.load(bad_data, id_field="key")


def test_check_index_exists_before_delete(client, index):
    index.set_client(client)
    index.create(overwrite=True, drop=True)
    index.delete(drop=True)
    with pytest.raises(RedisSearchError):
        index.delete()


def test_check_index_exists_before_search(client, index):
    index.set_client(client)
    index.create(overwrite=True, drop=True)
    index.delete(drop=True)

    query = VectorQuery(
        [0.1, 0.1, 0.5],
        "user_embedding",
        return_fields=["user", "credit_score", "age", "job", "location"],
        num_results=7,
    )
    with pytest.raises(RedisSearchError):
        index.search(query.query, query_params=query.params)
```

```
def test_check_index_exists_before_info(client, index):
    index.set_client(client)
    index.create(overwrite=True, drop=True)
    index.delete(drop=True)

    with pytest.raises(RedisSearchError):
        index.info()


def test_index_needs_valid_schema():
    with pytest.raises(ValueError, match=r"Must provide a valid IndexSchema object"):
        index = SearchIndex(schema="Not A Valid Schema")
}
```

tests/integration/test_search_results.py

```
import os

import pytest

from redisvl.index import SearchIndex
from redisvl.query import FilterQuery
from redisvl.query.filter import Tag


@pytest.fixture
def filter_query():
    return FilterQuery(
        return_fields=None,
        filter_expression=Tag("credit_score") == "high",
    )


@pytest.fixture
def index(sample_data):
    fields_spec = [
        {"name": "credit_score", "type": "tag"},
        {"name": "user", "type": "tag"},
        {"name": "job", "type": "text"},
        {"name": "age", "type": "numeric"},
        {
            "name": "user_embedding",
            "type": "vector",
            "attrs": {
                "dims": 3,
                "distance_metric": "cosine",
                "algorithm": "flat",
                "datatype": "float32",
            },
        },
    ]

    json_schema = {
        "index": {
            "name": "user_index_json",
            "prefix": "users_json",
            "storage_type": "json",
        },
        "fields": fields_spec,
```

```
    }

    # construct a search index from the schema
    index = SearchIndex.from_dict(json_schema)

    # connect to local redis instance
    index.connect(os.environ["REDIS_URL"])

    # create the index (no data yet)
    index.create(overwrite=True)

    # Prepare and load the data
    index.load(sample_data)

    # run the test
    yield index

    # clean up
    index.delete(drop=True)


def test_process_results_unpacks_json_properly(index, filter_query):
    results = index.query(filter_query)
    assert len(results) == 4
}
```

tests/integration/test_semantic_router.py

```
import os
import pathlib

import pytest
from redis.exceptions import ConnectionError

from redisvl.exceptions import RedisModuleVersionError
from redisvl.extensions.router import SemanticRouter
from redisvl.extensions.router.schema import Route, RoutingConfig
from redisvl.redis.connection import compare_versions


def get_base_path():
    return pathlib.Path(__file__).parent.resolve()


@pytest.fixture
def routes():
    return [
        Route(
            name="greeting",
            references=["hello", "hi"],
            metadata={"type": "greeting"},
            distance_threshold=0.3,
        ),
        Route(
            name="farewell",
            references=["bye", "goodbye"],
            metadata={"type": "farewell"},
            distance_threshold=0.3,
        ),
    ]
```

```
@pytest.fixture
def semantic_router(client, routes):
    router = SemanticRouter(
        name="test-router",
        routes=routes,
        routing_config=RoutingConfig(distance_threshold=0.3, max_k=2),
        redis_client=client,
        overwrite=False,
    )
    yield router
    router.delete()


def test_initialize_router(semantic_router):
    assert semantic_router.name == "test-router"
    assert len(semantic_router.routes) == 2
    assert semantic_router.routing_config.distance_threshold == 0.3
    assert semantic_router.routing_config.max_k == 2


def test_router_properties(semantic_router):
    route_names = semantic_router.route_names
    assert "greeting" in route_names
    assert "farewell" in route_names

    thresholds = semantic_router.route_thresholds
    assert thresholds["greeting"] == 0.3
    assert thresholds["farewell"] == 0.3


def test_get_route(semantic_router):
    route = semantic_router.get("greeting")
    assert route is not None
    assert route.name == "greeting"
    assert "hello" in route.references


def test_get_non_existing_route(semantic_router):
    route = semantic_router.get("non_existent_route")
    assert route is None


def test_single_query(semantic_router):
    redis_version = semantic_router._index.client.info()["redis_version"]
    if not compare_versions(redis_version, "7.0.0"):
        pytest.skip("Not using a late enough version of Redis")

    match = semantic_router("hello")
    assert match.name == "greeting"
    assert match.distance <= semantic_router.route_thresholds["greeting"]


def test_single_query_no_match(semantic_router):
    redis_version = semantic_router._index.client.info()["redis_version"]
    if not compare_versions(redis_version, "7.0.0"):
        pytest.skip("Not using a late enough version of Redis")

    match = semantic_router("unknown_phrase")
    assert match.name is None


def test_multiple_query(semantic_router):
```

```
    redis_version = semantic_router._index.client.info()["redis_version"]
    if not compare_versions(redis_version, "7.0.0"):
        pytest.skip("Not using a late enough version of Redis")

    matches = semantic_router.route_many("hello", max_k=2)
    assert len(matches) > 0
    assert matches[0].name == "greeting"


def test_update_routing_config(semantic_router):
    new_config = RoutingConfig(distance_threshold=0.5, max_k=1)
    semantic_router.update_routing_config(new_config)
    assert semantic_router.routing_config.distance_threshold == 0.5
    assert semantic_router.routing_config.max_k == 1


def test_vector_query(semantic_router):
    redis_version = semantic_router._index.client.info()["redis_version"]
    if not compare_versions(redis_version, "7.0.0"):
        pytest.skip("Not using a late enough version of Redis")

    vector = semantic_router.vectorizer.embed("goodbye")
    match = semantic_router(vector=vector)
    assert match.name == "farewell"


def test_vector_query_no_match(semantic_router):
    redis_version = semantic_router._index.client.info()["redis_version"]
    if not compare_versions(redis_version, "7.0.0"):
        pytest.skip("Not using a late enough version of Redis")

    vector = [
        0.0
    ] * semantic_router.vectorizer.dims  # Random vector unlikely to match any route
    match = semantic_router(vector=vector)
    assert match.name is None


def test_add_route(semantic_router):
    new_routes = [
        Route(
            name="politics",
            references=[
                "are you liberal or conservative?",
                "who will you vote for?",
                "political speech",
            ],
            metadata={"type": "greeting"},
        )
    ]
    semantic_router._add_routes(new_routes)

    route = semantic_router.get("politics")
    assert route is not None
    assert route.name == "politics"
    assert "political speech" in route.references

    redis_version = semantic_router._index.client.info()["redis_version"]
    if compare_versions(redis_version, "7.0.0"):
        match = semantic_router("political speech")
        print(match, flush=True)
        assert match is not None
        assert match.name == "politics"
```

```
def test_remove_routes(semantic_router):
    semantic_router.remove_route("greeting")
    assert semantic_router.get("greeting") is None

    semantic_router.remove_route("unknown_route")
    assert semantic_router.get("unknown_route") is None


def test_to_dict(semantic_router):
    router_dict = semantic_router.to_dict()
    assert router_dict["name"] == semantic_router.name
    assert len(router_dict["routes"]) == len(semantic_router.routes)
    assert router_dict["vectorizer"]["type"] == semantic_router.vectorizer.type


def test_from_dict(semantic_router):
    router_dict = semantic_router.to_dict()
    new_router = SemanticRouter.from_dict(
        router_dict, redis_client=semantic_router._index.client, overwrite=True
    )
    assert new_router == semantic_router


def test_to_yaml(semantic_router):
    yaml_file = str(get_base_path().joinpath("../../schemas/semantic_router.yaml"))
    semantic_router.to_yaml(yaml_file, overwrite=True)
    assert pathlib.Path(yaml_file).exists()


def test_from_yaml(semantic_router):
    yaml_file = str(get_base_path().joinpath("../../schemas/semantic_router.yaml"))
    new_router = SemanticRouter.from_yaml(
        yaml_file, redis_client=semantic_router._index.client, overwrite=True
    )
    assert new_router == semantic_router


def test_to_dict_missing_fields():
    data = {
        "name": "incomplete-router",
        "routes": [],
        "vectorizer": {"type": "HFTextVectorizer", "model": "bert-base-uncased"},
    }
    with pytest.raises(ValueError):
        SemanticRouter.from_dict(data)


def test_invalid_vectorizer():
    data = {
        "name": "invalid-router",
        "routes": [],
        "vectorizer": {"type": "InvalidVectorizer", "model": "invalid-model"},
        "routing_config": {},
    }
    with pytest.raises(ValueError):
        SemanticRouter.from_dict(data)


def test_yaml_invalid_file_path():
    with pytest.raises(FileNotFoundError):
        SemanticRouter.from_yaml("invalid_path.yaml", redis_client=None)
```

```
def test_idempotent_to_dict(semantic_router):
    router_dict = semantic_router.to_dict()
    new_router = SemanticRouter.from_dict(
        router_dict, redis_client=semantic_router._index.client, overwrite=True
    )
    assert new_router.to_dict() == router_dict


def test_bad_connection_info(routes):
    with pytest.raises(ConnectionError):
        SemanticRouter(
            name="test-router",
            routes=routes,
            routing_config=RoutingConfig(distance_threshold=0.3, max_k=2),
            redis_url="redis://localhost:6389",  # bad connection url
            overwrite=False,
        )


def test_different_vector_dtypes(routes):
    try:
        bfloat_router = SemanticRouter(
            name="bfloat_router",
            routes=routes,
            dtype="bfloat16",
        )

        float16_router = SemanticRouter(
            name="float16_router",
            routes=routes,
            dtype="float16",
        )

        float32_router = SemanticRouter(
            name="float32_router",
            routes=routes,
            dtype="float32",
        )

        float64_router = SemanticRouter(
            name="float64_router",
            routes=routes,
            dtype="float64",
        )

        for router in [bfloat_router, float16_router, float32_router, float64_router]:
            assert len(router.route_many("hello", max_k=5)) == 1
    except:
        pytest.skip("Not using a late enough version of Redis")


def test_bad_dtype_connecting_to_exiting_router(routes):
    try:
        router = SemanticRouter(
            name="float64 router",
            routes=routes,
            dtype="float64",
        )

        same_type = SemanticRouter(
            name="float64 router",
            routes=routes,
            dtype="float64",
        )
```

```
        # under the hood uses from_existing
    except RedisModuleVersionError:
        pytest.skip("Not using a late enough version of Redis")

    with pytest.raises(ValueError):
        bad_type = SemanticRouter(
            name="float64 router",
            routes=routes,
            dtype="float16",
        )
}
```

tests/integration/test_session_manager.py

```
import os

import pytest
from redis.exceptions import ConnectionError

from redisvl.exceptions import RedisModuleVersionError
from redisvl.extensions.constants import ID_FIELD_NAME
from redisvl.extensions.session_manager import (
    SemanticSessionManager,
    StandardSessionManager,
)


@pytest.fixture
def standard_session(app_name, client):
    session = StandardSessionManager(app_name, redis_client=client)
    yield session
    session.clear()


@pytest.fixture
def semantic_session(app_name, client):
    session = SemanticSessionManager(app_name, redis_client=client, overwrite=True)
    yield session
    session.clear()
    session.delete()


# test standard session manager
def test_specify_redis_client(client):
    session = StandardSessionManager(name="test_app", redis_client=client)
    assert isinstance(session._index.client, type(client))


def test_specify_redis_url(client, redis_url):
    session = StandardSessionManager(
        name="test_app",
        session_tag="abc",
        redis_url=redis_url,
    )
    assert isinstance(session._index.client, type(client))


def test_standard_bad_connection_info():
    with pytest.raises(ConnectionError):
        StandardSessionManager(
```

```
            name="test_app",
            session_tag="abc",
            redis_url="redis://localhost:6389",  # bad url
        )


def test_standard_store(standard_session):
    context = standard_session.get_recent()
    assert len(context) == 0

    standard_session.store(prompt="first prompt", response="first response")
    standard_session.store(prompt="second prompt", response="second response")
    standard_session.store(prompt="third prompt", response="third response")
    standard_session.store(prompt="fourth prompt", response="fourth response")
    standard_session.store(prompt="fifth prompt", response="fifth response")

    # test that order is maintained
    full_context = standard_session.get_recent(top_k=10)
    assert full_context == [
        {"role": "user", "content": "first prompt"},
        {"role": "llm", "content": "first response"},
        {"role": "user", "content": "second prompt"},
        {"role": "llm", "content": "second response"},
        {"role": "user", "content": "third prompt"},
        {"role": "llm", "content": "third response"},
        {"role": "user", "content": "fourth prompt"},
        {"role": "llm", "content": "fourth response"},
        {"role": "user", "content": "fifth prompt"},
        {"role": "llm", "content": "fifth response"},
    ]


def test_standard_add_and_get(standard_session):
    context = standard_session.get_recent()
    assert len(context) == 0

    standard_session.add_message({"role": "user", "content": "first prompt"})
    standard_session.add_message({"role": "llm", "content": "first response"})
    standard_session.add_message({"role": "user", "content": "second prompt"})
    standard_session.add_message({"role": "llm", "content": "second response"})
    standard_session.add_message(
        {
            "role": "tool",
            "content": "tool result 1",
            "tool_call_id": "tool call one",
        }
    )
    standard_session.add_message(
        {
            "role": "tool",
            "content": "tool result 2",
            "tool_call_id": "tool call two",
        }
    )
    standard_session.add_message({"role": "user", "content": "third prompt"})
    standard_session.add_message({"role": "llm", "content": "third response"})

    # test default context history size
    default_context = standard_session.get_recent()
    assert len(default_context) == 5  #  default is 5

    # test specified context history size
    partial_context = standard_session.get_recent(top_k=3)
    assert len(partial_context) == 3
```

```
    assert partial_context == [
        {"role": "tool", "content": "tool result 2", "tool_call_id": "tool call two"},
        {"role": "user", "content": "third prompt"},
        {"role": "llm", "content": "third response"},
    ]

    # test that order is maintained
    full_context = standard_session.get_recent(top_k=10)
    assert full_context == [
        {"role": "user", "content": "first prompt"},
        {"role": "llm", "content": "first response"},
        {"role": "user", "content": "second prompt"},
        {"role": "llm", "content": "second response"},
        {"role": "tool", "content": "tool result 1", "tool_call_id": "tool call one"},
        {"role": "tool", "content": "tool result 2", "tool_call_id": "tool call two"},
        {"role": "user", "content": "third prompt"},
        {"role": "llm", "content": "third response"},
    ]

    # test that a ValueError is raised when top_k is invalid
    with pytest.raises(ValueError):
        bad_context = standard_session.get_recent(top_k=-2)

    with pytest.raises(ValueError):
        bad_context = standard_session.get_recent(top_k=-2.0)

    with pytest.raises(ValueError):
        bad_context = standard_session.get_recent(top_k=1.3)

    with pytest.raises(ValueError):
        bad_context = standard_session.get_recent(top_k="3")


def test_standard_add_messages(standard_session):
    context = standard_session.get_recent()
    assert len(context) == 0

    standard_session.add_messages(
        [
            {"role": "user", "content": "first prompt"},
            {"role": "llm", "content": "first response"},
            {"role": "user", "content": "second prompt"},
            {"role": "llm", "content": "second response"},
            {
                "role": "tool",
                "content": "tool result 1",
                "tool_call_id": "tool call one",
            },
            {
                "role": "tool",
                "content": "tool result 2",
                "tool_call_id": "tool call two",
            },
            {"role": "user", "content": "fourth prompt"},
            {"role": "llm", "content": "fourth response"},
        ]
    )

    full_context = standard_session.get_recent(top_k=10)
    assert len(full_context) == 8
    assert full_context == [
        {"role": "user", "content": "first prompt"},
        {"role": "llm", "content": "first response"},
        {"role": "user", "content": "second prompt"},
```

```
        {"role": "llm", "content": "second response"},
        {"role": "tool", "content": "tool result 1", "tool_call_id": "tool call one"},
        {"role": "tool", "content": "tool result 2", "tool_call_id": "tool call two"},
        {"role": "user", "content": "fourth prompt"},
        {"role": "llm", "content": "fourth response"},
    ]


def test_standard_messages_property(standard_session):
    standard_session.add_messages(
        [
            {"role": "user", "content": "first prompt"},
            {"role": "llm", "content": "first response"},
            {"role": "user", "content": "second prompt"},
            {"role": "llm", "content": "second response"},
            {"role": "user", "content": "third prompt"},
        ]
    )

    assert standard_session.messages == [
        {"role": "user", "content": "first prompt"},
        {"role": "llm", "content": "first response"},
        {"role": "user", "content": "second prompt"},
        {"role": "llm", "content": "second response"},
        {"role": "user", "content": "third prompt"},
    ]


def test_standard_scope(standard_session):
    # store entries under default session tag
    standard_session.store("some prompt", "some response")

    # test that changing session tag does indeed change access scope
    new_session = "def"
    standard_session.store(
        "new user prompt", "new user response", session_tag=new_session
    )
    context = standard_session.get_recent(session_tag=new_session)
    assert context == [
        {"role": "user", "content": "new user prompt"},
        {"role": "llm", "content": "new user response"},
    ]

    # test that default session data is still accessible
    context = standard_session.get_recent()
    assert context == [
        {"role": "user", "content": "some prompt"},
        {"role": "llm", "content": "some response"},
    ]

    bad_session = "xyz"
    no_context = standard_session.get_recent(session_tag=bad_session)
    assert no_context == []


def test_standard_get_text(standard_session):
    standard_session.store("first prompt", "first response")
    text = standard_session.get_recent(as_text=True)
    assert text == ["first prompt", "first response"]

    standard_session.add_message({"role": "system", "content": "system level prompt"})
    text = standard_session.get_recent(as_text=True)
    assert text == ["first prompt", "first response", "system level prompt"]
```

```
def test_standard_get_raw(standard_session):
    standard_session.store("first prompt", "first response")
    standard_session.store("second prompt", "second response")
    raw = standard_session.get_recent(raw=True)
    assert len(raw) == 4
    assert raw[0]["role"] == "user"
    assert raw[0]["content"] == "first prompt"
    assert raw[1]["role"] == "llm"
    assert raw[1]["content"] == "first response"


def test_standard_drop(standard_session):
    standard_session.store("first prompt", "first response")
    standard_session.store("second prompt", "second response")
    standard_session.store("third prompt", "third response")
    standard_session.store("fourth prompt", "fourth response")

    # test drop() with no arguments removes the last element
    standard_session.drop()
    context = standard_session.get_recent(top_k=3)
    assert context == [
        {"role": "user", "content": "third prompt"},
        {"role": "llm", "content": "third response"},
        {"role": "user", "content": "fourth prompt"},
    ]

    # test drop(id) removes the specified element
    context = standard_session.get_recent(top_k=10, raw=True)
    middle_id = context[3][ID_FIELD_NAME]
    standard_session.drop(middle_id)
    context = standard_session.get_recent(top_k=6)
    assert context == [
        {"role": "user", "content": "first prompt"},
        {"role": "llm", "content": "first response"},
        {"role": "user", "content": "second prompt"},
        {"role": "user", "content": "third prompt"},
        {"role": "llm", "content": "third response"},
        {"role": "user", "content": "fourth prompt"},
    ]


def test_standard_clear(standard_session):
    standard_session.store("some prompt", "some response")
    standard_session.clear()
    empty_context = standard_session.get_recent(top_k=10)
    assert empty_context == []


# test semantic session manager
def test_semantic_specify_client(client):
    session = SemanticSessionManager(
        name="test_app", session_tag="abc", redis_client=client, overwrite=True
    )
    assert isinstance(session._index.client, type(client))


def test_semantic_bad_connection_info():
    with pytest.raises(ConnectionError):
        SemanticSessionManager(
            name="test_app",
            session_tag="abc",
            redis_url="redis://localhost:6389",
        )
```

```
def test_semantic_scope(semantic_session):
    # store entries under default session tag
    semantic_session.store("some prompt", "some response")

    # test that changing session tag does indeed change access scope
    new_session = "def"
    semantic_session.store(
        "new user prompt", "new user response", session_tag=new_session
    )
    context = semantic_session.get_recent(session_tag=new_session)
    assert context == [
        {"role": "user", "content": "new user prompt"},
        {"role": "llm", "content": "new user response"},
    ]

    # test that previous session data is still accessible
    context = semantic_session.get_recent()
    assert context == [
        {"role": "user", "content": "some prompt"},
        {"role": "llm", "content": "some response"},
    ]

    bad_session = "xyz"
    no_context = semantic_session.get_recent(session_tag=bad_session)
    assert no_context == []


def test_semantic_store_and_get_recent(semantic_session):
    context = semantic_session.get_recent()
    assert len(context) == 0

    semantic_session.store(prompt="first prompt", response="first response")
    semantic_session.store(prompt="second prompt", response="second response")
    semantic_session.store(prompt="third prompt", response="third response")
    semantic_session.store(prompt="fourth prompt", response="fourth response")
    semantic_session.add_message(
        {"role": "tool", "content": "tool result", "tool_call_id": "tool id"}
    )
    # test default context history size
    default_context = semantic_session.get_recent()
    assert len(default_context) == 5  # 5 is default

    # test specified context history size
    partial_context = semantic_session.get_recent(top_k=4)
    assert len(partial_context) == 4

    # test larger context history returns full history
    too_large_context = semantic_session.get_recent(top_k=100)
    assert len(too_large_context) == 9

    # test that order is maintained
    full_context = semantic_session.get_recent(top_k=9)
    assert full_context == [
        {"role": "user", "content": "first prompt"},
        {"role": "llm", "content": "first response"},
        {"role": "user", "content": "second prompt"},
        {"role": "llm", "content": "second response"},
        {"role": "user", "content": "third prompt"},
        {"role": "llm", "content": "third response"},
        {"role": "user", "content": "fourth prompt"},
        {"role": "llm", "content": "fourth response"},
        {"role": "tool", "content": "tool result", "tool_call_id": "tool id"},
```

```
    ]

    # test that more recent entries are returned
    context = semantic_session.get_recent(top_k=4)
    assert context == [
        {"role": "llm", "content": "third response"},
        {"role": "user", "content": "fourth prompt"},
        {"role": "llm", "content": "fourth response"},
        {"role": "tool", "content": "tool result", "tool_call_id": "tool id"},
    ]

    # test no entries are returned and no error is raised if top_k == 0
    context = semantic_session.get_recent(top_k=0)
    assert context == []

    # test that a ValueError is raised when top_k is invalid
    with pytest.raises(ValueError):
        bad_context = semantic_session.get_recent(top_k=0.5)

    with pytest.raises(ValueError):
        bad_context = semantic_session.get_recent(top_k=-1)

    with pytest.raises(ValueError):
        bad_context = semantic_session.get_recent(top_k=-2.0)

    with pytest.raises(ValueError):
        bad_context = semantic_session.get_recent(top_k=1.3)

    with pytest.raises(ValueError):
        bad_context = semantic_session.get_recent(top_k="3")


def test_semantic_messages_property(semantic_session):
    semantic_session.add_messages(
        [
            {"role": "user", "content": "first prompt"},
            {"role": "llm", "content": "first response"},
            {
                "role": "tool",
                "content": "tool result 1",
                "tool_call_id": "tool call one",
            },
            {
                "role": "tool",
                "content": "tool result 2",
                "tool_call_id": "tool call two",
            },
            {"role": "user", "content": "second prompt"},
            {"role": "llm", "content": "second response"},
            {"role": "user", "content": "third prompt"},
        ]
    )

    assert semantic_session.messages == [
        {"role": "user", "content": "first prompt"},
        {"role": "llm", "content": "first response"},
        {"role": "tool", "content": "tool result 1", "tool_call_id": "tool call one"},
        {"role": "tool", "content": "tool result 2", "tool_call_id": "tool call two"},
        {"role": "user", "content": "second prompt"},
        {"role": "llm", "content": "second response"},
        {"role": "user", "content": "third prompt"},
    ]
```

```
def test_semantic_add_and_get_relevant(semantic_session):
    semantic_session.add_message(
        {"role": "system", "content": "discussing common fruits and vegetables"}
    )
    semantic_session.store(
        prompt="list of common fruits",
        response="apples, oranges, bananas, strawberries",
    )
    semantic_session.store(
        prompt="list of common vegetables",
        response="carrots, broccoli, onions, spinach",
    )
    semantic_session.store(
        prompt="winter sports in the olympics",
        response="downhill skiing, ice skating, luge",
    )
    semantic_session.add_message(
        {
            "role": "tool",
            "content": "skiing, skating, luge",
            "tool_call_id": "winter_sports()",
        }
    )

    # test default distance metric
    default_context = semantic_session.get_relevant(
        "set of common fruits like apples and bananas"
    )
    assert len(default_context) == 2
    assert default_context[0] == {"role": "user", "content": "list of common fruits"}
    assert default_context[1] == {
        "role": "llm",
        "content": "apples, oranges, bananas, strawberries",
    }

    # test increasing distance metric broadens results
    semantic_session.set_distance_threshold(0.5)
    default_context = semantic_session.get_relevant("list of fruits and vegetables")
    assert len(default_context) == 5  # 2 pairs of prompt:response, and system
    assert default_context == semantic_session.get_relevant(
        "list of fruits and vegetables", distance_threshold=0.5
    )

    # test tool calls can also be returned
    context = semantic_session.get_relevant("winter sports like skiing")
    assert context == [
        {
            "role": "user",
            "content": "winter sports in the olympics",
        },
        {
            "role": "tool",
            "content": "skiing, skating, luge",
            "tool_call_id": "winter_sports()",
        },
        {
            "role": "llm",
            "content": "downhill skiing, ice skating, luge",
        },
    ]

    # test that a ValueError is raised when top_k is invalid
    with pytest.raises(ValueError):
        bad_context = semantic_session.get_relevant("test prompt", top_k=-1)
```

```
    with pytest.raises(ValueError):
        bad_context = semantic_session.get_relevant("test prompt", top_k=-2.0)

    with pytest.raises(ValueError):
        bad_context = semantic_session.get_relevant("test prompt", top_k=1.3)

    with pytest.raises(ValueError):
        bad_context = semantic_session.get_relevant("test prompt", top_k="3")


def test_semantic_get_raw(semantic_session):
    semantic_session.store("first prompt", "first response")
    semantic_session.store("second prompt", "second response")
    raw = semantic_session.get_recent(raw=True)
    assert len(raw) == 4
    assert raw[0]["role"] == "user"
    assert raw[0]["content"] == "first prompt"
    assert raw[1]["role"] == "llm"
    assert raw[1]["content"] == "first response"


def test_semantic_drop(semantic_session):
    semantic_session.store("first prompt", "first response")
    semantic_session.store("second prompt", "second response")
    semantic_session.store("third prompt", "third response")
    semantic_session.store("fourth prompt", "fourth response")

    # test drop() with no arguments removes the last element
    semantic_session.drop()
    context = semantic_session.get_recent(top_k=3)
    assert context == [
        {"role": "user", "content": "third prompt"},
        {"role": "llm", "content": "third response"},
        {"role": "user", "content": "fourth prompt"},
    ]

    # test drop(id) removes the specified element
    context = semantic_session.get_recent(top_k=5, raw=True)
    middle_id = context[2][ID_FIELD_NAME]
    semantic_session.drop(middle_id)
    context = semantic_session.get_recent(top_k=4)
    assert context == [
        {"role": "user", "content": "second prompt"},
        {"role": "llm", "content": "second response"},
        {"role": "llm", "content": "third response"},
        {"role": "user", "content": "fourth prompt"},
    ]


def test_different_vector_dtypes():
    try:
        bfloat_sess = SemanticSessionManager(name="bfloat_session", dtype="bfloat16")
        bfloat_sess.add_message({"role": "user", "content": "bfloat message"})

        float16_sess = SemanticSessionManager(name="float16_session", dtype="float16")
        float16_sess.add_message({"role": "user", "content": "float16 message"})

        float32_sess = SemanticSessionManager(name="float32_session", dtype="float32")
        float32_sess.add_message({"role": "user", "content": "float32 message"})

        float64_sess = SemanticSessionManager(name="float64_session", dtype="float64")
        float64_sess.add_message({"role": "user", "content": "float64 message"})
```

```
        for sess in [bfloat_sess, float16_sess, float32_sess, float64_sess]:
            sess.set_distance_threshold(0.7)
            assert len(sess.get_relevant("float message")) == 1
    except:
        pytest.skip("Not using a late enough version of Redis")


def test_bad_dtype_connecting_to_exiting_session():
    try:
        session = SemanticSessionManager(name="float64 session", dtype="float64")
        same_type = SemanticSessionManager(name="float64 session", dtype="float64")
        # under the hood uses from_existing
    except RedisModuleVersionError:
        pytest.skip("Not using a late enough version of Redis")

    with pytest.raises(ValueError):
        bad_type = SemanticSessionManager(name="float64 session", dtype="float16")
}
```

tests/integration/test_vectorizers.py

```
import os

import pytest

from redisvl.utils.vectorize import (
    AzureOpenAITextVectorizer,
    CohereTextVectorizer,
    CustomTextVectorizer,
    HFTextVectorizer,
    MistralAITextVectorizer,
    OpenAITextVectorizer,
    VertexAITextVectorizer,
)


@pytest.fixture
def skip_vectorizer() -> bool:
    # os.getenv returns a string
    v = os.getenv("SKIP_VECTORIZERS", "False").lower() == "true"
    return v


@pytest.fixture(
    params=[
        HFTextVectorizer,
        OpenAITextVectorizer,
        VertexAITextVectorizer,
        CohereTextVectorizer,
        AzureOpenAITextVectorizer,
        # MistralAITextVectorizer,
        CustomTextVectorizer,
    ]
)
def vectorizer(request, skip_vectorizer):
    if skip_vectorizer:
        pytest.skip("Skipping vectorizer instantiation...")

    if request.param == HFTextVectorizer:
        return request.param()
```

```
    elif request.param == OpenAITextVectorizer:
        return request.param()
    elif request.param == VertexAITextVectorizer:
        return request.param()
    elif request.param == CohereTextVectorizer:
        return request.param()
    elif request.param == MistralAITextVectorizer:
        return request.param()
    elif request.param == AzureOpenAITextVectorizer:
        return request.param(
            model=os.getenv("AZURE_OPENAI_DEPLOYMENT_NAME", "text-embedding-ada-002")
        )
    elif request.param == CustomTextVectorizer:

        def embed(text):
            return [1.1, 2.2, 3.3, 4.4]

        def embed_many(texts):
            return [[1.1, 2.2, 3.3, 4.4]] * len(texts)

        return request.param(embed=embed, embed_many=embed_many)


@pytest.fixture
def custom_embed_func():
    def embed(text: str):
        return [1.1, 2.2, 3.3, 4.4]

    return embed


@pytest.fixture
def custom_embed_class():
    class embedder:
        def embed(self, text: str):
            return [1.1, 2.2, 3.3, 4.4]

        def embed_with_args(self, text: str, max_len=None):
            return [1.1, 2.2, 3.3, 4.4][0:max_len]

        def embed_many(self, text_list):
            return [[1.1, 2.2, 3.3], [4.4, 5.5, 6.6]]

        def embed_many_with_args(self, texts, param=True):
            if param:
                return [[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]]
            else:
                return [[6.0, 5.0, 4.0], [3.0, 2.0, 1.0]]

    return embedder


def test_vectorizer_embed(vectorizer):
    text = "This is a test sentence."
    if isinstance(vectorizer, CohereTextVectorizer):
        embedding = vectorizer.embed(text, input_type="search_document")
    else:
        embedding = vectorizer.embed(text)

    assert isinstance(embedding, list)
    assert len(embedding) == vectorizer.dims


def test_vectorizer_embed_many(vectorizer):
```

```
    texts = ["This is the first test sentence.", "This is the second test sentence."]
    if isinstance(vectorizer, CohereTextVectorizer):
        embeddings = vectorizer.embed_many(texts, input_type="search_document")
    else:
        embeddings = vectorizer.embed_many(texts)

    assert isinstance(embeddings, list)
    assert len(embeddings) == len(texts)
    assert all(
        isinstance(emb, list) and len(emb) == vectorizer.dims for emb in embeddings
    )


def test_vectorizer_bad_input(vectorizer):
    with pytest.raises(TypeError):
        vectorizer.embed(1)

    with pytest.raises(TypeError):
        vectorizer.embed({"foo": "bar"})

    with pytest.raises(TypeError):
        vectorizer.embed_many(42)


def test_custom_vectorizer_embed(custom_embed_class, custom_embed_func):
    # test we can pass a stand alone function as embedder callable
    custom_wrapper = CustomTextVectorizer(embed=custom_embed_func)
    embedding = custom_wrapper.embed("This is a test sentence.")
    assert embedding == [1.1, 2.2, 3.3, 4.4]

    # test we can pass an instance of a class method as embedder callable
    custom_wrapper = CustomTextVectorizer(embed=custom_embed_class().embed)
    embedding = custom_wrapper.embed("This is a test sentence.")
    assert embedding == [1.1, 2.2, 3.3, 4.4]

    # test we can pass additional parameters and kwargs to embedding methods
    custom_wrapper = CustomTextVectorizer(embed=custom_embed_class().embed_with_args)
    embedding = custom_wrapper.embed("This is a test sentence.", max_len=4)
    assert embedding == [1.1, 2.2, 3.3, 4.4]
    embedding = custom_wrapper.embed("This is a test sentence.", max_len=2)
    assert embedding == [1.1, 2.2]

    # test that correct error is raised if a non-callable is passed
    with pytest.raises(TypeError):
        bad_wrapper = CustomTextVectorizer(embed="hello")

    with pytest.raises(TypeError):
        bad_wrapper = CustomTextVectorizer(embed=42)

    with pytest.raises(TypeError):
        bad_wrapper = CustomTextVectorizer(embed={"foo": "bar"})

    # test that correct error is raised if passed function has incorrect types
    def bad_arg_type(value: int):
        return [value]

    with pytest.raises(ValueError):
        bad_wrapper = CustomTextVectorizer(embed=bad_arg_type)

    def bad_return_type(text: str) -> str:
        return text

    with pytest.raises(ValueError):
        bad_wrapper = CustomTextVectorizer(embed=bad_return_type)
```

```
def test_custom_vectorizer_embed_many(custom_embed_class, custom_embed_func):
    # test we can pass a stand alone function as embed_many callable
    custom_wrapper = CustomTextVectorizer(
        custom_embed_func, embed_many=custom_embed_class().embed_many
    )
    embeddings = custom_wrapper.embed_many(["test one.", "test two"])
    assert embeddings == [[1.1, 2.2, 3.3], [4.4, 5.5, 6.6]]

    # test we can pass a class method as embedder callable
    custom_wrapper = CustomTextVectorizer(
        custom_embed_func, embed_many=custom_embed_class().embed_many
    )
    embeddings = custom_wrapper.embed_many(["test one.", "test two"])
    assert embeddings == [[1.1, 2.2, 3.3], [4.4, 5.5, 6.6]]

    # test we can pass additional parameters and kwargs to embedding methods
    custom_wrapper = CustomTextVectorizer(
        custom_embed_func, embed_many=custom_embed_class().embed_many_with_args
    )
    embeddings = custom_wrapper.embed_many(["test one.", "test two"], param=True)
    assert embeddings == [[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]]
    embeddings = custom_wrapper.embed_many(["test one.", "test two"], param=False)
    assert embeddings == [[6.0, 5.0, 4.0], [3.0, 2.0, 1.0]]

    # test that correct error is raised if a non-callable is passed
    with pytest.raises(TypeError):
        bad_wrapper = CustomTextVectorizer(custom_embed_func, embed_many="hello")

    with pytest.raises(TypeError):
        bad_wrapper = CustomTextVectorizer(custom_embed_func, embed_many=42)

    with pytest.raises(TypeError):
        bad_wrapper = CustomTextVectorizer(custom_embed_func, embed_many=
{"foo": "bar"})

    # test that correct error is raised if passed function has incorrect types
    def bad_arg_type(value: int):
        return [value]

    with pytest.raises(ValueError):
        bad_wrapper = CustomTextVectorizer(custom_embed_func, embed_many=bad_arg_type)

    def bad_return_type(text: str) -> str:
        return text

    with pytest.raises(ValueError):
        bad_wrapper = CustomTextVectorizer(
            custom_embed_func, embed_many=bad_return_type
        )


@pytest.fixture(
    params=[
        OpenAITextVectorizer,
        # MistralAITextVectorizer,
        CustomTextVectorizer,
    ]
)
def avectorizer(request, skip_vectorizer):
    if skip_vectorizer:
        pytest.skip("Skipping vectorizer instantiation...")
```

```
    # Here we use actual models for integration test
    if request.param == OpenAITextVectorizer:
        return request.param()
    elif request.param == MistralAITextVectorizer:
        return request.param()

    # Here we use actual models for integration test
    if request.param == CustomTextVectorizer:

        def embed_func(text):
            return [1.1, 2.2, 3.3, 4.4]

        async def aembed_func(text):
            return [1.1, 2.2, 3.3, 4.4]

        async def aembed_many_func(texts):
            return [[1.1, 2.2, 3.3, 4.4]] * len(texts)

        return request.param(
            embed=embed_func, aembed=aembed_func, aembed_many=aembed_many_func
        )


@pytest.mark.asyncio
async def test_vectorizer_aembed(avectorizer):
    text = "This is a test sentence."
    embedding = await avectorizer.aembed(text)

    assert isinstance(embedding, list)
    assert len(embedding) == avectorizer.dims


@pytest.mark.asyncio
async def test_vectorizer_aembed_many(avectorizer):
    texts = ["This is the first test sentence.", "This is the second test sentence."]
    embeddings = await avectorizer.aembed_many(texts)

    assert isinstance(embeddings, list)
    assert len(embeddings) == len(texts)
    assert all(
        isinstance(emb, list) and len(emb) == avectorizer.dims for emb in embeddings
    )


@pytest.mark.asyncio
async def test_avectorizer_bad_input(avectorizer):
    with pytest.raises(TypeError):
        avectorizer.embed(1)

    with pytest.raises(TypeError):
        avectorizer.embed({"foo": "bar"})

    with pytest.raises(TypeError):
        avectorizer.embed_many(42)
}
```

Chapter 4.2.0

tests/unit

tests/unit/test_cross_encoder_reranker.py

```
import pytest
from sentence_transformers import CrossEncoder

from redisvl.utils.rerank.hf_cross_encoder import HFCrossEncoderReranker


@pytest.fixture
def reranker():
    return HFCrossEncoderReranker()


def test_rank_documents(reranker):
    docs = ["document one", "document two", "document three"]
    query = "search query"

    reranked_docs, scores = reranker.rank(query, docs)

    assert isinstance(reranked_docs, list)
    assert len(reranked_docs) == reranker.limit
    assert all(isinstance(score, float) for score in scores)


@pytest.mark.asyncio
async def test_async_rank_documents(reranker):
    docs = ["document one", "document two", "document three"]
    query = "search query"

    reranked_docs, scores = await reranker.arank(query, docs)

    assert isinstance(reranked_docs, list)
    assert len(reranked_docs) == reranker.limit
    assert all(isinstance(score, float) for score in scores)


def test_bad_input(reranker):
    with pytest.raises(ValueError):
        reranker.rank("", [])  # Empty query

    with pytest.raises(TypeError):
        reranker.rank(123, ["valid document"])  # Invalid type for query

    with pytest.raises(TypeError):
        reranker.rank("valid query", "not a list")  # Invalid type for documents


def test_rerank_empty(reranker):
    docs = []
    query = "search query"

    reranked_docs = reranker.rank(query, docs, return_score=False)

    assert isinstance(reranked_docs, list)
```

```
    assert len(reranked_docs) == 0
}
```

tests/unit/test_fields.py

```
import pytest
from redis.commands.search.field import GeoField as RedisGeoField
from redis.commands.search.field import NumericField as RedisNumericField
from redis.commands.search.field import TagField as RedisTagField
from redis.commands.search.field import TextField as RedisTextField
from redis.commands.search.field import VectorField as RedisVectorField

from redisvl.schema.fields import (
    FieldFactory,
    FlatVectorField,
    GeoField,
    HNSWVectorField,
    NumericField,
    TagField,
    TextField,
)


# Utility functions to create schema instances with default values
def create_text_field_schema(**kwargs):
    defaults = {
        "name": "example_textfield",
        "attrs": {"sortable": False, "weight": 1.0},
    }
    defaults.update(kwargs)
    return TextField(**defaults)


def create_tag_field_schema(**kwargs):
    defaults = {
        "name": "example_tagfield",
        "attrs": {"sortable": False, "separator": ","},
    }
    defaults.update(kwargs)
    return TagField(**defaults)


def create_numeric_field_schema(**kwargs):
    defaults = {"name": "example_numericfield", "attrs": {"sortable": False}}
    defaults.update(kwargs)
    return NumericField(**defaults)


def create_geo_field_schema(**kwargs):
    defaults = {"name": "example_geofield", "attrs": {"sortable": False}}
    defaults.update(kwargs)
    return GeoField(**defaults)


def create_flat_vector_field(**kwargs):
    defaults = {
        "name": "example_flatvectorfield",
        "attrs": {"dims": 128, "algorithm": "FLAT"},
    }
    defaults["attrs"].update(kwargs)
```

```
    return FlatVectorField(**defaults)


def create_hnsw_vector_field(**kwargs):
    defaults = {
        "name": "example_hnswvectorfield",
        "attrs": {
            "dims": 128,
            "algorithm": "HNSW",
            "m": 16,
            "ef_construction": 200,
            "ef_runtime": 10,
            "epsilon": 0.01,
        },
    }
    defaults["attrs"].update(kwargs)
    return HNSWVectorField(**defaults)


# Tests for field schema creation and validation
@pytest.mark.parametrize(
    "schema_func,field_class",
    [
        (create_text_field_schema, RedisTextField),
        (create_tag_field_schema, RedisTagField),
        (create_numeric_field_schema, RedisNumericField),
        (create_geo_field_schema, RedisGeoField),
    ],
)
def test_field_schema_as_field(schema_func, field_class):
    schema = schema_func()
    field = schema.as_redis_field()
    assert isinstance(field, field_class)
    assert field.name == f"example_{field_class.__name__.lower()}"


def test_vector_fields_as_field():
    flat_vector_schema = create_flat_vector_field()
    flat_vector_field = flat_vector_schema.as_redis_field()
    assert isinstance(flat_vector_field, RedisVectorField)
    assert flat_vector_field.name == "example_flatvectorfield"

    hnsw_vector_schema = create_hnsw_vector_field()
    hnsw_vector_field = hnsw_vector_schema.as_redis_field()
    assert isinstance(hnsw_vector_field, RedisVectorField)
    assert hnsw_vector_field.name == "example_hnswvectorfield"


@pytest.mark.parametrize(
    "vector_schema_func,extra_params",
    [
        (create_flat_vector_field, {"block_size": 100}),
        (create_hnsw_vector_field, {"m": 24, "ef_construction": 300}),
    ],
)
def test_vector_fields_with_optional_params(vector_schema_func, extra_params):
    # Create a vector schema with additional parameters set.
    vector_schema = vector_schema_func(**extra_params)
    vector_field = vector_schema.as_redis_field()

    # Assert that the field is correctly created and the optional parameters are set.
    assert isinstance(vector_field, RedisVectorField)
    for param, value in extra_params.items():
        assert param.upper() in vector_field.args
```

```
        i = vector_field.args.index(param.upper())
        assert vector_field.args[i + 1] == value


def test_hnsw_vector_field_optional_params_not_set():
    # Create HNSW vector field without setting optional params
    hnsw_field = HNSWVectorField(
        name="example_vector", attrs={"dims": 128, "algorithm": "hnsw"}
    )

    assert hnsw_field.attrs.m == 16  # default value
    assert hnsw_field.attrs.ef_construction == 200  # default value
    assert hnsw_field.attrs.ef_runtime == 10  # default value
    assert hnsw_field.attrs.epsilon == 0.01  # default value

    field_exported = hnsw_field.as_redis_field()

    # Check the default values are correctly applied in the exported object
    assert field_exported.args[field_exported.args.index("M") + 1] == 16
    assert field_exported.args[field_exported.args.index("EF_CONSTRUCTION") + 1] == 200
    assert field_exported.args[field_exported.args.index("EF_RUNTIME") + 1] == 10
    assert field_exported.args[field_exported.args.index("EPSILON") + 1] == 0.01


def test_flat_vector_field_block_size_not_set():
    # Create Flat vector field without setting block_size
    flat_field = FlatVectorField(
        name="example_vector", attrs={"dims": 128, "algorithm": "flat"}
    )
    field_exported = flat_field.as_redis_field()

    # block_size and initial_cap should not be in the exported field if it was not set
    assert "BLOCK_SIZE" not in field_exported.args
    assert "INITIAL_CAP" not in field_exported.args


# Tests for standard field creation
@pytest.mark.parametrize(
    "field_type, expected_class",
    [
        ("tag", TagField),
        ("text", TextField),
        ("numeric", NumericField),
        ("geo", GeoField),
    ],
)
def test_create_standard_field(field_type, expected_class):
    field = FieldFactory.create_field(field_type, "example_field")
    assert isinstance(field, expected_class)
    assert field.name == "example_field"


# Tests for vector field creation
@pytest.mark.parametrize(
    "algorithm, expected_class",
    [
        ("flat", FlatVectorField),
        ("hnsw", HNSWVectorField),
    ],
)
def test_create_vector_field(algorithm, expected_class):
    field = FieldFactory.create_field(
        "vector", "example_vector_field", attrs={"algorithm": algorithm, "dims": 128}
    )
```

```
    assert isinstance(field, expected_class)
    assert field.name == "example_vector_field"


def test_create_vector_field_with_unknown_algorithm():
    """Test for unknown vector field algorithm."""
    with pytest.raises(ValueError) as e:
        FieldFactory.create_field(
            "vector",
            "example_vector_field",
            attrs={"algorithm": "unknown", "dims": 128},
        )
    assert "Unknown vector field algorithm" in str(e.value)


def test_missing_vector_field_algorithm():
    """Test for missing vector field algorithm."""
    with pytest.raises(ValueError) as e:
        FieldFactory.create_field("vector", "example_vector_field", attrs=
{"dims": 128})
    assert "Must provide algorithm param" in str(e.value)


def test_missing_vector_field_dims():
    """Test for missing vector field algorithm."""
    with pytest.raises(ValueError) as e:
        FieldFactory.create_field(
            "vector", "example_vector_field", attrs={"algorithm": "flat"}
        )
    assert "Must provide dims param" in str(e.value)


def test_create_unknown_field_type():
    """Test for unknown field type."""
    with pytest.raises(ValueError) as excinfo:
        FieldFactory.create_field("unknown", "example_field")
    assert "Unknown field type: unknown" in str(excinfo.value)
}
```

tests/unit/test_filter.py

```
import pytest

from redisvl.query.filter import Geo, GeoRadius, Num, Tag, Text


# Test cases for various scenarios of tag usage, combinations, and their
string representations.
@pytest.mark.parametrize(
    "operation,tags,expected",
    [
        # Testing single tags
        ("==", "simpletag", "@tag_field:{simpletag}"),
        (
            "==",
            "tag with space",
            "@tag_field:{tag\\ with\\ space}",
        ),  # Escaping spaces within quotes
        (
            "==",
```

```
            "special$char",
            "@tag_field:{special\\$char}",
        ),  # Escaping a special character
        ("!=", "negated", "(-@tag_field:{negated})"),
        # Testing multiple tags
        ("==", ["tag1", "tag2"], "@tag_field:{tag1|tag2}"),
        (
            "==",
            ["alpha", "beta with space", "gamma$special"],
            "@tag_field:{alpha|beta\\ with\\ space|gamma\\$special}",
        ),  # Multiple tags with spaces and special chars
        ("!=", ["tagA", "tagB"], "(-@tag_field:{tagA|tagB})"),
        # Complex tag scenarios with special characters
        ("==", "weird:tag", "@tag_field:{weird\\:tag}"),  # Tags with colon
        ("==", "tag&another", "@tag_field:{tag\\&another}"),  # Tags with ampersand
        # Escaping various special characters within tags
        ("==", "tag/with/slashes", "@tag_field:{tag\\/with\\/slashes}"),
        (
            "==",
            ["hypen-tag", "under_score", "dot.tag"],
            "@tag_field:{hypen\\-tag|under_score|dot\\.tag}",
        ),
        # ...additional unique cases as desired...
    ],
)
def test_tag_filter_varied(operation, tags, expected):
    if operation == "==":
        tf = Tag("tag_field") == tags
    elif operation == "!=":
        tf = Tag("tag_field") != tags
    else:
        raise ValueError(f"Unsupported operation: {operation}")

    # Verify the string representation matches the expected RediSearch query part
    assert str(tf) == expected


def test_nullable():
    tag = Tag("tag_field") == None
    assert str(tag) == "*"

    tag = Tag("tag_field") != None
    assert str(tag) == "*"

    tag = Tag("tag_field") == []
    assert str(tag) == "*"

    tag = Tag("tag_field") != []
    assert str(tag) == "*"

    tag = Tag("tag_field") == ""
    assert str(tag) == "*"

    tag = Tag("tag_field") != ""
    assert str(tag) == "*"

    tag = Tag("tag_field") == [None]
    assert str(tag) == "*"

    tag = Tag("tag_field") == [None, "tag"]
    assert str(tag) == "@tag_field:{tag}"


def test_numeric_filter():
```

```
    nf = Num("numeric_field") == 5
    assert str(nf) == "@numeric_field:[5 5]"

    nf = Num("numeric_field") != 5
    assert str(nf) == "(-@numeric_field:[5 5])"

    nf = Num("numeric_field") > 5
    assert str(nf) == "@numeric_field:[(5 +inf]"

    nf = Num("numeric_field") >= 5
    assert str(nf) == "@numeric_field:[5 +inf]"

    nf = Num("numeric_field") < 5
    assert str(nf) == "@numeric_field:[-inf (5]"

    nf = Num("numeric_field") <= 5
    assert str(nf) == "@numeric_field:[-inf 5]"

    nf = Num("numeric_field") > 5.5
    assert str(nf) == "@numeric_field:[-inf 5.5]"

    nf = Num("numeric_field") <= None
    assert str(nf) == "*"

    nf = Num("numeric_field") == None
    assert str(nf) == "*"

    nf = Num("numeric_field") != None
    assert str(nf) == "*"


def test_text_filter():
    txt_f = Text("text_field") == "text"
    assert str(txt_f) == '@text_field:("text")'

    txt_f = Text("text_field") != "text"
    assert str(txt_f) == '(-@text_field:"text")'

    txt_f = Text("text_field") % "text"
    assert str(txt_f) == "@text_field:(text)"

    txt_f = Text("text_field") % "tex*"
    assert str(txt_f) == "@text_field:(tex*)"

    txt_f = Text("text_field") % "%text%"
    assert str(txt_f) == "@text_field:(%text%)"

    txt_f = Text("text_field") % ""
    assert str(txt_f) == "*"


def test_geo_filter():
    geo_f = Geo("geo_field") == GeoRadius(1.0, 2.0, 3, "km")
    assert str(geo_f) == "@geo_field:[1.0 2.0 3 km]"

    geo_f = Geo("geo_field") != GeoRadius(1.0, 2.0, 3, "km")
    assert str(geo_f) != "(-@geo_field:[1.0 2.0 3 m])"


@pytest.mark.parametrize(
    "value, expected",
    [
        (None, "*"),
        ([], "*"),
```

```
        ("", "*"),
        ([None], "*"),
        ([None, "tag"], "@tag_field:{tag}"),
    ],
    ids=[
        "none",
        "empty_list",
        "empty_string",
        "list_with_none",
        "list_with_none_and_tag",
    ],
)
def test_nullable(value, expected):
    tag = Tag("tag_field")
    assert str(tag == value) == expected


@pytest.mark.parametrize(
    "operation, value, expected",
    [
        ("__eq__", 5, "@numeric_field:[5 5]"),
        ("__ne__", 5, "(-@numeric_field:[5 5])"),
        ("__gt__", 5, "@numeric_field:[(5 +inf]"),
        ("__ge__", 5, "@numeric_field:[5 +inf]"),
        ("__lt__", 5, "@numeric_field:[-inf (5]"),
        ("__le__", 5, "@numeric_field:[-inf 5]"),
        ("__le__", None, "*"),
        ("__eq__", None, "*"),
        ("__ne__", None, "*"),
    ],
    ids=["eq", "ne", "gt", "ge", "lt", "le", "le_none", "eq_none", "ne_none"],
)
def test_numeric_filter(operation, value, expected):
    nf = Num("numeric_field")
    assert str(getattr(nf, operation)(value)) == expected


@pytest.mark.parametrize(
    "operation, value, expected",
    [
        ("__eq__", "text", '@text_field:("text")'),
        ("__ne__", "text", '(-@text_field:"text")'),
        ("__eq__", "", "*"),
        ("__ne__", "", "*"),
        ("__eq__", None, "*"),
        ("__ne__", None, "*"),
        ("__mod__", "text", "@text_field:(text)"),
        ("__mod__", "tex*", "@text_field:(tex*)"),
        ("__mod__", "%text%", "@text_field:(%text%)"),
        ("__mod__", "", "*"),
        ("__mod__", None, "*"),
    ],
    ids=[
        "eq",
        "ne",
        "eq-empty",
        "ne-empty",
        "eq-none",
        "ne-none",
        "like",
        "like_wildcard",
        "like_full",
        "like_empty",
        "like_none",
```

```
    ],
)
def test_text_filter(operation, value, expected):
    txt_f = getattr(Text("text_field"), operation)(value)
    assert str(txt_f) == expected


@pytest.mark.parametrize(
    "operation, expected",
    [
        ("__eq__", "@geo_field:[1.0 2.0 3 km]"),
        ("__ne__", "(-@geo_field:[1.0 2.0 3 km])"),
    ],
    ids=["eq", "ne"],
)
def test_geo_filter(operation, expected):
    geo_radius = GeoRadius(1.0, 2.0, 3, "km")
    geo_f = Geo("geo_field")
    assert str(getattr(geo_f, operation)(geo_radius)) == expected


def test_filters_combination():
    tf1 = Tag("tag_field") == ["tag1", "tag2"]
    tf2 = Tag("tag_field") == "tag3"
    combined = tf1 & tf2
    assert str(combined) == "(@tag_field:{tag1|tag2} @tag_field:{tag3})"

    combined = tf1 | tf2
    assert str(combined) == "(@tag_field:{tag1|tag2} | @tag_field:{tag3})"

    tf1 = Tag("tag_field") == []
    assert str(tf1) == "*"
    assert str(tf1 & tf2) == str(tf2)
    assert str(tf1 | tf2) == str(tf2)

    # test combining filters with None values and empty strings
    tf1 = Tag("tag_field") == None
    tf2 = Tag("tag_field") == ""
    assert str(tf1 & tf2) == "*"

    tf1 = Tag("tag_field") == None
    tf2 = Tag("tag_field") == "tag"
    assert str(tf1 & tf2) == str(tf2)

    tf1 = Tag("tag_field") == None
    tf2 = Tag("tag_field") == ["tag1", "tag2"]
    assert str(tf1 & tf2) == str(tf2)

    tf1 = Tag("tag_field") == None
    tf2 = Tag("tag_field") != None
    assert str(tf1 & tf2) == "*"

    tf1 = Tag("tag_field") == ""
    tf2 = Tag("tag_field") == "tag"
    tf3 = Tag("tag_field") == ["tag1", "tag2"]
    assert str(tf1 & tf2 & tf3) == str(tf2 & tf3)

    # test none filters for Tag Num Text and Geo
    tf1 = Tag("tag_field") == None
    tf2 = Num("num_field") == None
    tf3 = Text("text_field") == None
    tf4 = Geo("geo_field") == None
    assert str(tf1 & tf2 & tf3 & tf4) == "*"
```

```
    tf1 = Tag("tag_field") != None
    tf2 = Num("num_field") != None
    tf3 = Text("text_field") != None
    tf4 = Geo("geo_field") != None
    assert str(tf1 & tf2 & tf3 & tf4) == "*"

    # test combinations of real and None filters across tag
    # text and geo filters
    tf1 = Tag("tag_field") == "tag"
    tf2 = Num("num_field") == None
    tf3 = Text("text_field") == None
    tf4 = Geo("geo_field") == GeoRadius(1.0, 2.0, 3, "km")
    assert str(tf1 & tf2 & tf3 & tf4) == str(tf1 & tf4)


def test_num_filter_zero():
    num_filter = Num("chunk_number") == 0
    assert (
        str(num_filter) == "@chunk_number:[0 0]"
    ), "Num filter should handle zero correctly"
}
```

tests/unit/test_llmcache_schema.py

```
import json

import pytest
from pydantic.v1 import ValidationError

from redisvl.extensions.llmcache.schema import CacheEntry, CacheHit
from redisvl.redis.utils import array_to_buffer, hashify


def test_valid_cache_entry_creation():
    entry = CacheEntry(
        prompt="What is AI?",
        response="AI is artificial intelligence.",
        prompt_vector=[0.1, 0.2, 0.3],
    )
    assert entry.entry_id == hashify("What is AI?")
    assert entry.prompt == "What is AI?"
    assert entry.response == "AI is artificial intelligence."
    assert entry.prompt_vector == [0.1, 0.2, 0.3]


def test_cache_entry_with_given_entry_id():
    entry = CacheEntry(
        entry_id="custom_id",
        prompt="What is AI?",
        response="AI is artificial intelligence.",
        prompt_vector=[0.1, 0.2, 0.3],
    )
    assert entry.entry_id == "custom_id"


def test_cache_entry_with_invalid_metadata():
    with pytest.raises(ValidationError):
        CacheEntry(
            prompt="What is AI?",
            response="AI is artificial intelligence.",
```

```
            prompt_vector=[0.1, 0.2, 0.3],
            metadata="invalid_metadata",
        )


def test_cache_entry_to_dict():
    entry = CacheEntry(
        prompt="What is AI?",
        response="AI is artificial intelligence.",
        prompt_vector=[0.1, 0.2, 0.3],
        metadata={"author": "John"},
        filters={"category": "technology"},
    )
    result = entry.to_dict(dtype="float32")
    assert result["entry_id"] == hashify("What is AI?", {"category": "technology"})
    assert result["metadata"] == json.dumps({"author": "John"})
    assert result["prompt_vector"] == array_to_buffer([0.1, 0.2, 0.3], "float32")
    assert result["category"] == "technology"
    assert "filters" not in result


def test_valid_cache_hit_creation():
    hit = CacheHit(
        entry_id="entry_1",
        prompt="What is AI?",
        response="AI is artificial intelligence.",
        vector_distance=0.1,
        inserted_at=1625819123.123,
        updated_at=1625819123.123,
    )
    assert hit.entry_id == "entry_1"
    assert hit.prompt == "What is AI?"
    assert hit.response == "AI is artificial intelligence."
    assert hit.vector_distance == 0.1
    assert hit.inserted_at == hit.updated_at == 1625819123.123


def test_cache_hit_with_serialized_metadata():
    hit = CacheHit(
        entry_id="entry_1",
        prompt="What is AI?",
        response="AI is artificial intelligence.",
        vector_distance=0.1,
        inserted_at=1625819123.123,
        updated_at=1625819123.123,
        metadata=json.dumps({"author": "John"}),
    )
    assert hit.metadata == {"author": "John"}


def test_cache_hit_to_dict():
    hit = CacheHit(
        entry_id="entry_1",
        prompt="What is AI?",
        response="AI is artificial intelligence.",
        vector_distance=0.1,
        inserted_at=1625819123.123,
        updated_at=1625819123.123,
        filters={"category": "technology"},
    )
    result = hit.to_dict()
    assert result["entry_id"] == "entry_1"
    assert result["prompt"] == "What is AI?"
    assert result["response"] == "AI is artificial intelligence."
```

```
    assert result["vector_distance"] == 0.1
    assert result["category"] == "technology"
    assert "filters" not in result


def test_cache_entry_with_empty_optional_fields():
    entry = CacheEntry(
        prompt="What is AI?",
        response="AI is artificial intelligence.",
        prompt_vector=[0.1, 0.2, 0.3],
    )
    result = entry.to_dict(dtype="float32")
    assert "metadata" not in result
    assert "filters" not in result


def test_cache_hit_with_empty_optional_fields():
    hit = CacheHit(
        entry_id="entry_1",
        prompt="What is AI?",
        response="AI is artificial intelligence.",
        vector_distance=0.1,
        inserted_at=1625819123.123,
        updated_at=1625819123.123,
    )
    result = hit.to_dict()
    assert "metadata" not in result
    assert "filters" not in result
}
```

tests/unit/test_query_types.py

```
import pytest
from redis.commands.search.query import Query
from redis.commands.search.result import Result

from redisvl.index.index import process_results
from redisvl.query import CountQuery, FilterQuery, RangeQuery, VectorQuery
from redisvl.query.filter import Tag

# Sample data for testing
sample_vector = [0.1, 0.2, 0.3, 0.4]


# Test Cases


def test_count_query():
    # Create a filter expression
    filter_expression = Tag("brand") == "Nike"
    count_query = CountQuery(filter_expression)

    # Check properties
    assert isinstance(count_query, Query)
    assert isinstance(count_query.query, Query)
    assert isinstance(count_query.params, dict)
    assert count_query.params == {}

    # Test set_filter functionality
    new_filter_expression = Tag("category") == "Sportswear"
```

```
    count_query.set_filter(new_filter_expression)
    assert count_query.filter == new_filter_expression

    fake_result = Result([2], "")
    assert process_results(fake_result, count_query, "json") == 2


def test_filter_query():
    # Create a filter expression
    filter_expression = Tag("brand") == "Nike"
    return_fields = ["brand", "price"]
    filter_query = FilterQuery(filter_expression, return_fields, 10)

    # Check properties
    assert filter_query._return_fields == return_fields
    assert filter_query._num_results == 10
    assert filter_query.filter == filter_expression
    assert isinstance(filter_query, Query)
    assert isinstance(filter_query.query, Query)
    assert isinstance(filter_query.params, dict)
    assert filter_query.params == {}
    assert filter_query._dialect == 2
    assert filter_query._sortby == None
    assert filter_query._in_order == False

    # Test set_filter functionality
    new_filter_expression = Tag("category") == "Sportswear"
    filter_query.set_filter(new_filter_expression)
    assert filter_query.filter == new_filter_expression

    # Test paging functionality
    filter_query.paging(5, 7)
    assert filter_query._offset == 5
    assert filter_query._num == 7
    assert filter_query._num_results == 10

    # Test sort_by functionality
    filter_query = FilterQuery(
        filter_expression, return_fields, num_results=10, sort_by="price"
    )
    assert filter_query._sortby is not None

    # Test in_order functionality
    filter_query = FilterQuery(
        filter_expression, return_fields, num_results=10, in_order=True
    )
    assert filter_query._in_order


def test_vector_query():
    # Create a vector query
    vector_query = VectorQuery(
        sample_vector, "vector_field", ["field1", "field2"], dialect=3, num_results=10
    )

    # Check properties
    assert vector_query._vector == sample_vector
    assert vector_query._vector_field_name == "vector_field"
    assert vector_query._num_results == 10
    assert vector_query._return_fields == ["field1", "field2", "vector_distance"]
    assert isinstance(vector_query, Query)
    assert isinstance(vector_query.query, Query)
    assert isinstance(vector_query.params, dict)
    assert vector_query.params != {}
```

```
    assert vector_query._dialect == 3
    assert vector_query._sortby.args[0] == VectorQuery.DISTANCE_ID
    assert vector_query._in_order == False

    # Test set_filter functionality
    new_filter_expression = Tag("category") == "Sportswear"
    vector_query.set_filter(new_filter_expression)
    assert vector_query.filter == new_filter_expression

    # Test paging functionality
    vector_query.paging(5, 7)
    assert vector_query._offset == 5
    assert vector_query._num == 7
    assert vector_query._num_results == 10

    # Test sort_by functionality
    vector_query = VectorQuery(
        sample_vector,
        "vector_field",
        ["field1", "field2"],
        dialect=3,
        num_results=10,
        sort_by="field2",
    )
    assert vector_query._sortby.args[0] == "field2"

    # Test in_order functionality
    vector_query = VectorQuery(
        sample_vector,
        "vector_field",
        ["field1", "field2"],
        dialect=3,
        num_results=10,
        in_order=True,
    )
    assert vector_query._in_order


def test_range_query():
    # Create a filter expression
    filter_expression = Tag("brand") == "Nike"

    # Create a RangeQuery instance
    range_query = RangeQuery(
        sample_vector, "vector_field", ["field1"], filter_expression, num_results=10
    )

    # Check properties
    assert range_query._vector == sample_vector
    assert range_query._vector_field_name == "vector_field"
    assert range_query._num_results == 10
    assert range_query.distance_threshold == 0.2
    assert "field1" in range_query._return_fields
    assert isinstance(range_query, Query)
    assert isinstance(range_query.query, Query)
    assert isinstance(range_query.params, dict)
    assert range_query.params != {}
    assert range_query._sortby.args[0] == RangeQuery.DISTANCE_ID

    # Test set_distance_threshold functionality
    range_query.set_distance_threshold(0.1)
    assert range_query.distance_threshold == 0.1

    # Test set_filter functionality
```

```
    new_filter_expression = Tag("category") == "Outdoor"
    range_query.set_filter(new_filter_expression)
    assert range_query.filter == new_filter_expression

    # Test paging functionality
    range_query.paging(5, 7)
    assert range_query._offset == 5
    assert range_query._num == 7
    assert range_query._num_results == 10

    # Test sort_by functionality
    range_query = RangeQuery(
        sample_vector,
        "vector_field",
        ["field1"],
        filter_expression,
        num_results=10,
        sort_by="field1",
    )
    assert range_query._sortby.args[0] == "field1"

    # Test in_order functionality
    range_query = RangeQuery(
        sample_vector,
        "vector_field",
        ["field1"],
        filter_expression,
        num_results=10,
        in_order=True,
    )
    assert range_query._in_order


@pytest.mark.parametrize(
    "query",
    [
        CountQuery(),
        FilterQuery(),
        VectorQuery(vector=[1, 2, 3], vector_field_name="vector"),
        RangeQuery(vector=[1, 2, 3], vector_field_name="vector"),
    ],
)
def test_query_modifiers(query):
    query.paging(3, 5)
    assert query._offset == 3
    assert query._num == 5

    query.dialect(4)
    assert query._dialect == 4

    query.in_order()
    assert query._in_order

    query.sort_by("time")
    assert query._sortby.args[0] == "time"

    query.scorer("BM25")
    assert query._scorer == "BM25"

    query.timeout(20)
    assert query._timeout == 20

    query.slop(10)
    assert query._slop == 10
```

```
    query.verbatim()
    assert query._verbatim

    query.no_content()
    assert query._no_content

    query.no_stopwords()
    assert query._no_stopwords

    query.with_scores()
    assert query._with_scores

    query.limit_fields("test")
    assert query._fields == ("test",)

    f = Tag("test") == "foo"
    query.set_filter(f)
    assert query._filter_expression == f

    # double check all other states
    assert query._offset == 3
    assert query._num == 5
    assert query._dialect == 4
    assert query._in_order
    assert query._sortby.args[0] == "time"
    assert query._scorer == "BM25"
    assert query._timeout == 20
    assert query._slop == 10
    assert query._verbatim
    assert query._no_content
    assert query._no_stopwords
    assert query._with_scores
    assert query._fields == ("test",)


@pytest.mark.parametrize(
    "query",
    [
        CountQuery(),
        FilterQuery(),
        VectorQuery(vector=[1, 2, 3], vector_field_name="vector"),
        RangeQuery(vector=[1, 2, 3], vector_field_name="vector"),
    ],
)
def test_string_filter_expressions(query):
    # No filter
    query.set_filter("*")
    assert query._filter_expression == "*"

    # Simple full text search
    query.set_filter("hello world")
    assert query._filter_expression == "hello world"
    assert query.query_string().__contains__("hello world")

    # Optional flag
    query.set_filter("~(@desciption:(hello | world))")
    assert query._filter_expression == "~(@desciption:(hello | world))"
    assert query.query_string().__contains__("~(@desciption:(hello | world))")
}
```

tests/unit/test_route_schema.py

```
import pytest
from pydantic.v1 import ValidationError

from redisvl.extensions.router.schema import (
    DistanceAggregationMethod,
    Route,
    RouteMatch,
    RoutingConfig,
)


def test_route_valid():
    route = Route(
        name="Test Route",
        references=["reference1", "reference2"],
        metadata={"key": "value"},
        distance_threshold=0.3,
    )
    assert route.name == "Test Route"
    assert route.references == ["reference1", "reference2"]
    assert route.metadata == {"key": "value"}
    assert route.distance_threshold == 0.3


def test_route_empty_name():
    with pytest.raises(ValidationError) as excinfo:
        Route(
            name="",
            references=["reference1", "reference2"],
            metadata={"key": "value"},
            distance_threshold=0.3,
        )
    assert "Route name must not be empty" in str(excinfo.value)


def test_route_empty_references():
    with pytest.raises(ValidationError) as excinfo:
        Route(
            name="Test Route",
            references=[],
            metadata={"key": "value"},
            distance_threshold=0.3,
        )
    assert "References must not be empty" in str(excinfo.value)


def test_route_non_empty_references():
    with pytest.raises(ValidationError) as excinfo:
        Route(
            name="Test Route",
            references=["reference1", ""],
            metadata={"key": "value"},
            distance_threshold=0.3,
        )
    assert "All references must be non-empty strings" in str(excinfo.value)


def test_route_valid_no_threshold():
    route = Route(
        name="Test Route",
        references=["reference1", "reference2"],
```

```
        metadata={"key": "value"},
    )
    assert route.name == "Test Route"
    assert route.references == ["reference1", "reference2"]
    assert route.metadata == {"key": "value"}
    assert route.distance_threshold is None


def test_route_invalid_threshold_zero():
    with pytest.raises(ValidationError) as excinfo:
        Route(
            name="Test Route",
            references=["reference1", "reference2"],
            metadata={"key": "value"},
            distance_threshold=0,
        )
    assert "Route distance threshold must be greater than zero" in str(excinfo.value)


def test_route_invalid_threshold_negative():
    with pytest.raises(ValidationError) as excinfo:
        Route(
            name="Test Route",
            references=["reference1", "reference2"],
            metadata={"key": "value"},
            distance_threshold=-0.1,
        )
    assert "Route distance threshold must be greater than zero" in str(excinfo.value)


def test_route_match():
    route_match = RouteMatch(name="test", distance=0.25)
    assert route_match.name == "test"
    assert route_match.distance == 0.25


def test_route_match_no_route():
    route_match = RouteMatch()
    assert route_match.name is None
    assert route_match.distance is None


def test_distance_aggregation_method():
    assert DistanceAggregationMethod.avg == DistanceAggregationMethod("avg")
    assert DistanceAggregationMethod.min == DistanceAggregationMethod("min")
    assert DistanceAggregationMethod.sum == DistanceAggregationMethod("sum")


def test_routing_config_valid():
    config = RoutingConfig(distance_threshold=0.6, max_k=5)
    assert config.distance_threshold == 0.6
    assert config.max_k == 5


def test_routing_config_invalid_max_k():
    with pytest.raises(ValidationError) as excinfo:
        RoutingConfig(distance_threshold=0.6, max_k=0)
    assert "max_k must be a positive integer" in str(excinfo.value)


def test_routing_config_invalid_distance_threshold():
    with pytest.raises(ValidationError) as excinfo:
        RoutingConfig(distance_threshold=1.5, max_k=5)
```

```
    assert "distance_threshold must be between 0 and 1" in str(excinfo.value)
}
```

tests/unit/test_schema.py

```
import os
import pathlib

import pytest

from redisvl.schema.fields import TagField, TextField
from redisvl.schema.schema import IndexSchema, StorageType


def get_base_path():
    return pathlib.Path(__file__).parent.resolve()


# Sample data for testing
def create_sample_index_schema():
    sample_fields = [
        {"name": "example_text", "type": "text", "attrs": {"sortable": False}},
        {"name": "example_numeric", "type": "numeric", "attrs": {"sortable": True}},
        {"name": "example_tag", "type": "tag", "attrs": {"sortable": True}},
        {
            "name": "example_vector",
            "type": "vector",
            "attrs": {"dims": 1024, "algorithm": "flat"},
        },
    ]
    return IndexSchema.from_dict({"index": {"name": "test"}, "fields": sample_fields})


# Tests for IndexSchema


def test_initialization_with_default_params():
    """Test basic schema init with defaults."""
    default_schema = IndexSchema.from_dict({"index": {"name": "test"}})
    assert default_schema.version == "0.1.0"
    assert default_schema.index.name == "test"
    assert default_schema.index.prefix == "rvl"  # Default value
    assert default_schema.index.key_separator == ":"  # Default value
    assert default_schema.index.storage_type == StorageType.HASH  # Default value
    assert default_schema.fields == {}  # Default value


def test_initialization_with_custom_params():
    """Test custom schema params."""
    custom_schema = IndexSchema.from_dict(
        {
            "index": {
                "name": "custom_schema",
                "prefix": "custom",
                "key_separator": "|",
                "storage_type": "json",
            }
        }
    )
    assert custom_schema.index.name == "custom_schema"
```

```
    assert custom_schema.index.prefix == "custom"
    assert custom_schema.index.key_separator == "|"
    assert custom_schema.index.storage_type == StorageType.JSON


def test_add_field():
    """Test field addition."""
    index_schema = create_sample_index_schema()
    index_schema.add_field({"name": "new_text_field", "type": "text"})
    assert "new_text_field" in index_schema.fields
    assert isinstance(index_schema.fields["new_text_field"], TextField)


def test_add_fields():
    """Test multiple field addition."""
    index_schema = create_sample_index_schema()
    index_schema.add_fields(
        [
            {"name": "new_text_field", "type": "text"},
            {"name": "new_tag_field", "type": "tag"},
        ]
    )
    assert "new_text_field" in index_schema.fields
    assert isinstance(index_schema.fields["new_text_field"], TextField)
    assert "new_tag_field" in index_schema.fields
    assert isinstance(index_schema.fields["new_tag_field"], TagField)


def test_add_duplicate_field():
    """Test adding a duplicate field."""
    index_schema = create_sample_index_schema()
    with pytest.raises(ValueError):
        index_schema.add_field({"name": "example_text", "type": "text"})


def test_remove_field():
    """Test field removal."""
    index_schema = create_sample_index_schema()
    index_schema.remove_field("example_text")
    assert "example_text" not in index_schema.field_names


def test_generate_fields():
    """Test field generation."""
    sample = {"name": "John", "age": 30, "tags": ["test", "test2"]}
    index_schema = IndexSchema.from_dict({"index": {"name": "test"}})
    generated_fields = index_schema.generate_fields(sample)
    expected_field_names = sample.keys()
    for field in generated_fields:
        assert field["name"] in expected_field_names
        assert field["path"] == None


def test_to_dict():
    """Test schema to dict serialization."""
    index_schema = create_sample_index_schema()
    index_dict = index_schema.to_dict()
    assert index_dict["index"]["name"] == "test"
    assert isinstance(index_dict["fields"], list)
    assert len(index_dict["fields"]) == 4 == len(index_schema.fields)


def test_from_dict():
    """Test loading schema from a dictionary."""
```

```
    sample_fields = [
        {"name": "example_text", "type": "text", "attrs": {"sortable": False}},
        {"name": "example_numeric", "type": "tag", "attrs": {"sortable": True}},
    ]
    index_schema = IndexSchema.from_dict(
        {
            "index": {
                "name": "example_index",
                "prefix": "ex",
                "key_separator": "|",
                "storage_type": "json",
            },
            "fields": sample_fields,
        }
    )
    assert index_schema.index.name == "example_index"
    assert index_schema.index.key_separator == "|"
    assert index_schema.index.prefix == "ex"
    assert index_schema.index.storage_type == StorageType.JSON
    assert len(index_schema.fields) == 2


def test_hash_index_from_yaml():
    """Test loading from yaml."""
    index_schema = IndexSchema.from_yaml(
        str(get_base_path().joinpath("../../schemas/test_hash_schema.yaml"))
    )
    assert index_schema.index.name == "hash-test"
    assert index_schema.index.prefix == "hash"
    assert index_schema.index.storage_type == StorageType.HASH
    assert len(index_schema.fields) == 2


def test_json_index_from_yaml():
    """Test loading from yaml."""
    index_schema = IndexSchema.from_yaml(
        str(get_base_path().joinpath("../../schemas/test_json_schema.yaml"))
    )
    assert index_schema.index.name == "json-test"
    assert index_schema.index.prefix == "json"
    assert index_schema.index.storage_type == StorageType.JSON
    assert len(index_schema.fields) == 2


def test_to_yaml_and_reload():
    index_schema = create_sample_index_schema()
    index_schema.to_yaml("temp_test.yaml")

    assert os.path.exists("temp_test.yaml")

    new_schema = IndexSchema.from_yaml("temp_test.yaml")
    assert new_schema == index_schema
    assert new_schema.to_dict() == index_schema.to_dict()

    os.remove("temp_test.yaml")


def test_from_yaml_file_not_found():
    """Test loading from yaml with file not found."""
    with pytest.raises(FileNotFoundError):
        IndexSchema.from_yaml("nonexistent_file")
}
```

tests/unit/test_session_schema.py

```
from uuid import uuid4

import pytest
from pydantic.v1 import ValidationError

from redisvl.extensions.session_manager.schema import ChatMessage
from redisvl.redis.utils import array_to_buffer
from redisvl.utils.utils import create_uuid, current_timestamp


def test_chat_message_creation():
    session_tag = create_uuid()
    timestamp = current_timestamp()
    content = "Hello, world!"

    chat_message = ChatMessage(
        entry_id=f"{session_tag}:{timestamp}",
        role="user",
        content=content,
        session_tag=session_tag,
        timestamp=timestamp,
    )

    assert chat_message.entry_id == f"{session_tag}:{timestamp}"
    assert chat_message.role == "user"
    assert chat_message.content == content
    assert chat_message.session_tag == session_tag
    assert chat_message.timestamp == timestamp
    assert chat_message.tool_call_id is None
    assert chat_message.vector_field is None


def test_chat_message_default_id_generation():
    session_tag = create_uuid()
    timestamp = current_timestamp()
    content = "Hello, world!"

    chat_message = ChatMessage(
        role="user",
        content=content,
        session_tag=session_tag,
        timestamp=timestamp,
    )

    assert chat_message.entry_id == f"{session_tag}:{timestamp}"


def test_chat_message_with_tool_call_id():
    session_tag = create_uuid()
    timestamp = current_timestamp()
    content = "Hello, world!"
    tool_call_id = create_uuid()

    chat_message = ChatMessage(
        entry_id=f"{session_tag}:{timestamp}",
        role="user",
        content=content,
        session_tag=session_tag,
        timestamp=timestamp,
```

```
        tool_call_id=tool_call_id,
    )

    assert chat_message.tool_call_id == tool_call_id


def test_chat_message_with_vector_field():
    session_tag = create_uuid()
    timestamp = current_timestamp()
    content = "Hello, world!"
    vector_field = [0.1, 0.2, 0.3]

    chat_message = ChatMessage(
        entry_id=f"{session_tag}:{timestamp}",
        role="user",
        content=content,
        session_tag=session_tag,
        timestamp=timestamp,
        vector_field=vector_field,
    )

    assert chat_message.vector_field == vector_field


def test_chat_message_to_dict():
    session_tag = create_uuid()
    timestamp = current_timestamp()
    content = "Hello, world!"
    vector_field = [0.1, 0.2, 0.3]

    chat_message = ChatMessage(
        entry_id=f"{session_tag}:{timestamp}",
        role="user",
        content=content,
        session_tag=session_tag,
        timestamp=timestamp,
        vector_field=vector_field,
    )

    data = chat_message.to_dict(dtype="float32")

    assert data["entry_id"] == f"{session_tag}:{timestamp}"
    assert data["role"] == "user"
    assert data["content"] == content
    assert data["session_tag"] == session_tag
    assert data["timestamp"] == timestamp
    assert data["vector_field"] == array_to_buffer(vector_field, "float32")


def test_chat_message_missing_fields():
    session_tag = create_uuid()
    timestamp = current_timestamp()
    content = "Hello, world!"

    with pytest.raises(ValidationError):
        ChatMessage(
            content=content,
            session_tag=session_tag,
            timestamp=timestamp,
        )


def test_chat_message_invalid_role():
    session_tag = create_uuid()
```

```
    timestamp = current_timestamp()
    content = "Hello, world!"

    with pytest.raises(ValidationError):
        ChatMessage(
            entry_id=f"{session_tag}:{timestamp}",
            role=[1, 2, 3],  # Invalid role type
            content=content,
            session_tag=session_tag,
            timestamp=timestamp,
        )
}
```

tests/unit/test_storage.py

```
import pytest

from redisvl.index.storage import BaseStorage, HashStorage, JsonStorage


@pytest.fixture(params=[JsonStorage, HashStorage])
def storage_instance(request):
    StorageClass = request.param
    instance = StorageClass(prefix="test", key_separator=":")
    return instance


def test_key_formatting(storage_instance):
    key = "1234"
    generated_key = storage_instance._key(key, "", "")
    assert generated_key == key, "The generated key does not match the
expected format."
    generated_key = storage_instance._key(key, "", ":")
    assert generated_key == key, "The generated key does not match the
expected format."
    generated_key = storage_instance._key(key, "test", ":")
    assert (
        generated_key == f"test:{key}"
    ), "The generated key does not match the expected format."


def test_create_key(storage_instance):
    id_field = "id"
    obj = {id_field: "1234"}
    expected_key = (
        f"{storage_instance.prefix}{storage_instance.key_separator}{obj[id_field]}"
    )
    generated_key = storage_instance._create_key(obj, id_field)
    assert (
        generated_key == expected_key
    ), "The generated key does not match the expected format."


def test_validate_success(storage_instance):
    data = {"foo": "bar"}
    try:
        storage_instance._validate(data)
    except Exception as e:
        pytest.fail(f"_validate should not raise an exception here, but raised {e}")
```

```
def test_validate_failure(storage_instance):
    data = "Some invalid data type"
    with pytest.raises(TypeError):
        storage_instance._validate(data)
    data = 12345
    with pytest.raises(TypeError):
        storage_instance._validate(data)


def test_preprocess(storage_instance):
    data = {"key": "value"}
    preprocessed_data = storage_instance._preprocess(preprocess=None, obj=data)
    assert preprocessed_data == data

    def fn(d):
        d["foo"] = "bar"
        return d

    preprocessed_data = storage_instance._preprocess(fn, data)
    assert "foo" in preprocessed_data
    assert preprocessed_data["foo"] == "bar"


@pytest.mark.asyncio
async def test_preprocess(storage_instance):
    data = {"key": "value"}
    preprocessed_data = await storage_instance._apreprocess(preprocess=None, obj=data)
    assert preprocessed_data == data

    async def fn(d):
        d["foo"] = "bar"
        return d

    preprocessed_data = await storage_instance._apreprocess(data, fn)
    assert "foo" in preprocessed_data
    assert preprocessed_data["foo"] == "bar"
}
```

tests/unit/test_token_escaper.py

```
import pytest

from redisvl.utils.token_escaper import TokenEscaper


@pytest.fixture
def escaper():
    return TokenEscaper()


@pytest.mark.parametrize(
    ("test_input,expected"),
    [
        (r"a [big] test.", r"a\ \[big\]\ test\."),
        (r"hello, world!", r"hello\,\ world\!"),
        (
            r'special "quotes" (and parentheses)',
            r"special\ \"quotes\"\ \(and\ parentheses\)",
        ),
```

```
        (
            r"& symbols, like * and ?",
            r"\&\ symbols\,\ like\ \*\ and\ ?",
        ),  # TODO: question marks are not caught?
        # underscores are ignored
        (r"-dashes_and_underscores-", r"\-dashes_and_underscores\-"),
    ],
    ids=["brackets", "commas", "quotes", "symbols", "underscores"],
)
def test_escape_text_chars(escaper, test_input, expected):
    assert escaper.escape(test_input) == expected


@pytest.mark.parametrize(
    ("test_input,expected"),
    [
        # Simple tags
        ("user:name", r"user\:name"),
        ("123#comment", r"123\#comment"),
        ("hyphen-separated", r"hyphen\-separated"),
        # Tags with special characters
        ("price$", r"price\$"),
        ("super*star", r"super\*star"),
        ("tag&value", r"tag\&value"),
        ("@username", r"\@username"),
        # Space-containing tags often used in search scenarios
        ("San Francisco", r"San\ Francisco"),
        ("New Zealand", r"New\ Zealand"),
        # Multi-special-character tags
        ("complex/tag:value", r"complex\/tag\:value"),
        ("$special$tag$", r"\$special\$tag\$"),
        ("tag-with-hyphen", r"tag\-with\-hyphen"),
        # Tags with less common, but legal characters
        ("_underscore_", r"_underscore_"),
        ("dot.tag", r"dot\.tag"),
        # ("pipe|tag", r"pipe\|tag"), #TODO - pipes are not caught?
        # More edge cases with special characters
        ("(parentheses)", r"\(parentheses\)"),
        ("[brackets]", r"\[brackets\]"),
        ("{braces}", r"\{braces\}"),
        # ("question?mark", r"question\?mark"),  #TODO - question marks are not caught?
        # Unicode characters in tags
        ("你好", r"你好"),  # Assuming non-Latin characters don't need escaping
        ("emoji:😊", r"emoji\:😊"),
        # ...other cases as needed...
    ],
    ids=[
        ":",
        "#",
        "-",
        "$",
        "*",
        "&",
        "@",
        "space",
        "space-2",
        "complex",
        "special",
        "hyphen",
        "underscore",
        "dot",
        "parentheses",
        "brackets",
        "braces",
```

```
        "non-latin",
        "emoji",
    ],
)
def test_escape_tag_like_values(escaper, test_input, expected):
    assert escaper.escape(test_input) == expected


@pytest.mark.parametrize("test_input", [123, 45.67, None, [], {}])
def test_escape_non_string_input(escaper, test_input):
    with pytest.raises(TypeError):
        escaper.escape(test_input)


@pytest.mark.parametrize(
    "test_input,expected",
    [
        # ('你好，世界！', r'你好\，世界\！'), # TODO - non latin chars?
        ("😊 ❤️ 👍", r"😊\ ❤️\ 👍"),
        # ...other cases as needed...
    ],
    ids=["emoji"],
)
def test_escape_unicode_characters(escaper, test_input, expected):
    assert escaper.escape(test_input) == expected


def test_escape_empty_string(escaper):
    assert escaper.escape("") == ""


def test_escape_long_string(escaper):
    # Construct a very long string
    long_str = "a," * 1000  # This creates a string "a,a,a,a,...a,"
    expected = r"a\," * 1000  # Expected escaped string

    # Use pytest's benchmark fixture to check performance
    escaped = escaper.escape(long_str)
    assert escaped == expected
}
```

tests/unit/test_utils.py

```
import numpy as np
import pytest
from ml_dtypes import bfloat16

from redisvl.redis.utils import (
    array_to_buffer,
    buffer_to_array,
    convert_bytes,
    make_dict,
)


def test_even_number_of_elements():
    """Test with an even number of elements"""
    values = ["key1", "value1", "key2", "value2"]
    expected = {"key1": "value1", "key2": "value2"}
    assert make_dict(values) == expected
```

```
def test_odd_number_of_elements():
    """Test with an odd number of elements - expecting the last element to
be ignored"""
    values = ["key1", "value1", "key2"]
    expected = {"key1": "value1"}  # 'key2' has no pair, so it's ignored
    assert make_dict(values) == expected


def test_different_data_types():
    """Test with different data types as keys and values"""
    values = [1, "one", 2.0, "two"]
    expected = {1: "one", 2.0: "two"}
    assert make_dict(values) == expected


def test_empty_list():
    """Test with an empty list"""
    values = []
    expected = {}
    assert make_dict(values) == expected


def test_with_complex_objects():
    """Test with complex objects like lists and dicts as values"""
    key = "a list"
    value = [1, 2, 3]
    values = [key, value]
    expected = {key: value}
    assert make_dict(values) == expected


def test_simple_byte_buffer_to_floats():
    """Test conversion of a simple byte buffer into floats"""
    buffer = np.array([1.0, 2.0, 3.0], dtype=np.float32).tobytes()
    expected = [1.0, 2.0, 3.0]
    assert buffer_to_array(buffer, dtype="float32") == expected


def test_converting_different_data_types():
    """Test conversion with different data types"""
    # Float64 test
    buffer = np.array([1.0, 2.0, 3.0], dtype=np.float64).tobytes()
    expected = [1.0, 2.0, 3.0]
    assert buffer_to_array(buffer, dtype="float64") == expected


def test_empty_byte_buffer():
    """Test conversion of an empty byte buffer"""
    buffer = b""
    expected = []
    assert buffer_to_array(buffer, dtype="float32") == expected


def test_plain_bytes_to_string():
    """Test conversion of plain bytes to string"""
    data = b"hello world"
    expected = "hello world"
    assert convert_bytes(data) == expected


def test_bytes_in_dict():
    """Test conversion of bytes in a dictionary, including nested dictionaries"""
```

```
    data = {"key": b"value", "nested": {"nkey": b"nvalue"}}
    expected = {"key": "value", "nested": {"nkey": "nvalue"}}
    assert convert_bytes(data) == expected


def test_bytes_in_list():
    """Test conversion of bytes in a list, including nested lists"""
    data = [b"item1", b"item2", ["nested", b"nested item"]]
    expected = ["item1", "item2", ["nested", "nested item"]]
    assert convert_bytes(data) == expected


def test_bytes_in_tuple():
    """Test conversion of bytes in a tuple, including nested tuples"""
    data = (b"item1", b"item2", ("nested", b"nested item"))
    expected = ("item1", "item2", ("nested", "nested item"))
    assert convert_bytes(data) == expected


def test_non_bytes_data():
    """Test handling of non-bytes data types"""
    data = "already a string"
    expected = "already a string"
    assert convert_bytes(data) == expected


def test_bytes_with_invalid_utf8():
    """Test handling bytes that cannot be decoded with UTF-8"""
    data = b"\xff\xff"  # Invalid in UTF-8
    expected = data
    assert convert_bytes(data) == expected


def test_simple_list_to_bytes_default_dtype():
    """Test conversion of a simple list of floats to bytes using the default dtype"""
    array = [1.0, 2.0, 3.0]
    expected = np.array(array, dtype=np.float32).tobytes()
    assert array_to_buffer(array, "float32") == expected


def test_list_to_bytes_non_default_dtype():
    """Test conversion with a non-default dtype"""
    array = [1.0, 2.0, 3.0]
    dtype = np.float64
    expected = np.array(array, dtype=dtype).tobytes()
    assert array_to_buffer(array, dtype="float64") == expected


def test_empty_list_to_bytes():
    """Test conversion of an empty list"""
    array = []
    expected = np.array(array, dtype=np.float32).tobytes()
    assert array_to_buffer(array, dtype="float32") == expected


@pytest.mark.parametrize("dtype", ["float64", "float32", "float16", "bfloat16"])
def test_conversion_with_various_dtypes(dtype):
    """Test conversion of a list of floats to bytes with various dtypes"""
    array = [1.0, -2.0, 3.5]
    expected = np.array(array, dtype=dtype).tobytes()
    assert array_to_buffer(array, dtype=dtype) == expected


def test_conversion_with_invalid_floats():
```

```
    """Test conversion with invalid float values (numpy should handle them)"""
    array = [float("inf"), float("-inf"), float("nan")]
    result = array_to_buffer(array, "float16")
    assert len(result) > 0  # Simple check to ensure it returns anything
}
```

The End

This PDF was generated by gitprint.me